Community Experience Distilled

Learning Image Processing with OpenCV

Exploit the amazing features of OpenCV to create powerful image processing applications through easy-to-follow examples

Gloria Bueno García
José Luis Espinosa Aranda
Ismael Serrano Gracia
Oscar Deniz Suarez
Jesus Salido Tercero
Noelia Vállez Enano

[PACKT] open source*
PUBLISHING
community experience distilled

Learning Image Processing with OpenCV

Exploit the amazing features of OpenCV to create powerful image processing applications through easy-to-follow examples

Gloria Bueno García

Oscar Deniz Suarez

José Luis Espinosa Aranda

Jesus Salido Tercero

Ismael Serrano Gracia

Noelia Vállez Enano

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Image Processing with OpenCV

First published: March 2015

Production reference: 1230315

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

ISBN 978-1-78328-765-9

www.packtpub.com

Credits

Authors
Gloria Bueno García
Oscar Deniz Suarez
José Luis Espinosa Aranda
Jesus Salido Tercero
Ismael Serrano Gracia
Noelia Vállez Enano

Reviewers
Walter Lucetti
André de Souza Moreira
Marvin Smith

Commissioning Editor
Julian Ursell

Acquisition Editor
Sam Wood

Content Development Editor
Kirti Patil

Technical Editor
Faisal Siddiqui

Copy Editor
Stuti Srivastava

Project Coordinator
Nidhi Joshi

Proofreaders
Martin Diver
Maria Gould
Samantha Lyon

Indexer
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Authors

Gloria Bueno García holds a PhD in machine vision from Coventry University, UK. She has experience working as the principal researcher in several research centers, such as UMR 7005 research unit CNRS/ Louis Pasteur Univ. Strasbourg (France), Gilbert Gilkes & Gordon Technology (UK), and CEIT San Sebastian (Spain). She is the author of two patents, one registered type of software, and more than 100 refereed papers. Her interests are in 2D/3D multimodality image processing and artificial intelligence. She leads the VISILAB research group at the University of Castilla-La Mancha. She has coauthored a book on OpenCV programming for mobile devices: *OpenCV essentials*, *Packt Publishing*.

This is dedicated to our sons for the time we have not been able to play with them and our parents for their unconditional support during our lifetime. Thanks from Gloria and Oscar.

Oscar Deniz Suarez's research interests are mainly focused on computer vision and pattern recognition. He is the author of more than 50 refereed papers in journals and conferences. He received the runner-up award for the best PhD work on computer vision and pattern recognition by AERFAI and the Image File and Reformatting Software Challenge Award by Innocentive Inc. He has been a national finalist for the 2009 Cor Baayen award. His work is used by cutting-edge companies, such as Existor, Gliif, Tapmedia, E-Twenty, and others, and has also been added to OpenCV. Currently, he works as an associate professor at the University of Castilla-La Mancha and contributes to VISILAB. He is a senior member of IEEE and is affiliated with AAAI, SIANI, CEA-IFAC, AEPIA, and AERFAI-IAPR. He serves as an academic editor of the PLoS ONE journal. He has been a visiting researcher at Carnegie Mellon University, Imperial College London, and Leica Biosystems. He has coauthored two books on OpenCV previously.

José Luis Espinosa Aranda holds a PhD in computer science from the University of Castilla-La Mancha. He has been a finalist for Certamen Universitario Arquímedes de Introducción a la Investigación científica in 2009 for his final degree project in Spain. His research interests involve computer vision, heuristic algorithms, and operational research. He is currently working at the VISILAB group as an assistant researcher and developer in computer vision topics.

This is dedicated to my parents and my brothers.

Jesus Salido Tercero gained his electrical engineering degree and PhD (1996) from Universidad Politécnica de Madrid (Spain). He then spent 2 years (1997 and 1998) as a visiting scholar at the Robotics Institute (Carnegie Mellon University, Pittsburgh, USA), working on cooperative multirobot systems. Since his return to the Spanish University of Castilla-La Mancha, he spends his time teaching courses on robotics and industrial informatics, along with research on vision and intelligent systems. Over the last 3 years, his efforts have been directed to develop vision applications on mobile devices. He has coauthored a book on OpenCV programming for mobile devices.

This is dedicated to those to whom I owe all I am: my parents, Sagrario and Maria.

Ismael Serrano Gracia received his degree in computer science in 2012 from the University of Castilla-La Mancha. He got the highest marks for his final degree project on person detection. This application uses depth cameras with OpenCV libraries. Currently, he is a PhD candidate at the same university, holding a research grant from the Spanish Ministry of Science and Research. He is also working at the VISILAB group as an assistant researcher and developer on different computer vision topics.

This is dedicated to my parents, who have given me the opportunity of education and have supported me throughout my life. It is also dedicated to my supervisor, Dr. Oscar Deniz, who has been a friend, guide, and helper. Finally, it is dedicated to my friends and my girlfriend, who have always helped me and believed that I could do this.

Noelia Vállez Enano has liked computers since her childhood, though she didn't have one before her mid-teens. In 2009, she finished her studies in computer science at the University of Castilla-La Mancha, where she graduated with top honors. She started working at the VISILAB group through a project on mammography CAD systems and electronic health records. Since then, she has obtained a master's degree in physics and mathematics and has enrolled for a PhD degree. Her work involves using image processing and pattern recognition methods. She also likes teaching and working in other areas of artificial intelligence.

About the Reviewers

Walter Lucetti, known on the Internet as Myzhar, is an Italian computer engineer with a specialization in robotics and robotics perception. He received the laurea degree in 2005 while studying at Research Center "E.Piaggio" in Pisa (Italy), where he wrote a thesis about 3D mapping of the real world using a 2D laser tilted with a servo motor, fusing 3D with RGB data. While writing the thesis, he encountered OpenCV for the first time—it was early 2004 and OpenCV was at its larval stage.

After the laurea, he started working as software developer for low-level embedded systems and high-level desktop systems. He greatly improved his knowledge of computer vision and machine learning as a researcher at Gustavo Stefanini Advanced Robotics Center in La Spezia (Italy), a spinoff of PERCRO Laboratory of Scuola Superiore Sant'Anna of Pisa (Italy).

Currently, he is working in the software industry, writing firmware for embedded ARM systems, software for desktop systems based on the Qt framework, and intelligent algorithms for video surveillance systems based on OpenCV and CUDA.

He is also working on a personal robotic project: MyzharBot. MyzharBot is a tracked ground mobile robot that uses computer vision to detect obstacles and to analyze and explore the environment. The robot is guided by algorithms based on ROS, CUDA, and OpenCV. You can follow the project on this website: `http://myzharbot.robot-home.it`.

André de Souza Moreira has a master's degree in computer science, with emphasis on computer graphics from the Pontifical Catholic University of Rio de Janeiro (Brazil).

He graduated with a bachelor of computer science degree from Universidade Federal do Maranhão (UFMA) in Brazil. During his undergraduate degree, he was a member of Labmint's research team and worked with medical imaging, specifically, breast cancer detection and diagnosis using image processing.

Currently, he works as a researcher and system analyst at Instituto Tecgraf, one of the major research and development labs in computer graphics in Brazil. He has been working extensively with PHP, HTML, and CSS since 2007, and nowadays, he develops projects in C ++ 11 / C ++ 14, along with SQLite, Qt, Boost, and OpenGL. More information about him can be acquired on his personal website at `www.andredsm.com`.

Marvin Smith is currently a software engineer in the defense industry, specializing in photogrammetry and remote sensing. He received his BS degree in computer science from the University of Nevada Reno. His technical interests include high performance computing, distributed image processing, and multispectral imagery exploitation. Prior to working in defense, Marvin held internships with the Intelligent Robotics Group at the NASA Ames Research Center and the Nevada Automotive Test Center.

Table of Contents

Preface

OpenCV, arguably the most widely used computer vision library, includes hundreds of ready-to-use imaging and vision functions and is used in both academia and industry. As cameras get cheaper and imaging features grow in demand, the range of applications using OpenCV increases significantly, both for desktop and mobile platforms.

This book provides an example-based tour of OpenCV's main image processing algorithms. While other OpenCV books try to explain the underlying theory or provide large examples of nearly complete applications, This book is aimed at people who want to have an easy-to-understand working example as soon as possible, and possibly develop additional features on top of that.

The book starts with an introductory chapter in which the library installation is explained, the structure of the library is described, and basic image and video reading and writing examples are given. From this, the following functionalities are covered: handling of images and videos, basic image processing tools, correcting and enhancing images, color, video processing, and computational photography. Last but not least, advanced features such as GPU-based accelerations are also considered in the final chapter. New functions and techniques in the latest major release, OpenCV 3, are explained throughout.

What this book covers

Chapter 1, *Handling Image and Video Files*, shows you how to read image and video files. It also shows basic user-interaction tools, which are very useful in image processing to change a parameter value, select regions of interest, and so on.

Chapter 2, *Establishing Image Processing Tools*, describes the main data structures and basic procedures needed in subsequent chapters.

Chapter 3, *Correcting and Enhancing Images*, deals with transformations typically used to correct image defects. This chapter covers filtering, point transformations using Look Up Tables, geometrical transformations, and algorithms for inpainting and denoising images.

Chapter 4, *Processing Color*, deals with color topics in image processing. This chapter explains how to use different color spaces and perform color transfers between two images.

Chapter 5, *Image Processing for Video*, covers techniques that use a video or a sequence of images. This chapter is focused on algorithms' implementation for video stabilization, superresolution, and stitching.

Chapter 6, *Computational Photography*, explains how to read HDR images and perform tone mapping on them.

Chapter 7, *Accelerating Image Processing*, covers an important topic in image processing: speed. Modern GPUs are the best available technology to accelerate time-consuming image processing tasks.

What you need for this book

The purpose of this book is to teach you OpenCV image processing by taking you through a number of practical image processing projects. The latest version, Version 3.0 of OpenCV, will be used.

Each chapter provides several ready-to-use examples to illustrate the concepts covered in it. The book is, therefore, focused on providing you with a working example as soon as possible so that they can develop additional features on top of that.

To use this book, only free software is needed. All the examples have been developed and tested with the freely available Qt Creator IDE and GNU/GCC compiler. The CMake tool is also used to configure the build process of the OpenCV library on the target platform. Moreover, the freely available OpenCL SDK is required for the GPU acceleration examples shown in *Chapter 7*, *Accelerating Image Processing*.

Who this book is for

This book is intended for readers who already know C++ programming and want to learn how to do image processing using OpenCV. You are expected to have a minimal background in the theory of image processing. The book does not cover topics that are more related to computer vision, such as feature and object detection, tracking, or machine learning.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, folder names, filenames, file extensions, pathnames, system variables, URLs, and user input are shown as follows: "Each module has an associated header file (for example `core.hpp`)."

A block of code is set as follows:

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

int main(int argc, char *argv[])
{
    Mat frame; // Container for each frame
```

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

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

using namespace std;
using namespace cv;

int main(int argc, char *argv[])
{
```

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

```
C:\opencv-buildQt\install
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/ImageProcessingwithOpenCV_Graphics.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Handling Image and Video Files

This chapter is intended as a first contact with OpenCV, its installation, and first basic programs. We will cover the following topics:

- A brief introduction to OpenCV for the novice, followed by an easy step-by-step guide to the installation of the library
- A quick tour of OpenCV's structure after the installation in the user's local disk
- Quick recipes to create projects using the library with some common programming frameworks
- How to use the functions to read and write images and videos
- Finally, we describe the library functions to add rich user interfaces to the software projects, including mouse interaction, drawing primitives, and Qt support

An introduction to OpenCV

Initially developed by Intel, **OpenCV** (**Open Source Computer Vision**) is a free cross-platform library for real-time image processing that has become a *de facto* standard tool for all things related to Computer Vision. The first version was released in 2000 under **BSD license** and since then, its functionality has been very much enriched by the scientific community. In 2012, the nonprofit foundation OpenCV.org took on the task of maintaining a support site for developers and users.

At the time of writing this book, a new major version of OpenCV (Version 3.0) is available, still on beta status. Throughout the book, we will present the most relevant changes brought with this new version.

OpenCV is available for the most popular operating systems, such as GNU/Linux, OS X, Windows, Android, iOS, and some more. The first implementation was in the **C** programming language; however, its popularity grew with its C++ implementation as of Version 2.0. New functions are programmed with C++. However, nowadays, the library has a full interface for other programming languages, such as Java, Python, and MATLAB/Octave. Also, wrappers for other languages (such as C#, Ruby, and Perl) have been developed to encourage adoption by programmers.

In an attempt to maximize the performance of computing intensive vision tasks, OpenCV includes support for the following:

- Multithreading on multicore computers using **Threading Building Blocks** (**TBB**)—a template library developed by Intel.
- A subset of **Integrated Performance Primitives** (**IPP**) on Intel processors to boost performance. Thanks to Intel, these primitives are freely available as of Version 3.0 beta.
- Interfaces for processing on **Graphic Processing Unit** (**GPU**) using **Compute Unified Device Architecture** (**CUDA**) and **Open Computing Language** (**OpenCL**).

The applications for OpenCV cover areas such as segmentation and recognition, 2D and 3D feature toolkits, object identification, facial recognition, motion tracking, gesture recognition, image stitching, **high dynamic range** (**HDR**) imaging, augmented reality, and so on. Moreover, to support some of the previous application areas, a module with statistical machine learning functions is included.

Downloading and installing OpenCV

OpenCV is freely available for download at `http://opencv.org`. This site provides the last version for distribution (currently, 3.0 beta) and older versions.

Special care should be taken with possible errors when the downloaded version is a nonstable release, for example, the current 3.0 beta version.

On `http://opencv.org/downloads.html`, suitable versions of OpenCV for each platform can be found. The code and information of the library can be obtained from different repositories depending on the final purpose:

- The main repository (at `http://sourceforge.net/projects/opencvlibrary`), devoted to final users. It contains binary versions of the library and ready-to-compile sources for the target platform.
- The test data repository (at `https://github.com/itseez/opencv_extra`) with sets of data to test purposes of some library modules.
- The contributions repository (at `http://github.com/itseez/opencv_contrib`) with the source code corresponding to extra and cutting-edge features supplied by contributors. This code is more error-prone and less tested than the main trunk.

> With the last version, OpenCV 3.0 beta, the extra contributed modules are not included in the main package. They should be downloaded separately and explicitly included in the compilation process through the proper options. Be cautious if you include some of those contributed modules, because some of them have dependencies on third-party software not included with OpenCV.

- The documentation site (at `http://docs.opencv.org/master/`) for each of the modules, including the contributed ones.
- The development repository (at `https://github.com/Itseez/opencv`) with the current development version of the library. It is intended for developers of the main features of the library and the "impatient" user who wishes to use the last update even before it is released.

Rather than GNU/Linux and OS X, where OpenCV is distributed as source code only, in the Windows distribution, one can find precompiled (with Microsoft Visual C++ v10, v11, and v12) versions of the library. Each precompiled version is ready to be used with Microsoft compilers. However, if the primary intention is to develop projects with a different compiler framework, we need to compile the library for that specific compiler (for example, GNU GCC).

The fastest route to working with OpenCV is to use one of the precompiled versions included with the distribution. Then, a better choice is to build a fine-tuned version of the library with the best settings for the local platform used for software development. This chapter provides the information to build and install OpenCV on Windows. Further information to set the library on Linux can be found at `http://docs.opencv.org/doc/tutorials/introduction/linux_install` and `https://help.ubuntu.com/community/OpenCV`.

Getting a compiler and setting CMake

A good choice for cross-platform development with OpenCV is to use the **GNU toolkit** (including gmake, g++, and gdb). The GNU toolkit can be easily obtained for the most popular operating systems. Our preferred choice for a development environment consists of the GNU toolkit and the cross-platform **Qt framework**, which includes the Qt library and the Qt Creator **Integrated Development Environment (IDE)**. The Qt framework is freely available at `http://qt-project.org/`.

After installing the compiler on Windows, remember to properly set the `Path` environment variable, adding the path for the compiler's executable, for example, `C:\Qt\Qt5.2.1\5.2.1\mingw48_32\bin` for the GNU/compilers included with the Qt framework. On Windows, the free **Rapid Environment Editor** tool (available at `http://www.rapidee.com`) provides a convenient way to change `Path` and other environment variables.

To manage the build process for the OpenCV library in a compiler-independent way, CMake is the recommended tool. CMake is a free and open source cross-platform tool available at `http://www.cmake.org/`.

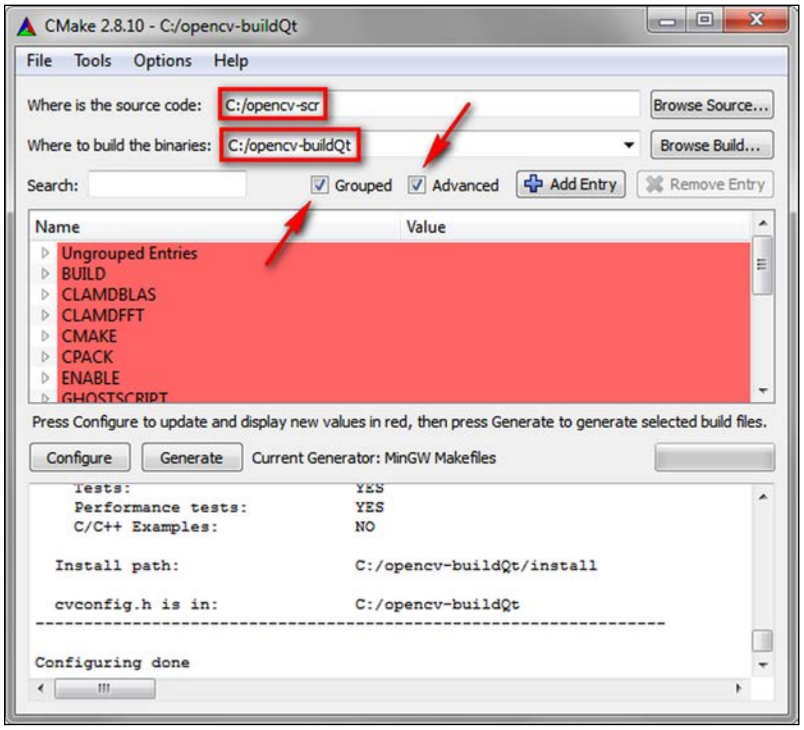
Configuring OpenCV with CMake

Once the sources of the library have been downloaded into the local disk, it is required that you configure the makefiles for the compilation process of the library. CMake is the key tool for an easy configuration of OpenCV's installation process. It can be used from the command line or in a more user-friendly way with its **Graphical User Interface (GUI)** version.

The steps to configure OpenCV with CMake can be summarized as follows:

1. Choose the source (let's call it `OPENCV_SRC` in what follows) and target (`OPENCV_BUILD`) directories. The target directory is where the compiled binaries will be located.
2. Mark the **Grouped** and **Advanced** checkboxes and click on the **Configure** button.
3. Choose the desired compiler (for example, GNU default compilers, MSVC, and so on).
4. Set the preferred options and unset those not desired.
5. Click on the **Configure** button and repeat steps 4 and 5 until no errors are obtained.
6. Click on the **Generate** button and close CMake.

The following screenshot shows you the main window of CMake with the source and target directories and the checkboxes to group all the available options:



Where is the source code: C:/opencv-scr

Where to build the binaries: C:/opencv-buildQt

☑ Grouped ☑ Advanced

Ungrouped Entries
BUILD
CLAMDBLAS
CLAMDFFT
CMAKE
CPACK
ENABLE
GHOSTSCRIPT

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: MinGW Makefiles

```
    Tests:                      YES
    Performance tests:          YES
    C/C++ Examples:             NO

  Install path:                 C:/opencv-buildQt/install

  cvconfig.h is in:             C:/opencv-buildQt
-----------------------------------------------------------------

Configuring done
```

The main window of CMake after the preconfiguration step

For brevity, we use OPENCV_BUILD and OPENCV_SRC in this text to denote the target and source directories of the OpenCV local setup, respectively. Keep in mind that all directories should match your current local configuration.

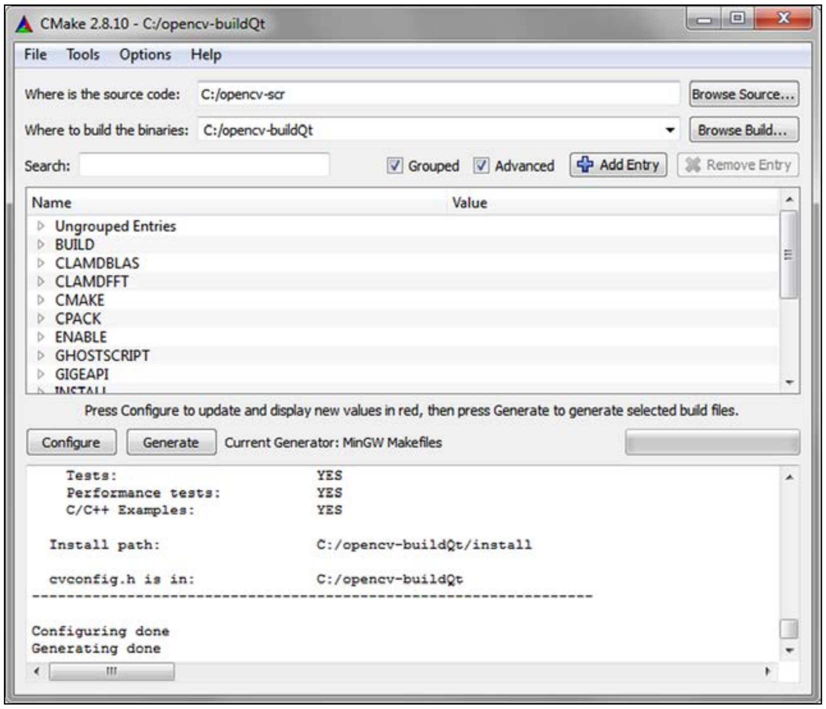
During the preconfiguration process, CMake detects the compilers present and many other local properties to set the build process of OpenCV. The previous screenshot displays the main **CMake** window after the preconfiguration process, showing the grouped options in red.

It is possible to leave the default options unchanged and continue the configuration process. However, some convenient options can be set:

- BUILD_EXAMPLES: This is set to build some examples using OpenCV.
- BUILD_opencv_<module_name>: This is set to include the module (module_name) in the build process.
- OPENCV_EXTRA_MODULES_PATH: This is used when you need some extra contributed module; set the path for the source code of the extra modules here (for example, C:/opencv_contrib-master/modules).
- WITH_QT: This is turned on to include the Qt functionality into the library.
- WITH_IPP: This option is turned on by default. The current OpenCV 3.0 version includes a subset of the **Intel Integrated Performance Primitives** (**IPP**) that speed up the execution time of the library.

If you compile the new OpenCV 3.0 (beta), be cautious because some unexpected errors have been reported related to the IPP inclusion (that is, with the default value of this option). We recommend that you unset the WITH_IPP option.

If the configuration steps with CMake (loop through steps 4 and 5) don't produce any further errors, it is possible to generate the final makefiles for the build process. The following screenshot shows you the main window of CMake after a generation step without errors:



Where is the source code: C:/opencv-scr
Where to build the binaries: C:/opencv-buildQt
Search:

Ungrouped Entries
BUILD
CLAMDBLAS
CLAMDFFT
CMAKE
CPACK
ENABLE
GHOSTSCRIPT
GIGEAPI
INSTALL

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: MinGW Makefiles

```
    Tests:                          YES
    Performance tests:              YES
    C/C++ Examples:                 YES

  Install path:                     C:/opencv-buildQt/install

  cvconfig.h is in:                 C:/opencv-buildQt
-----------------------------------------------------------------

Configuring done
Generating done
```

Compiling and installing the library

The next step after the generation process of makefiles with CMake is the compilation with the proper `make` tool. This tool is usually executed on the command line (the console) from the target directory (the one set at the CMake configuration step). For example, in **Windows**, the compilation should be launched from the command line as follows:

```
OPENCV_BUILD>mingw32-make
```

This command launches a build process using the makefiles generated by CMake. The whole compilation typically takes several minutes. If the compilation ends without errors, the installation continues with the execution of the following command:

```
OPENCV_BUILD>mingw32-make install
```

This command copies the OpenCV binaries to the `OPENCV_BUILD\install` directory.

If something went wrong during the compilation, we should run CMake again to change the options selected during the configuration. Then, we should regenerate the makefiles.

The installation ends by adding the location of the library binaries (for example, in **Windows**, the resulting **DLL** files are located at `OPENCV_BUILD\install\x64\mingw\bin`) to the **Path** environment variable. Without this directory in the **Path** field, the execution of every OpenCV executable will give an error as the library binaries won't be found.

To check the success of the installation process, it is possible to run some of the examples compiled along with the library (if the `BUILD_EXAMPLES` option was set using CMake). The code samples (written in C++) can be found at `OPENCV_BUILD\install\x64\mingw\samples\cpp`.

The short instructions given to install OpenCV apply to **Windows**. A detailed description with the prerequisites for **Linux** can be read at `http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html`. Although the tutorial applies to OpenCV 2.0, almost all the information is still valid for Version 3.0.

The structure of OpenCV

Once OpenCV is installed, the `OPENCV_BUILD\install` directory will be populated with three types of files:

- **Header files**: These are located in the `OPENCV_BUILD\install\include` subdirectory and are used to develop new projects with OpenCV.
- **Library binaries**: These are static or dynamic libraries (depending on the option selected with CMake) with the functionality of each of the OpenCV modules. They are located in the `bin` subdirectory (for example, `x64\mingw\bin` when the GNU compiler is used).
- **Sample binaries**: These are executables with examples that use the libraries. The sources for these samples can be found in the source package (for example, `OPENCV_SRC\sources\samples`).

OpenCV has a modular structure, which means that the package includes a static or dynamic (DLL) library for each module. The official documentation for each module can be found at `http://docs.opencv.org/master/`. The main modules included in the package are:

- `core`: This defines the basic functions used by all the other modules and the fundamental data structures including the important multidimensional array `Mat`.

- `highgui`: This provides simple **user interface** (**UI**) capabilities. Building the library with Qt support (the `WITH_QT` CMake option) allows UI compatibility with such a framework.
- `imgproc`: These are image processing functions that include filtering (linear and nonlinear), geometric transformations, color space conversion, histograms, and so on.
- `imgcodecs`: This is an easy-to-use interface to read and write images.

Pay attention to the changes in modules since OpenCV 3.0 as some functionality has been moved to a new module (for example, reading and writing images functions were moved from `highgui` to `imgcodecs`).

- `photo`: This includes Computational Photography including inpainting, denoising, `High` **Dynamic Range** (**HDR**) imaging, and some others.
- `stitching`: This is used for image stitching.
- `videoio`: This is an easy-to-use interface for video capture and video codecs.
- `video`: This supplies the functionality of video analysis (motion estimation, background extraction, and object tracking).
- `features2d`: These are functions for feature detection (corners and planar objects), feature description, feature matching, and so on.
- `objdetect`: These are functions for object detection and instances of predefined detectors (such as faces, eyes, smile, people, cars, and so on).

Some other modules are `calib3d` (camera calibration), `flann` (clustering and search), `ml` (machine learning), `shape` (shape distance and matching), `superres` (super resolution), `video` (video analysis), and `videostab` (video stabilization).

As of Version 3.0 beta, the new contributed modules are distributed in a separate package (`opencv_contrib-master.zip`) that can be downloaded from `https://github.com/itseez/opencv_contrib`. These modules provide extra features that should be fully understood before using them. For a quick overview of the new functionality in the new release of OpenCV (Version 3.0), refer to the document at `http://opencv.org/opencv-3-0-beta.html`.

Creating user projects with OpenCV

In this book, we assume that C++ is the main language for programming image processing applications, although interfaces and wrappers for other programming languages are actually provided (for instance, Python, Java, MATLAB/Octave, and some more).

In this section, we explain how to develop applications with OpenCV's C++ API using an easy-to-use cross-platform framework.

General usage of the library

To develop an OpenCV application with **C++,** we require our code to:

- Include the OpenCV header files with definitions
- Link the OpenCV libraries (binaries) to get the final executable

The OpenCV header files are located in the `OPENCV_BUILD\install\include\opencv2` directory where there is a file (`*.hpp`) for each of the modules. The inclusion of the header file is done with the `#include` directive, as shown here:

```
#include <opencv2/<module_name>/<module_name>.hpp>
// Including the header file for each module used in the code
```

With this directive, it is possible to include every header file needed by the user program. On the other hand, if the `opencv.hpp` header file is included, all the header files will be automatically included as follows:

```
#include <opencv2/opencv.hpp>
// Including all the OpenCV's header files in the code
```

Remember that all the modules installed locally are defined in the `OPENCV_BUILD\install\include\opencv2\opencv_modules.hpp` header file, which is generated automatically during the building process of OpenCV.

The use of the `#include` directive is not always a guarantee for the correct inclusion of the header files, because it is necessary to tell the compiler where to find the include files. This is achieved by passing a special argument with the location of the files (such as `I\<location>` for GNU compilers).

The linking process requires you to provide the linker with the libraries (dynamic or static) where the required OpenCV functionality can be found. This is usually done with two types of arguments for the linker: the location of the library (such as `-L<location>` for GNU compilers) and the name of the library (such as `-l<module_name>`).

> You can find a complete list of available online documentation for GNU GCC and Make at `https://gcc.gnu.org/onlinedocs/` and `https://www.gnu.org/software/make/manual/`.

Tools to develop new projects

The main prerequisites to develop our own OpenCV **C++** applications are:

- **OpenCV header files and library binaries**: Of course we need to compile OpenCV, and the auxiliary libraries are prerequisites for such a compilation. The package should be compiled with the same compiler used to generate the user application.
- **A C++ compiler**: Some associate tools are convenient as the *code editor*, *debugger*, *project manager*, *build process manager* (for instance CMake), *revision control system* (such as Git, Mercurial, SVN, and so on), and *class inspector*, among others. Usually, these tools are deployed together in a so-called **Integrated Development Environment** (**IDE**).
- **Any other auxiliary libraries**: Optionally, any other auxiliary libraries needed to program the final application, such as graphical, statistical, and so on will be required.

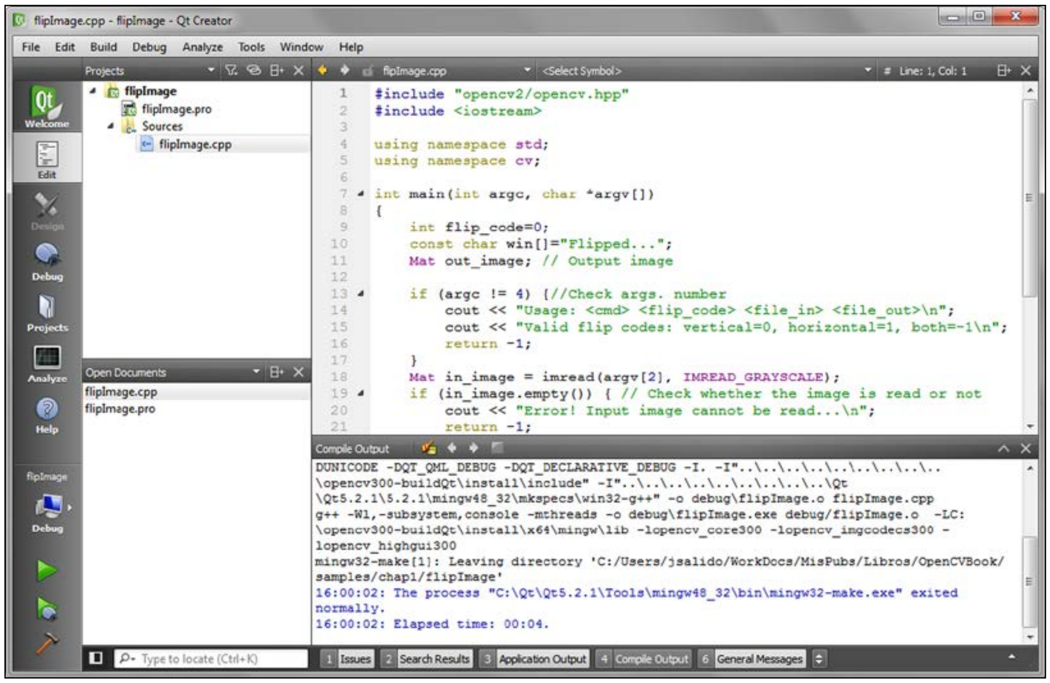
The most popular available compiler kits to program OpenCV C++ applications are:

- **Microsoft Visual C (MSVC)**: This is only supported on Windows and it is very well integrated with the IDE Visual Studio, although it can be also integrated with other cross-platform IDEs, such as Qt Creator or Eclipse. Versions of MSVC that currently compatible with the latest OpenCV release are VC 10, VC 11, and VC 12 (Visual Studio 2010, 2012, and 2013).
- **GNU Compiler Collection GNU GCC**: This is a cross-platform compiler system developed by the GNU project. For Windows, this kit is known as **MinGW** (**Minimal GNU GCC**). The version compatible with the current OpenCV release is GNU GCC 4.8. This kit may be used with several IDEs, such as Qt Creator, Code::Blocks, Eclipse, among others.

For the examples presented in this book, we used the MinGW 4.8 compiler kit for Windows plus the Qt 5.2.1 library and the Qt Creator IDE (3.0.1). The cross-platform Qt library is required to compile OpenCV with the new UI capabilities provided by such a library.

> For Windows, it is possible to download a Qt bundle (including Qt library, Qt Creator, and the MinGW kit) from `http://qt-project.org/`. The bundle is approximately 700 MB.

Qt Creator is a cross-platform IDE for **C++** that integrates the tools we need to code applications. In Windows, it may be used with MinGW or MSVC. The following screenshot shows you the Qt Creator main window with the different panels and views for an OpenCV **C++** project:



```cpp
#include "opencv2/opencv.hpp"
#include <iostream>

using namespace std;
using namespace cv;

int main(int argc, char *argv[])
{
    int flip_code=0;
    const char win[]="Flipped...";
    Mat out_image; // Output image

    if (argc != 4) {//Check args. number
        cout << "Usage: <cmd> <flip_code> <file_in> <file_out>\n";
        cout << "Valid flip codes: vertical=0, horizontal=1, both=-1\n";
        return -1;
    }
    Mat in_image = imread(argv[2], IMREAD_GRAYSCALE);
    if (in_image.empty()) { // Check whether the image is read or not
        cout << "Error! Input image cannot be read...\n";
        return -1;
```

```
DUNICODE -DQT_QML_DEBUG -DQT_DECLARATIVE_DEBUG -I. -I"..\..\..\..\..\..\..\..\..
\opencv300-buildQt\install\include" -I"..\..\..\..\..\..\..\..\..\Qt
\Qt5.2.1\5.2.1\mingw48_32\mkspecs\win32-g++" -o debug\flipImage.o flipImage.cpp
g++ -Wl,-subsystem,console -mthreads -o debug\flipImage.exe debug/flipImage.o  -LC:
\opencv300-buildQt\install\x64\mingw\lib -lopencv_core300 -lopencv_imgcodecs300 -
lopencv_highgui300
mingw32-make[1]: Leaving directory 'C:/Users/jsalido/WorkDocs/MisPubs/Libros/OpenCVBook/
samples/chap1/flipImage'
16:00:02: The process "C:\Qt\Qt5.2.1\Tools\mingw48_32\bin\mingw32-make.exe" exited
normally.
16:00:02: Elapsed time: 00:04.
```

The main window of Qt Creator with some views from an OpenCV C++ project

Creating an OpenCV C++ program with Qt Creator

Next, we explain how to create a code project with the Qt Creator IDE. In particular, we apply this description to an OpenCV example.

We can create a project for any OpenCV application using Qt Creator by navigating to **File | New File** or **File | Project…** and then navigating to **Non-Qt Project | Plain C++ Project**. Then, we have to choose a project name and the location at which it will be stored. The next step is to pick a kit (that is, the compiler) for the project (in our case, **Desktop Qt 5.2.1 MinGW 32 bit**) and the location for the binaries generated. Usually, two possible build configurations (profiles) are used: `debug` and `release`. These profiles set the appropriate flags to build and run the binaries.

When a project is created using Qt Creator, two special files (with `.pro` and `.pro.user` extensions) are generated to configure the build and run processes. The build process is determined by the kit chosen during the creation of the project. With the **Desktop Qt 5.2.1 MinGW 32 bit** kit, this process relies on the qmake and mingw32-make tools. Using the `*.pro` file as the input, qmake generates the makefile that drives the build process for each profile (that is, `release` and `debug`). The qmake tool is used from the Qt Creator IDE as an alternative to CMake to simplify the build process of software projects. It automates the generation of makefiles from a few lines of information.

The following lines represent an example of a `*.pro` file (for example, `showImage.pro`):

```
TARGET: showImage
TEMPLATE = app
CONFIG += console
CONFIG -= app_bundle
CONFIG -= qt
SOURCES += \
    showImage.cpp
INCLUDEPATH += C:/opencv300-buildQt/install/include
LIBS += -LC:/opencv300-buildQt/install/x64/mingw/lib \
    -lopencv_core300.dll \
    -lopencv_imgcodecs300.dll\
    -lopencv_highgui300.dll\
    -lopencv_imgproc300.dll
```

The preceding file illustrates the options that qmake needs to generate the appropriate makefiles to build the binaries for our project. Each line starts with a tag indicating an option (`TARGET`, `CONFIG`, `SOURCES`, `INCLUDEPATH`, and `LIBS`) followed with a mark to add (`+=`) or remove (`-=`) the value of the option. In this sample project, we use the non-Qt console application. The executable file is `showImage.exe` (`TARGET`) and the source file is `showImage.cpp` (`SOURCES`). As this project is an OpenCV-based application, the two last tags indicate the location of the header files (`INCLUDEPATH`) and the OpenCV libraries (`LIBS`) used by this particular project (`core`, `imgcodecs`, `highgui`, and `imgproc`). Note that a backslash at the end of the line denotes continuation in the next line.

For a detailed description of the tools (including Qt Creator and **qmake**) developed within the Qt project, visit `http://doc.qt.io/`.

Reading and writing image files

Image processing relies on getting an image (for instance, a photograph or a video fame) and "playing" with it by applying signal processing techniques on it to get the desired results. In this section, we show you how to read images from files using the functions supplied by OpenCV.

The basic API concepts

The `Mat` class is the main data structure that stores and manipulates images in OpenCV. This class is defined in the `core` module. OpenCV has implemented mechanisms to allocate and release memory automatically for these data structures. However, the programmer should still take special care when data structures share the same buffer memory. For instance, the assignment operator does not copy the memory content from an object (`Mat A`) to another (`Mat B`); it only copies the reference (the memory address of the content). Then, a change in one object (`A` or `B`) affects both objects. To duplicate the memory content of a `Mat` object, the `Mat::clone()` member function should be used.

Many functions in OpenCV process dense single or multichannel arrays, usually using the Mat class. However, in some cases, a different datatype may be convenient, such as std::vector<>, Matx<>, Vec<>, or Scalar. For this purpose, OpenCV provides the proxy classes InputArray and OutputArray, which allow any of the previous types to be used as parameters for functions.

The Mat class is used for dense n-dimensional single or multichannel arrays. It can actually store real or complex-valued vectors and matrices, colored or grayscale images, histograms, point clouds, and so on.

There are many different ways to create a Mat object, the most popular being the constructor where the size and type of the array are specified as follows:

```
Mat(nrows, ncols, type, fillValue)
```

The initial value for the array elements might be set by the Scalar class as a typical four-element vector (for each RGB and transparency component of the image stored in the array). Next, we show you a usage example of Mat as follows:

```
Mat img_A(4, 4, CV_8U, Scalar(255));
// White image:
// 4 x 4 single-channel array with 8 bits of unsigned integers
// (up to 255 values, valid for a grayscale image, for example,
// 255=white)
```

The DataType class defines the primitive datatypes for OpenCV. The primitive datatypes can be bool, unsigned char, signed char, unsigned short, signed short, int, float, double, or a tuple of values of one of these primitive types. Any primitive type can be defined by an identifier in the following form:

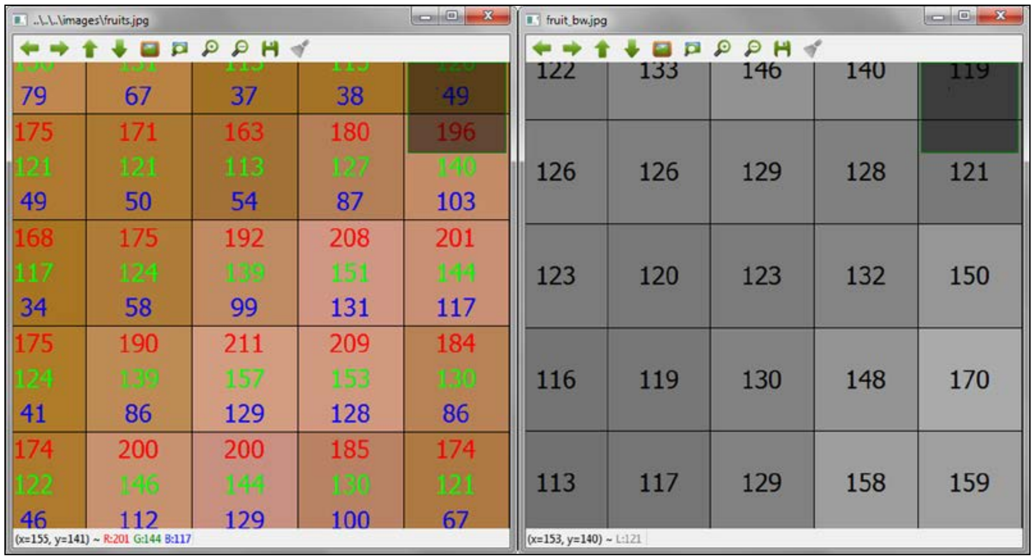
```
CV_<bit depth>{U|S|F}C(<number of channels>)
```

In the preceding code U, S, and F stand for unsigned, signed, and float, respectively. For the single channel arrays, the following enumeration is applied, describing the datatypes:

```
enum {CV_8U=0, CV_8S=1, CV_16U=2, CV_16S=3,
  CV_32S=4, CV_32F=5, CV_64F=6};
```

Here, it should be noted that these three declarations are equivalent: `CV_8U`, `CV_8UC1`, and `CV_8UC(1)`. The single-channel declaration fits well for integer arrays devoted to grayscale images, whereas the three channel declaration of an array is more appropriate for images with three components (for example, RGB, BRG, HSV, and so on). For linear algebra operations, the arrays of type `float` (F) might be used.

We can define all of the preceding datatypes for multichannel arrays (up to 512 channels). The following screenshots illustrate an image's internal representation with one single channel (`CV_8U`, `grayscale`) and the same image represented with three channels (`CV_8UC3`, `RGB`). These screenshots are taken by zooming in on an image displayed in the window of an OpenCV executable (the **showImage** example):



79	67	37	38	49
175 121 49	171 121 50	163 113 54	180 127 87	196 140 103
168 117 34	175 124 58	192 139 99	208 151 131	201 144 117
175 124 41	190 139 86	211 157 129	209 153 128	184 130 86
174 122 46	200 146 112	200 144 129	185 130 100	174 121 67

(x=155, y=141) ~ R:201 G:144 B:117

122	133	146	140	119
126	126	129	128	121
123	120	123	132	150
116	119	130	148	170
113	117	129	158	159

(x=153, y=140) ~ L:121

An 8-bit representation of an image in RGB color and grayscale

It is important to notice that to properly save a RGB image with OpenCV functions, the image must be stored in memory with its channels ordered as BGR. In the same way, when an RGB image is read from a file, it is stored in memory with its channels in a BGR order. Moreover, it needs a supplementary fourth channel (alpha) to manipulate images with three channels, RGB, plus a transparency. For RGB images, a larger integer value means a brighter pixel or more transparency for the alpha channel.

All OpenCV classes and functions are in the `cv` namespace, and consequently, we will have the following two options in our source code:

- Add the `using namespace cv` declaration after including the header files (this is the option used in all the code examples in this book).
- Append the `cv::` prefix to all the OpenCV classes, functions, and data structures that we use. This option is recommended if the external names provided by OpenCV conflict with the often-used **standard template library** (**STL**) or other libraries.

Image file-supported formats

OpenCV supports the most common image formats. However, some of them need (freely available) third-party libraries. The main formats supported by OpenCV are:

- Windows bitmaps (`*.bmp`, `*dib`)
- Portable image formats (`*.pbm`, `*.pgm`, `*.ppm`)
- Sun rasters (`*.sr`, `*.ras`)

The formats that need auxiliary libraries are:

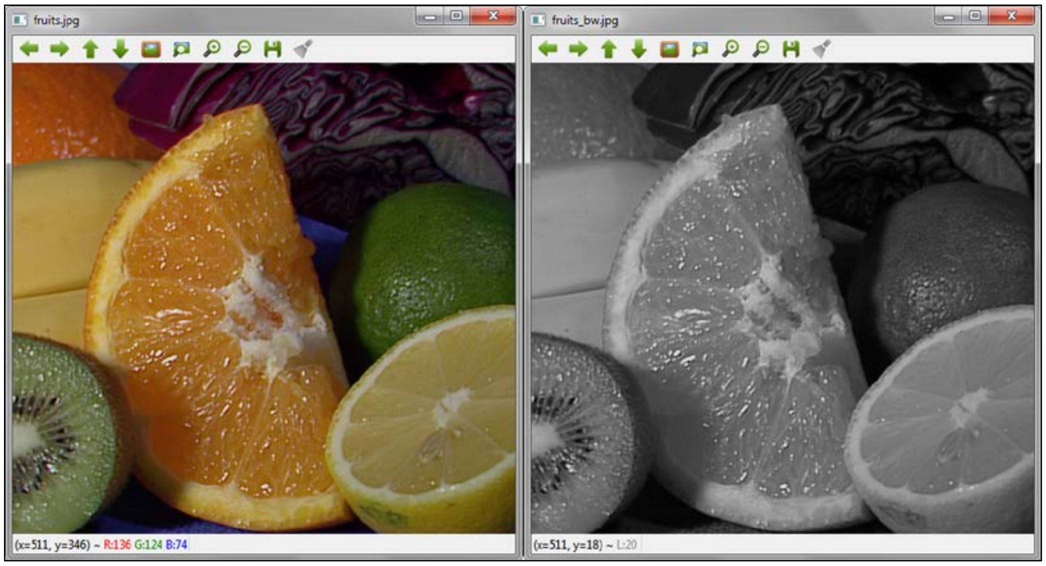
- **JPEG** (`*.jpeg`, `*.jpg`, `*.jpe`)
- **JPEG 2000** (`*.jp2`)
- **Portable Network Graphics** (`*.png`)
- **TIFF** (`*.tiff`, `*.tif`)
- **WebP** (`*.webp`).

In addition to the preceding listed formats, with the OpenCV 3.0 version, it includes a driver for the formats (**NITF**, **DTED**, **SRTM**, and others) supported by the **Geographic Data Abstraction Library (GDAL)** set with the CMake option, `WITH_GDAL`. Notice that the GDAL support has not been extensively tested on Windows OSes yet. In Windows and OS X, codecs shipped with OpenCV are used by default (`libjpeg`, `libjasper`, `libpng`, and `libtiff`). Then, in these OSes, it is possible to read the *JPEG*, *PNG*, and *TIFF* formats. Linux (and other Unix-like open source OSes) looks for codecs installed in the system. The codecs can be installed before OpenCV or else the libraries can be built from the OpenCV package by setting the proper options in CMake (for example, `BUILD_JASPER`, `BUILD_JPEG`, `BUILD_PNG`, and `BUILD_TIFF`).

The example code

To illustrate how to read and write image files with OpenCV, we will now describe the **showImage** example. The example is executed from the command line with the corresponding output windows as follows:

```
<bin_dir>\showImage.exe fruits.jpg fruits_bw.jpg
```



The output window for the showImage example

In this example, two filenames are given as arguments. The first one is the input image file to be read. The second one is the image file to be written with a grayscale copy of the input image. Next, we show you the source code and its explanation:

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;
using namespace cv;

int main(int, char *argv[])
{
    Mat in_image, out_image;

    // Usage: <cmd> <file_in> <file_out>
    // Read original image
```

```
        in_image = imread(argv[1], IMREAD_UNCHANGED);
        if (in_image.empty()) {
        // Check whether the image is read or not
        cout << "Error! Input image cannot be read...\n";
        return -1;
    }
    // Creates two windows with the names of the images
        namedWindow(argv[1], WINDOW_AUTOSIZE);
        namedWindow(argv[2], WINDOW_AUTOSIZE);
        // Shows the image into the previously created window
        imshow(argv[1], in_image);
        cvtColor(in_image, out_image, COLOR_BGR2GRAY);
        imshow(argv[2], in_image);
        cout << "Press any key to exit...\n";
        waitKey(); // Wait for key press
        // Writing image
        imwrite(argv[2], in_image);
        return 0;
    }
```

Here, we use the `#include` directive with the `opencv.hpp` header file that, in fact, includes all the OpenCV header files. By including this single file, no more files need to be included. After declaring the use of `cv` namespace, all the variables and functions inside this namespace don't need the `cv::` prefix. The first thing to do in the main function is to check the number of arguments passed in the command line. Then, a help message is displayed if an error occurs.

Reading image files

If the number of arguments is correct, the image file is read into the `Mat in_image` object with the `imread(argv[1], IMREAD_UNCHANGED)` function, where the first parameter is the first argument (`argv[1]`) passed in the command line and the second parameter is a flag (`IMREAD_UNCHANGED`), which means that the image stored into the memory object should be unchanged. The `imread` function determines the type of image (codec) from the file content rather than from the file extension.

The prototype for the `imread` function is as follows:

```
Mat imread(const String& filename,
int flags = IMREAD_COLOR )
```

The flag specifies the color of the image read and they are defined and explained by the following enumeration in the imgcodecs.hpp header file:

```
enum { IMREAD_UNCHANGED = -1, // 8bit, color or not
  IMREAD_GRAYSCALE = 0, // 8bit, gray
  IMREAD_COLOR = 1, // unchanged depth, color
  IMREAD_ANYDEPTH = 2, // any depth, unchanged color
  IMREAD_ANYCOLOR = 4, // unchanged depth, any color
  IMREAD_LOAD_GDAL = 8 // Use gdal driver
};
```

As of Version 3.0 of OpenCV, the imread function is in the imgcodecs module and not in highgui like in OpenCV 2.x.

As several functions and declarations are moved into OpenCV 3.0, it is possible to get some compilation errors as one or more declarations (symbols and/or functions) are not found by the linker. To figure out where (*.hpp) a symbol is defined and which library to link, we recommend the following trick using the Qt Creator IDE:

Add the #include <opencv2/opencv.hpp> declaration to the code. Press the *F2* function key with the mouse cursor over the symbol or function; this opens the *.hpp file where the symbol or function is declared.

After the input image file is read, check to see whether the operation succeeded. This check is achieved with the in_image.empty() member function. If the image file is read without errors, two windows are created to display the input and output images, respectively. The creation of windows is carried out with the following function:

```
void namedWindow(const String& winname,
  int flags = WINDOW_AUTOSIZE )
```

OpenCV windows are identified by a univocal name in the program. The flags' definition and their explanation are given by the following enumeration in the highgui.hpp header file:

```
enum { WINDOW_NORMAL = 0x00000000,
  // the user can resize the window (no constraint)
  // also use to switch a fullscreen window to a normal size
  WINDOW_AUTOSIZE = 0x00000001,
  // the user cannot resize the window,
  // the size is constrained by the image displayed
  WINDOW_OPENGL = 0x00001000, // window with opengl support
  WINDOW_FULLSCREEN = 1,
  WINDOW_FREERATIO = 0x00000100,
  // the image expends as much as it can (no ratio constraint)
  WINDOW_KEEPRATIO = 0x00000000
  // the ratio of the image is respected
};
```

The creation of a window does not show anything on screen. The function (belonging to the `highgui` module) to display an image in a window is:

```
    void imshow(const String& winname, InputArray mat)
```

The image (`mat`) is shown with its original size if the window (`winname`) was created with the `WINDOW_AUTOSIZE` flag.

In the **showImage** example, the second window shows a grayscale copy of the input image. To convert a color image to grayscale, the `cvtColor` function from the `imgproc` module is used. This function can actually be used to change the image color space.

Any window created in a program can be resized and moved from its default settings. When any window is no longer required, it should be destroyed in order to release its resources. This resource liberation is done implicitly at the end of a program, like in the example.

Event handling into the intrinsic loop

If we do nothing more after showing an image on a window, surprisingly, the image will not be shown at all. After showing an image on a window, we should start a loop to fetch and handle events related to user interaction with the window. Such a task is carried out by the following function (from the `highgui` module):

```
    int waitKey(int delay=0)
```

This function waits for a key pressed during a number of milliseconds (`delay` > `0`) returning the code of the key or `-1` if the delay ends without a key pressed. If `delay` is `0` or negative, the function waits forever until a key is pressed.

Remember that the `waitKey` function only works if there is a created and active window at least.

Writing image files

Another important function in the `imgcodecs` module is:

```
bool imwrite(const String& filename,
  InputArray img,
  const vector<int>& params=vector<int>())
```

This function saves the image (`img`) into a file (`filename`), being the third optional argument a vector of property-value pairs specifying the parameters of the codec (leave it empty to use the default values). The codec is determined by the extension of the file.

[For a detailed list of codec properties, take a look at the `imgcodecs.hpp` header file and the OpenCV API reference at `http://docs.opencv.org/master/modules/refman.html`.]

Reading and writing video files

Rather than still images, a video deals with moving images. The sources of video can be a dedicated camera, a webcam, a video file, or a sequence of image files. In OpenCV, the `VideoCapture` and `VideoWriter` classes provide an easy-to-use C++ API for the task of capturing and recording involved in video processing.

The example code

The **recVideo** example is a short snippet of code where you can see how to use a default camera as a capture device to grab frames, process them for edge detection, and save this new converted frame to a file. Also, two windows are created to simultaneously show you the original frame and the processed one. The example code is:

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;
using namespace cv;

int main(int, char **)
{
  Mat in_frame, out_frame;
  const char win1[]="Grabbing...", win2[]="Recording...";
  double fps=30; // Frames per second
```

```
    char file_out[]="recorded.avi";

    VideoCapture inVid(0); // Open default camera
    if (!inVid.isOpened()) { // Check error
      cout << "Error! Camera not ready...\n";
      return -1;
    }
    // Gets the width and height of the input video
    int width = (int)inVid.get(CAP_PROP_FRAME_WIDTH);
    int height = (int)inVid.get(CAP_PROP_FRAME_HEIGHT);
    VideoWriter recVid(file_out,
      VideoWriter::fourcc('M','S','V','C'),
      fps, Size(width, height));
    if (!recVid.isOpened()) {
      cout << "Error! Video file not opened...\n";
      return -1;
    }
    // Create two windows for orig. and final video
    namedWindow(win1);
    namedWindow(win2);
    while (true) {
      // Read frame from camera (grabbing and decoding)
      inVid >> in_frame;
      // Convert the frame to grayscale
      cvtColor(in_frame, out_frame, COLOR_BGR2GRAY);
      // Write frame to video file (encoding and saving)
      recVid << out_frame;
      imshow(win1, in_frame); // Show frame in window
      imshow(win2, out_frame); // Show frame in window
      if (waitKey(1000/fps) >= 0)
      break;
    }
    inVid.release(); // Close camera
    return 0;
  }
```

In this example, the following functions deserve a quick review:

- `double VideoCapture::get(int propId)`: This returns the value of the specified property for a `VideoCapture` object. A complete list of properties based on DC1394 (IEEE 1394 Digital Camera Specifications) is included with the `videoio.hpp` header file.

- `static int VideoWriter::fourcc(char c1, char c2, char c3, char c4)`: This concatenates four characters to a **fourcc** code. In the example, MSVC stands for Microsoft Video (only available for Windows). The list of valid fourcc codes is published at `http://www.fourcc.org/codecs.php`.
- `bool VideoWriter::isOpened()`: This returns `true` if the object for writing the video was successfully initialized. For instance, using an improper codec produces an error.

> Be cautious; the valid fourcc codes in a system depend on the locally installed codecs. To know the installed fourcc codecs available in the local system, we recommend the open source tool MediaInfo, available for many platforms at `http://mediaarea.net/en/MediaInfo`.

- `VideoCapture& VideoCapture::operator>>(Mat& image)`: This grabs, decodes, and returns the next frame. This method has the equivalent `bool VideoCapture::read(OutputArray image)` function. It can be used rather than using the `VideoCapture::grab()` function, followed by `VideoCapture::retrieve()`.
- `VideoWriter& VideoWriter::operator<<(const Mat& image)`: This writes the next frame. This method has the equivalent `void VideoWriter::write(const Mat& image)` function.

  In this example, there is a reading/writing loop where the window events are fetched and handled as well. The `waitKey(1000/fps)` function call is in charge of that; however, in this case, `1000/fps` indicates the number of milliseconds to wait before returning to the external loop. Although not exact, an approximate measure of frames per second is obtained for the recorded video.
- `void VideoCapture::release()`: This releases the video file or capturing device. Although not explicitly necessary in this example, we include it to illustrate its use.

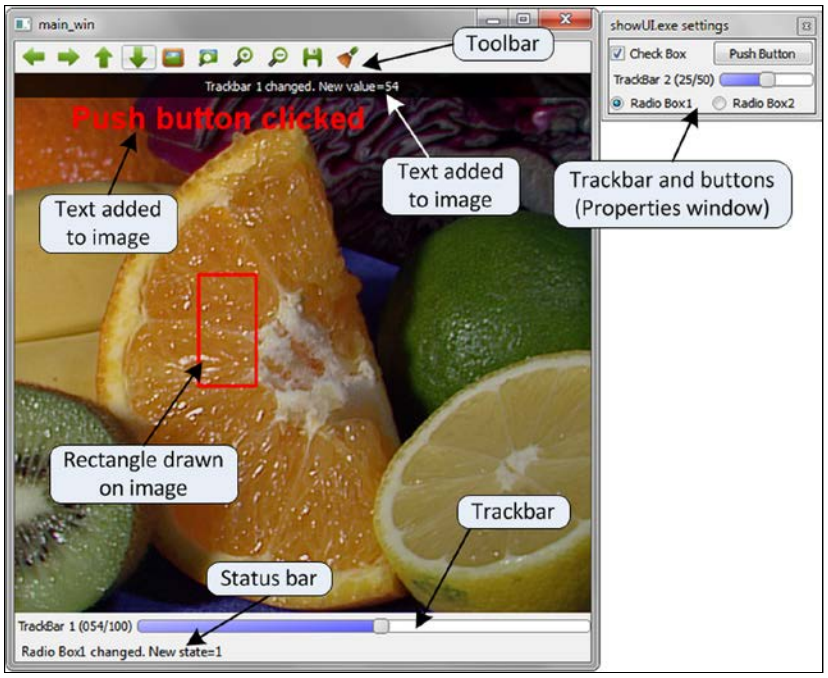
User-interactions tools

In the previous sections, we explained how to create (`namedWindow`) a window to display (`imshow`) an image and fetch/handle events (`waitKey`). The examples we provide show you a very easy method for user interaction with OpenCV applications through the keyboard. The `waitKey` function returns the code of a key pressed before a timeout expires.

Fortunately, OpenCV provides more flexible ways for user interaction, such as trackbars and mouse interaction, which can be combined with some drawing functions to provide a richer user experience. Moreover, if OpenCV is locally compiled with Qt support (the `WITH_QT` option of CMake), a set of new functions are available to program an even better UI.

In this section, we provide a quick review of the available functionality to program user interfaces in an OpenCV project with Qt support. We illustrate this review on OpenCV UI support with the next example named **showUI**.

The example shows you a color image in a window, illustrating how to use some basic elements to enrich the user interaction. The following screenshot displays the UI elements created in the example:



Trackbar 1 changed. New value=54

Push button clicked

TrackBar 1 (054/100)

Radio Box1 changed. New state=1

TrackBar 2 (25/50)

The output window for the showUI example

The source code of the **showUI** example (without the callback functions) is as follows:

```
#include <opencv2/opencv.hpp>
#include <iostream>
```

```
using namespace std;
using namespace cv;

// Callback functions declarations
void cbMouse(int event, int x, int y, int flags, void*);
void tb1_Callback(int value, void *);
void tb2_Callback(int value, void *);
void checkboxCallBack(int state, void *);
void radioboxCallBack(int state, void *id);
void pushbuttonCallBack(int, void *font);

// Global definitions and variables
Mat orig_img, tmp_img;
const char main_win[]="main_win";
char msg[50];

int main(int, char* argv[]) {
  const char track1[]="TrackBar 1";
  const char track2[]="TrackBar 2";
  const char checkbox[]="Check Box";
  const char radiobox1[]="Radio Box1";
  const char radiobox2[]="Radio Box2";
  const char pushbutton[]="Push Button";
  int tb1_value = 50; // Initial value of trackbar 1
  int tb2_value = 25; // Initial value of trackbar 1

  orig_img = imread(argv[1]); // Open and read the image
  if (orig_img.empty()) {
    cout << "Error!!! Image cannot be loaded..." << endl;
    return -1;
  }
  namedWindow(main_win); // Creates main window
  // Creates a font for adding text to the image
  QtFont font = fontQt("Arial", 20, Scalar(255,0,0,0),
    QT_FONT_BLACK, QT_STYLE_NORMAL);
  // Creation of CallBack functions
    setMouseCallback(main_win, cbMouse, NULL);
    createTrackbar(track1, main_win, &tb1_value,
      100, tb1_Callback);
    createButton(checkbox, checkboxCallBack, 0,
     QT_CHECKBOX);
    // Passing values (font) to the CallBack
    createButton(pushbutton, pushbuttonCallBack,
      (void *)&font, QT_PUSH_BUTTON);
    createTrackbar(track2, NULL, &tb2_value,
      50, tb2_Callback);
    // Passing values to the CallBack
    createButton(radiobox1, radioboxCallBack,
      (void *)radiobox1, QT_RADIOBOX);
    createButton(radiobox2, radioboxCallBack,
      (void *)radiobox2, QT_RADIOBOX);

  imshow(main_win, orig_img); // Shows original image
  cout << "Press any key to exit..." << endl;
  waitKey(); // Infinite loop with handle for events
  return 0;
}
```

When OpenCV is built with Qt support, every created window—through the `highgui` module—shows a default **toolbar** (see the preceding figure) with options (from left to right) for panning, zooming, saving, and opening the properties window.

Additional to the mentioned toolbar (only available with Qt), in the next subsections, we comment the different UI elements created in the example and the code to implement them.

Trackbars

Trackbars are created with the `createTrackbar(const String& trackbarname, const String& winname, int* value, int count, TrackbarCallback onChange=0, void* userdata=0)` function in the specified window (`winname`), with a linked integer value (`value`), a maximum value (`count`), an optional **callback** function (`onChange`) to be called on changes of the slider, and an argument (`userdata`) to the callback function. The callback function itself gets two arguments: `value` (selected by the slider) and a pointer to `userdata` (optional).With Qt support, if no window is specified, the **trackbar** is created in the properties window. In the **showUI** example, we create two trackbars: the first in the main window and the second one in the properties window. The code for the trackbar callbacks is:

```
  void tb1_Callback(int value, void *) {

    sprintf(msg, "Trackbar 1 changed. New value=%d", value);
    displayOverlay(main_win, msg);
    return;
  }
```

```
void tb2_Callback(int value, void *) {

  sprintf(msg, "Trackbar 2 changed. New value=%d", value);
  displayStatusBar(main_win, msg, 1000);
  return;
}
```

Mouse interaction

Mouse events are always generated so that the user interacts with the mouse (moving and clicking). By setting the proper handler or callback functions, it is possible to implement actions such as select, drag and drop, and so on. The callback function (`onMouse`) is enabled with the `setMouseCallback(const String& winname, MouseCallback onMouse, void* userdata=0 )` function in the specified window (`winname`) and optional argument (`userdata`).

The source code for the callback function that handles the mouse event is:

```
void cbMouse(int event, int x, int y, int flags, void*) {
  // Static vars hold values between calls
  static Point p1, p2;
  static bool p2set = false;

  // Left mouse button pressed
  if (event == EVENT_LBUTTONDOWN) {
    p1 = Point(x, y); // Set orig. point
    p2set = false;
  } else if (event == EVENT_MOUSEMOVE &&
  flags == EVENT_FLAG_LBUTTON) {
    // Check moving mouse and left button down
    // Check out bounds
    if (x > orig_img.size().width)
      x = orig_img.size().width;
    else if (x < 0)
      x = 0;
    // Check out bounds
    if (y > orig_img.size().height)
      y = orig_img.size().height;
    else if (y < 0)
      y = 0;
    p2 = Point(x, y); // Set final point
    p2set = true;
    // Copy orig. to temp. image
    orig_img.copyTo(tmp_img);
```

```
        // Draws rectangle
        rectangle(tmp_img, p1, p2, Scalar(0, 0 ,255));
        // Draw temporal image with rect.
        imshow(main_win, tmp_img);
      } else if (event == EVENT_LBUTTONUP
      && p2set) {
        // Check if left button is released
        // and selected an area
        // Set subarray on orig. image
        // with selected rectangle
        Mat submat = orig_img(Rect(p1, p2));
        // Here some processing for the submatrix
        //...
        // Mark the boundaries of selected rectangle
        rectangle(orig_img, p1, p2, Scalar(0, 0, 255), 2);
        imshow("main_win", orig_img);
      }
      return;
    }
```

In the **showUI** example, the mouse events are used to control through a callback function (`cbMouse`), the selection of a rectangular region by drawing a rectangle around it. In the example, this function is declared as `void cbMouse(int event, int x, int y, int flags, void*)`, the arguments being the position of the pointer (`x`, `y`) where the event occurs, the condition when the event occurs (`flags`), and optionally, `userdata`.

The available events, flags, and their corresponding definition symbols can be found in the `highgui.hpp` header file.

Buttons

OpenCV (only with Qt support) allows you to create three types of buttons: **checkbox** (`QT_CHECKBOX`), **radiobox** (`QT_RADIOBOX`), and **push button** (`QT_PUSH_BUTTON`). These types of button can be used respectively to set options, set exclusive options, and take actions on push. The three are created with the `createButton(const String& button_name, ButtonCallback on_change, void* userdata=0, int type=QT_PUSH_BUTTON, bool init_state=false )` function in the properties window arranged in a buttonbar after the last trackbar created in this window. The arguments for the button are its name (`button_name`), the callback function called on the status change (`on_change`), and optionally, an argument (`userdate`) to the callback, the type of button (`type`), and the initial state of the button (`init_state`).

Next, we show you the source code for the callback functions corresponding to buttons in the example:

```
void checkboxCallBack(int state, void *) {

  sprintf(msg, "Check box changed. New state=%d", state);
  displayStatusBar(main_win, msg);
  return;
}

void radioboxCallBack(int state, void *id) {

  // Id of the radio box passed to the callBack
  sprintf(msg, "%s changed. New state=%d",
    (char *)id, state);
  displayStatusBar(main_win, msg);
  return;
}

void pushbuttonCallBack(int, void *font) {

  // Add text to the image
  addText(orig_img, "Push button clicked",
    Point(50,50), *((QtFont *)font));
  imshow(main_win, orig_img); // Shows original image
  return;
}
```

The callback function for a button gets two arguments: its status and, optionally, a pointer to user data. In the **showUI** example, we show you how to pass an integer (`radioboxCallBack(int state, void *id)`) to identify the button and a more complex object (`pushbuttonCallBack(int, void *font)`).

Drawing and displaying text

A very efficient way to communicate the results of some image processing to the user is by drawing shapes or/and displaying text over the figure being processed. Through the `imgproc` module, OpenCV provides some convenient functions to achieve such tasks as putting text, drawing lines, circles, ellipses, rectangles, polygons, and so on. The **showUI** example illustrates how to select a rectangular region over an image and draw a rectangle to mark the selected area. The following function draws (`img`) a rectangle defined by two points (`p1`, `p2`) over an image with the specified color and other optional parameters as thickness (negative for a fill shape) and the type of lines:

```
void rectangle(InputOutputArray img, Point pt1, Point pt2,
  const Scalar& color, int thickness=1,
  int lineType=LINE_8, int shift=0 )
```

Additional to shapes' drawing support, the `imgproc` module provides a function to put text over an image with the function:

```
void putText(InputOutputArray img, const String& text,
Point org, int fontFace, double fontScale,
Scalar color, int thickness=1,
int lineType=LINE_8, bool bottomLeftOrigin=false )
```

[The available font faces for the text can be inspected in the `core.hpp` header file.]

Qt support, in the `highgui` module, adds some additional ways to show text on the main window of an OpenCV application:

- **Text over the image**: We get this result using the `addText(const Mat& img, const String& text, Point org, const QtFont& font)` function. This function allows you to select the origin point for the displayed text with a font previously created with the `fontQt(const String& nameFont, int pointSize=-1, Scalar color=Scalar::all(0), int weight=QT_FONT_NORMAL, int style=QT_STYLE_NORMAL, int spacing=0)` function. In the **showUI** example, this function is used to put text over the image when the push button is clicked on, calling the `addText` function inside the callback function.
- **Text on the status bar**: Using the `displayStatusBar(const String& winname, const String& text, int delayms=0 )` function, we display text in the status bar for a number of milliseconds given by the last argument (`delayms`). In the **showUI** example, this function is used (in the callback functions) to display an informative text when the buttons and trackbar of the properties window change their state.
- **Text overlaid on the image**: Using the `displayOverlay(const String& winname, const String& text, int delayms=0)` function, we display text overlaid on the image for a number of milliseconds given by the last argument. In the **showUI** example, this function is used (in the callback function) to display informative text when the main window trackbar changes its value.

Summary

In this chapter, you got a quick review of the main purpose of the OpenCV library and its modules. You learned the foundations of how to compile, install, and use the library in your local system to develop **C++** OpenCV applications with Qt support. To develop your own software, we explained how to start with the free Qt Creator IDE and the GNU compiler kit.

To start with, full code examples were provided in the chapter. These examples showed you how to read and write images and video. Finally, the chapter gave you an example of displaying some easy-to-implement user interface capabilities in OpenCV programs, such as trackbars, buttons, putting text on images, drawing shapes, and so on.

The next chapter will be devoted to establishing the main image processing tools and tasks that will set the basis for the remaining chapters.

2
Establishing Image Processing Tools

This chapter describes the main data structures and basic procedures that will be used in subsequent chapters:

- Image types
- Pixel access
- Basic operations with images
- Histograms

These are some of the most frequent operations that we will have to perform on images. Most of the functionality covered here is in the *core* module of the library.

Basic data types

The fundamental data type in OpenCV is `Mat`, as it is used to store images. Basically, an image is stored as a header plus a memory zone containing the pixel data. Images have a number of channels. Grayscale images have a single channel, while color images typically have three for the red, green, and blue components (although OpenCV stores them in a reverse order, that is blue, green, and red). A fourth transparency (alpha) channel can also be used. The number of channels for an `img` image can be retrieved with `img.channels()`.

Each pixel in an image is stored using a number of bits. This is called the image *depth*. For grayscale images, pixels are commonly stored in 8 bits, thus allowing 256 gray levels (integer values 0 to 255). For color images, each pixel is stored in three bytes, one per color channel. In some operations, it will be necessary to store pixels in a floating-point format. The image depth can be obtained with `img.depth()`, and the values returned are:

- CV_8U, 8-bit unsigned integers (0..255)
- CV_8S, 8-bit signed integers (-128..127)
- CV_16U, 16-bit unsigned integers (0..65,535)
- CV_16S, 16-bit signed integers (-32,768..32,767)
- CV_32S, 32-bit signed integers (-2,147,483,648..2,147,483,647)
- CV_32F, 32-bit floating-point numbers
- CV_64F, 64-bit floating-point numbers

Note that the most common image depth will be CV_8U for both grayscale and color images. It is possible to convert from one depth to another using the `convertTo` method:

```
Mat img = imread("lena.png", IMREAD_GRAYSCALE);
Mat fp;
img.convertTo(fp,CV_32F);
```

It is common to perform an operation on floating-point images (that is, pixel values are the result of a mathematical operation). If we use `imshow()` to display this image, we will not see anything meaningful. In this case, we have to convert pixels to the integer range 0..255. The `convertTo` function implements a linear transformation and has two additional parameters, alpha and beta, which represent a scale factor and a delta value to add, respectively. This means that each pixel *p* is converted with:

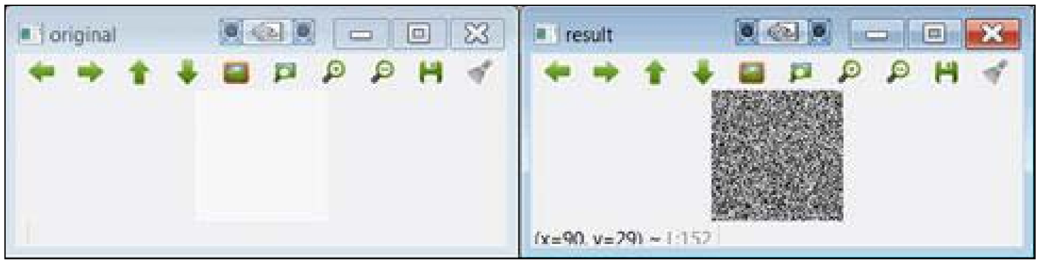
*newp = alpha*p + beta*

This can be used to display floating point images properly. Assuming that the `img` image has `m` and `M` minimum and maximum values (refer to the following code to see how to obtain these values), we would use this:

```
Mat m1 = Mat(100, 100, CV_32FC1);
randu(m1, 0, 1e6); // random values between 0 and 1e6
imshow("original", m1);
double minRange,MaxRange;
Point mLoc,MLoc;
minMaxLoc(m1,&minRange,&MaxRange,&mLoc,&MLoc);
Mat img1;
```

```
m1.convertTo(img1,CV_8U,255.0/(MaxRange-minRange),-255.0/minRange);
imshow("result", img1);
```

This code maps the range of the result image values to the range 0-255. The following image shows you the result of running the code:



The result of convertTo (note that the image on the left-hand side is displayed as white)

The image size can be obtained with the rows and cols attributes. There is also a `size` attribute that retrieves both:

```
MatSize s = img.size;
int r=l[0];
int c=l[1];
```

Apart from the image itself, other data types are very common; refer to the following table:

Type	**Type keyword**	**Example**
(Small) vector	`VecAB`, where `A` can be 2,3,4,5 or 6, `B` can be `b`,`s`,`i`,`f`, or `d`	`Vec3b rgb;` `rgb[0]=255;`
(Up to 4) scalars	`Scalar`	`Scalar a;` `a[0]=0;` `a[1]=0;`
Point	`PointAB`, where `A` can be `2` or `3` and `B` can be `i`, `f`, or `d`	`Point3d p;` `p.x=0;` `p.y=0;` `p.z=0;`
Size	`Size`	`Size s;` `s.width=30;` `s.height=40;`
Rectangle	`Rect`	`Rect r;` `r.x=r.y=0;` `r.width=r.height=100;`

Some of these types have additional operations. For example, we can check whether a point lies inside a rectangle:

```
p.inside(r)
```

The `p` and `r` arguments are (two-dimensional) point and rectangle, respectively. Note that in any case, the preceding table is not exhaustive; OpenCV provides many more support structures with associated methods.

Pixel-level access

To process images, we have to know how to access each pixel independently. OpenCV provides a number of ways to do this. In this section, we cover two methods; the first one is easy for the programmer, while the second one is more efficient.

The first method uses the `at<>` template function. In order to use it, we have to specify the type of matrix cells, such as in this short example:

```
Mat src1 = imread("lena.jpg", IMREAD_GRAYSCALE);
uchar pixel1=src1.at<uchar>(0,0);
cout << "Value of pixel (0,0): " << (unsigned int)pixel1 << endl;
Mat src2 = imread("lena.jpg", IMREAD_COLOR);
Vec3b pixel2 = src2.at<Vec3b>(0,0);
cout << "B component of pixel (0,0):" << (unsigned int)pixel2[0] <<
endl;
```

The example reads an image in both grayscale and color and accesses the first pixel at (0,0). In the first case, the pixel type is `unsigned char` (that is, `uchar`). In the second case, when the image is read in full color, we have to use the `Vec3b` type, which refers to a triplet of unsigned chars. Of course, the `at<>` function can also appear on the left-hand side of an assignment, that is, to change the value of a pixel.

The following is another short example in which a floating-point matrix is initialized to the Pi value using this method:

```
Mat M(200, 200, CV_64F);
for(int i = 0; i < M.rows; i++)
  for(int j = 0; j < M.cols; j++)
  M.at<double>(i,j)=CV_PI;
```

Note that the `at<>` method is not very efficient as it has to calculate the exact memory position from the pixel row and column. This can be very time consuming when we process the whole image pixel by pixel. The second method uses the `ptr` function, which returns a pointer to a specific image row. The following snippet obtains the pixel value of each pixel in a color image:

```
uchar R, G, B;
for (int i = 0; i < src2.rows; i++)
{
Vec3b* pixrow = src2.ptr<Vec3b>(i);
for (int j = 0; j < src2.cols; j++)
{
  B = pixrow[j][0];
  G = pixrow[j][1];
  R = pixrow[j][2];
}
}
```

In the example above, `ptr` is used to get a pointer to the first pixel in each row. Using this pointer, we can now access each column in the innermost loop.

Measuring the time

Processing images takes time (comparably much more than the time it takes to process 1D data). Often, processing time is the crucial factor that decides whether a solution is practical or not. OpenCV provides two functions to measure the elapsed time: `getTickCount()` and `getTickFrequency()`. You'll use them like this:

```
double t0 = (double)getTickCount();
// your stuff here ...
elapsed = ((double)getTickCount() – t0)/getTickFrequency();
```

Here, `elapsed` is in seconds.

Common operations with images

The following table summarizes the most typical operations with images:

Operation	**Code examples**

Set matrix values	`img.setTo(0); // for 1-channel img`
	`img.setTo(Scalar(B,G,R); // 3-channel img`

MATLAB-style matrix initialization	`Mat m1 = Mat::eye(100, 100, CV_64F);`
	`Mat m3 = Mat::zeros(100, 100, CV_8UC1);`
	`Mat m2 = Mat::ones(100, 100, CV_8UC1)*255;`

Random initialization	`Mat m1 = Mat(100, 100, CV_8UC1);`
	`randu(m1, 0, 255);`

Create a copy of the matrix	`Mat img1 = img.clone();`

Operation	Code examples
Create a copy of the matrix (with the mask)	`img.copy(img1, mask);`
Reference a submatrix (the data is not copied)	`Mat img1 = img (Range(r1,r2),Range(c1,c2));`

Image crop	`Rect roi(r1,c2, width, height);`
	`Mat img1 = img(roi).clone(); // data copied`

Resize image	`resize(img, imag1, Size(), 0.5, 0.5); // decimate by a factor of 2`

Flip image	`flip(imgsrc, imgdst, code);`
	`// code=0 => vertical flipping`
	`// code>0 => horizontal flipping`
	`// code<0 => vertical & horizontal flipping`

Split channels	`Mat channel[3];`
	`split(img, channel);`
	`imshow("B", channel[0]); // show blue`

Merge channels	`merge(channel,img);`
Count nonzero pixels	`int nz = countNonZero(img);`

Minimum and maximum	`double m,M;`
	`Point mLoc,MLoc;  minMaxLoc(img,&m,&M,&mLoc,&MLoc);`

The mean pixel value	`Scalar m, stdd;`
	`meanStdDev(img, m, stdd);`
	`uint mean_pxl = mean.val[0];`

Check whether the image data is null	`If (img.empty())`
	`   cout << "couldn't load image";`

Arithmetic operations

Arithmetic operators are overloaded. This means that we can operate on `Mat` images like we can in this example:

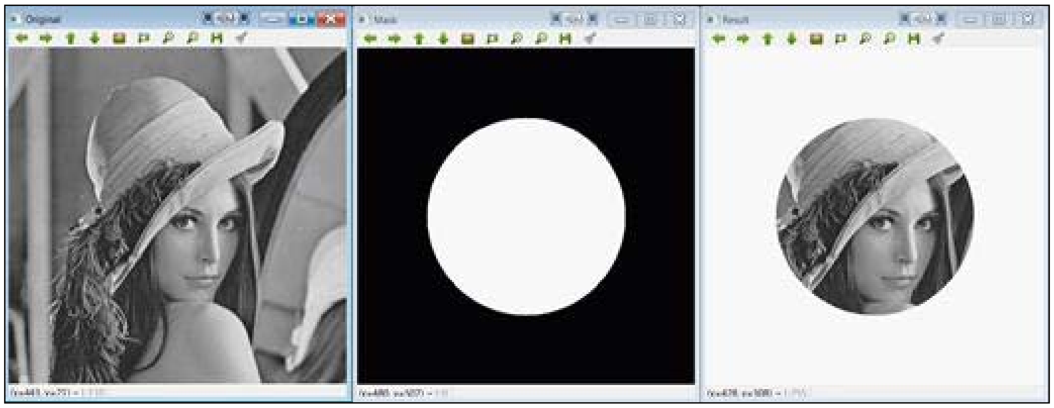
```
imgblend = 0.2*img1 + 0.8*img2;
```

In OpenCV, the result value of an operation is subject to the so-called saturation arithmetic. This means that the final value is actually the nearest integer in the 0..255 range.

Bitwise operations `bitwise_and()`, `bitwise_or()`, `bitwise_xor()`, and `bitwise_not()` can be very useful when working with masks. Masks are binary images that indicate the pixels in which an operation is to be performed (instead of the whole image). The following **bitwise_and** example shows you how to use the AND operation to crop part of an image:

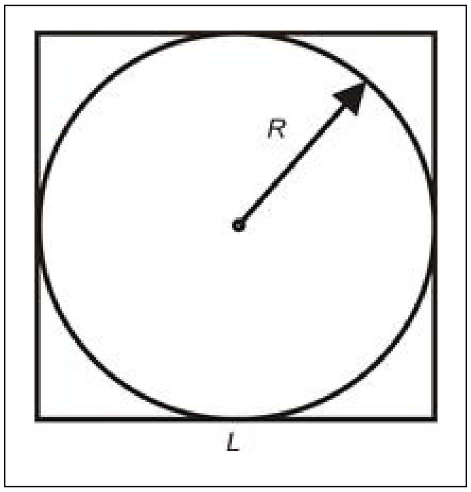
```
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main()
{
  Mat img1 = imread("lena.png", IMREAD_GRAYSCALE);
  if (img1.empty())
  {
  cout << "Cannot load image!" << endl;
  return -1;
  }

  imshow("Original", img1);  // Original

  // Create mask image
  Mat mask(img1.rows, img1.cols, CV_8UC1, Scalar(0,0,0));
  circle(mask, Point(img1.rows/2,img1.cols/2), 150, 255, -1);
  imshow("Mask",mask);

  // perform AND
  Mat r;
  bitwise_and(img1,mask,r);

  // fill outside with white
  const uchar white = 255;
  for(int i = 0; i < r.rows; i++)
  for(int j = 0; j < r.cols; j++)
    if (!mask.at<uchar>(i,j))
    r.at<uchar>(i,j)=white;

  imshow("Result",r);

  waitKey(0);
  return 0;
}
```

After reading and displaying the input image, we create a mask by drawing a filled white circle. In the AND operation, this mask is used. The logical operation is only applied in those pixels in which the mask value is not zero; other pixels are not affected. Finally, in this example, we fill the outer part of the result image (that is, outside the circle) with white. This is done using one of the pixel access methods explained previously. See the resulting images in the following screenshot:



The result of the bitwise_and example

Next, another cool example is shown in which we estimate the value of Pi. Let's consider a square and its enclosed circle:



Their areas are given by:

$$A_{circle} = Pi \cdot R^2$$

$$A_{square} = L \cdot L = 4 \cdot R^2$$

From this, we have:

$$Pi = 4 \cdot \frac{A_{circle}}{A_{square}}$$

Let's assume that we have a square image of unknown side length and an enclosed circle. We can estimate the area of the enclosed circle by painting many pixels in random positions within the image and counting those that fall inside the enclosed circle. On the other hand, the area of the square is estimated as the total number of pixels painted. This would allow you to estimate the value of Pi using the previous equation.

The following algorithm simulates this:

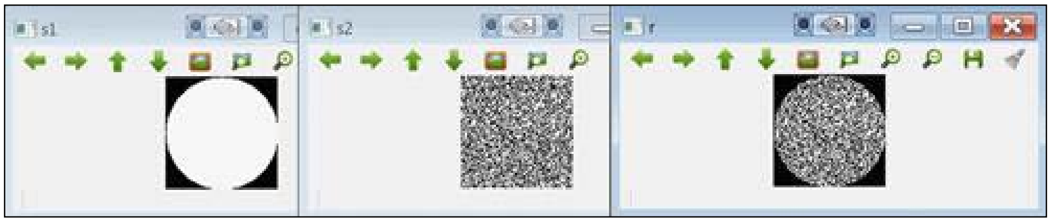
1. On a black square image, paint a solid white enclosed circle.
2. On another black square image (same dimensions), paint a large number of pixels at random positions.
3. Perform an AND operation between the two images and count nonzero pixels in the resulting image.
4. Estimate Pi using the equation.

The following is the code for the **estimatePi** example:

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

using namespace cv;
using namespace std;
```

```
int main()
{
  const int side=100;
  const int npixels=8000;

  int i,j;
  Mat s1=Mat::zeros(side, side, CV_8UC1);
  Mat s2=s1.clone();
  circle(s1, Point(side/2, side/2), side/2, 255, -1);

  imshow("s1",s1);

  for (int k=0;k<npixels;k++)
  {
  i = rand()%side;
  j = rand()%side;
  s2.at<uchar>(i,j)=255;
  }

  Mat r;
  bitwise_and(s1,s2,r);

  imshow("s2", s2);
  imshow("r", r);

  int Acircle = countNonZero(r);
  int Asquare = countNonZero(s2);
  float Pi=4*(float)Acircle/Asquare;
  cout << "Estimated value of Pi: " << Pi << endl;

  waitKey();
  return 0;
}
```

The program follows the preceding algorithm exactly. Note that we use the `countNonZero` function to count nonzero (white, in this case) pixels. For `npixels=8000`, the estimate was 3.125. The larger the value of `npixels`, the better the estimation.



The output of the estimatePi example

Data persistence

Apart from the specific functions to read and write images and video, in OpenCV, there is a more generic way to save/load the data. This is referred to as data persistence: the value of objects and variables in the program can be recorded (serialized) on the disk. This can be very useful to save results and load the configuration data. The main class is the aptly named `FileStorage`, which represents a file on a disk. Data is actually stored in XML or YAML formats.

These are the steps involved when writing data:

1. Call the `FileStorage` constructor, passing a filename and a flag with the `FileStorage::WRITE` value. The data format is defined by the file extension (that is, `.xml`, `.yml`, or `.yaml`).
2. Use the `<<` operator to write data to the file. Data is typically written as string-value pairs.
3. Close the file using the `release` method.

Reading data requires that you follow these steps:

1. Call the `FileStorage` constructor, passing a filename and a flag with the `FileStorage::READ` value.
2. Use the `[]` or `>>`operator to read data from the file.
3. Close the file using the `release` method.

The following example uses data persistence to save and load trackbar values.

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

using namespace cv;
using namespace std;
```

```
Mat img1;

void tb1_Callback(int value, void *)
{
  Mat temp = img1 + value;
  imshow("main_win", temp);
}

int main()
{
  img1 = imread("lena.png", IMREAD_GRAYSCALE);
  if (img1.empty())
  {
  cout << "Cannot load image!" << endl;
  return -1;
  }

  int tb1_value = 0;

  // load trackbar value
  FileStorage fs1("config.xml", FileStorage::READ);
  tb1_value=fs1["tb1_value"];   // method 1
  fs1["tb1_value"] >> tb1_value; // method 2
  fs1.release();

  // create trackbar
  namedWindow("main_win");
  createTrackbar("brightness", "main_win", &tb1_value,
       255, tb1_Callback);
  tb1_Callback(tb1_value, NULL);

  waitKey();

  // save trackbar value upon exiting
  FileStorage fs2("config.xml", FileStorage::WRITE);
  fs2 << "tb1_value" << tb1_value;
  fs2.release();

  return 0;
}
```

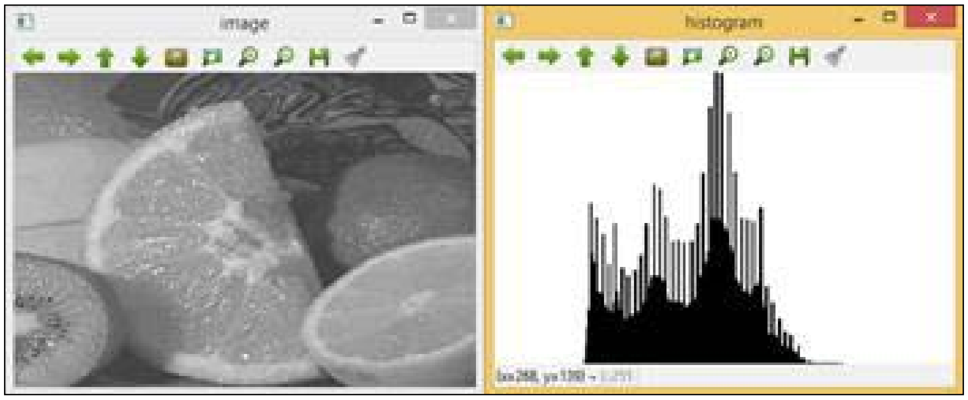
When OpenCV has been compiled with Qt support, window properties can be saved, including trackbar values, with the `saveWindowParameters()` function.

Once the trackbar is used to control an integer value, it is simply added to the original image, making it brighter. This value is read when the program starts (the value will be 0 the first time) and saved when the program exits normally. Note that two equivalent methods are shown to read the value of the `tb1_value` variable. The contents of the `config.xml` file are:

```
<?xml version="1.0"?>
<opencv_storage>
<tb1_value>112</tb1_value>
</opencv_storage>
```

Histograms

Once the image has been defined with a data type and we are able to access its gray level values, that is, the pixels, we may want to obtain a probability density function of the different gray levels, which is called the histogram. The image histogram represents the frequency of occurrence of the various gray levels in the image. The histogram can be modeled so that the image may change its contrast levels. This is known as **histogram equalization**. Histogram modeling is a powerful technique for image enhancement by means of contrast variation. The equalization allows for image areas of lower contrast to gain a higher contrast. The following image shows you an example of an equalized image and its histogram:



An example of an equalized image histogram

In OpenCV, the image histogram can be calculated with the `void calcHist` function and histogram equalization is performed with the `void equalizeHist` function.

The image histogram calculation is defined with ten parameters: `void calcHist(const Mat* images, int nimages, const int* channels, InputArray mask, OutputArray hist, int dims, const int* histSize, const float** ranges, bool uniform=true,` and `bool accumulate=false).`

- `const Mat* images`: The first parameter is the address of the first image from a collection. This can be used to process a batch of images.
- `int nimages`: The second parameter is the number of source images.
- `const int* channels`: The third input parameter is the list of the channels used to compute the histogram. The number of channels goes from 0 to 2.
- `InputArray mask`: This is an optional mask to indicate the image pixels counted in the histogram.
- `OutputArray hist`: The fifth parameter is the output histogram.
- `int dims`: This parameter allows you to indicate the dimension of the histogram.
- `const int* histSize`: This parameter is the array of histogram sizes in each dimension.
- `const float** ranges`: This parameter is the array of the dims arrays of the histogram bin boundaries in each dimension.
- `bool uniform=true`: By default, the Boolean value is `true`. It indicates that the histogram is uniform.
- `bool accumulate=false`: By default, the Boolean value is false. It indicates that the histogram is nonaccumulative.

The histogram equalization requires only two parameters, `void equalizeHist(InputArray src, OutputArray dst)`. The first parameter is the input image and the second one is the output image with the histogram equalized.

It is possible to calculate the histogram of more than one input image. This allows you to compare image histograms and calculate the joint histogram of several images. The comparison of two image histograms, `histImage1` and `histImage2`, can be performed with the `void compareHist(InputArray histImage1, InputArray histImage2, method)` function. The `Method` metric is the used to compute the matching between both histograms. There are four metrics implemented in OpenCV, that is, correlation (`CV_COMP_CORREL`), chi-square (`CV_COMP_CHISQR`), intersection or minimum distance (`CV_COMP_INTERSECT`), and Bhattacharyya distance (`CV_COMP_BHATTACHARYYA`).

It is possible to calculate the histogram of more than one channel of the same color image. This is possible thanks to the third parameter.

The following sections show you two example codes for color histogram calculation (**ColourImageEqualizeHist**) and comparison **ColourImageComparison**. In ColourImageEqualizeHist, it is also shown how to calculate the histogram equalization as well as the 2D histogram for two channels, that is, hue (H) and saturation (S), in the **ColourImageComparison** example.

The example code

The following **ColourImageEqualizeHist** example shows you how to equalize a color image and display the histogram of each channel at the same time. The histogram calculation of each color channel in the RGB image is done with the `histogramcalculation(InputArray Imagesrc, OutputArray histoImage)` function. To this end, the color image is split into the channels: R, G, and B. The histogram equalization is also applied to each channel that is then merged to form the equalized color image:

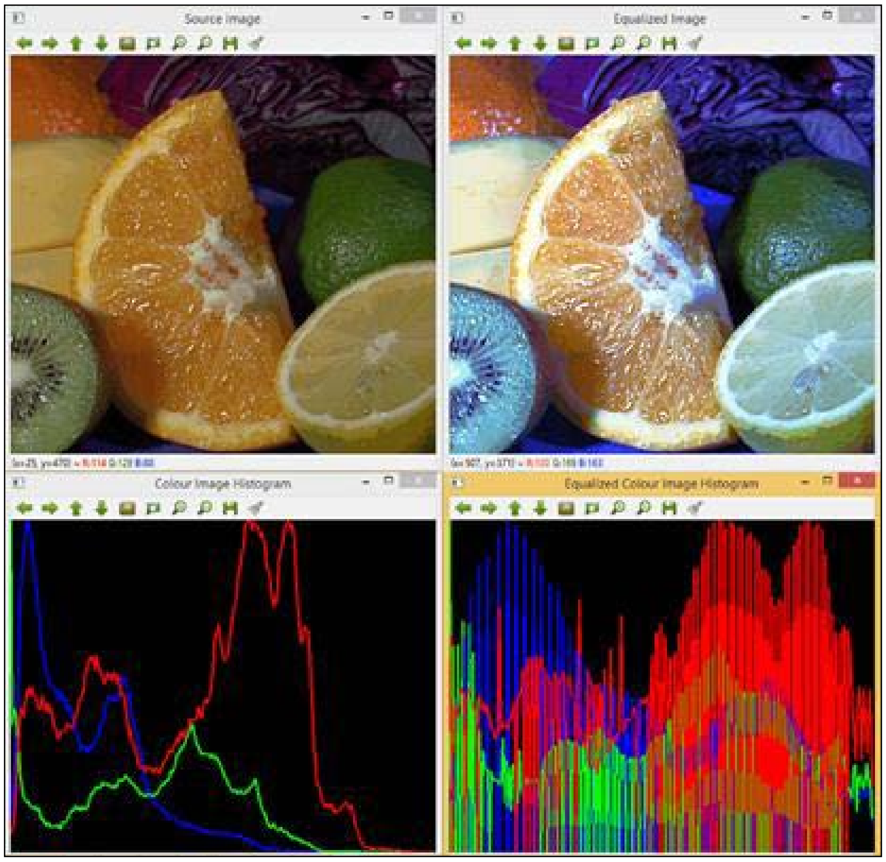
```
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
#include <stdio.h>

using namespace cv;
using namespace std;

void histogramcalculation(const Mat &Image, Mat &histoImage)
{
  int histSize = 255;

  // Set the ranges ( for B,G,R) )
  float range[] = { 0, 256 } ;
  const float* histRange = { range };

  bool uniform = true; bool accumulate = false;

  Mat b_hist, g_hist, r_hist;

  vector<Mat> bgr_planes;
  split(Image, bgr_planes );
  // Compute the histograms:
  calcHist( &bgr_planes[0], 1, 0, Mat(), b_hist, 1, &histSize, 
&histRange, uniform, accumulate );
  calcHist( &bgr_planes[1], 1, 0, Mat(), g_hist, 1, &histSize, 
&histRange, uniform, accumulate );
  calcHist( &bgr_planes[2], 1, 0, Mat(), r_hist, 1, &histSize, 
&histRange, uniform, accumulate );

  // Draw the histograms for B, G and R
  int hist_w = 512; int hist_h = 400;
  int bin_w = cvRound( (double) hist_w/histSize );

  Mat histImage( hist_h, hist_w, CV_8UC3, Scalar( 0,0,0) );

  // Normalize the result to [ 0, histImage.rows ]
  normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() 
);
  normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() 
);
  normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat() 
);

  // Draw for each channel
  for( int i = 1; i < histSize; i++ ){
  line( histImage, Point( bin_w*(i-1), hist_h - cvRound(b_hist.
at<float>(i-1)) ) , Point( bin_w*(i), hist_h - cvRound(b_hist.
at<float>(i)) ), Scalar( 255, 0, 0), 2, 8, 0  );
  line( histImage, Point( bin_w*(i-1), hist_h - cvRound(g_hist.
at<float>(i-1)) ) , Point( bin_w*(i), hist_h - cvRound(g_hist.
at<float>(i)) ), Scalar( 0, 255, 0), 2, 8, 0  );
  line( histImage, Point( bin_w*(i-1), hist_h - cvRound(r_hist.
at<float>(i-1)) ) , Point( bin_w*(i), hist_h - cvRound(r_hist.
at<float>(i)) ), Scalar( 0, 0, 255), 2, 8, 0  );
  }

  histoImage= histImage;
}


int main( int,  char *argv[] )
{
  Mat src, imageq;
  Mat histImage;
```

```
    // Read original image
    src = imread( "fruits.jpg");
    if(! src.data )
     {  printf("Error imagen\n"); exit(1); }

    // Separate the image in 3 places ( B, G and R )
    vector<Mat> bgr_planes;
    split( src, bgr_planes );

    // Display results
    imshow( "Source image", src );

    // Calculate the histogram to each channel of the source image
    histogramcalculation(src, histImage);

    // Display the histogram for each colour channel
    imshow("Colour Image Histogram", histImage );

    // Equalized Image

    // Apply Histogram Equalization to each channel
    equalizeHist(bgr_planes[0], bgr_planes[0]);
    equalizeHist(bgr_planes[1], bgr_planes[1]);
    equalizeHist(bgr_planes[2], bgr_planes[2]);

    // Merge the equalized image channels into the equalized image
    merge(bgr_planes, imageq );

    // Display Equalized Image
    imshow( "Equalized Image ", imageq );

    // Calculate the histogram to each channel of the equalized image
    histogramcalculation(imageq, histImage);

    // Display the Histogram of the Equalized Image
    imshow("Equalized Colour Image Histogram", histImage );


    // Wait until user exits the program
    waitKey();
    return 0;
  }
```

The example creates four windows with:

- **The source image**: This is shown in the following figure in the upper-left corner.
- **The equalized color image**: This is shown in the following figure in the upper-right corner.
- **The histogram of three channels**: Here, R= Read, G=Green and B= Blue, for the source image. This is shown in the following figure in the lower-left corner.
- **The histogram of RGB channel for the equalized image**: This is shown in next figure in the lower-right corner. The figure shows you how the most frequent intensity values for R, G, and B have been stretched out due to the equalization process.

The following figure shows you the results of the algorithm:



The example code

The following **ColourImageComparison** example shows you how to calculate a 2D histogram composed of two channels from the same color image. The example code also performs a comparison between the original image and the equalized image by means of histogram matching. The metrics used for the matching are the four metrics that have been mentioned previously, that is, Correlation, Chi-Square, Minimum distance, and Bhattacharyya distance. The 2D histogram calculation of the H and S color channel is done with the `histogram2Dcalculation(InputArray Imagesrc, OutputArray histo2D)` function. To perform the histogram comparison, the normalized 1D histogram has been calculated for the RGB image. In order to compare the histogram, they have been normalized. This is done in `histogramRGcalculation(InputArray Imagesrc, OutputArray histo)`:

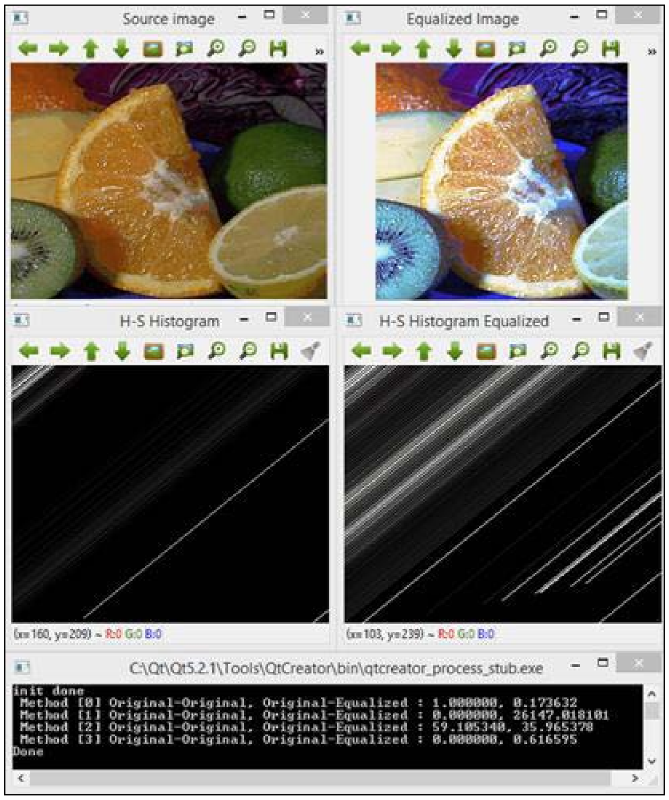
```
void histogram2Dcalculation(const Mat &src, Mat &histo2D)
{
  Mat hsv;

  cvtColor(src, hsv, CV_BGR2HSV);

  // Quantize the hue to 30 -255 levels
  // and the saturation to 32 - 255 levels
  int hbins = 255, sbins = 255;
  int histSize[] = {hbins, sbins};
  // hue varies from 0 to 179, see cvtColor
  float hranges[] = { 0, 180 };
  // saturation varies from 0 (black-gray-white) to
  // 255 (pure spectrum color)
  float sranges[] = { 0, 256 };
  const float* ranges[] = { hranges, sranges };
  MatND hist, hist2;
  // we compute the histogram from the 0-th and 1-st channels
  int channels[] = {0, 1};

   calcHist( &hsv, 1, channels, Mat(), hist, 1, histSize, ranges,
true, false );
    double maxVal=0;
    minMaxLoc(hist, 0, &maxVal, 0, 0);

    int scale = 1;
    Mat histImg = Mat::zeros(sbins*scale, hbins*scale, CV_8UC3);

    for( int h = 0; h < hbins; h++ )
      for( int s = 0; s < sbins; s++ )
      {
        float binVal = hist.at<float>(h, s);
```

```
            int intensity = cvRound(binVal*255/maxVal);
            rectangle( histImg, Point(h*scale, s*scale),
                  Point( (h+1)*scale - 1, (s+1)*scale - 1),
                  Scalar::all(intensity),
                  CV_FILLED );
        }
  histo2D=histImg;
}

void histogramRGcalculation(const Mat &src, Mat &histoRG)
{
    // Using 50 bins for red and 60 for green
    int r_bins = 50; int g_bins = 60;
    int histSize[] = { r_bins, g_bins };

    // red varies from 0 to 255, green from 0 to 255
    float r_ranges[] = { 0, 255 };
    float g_ranges[] = { 0, 255 };

    const float* ranges[] = { r_ranges, g_ranges };

    // Use the o-th and 1-st channels
    int channels[] = { 0, 1 };


    // Histograms
    MatND hist_base;

    // Calculate the histograms for the HSV images
    calcHist( &src, 1, channels, Mat(), hist_base, 2, histSize,
ranges, true, false );
    normalize( hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat() );

    histoRG=hist_base;

}

int main( int argc, char *argv[])
{
  Mat src, imageq;
  Mat histImg, histImgeq;
  Mat histHSorg, histHSeq;

  // Read original image
  src = imread( "fruits.jpg");
    if(! src.data )
```

```
        {  printf("Error imagen\n"); exit(1); }

     // Separate the image in 3 places ( B, G and R )
     vector<Mat> bgr_planes;
     split( src, bgr_planes );

     // Display results
     namedWindow("Source image", 0 );
     imshow( "Source image", src );

     // Calculate the histogram of the source image
     histogram2Dcalculation(src, histImg);

     // Display the histogram for each colour channel
     imshow("H-S Histogram", histImg );

     // Equalized Image

     // Apply Histogram Equalization to each channel
     equalizeHist(bgr_planes[0], bgr_planes[0] );
     equalizeHist(bgr_planes[1], bgr_planes[1] );
     equalizeHist(bgr_planes[2], bgr_planes[2] );

     // Merge the equalized image channels into the equalized image
     merge(bgr_planes, imageq );

     // Display Equalized Image
     namedWindow("Equalized Image", 0 );
     imshow("Equalized Image", imageq );

     // Calculate the 2D histogram for H and S channels
     histogram2Dcalculation(imageq, histImgeq);

     // Display the 2D Histogram
     imshow( "H-S Histogram Equalized", histImgeq );

     histogramRGcalculation(src, histHSorg);
     histogramRGcalculation(imageq, histHSeq);

     /// Apply the histogram comparison methods
      for( int i = 0; i < 4; i++ )
      {
        int compare_method = i;
        double orig_orig = compareHist( histHSorg, histHSorg, compare_
   method );
```

```
        double orig_equ = compareHist( histHSorg, histHSeq, compare_
method );

        printf( " Method [%d] Original-Original, Original-Equalized : %f,
%f \n", i, orig_orig, orig_equ );
    }

    printf( "Done \n" );

    waitKey();
}
```

The example creates four windows with the source image, the equalized color image and the 2D histogram for H and S channels for both images the original, and the equalized image. The algorithm also displays the four numerical matching parameters obtained from the comparison of the original RGB image histogram with itself and with the equalized RGB image. For the correlation and intersection methods, the higher the metric, the more accurate the match. For the chi-square and Bhattacharyya distance, the less the result, the better the match. The following figure shows you the output of the **ColourImageComparison** algorithm:


Source image
Equalized Image
H-S Histogram
H-S Histogram Equalized
(x=160, y=209) ~ R:0 G:0 B:0
(x=103, y=239) ~ R:0 G:0 B:0
C:\Qt\Qt5.2.1\Tools\QtCreator\bin\qtcreator_process_stub.exe

```
init done
 Method [0] Original-Original, Original-Equalized : 1.000000, 0.173632
 Method [1] Original-Original, Original-Equalized : 0.000000, 26147.018101
 Method [2] Original-Original, Original-Equalized : 59.105340, 35.965378
 Method [3] Original-Original, Original-Equalized : 0.000000, 0.616595
Done
```

Finally, you can refer to *Chapter 3*, *Correcting and Enhancing Images*, as well as the examples within to cover essential aspects of this broad topic, such as image enhancement by means of histogram modeling.

[For more information, refer to *OpenCV Essentials*, Deniz O., Fernández M.M., Vállez N., Bueno G., Serrano I., Patón .A., Salido J. by Packt Publishing, `https://www.packtpub.com/application-development/opencv-essentials`.]

Summary

This chapter covered and established the basis of applying image processing methods used in computer vision. Image processing is often the first step to further computer vision applications, and therefore, it has been covered here: basic data types, pixel level access, common operations with images, arithmetic operations, data persistence, and histograms.

You can also refer to *Chapter 3*, *Correcting and Enhancing Images*, of *OpenCV Essentials* by Packt Publishing to cover further essential aspects of this broad topic, such as image enhancement, image restoration by means of filtering, and geometrical correction.

The next chapter will cover further aspects of image processing to correct and enhance images by means of smoothing, sharpening, image resolution analysis, morphological and geometrical transforms, inpainting, and denoising.

3
Correcting and Enhancing Images

This chapter presents methods for image enhancement and correction. Sometimes, it is necessary to reduce the noise in an image or emphasize or suppress certain details in it. These procedures are usually carried out by modifying pixel values, performing some operations on them, or on their local neighborhood as well. By definition, image-enhancement operations are used to improve important image details. Enhancement operations include noise reduction, smoothing, and edge enhancement. On the other hand, image correction attempts to restore a damaged image. In OpenCV, the **imgproc** module contains functions for image processing.

In this chapter, we will cover:

- Image filtering. This includes image smoothing, image sharpening, and working with image pyramids.
- Applying morphological operations, such as dilation, erosion, opening, or closing.
- Geometrical transformations (affine and perspective transformations).
- Inpainting, which is used to reconstruct damaged parts of images.
- Denoising, which is necessary to reduce the image noise produced by the image-capture device.

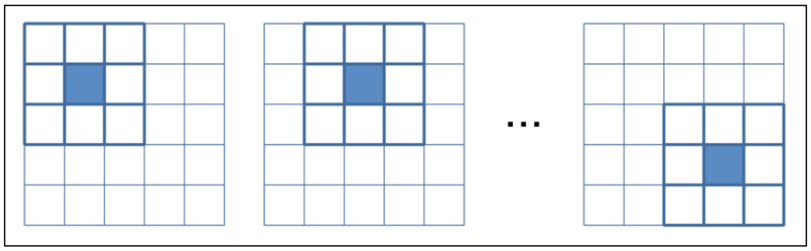
Image filtering

Image filtering is a process to modify or enhance images. Emphasizing certain features or removing others in an image are examples of image filtering. Filtering is a neighborhood operation. The neighborhood is a set of pixels around a selected one. Image filtering determines the output value of a certain pixel located at a position (x,y) by performing some operations with the values of the pixels in its neighborhood.

OpenCV provides several filtering functions for common image-processing operations, such as smoothing or sharpening.

Smoothing

Smoothing, also called blurring, is an image-processing operation that is frequently used to reduce noise, among other purposes. A smoothing operation is performed by applying linear filters to the image. Then, the pixel values of the output at positions (x_i,y_j) are computed as a weighted sum of the input pixel values at positions (x_i,y_j) and their neighborhoods. The weights for the pixels in the linear operation are usually stored in a matrix called **kernel**. Therefore, a filter could be represented as a sliding window of coefficients.



The representation of the pixel neighborhood

Let *K* be the kernel and *I* and *O* the input and output images, respectively. Then, each output pixel value at (i,j) is calculated as follows:

$$O(i,j) = \sum_{m,n} I(i+m, j+n) \cdot K(m,n)$$

Median, Gaussian, and bilateral are the most used OpenCV smoothing filters. Median filtering is very good to get rid of *salt-and-pepper* or *speckle* noise, while Gaussian is a much better preprocessing step for edge detection. On the other hand, bilateral filtering is a good technique to smooth an image while respecting strong edges.

The functions included in OpenCV for this purpose are:

- `void boxFilter(InputArray src, OutputArray dst, int ddepth, Size ksize, Point anchor = Point(-1,-1), bool normalize = true, int borderType = BORDER_DEFAULT)`: This is a box filter whose kernel coefficients are equal. With `normalize=true`, each output pixel value is the mean of its kernel neighbors with all coefficients equal to 1/n, where n = the number of elements. With `normalize=false`, all coefficients are equal to 1. The `src` argument is the input image, while the filtered image is stored in `dst`. The `ddepth` parameter indicates the output image depth that is -1 to use the same depth as the input image. The kernel size is indicated in `ksize`. The `anchor` point indicates the position of the so-called anchor pixel. The (-1, -1) default value means that the anchor is at the center of the kernel. Finally, the border-type treatment is indicated in the `borderType` parameter.
- `void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY = 0, int borderType=BORDER_DEFAULT)`: This is done by convolving each point in the `src` input array with a Gaussian kernel to produce the `dst` output. The `sigmaX` and `sigmaY` parameters indicate the Gaussian kernel standard deviation in X and Y directions. If `sigmaY` is zero, it is set to be equal to `sigmaX`, and if both are equal to zero, they are computed using the width and height given in `ksize`.

Convolution is defined as the integral of the product of two functions in which one of them is previously reversed and shifted.

- `void medianBlur(InputArray src, OutputArray dst, int ksize)`: This runs through each element of the image and replaces each pixel with the median of its neighboring pixels.
- `void bilateralFilter(InputArray src, OutputArray dst, int d, double sigmaColor, double sigmaSpace, int borderType=BORDER_DEFAULT)`: This is analogous to the Gaussian filter considering the neighboring pixels with weights assigned to each of them but having two components on each weight, which is the same used by the Gaussian filter, and another one that takes into account the difference in intensity between the neighboring and evaluated pixels. This function needs the diameter of the pixel neighborhood as parameter `d`, and `sigmaColor` `sigmaSpace` values. A larger value of the `sigmaColor` parameter means that farther colors within the pixel neighborhood will be mixed together, generating larger areas of semi-equal colors, whereas a larger value of the `sigmaSpace` parameter means that farther pixels will influence each other as long as their colors are close enough.

- `void blur(InputArray src, OutputArray dst, Size ksize, Point anchor=Point(-1,-1), int borderType=BORDER_DEFAULT)`: This blurs an image using the normalized box filter. It is equivalent to using `boxFilter` with `normalize = true`. The kernel used in this function is:

$$\frac{1}{ksize.width \cdot ksize.height}\begin{bmatrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ \vdots & \vdots & \ddots & \vdots\\ 1 & 1 & \cdots & 1\end{bmatrix}$$

[The `getGaussianKernel` and `getGaborKernel` functions can be used in OpenCV to generate custom kernels, which can then be passed on to `filter2D`.]

In all cases, it is necessary to extrapolate the values of the non-existent pixels outside the image boundary. OpenCV enables the specification of the extrapolation method in most of the filter functions. These methods are:

- `BORDER_REPLICATE`: This repeats the last known pixel value: aaaaaa | abcdefgh | hhhhhhh
- `BORDER_REFLECT`: This reflects the image border: fedcba | abcdefgh | hgfedcb
- `BORDER_REFLECT_101`: This reflects the image border without duplicating the last pixel of the border: gfedcb | abcdefgh | gfedcba
- `BORDER_WRAP`: This appends the value of the opposite border: cdefgh | abcdefgh | abcdefg
- `BORDER_CONSTANT`: This establishes a constant over the new border: kkkkkk | abcdefgh | kkkkkk

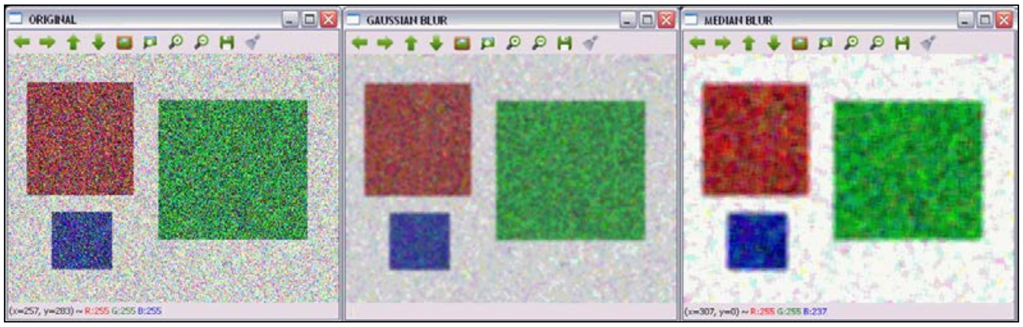
The example code

The following **Smooth** example shows you how to load an image and apply Gaussian and median blurring to it through `GaussianBlur` and `medianBlur` functions:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
```

```
        Mat src;
        src = imread(argv[1]);

        // Apply the filters
        Mat dst, dst2;
        GaussianBlur( src, dst, Size( 9, 9 ), 0, 0);
        medianBlur( src, dst2, 9);

        // Show the results
        namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
        imshow( " ORIGINAL ", src );
        namedWindow( " GAUSSIAN BLUR ", WINDOW_AUTOSIZE );
        imshow( " GAUSSIAN BLUR ", dst );
        namedWindow( " MEDIAN BLUR ", WINDOW_AUTOSIZE );
        imshow( " MEDIAN BLUR ", dst2 );

        waitKey();
        return 0;
    }
```

The following figure shows you the output of the code:


ORIGINAL
GAUSSIAN BLUR
MEDIAN BLUR
(x=257, y=283) ~ R:255 G:255 B:255
(x=307, y=0) ~ R:255 G:255 B:237

Original and blurred images from Gaussian and Median blurring transformations

Sharpening

Sharpening filters are used to highlight borders and other fine details within images. They are based on first- and second-order derivatives. The first derivative of an image computes an approximation of the image intensity gradient, whereas the second derivative is defined as the divergence of this gradient. Since digital image processing deals with discrete quantities (pixel values), the discrete versions of the first and second derivatives are used for sharpening.

First-order derivatives produce thicker image edges and are widely used for edge-extraction purposes. However, second-order derivatives are used for image enhancement due to their better response to fine details. Two popular operators used to obtain derivatives are the Sobel and the Laplacian.

The Sobel operator computes the first image derivative of an image, I, through:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

The Sobel gradient magnitude can be obtained by combining the gradient approximations in the two directions, as follows:

$$G = \sqrt{G_x^2 + G_y^2}$$

On the other hand, the discrete Laplacian of an image can be given as a convolution with the following kernel:

$$D_{xy}^2 = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 1 & 6 & 1 \\ 0.5 & 1 & 0.5 \end{bmatrix}$$

The functions included in OpenCV for this purpose are:

- `void Sobel(InputArray src, OutputArray dst, int ddepth, int dx, int dy, int ksize = 3, double scale = 1, double delta = 0, int borderType = BORDER_DEFAULT)`: This calculates the first, second, third, or mixed-image derivatives with the Sobel operator from an image in `src`. The `ddepth` parameter indicates the output image depth, that is, -1 to use the same depth as the input image. The kernel size is indicated in `ksize` and the desired derivative orders in `dx` and `dy`. A scale factor for the computed derivative vales can be established with `scale`. Finally, the border-type treatment is indicated in the `borderType` parameter and a `delta` value can be added to the results before storing them in `dst`.

- `void Scharr(InputArray src, OutputArray dst, int ddepth, int dx, int dy, double scale = 1, double delta = 0, int borderType = BORDER_DEFAULT )`: This calculates a more accurate derivative for a kernel of size 3 x 3. `Scharr(src, dst, ddepth, dx, dy, scale, delta, borderType)` is equivalent to `Sobel(src, dst, ddepth, dx, dy, CV_SCHARR, scale, delta, borderType)`.
- `void Laplacian(InputArray src, OutputArray dst, int ddepth, int ksize = 1, double scale = 1, double delta = 0, int borderType = BORDER_DEFAULT)`: This calculates the Laplacian of an image. All the parameters are equivalent to the ones from the `Sobel` and `Scharr` functions except for `ksize`. When `ksize`>1, it calculates the Laplacian of the image in `src` by adding up the second x and y derivatives calculated using `Sobel`. When `ksize`=1, the Laplacian is calculated by filtering the image with a 3 x 3 kernel that contains -4 for the center, 0 for the corners, and 1 for the rest of the coefficients.

[`getDerivKernels` can be used in OpenCV to generate custom derivative kernels, which can then be passed on to `sepFilter2D`.]

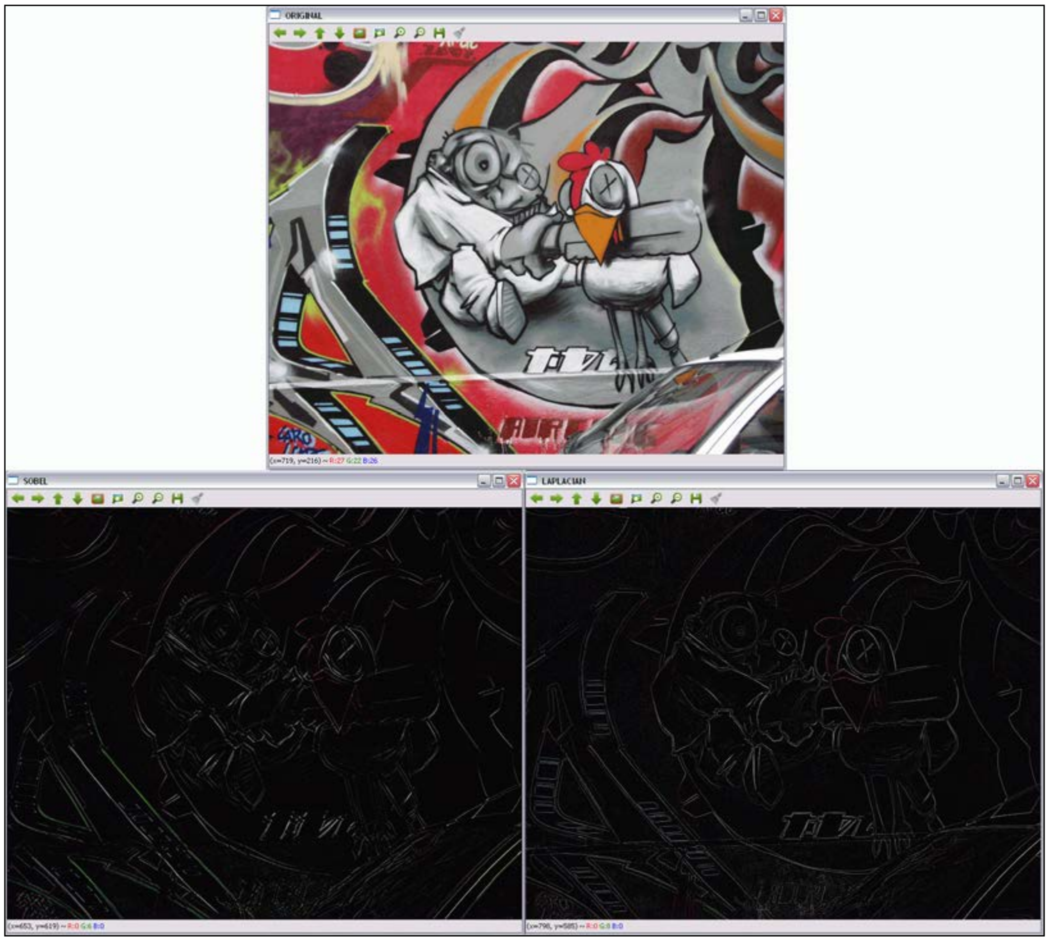
The example code

The following **Sharpen** example shows you how to compute Sobel and Laplacian derivatives from an image through `Sobel` and `Laplacian` functions. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply Sobel and Laplacian
    Mat dst, dst2;
    Sobel(src, dst, -1, 1, 1 );
    Laplacian(src, dst2, -1 );

    // Show the results
```

```
        namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
        imshow( " ORIGINAL ", src );
        namedWindow( " SOBEL ", WINDOW_AUTOSIZE );
        imshow( " SOBEL ", dst );
        namedWindow( " LAPLACIAN ", WINDOW_AUTOSIZE );
        imshow( " LAPLACIAN ", dst2 );

        waitKey();
        return 0;
    }
```

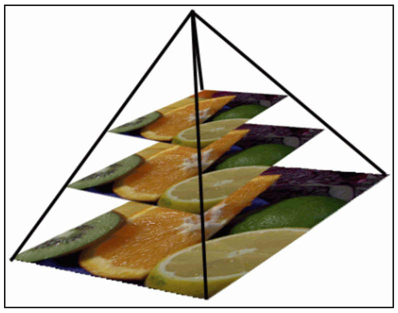
The following figure shows you the output of the code:



Contours obtained by Sobel and Laplacian derivatives

Working with image pyramids

On some occasions, working with a fixed image size is not possible, and we will need the original image at different resolutions. For example, in object-detection problems, examining the whole image trying to find the object takes too much time. In this case, searching for objects by starting at smaller resolutions is more efficient. This type of image set is called **pyramid** or **mipmap** due to the similarity with the pyramid structure type if images are organized from the largest to the smallest from the bottom to the top.


The Gaussian pyramid

There are two kinds of image pyramids: Gaussian pyramids and Laplacian pyramids.

Gaussian pyramids

Gaussian pyramids are created by alternately removing rows and columns in the lower level and then obtaining the value of the higher-level pixel by applying a Gaussian filter using the neighborhood from the underlying level. After each pyramid step, the image reduces its width and height by half and its area is a quarter of the previous level image area. In OpenCV, Gaussian pyramids can be computed using the `pyrDown`, `pyrUp`, and `buildPyramid` functions:

- `void pyrDown(InputArray src, OutputArray dst, const Size& dstsize = Size(), int borderType = BORDER_DEFAULT)`: This subsamples and blurs an `src` image, saving the result in `dst`. The size of the output image is computed as `Size((src.cols+1)/2, (src.rows+1)/2)` when it is not set with the `dstsize` parameter.
- `void pyrUp(InputArray src, OutputArray dst, const Size& dstsize = Size(), int borderType = BORDER_DEFAULT)`: This computes the opposite process of `pyrDown`.

- `void buildPyramid(InputArray src, OutputArrayOfArrays dst, int maxlevel, int borderType = BORDER_DEFAULT)`: This builds a Gaussian pyramid for an image stored in `src`, obtaining `maxlevel` new images and storing them in the `dst` array following the original image that is stored in `dst[0]`. Thus, `dst` stores `maxlevel` +1 images as a result.

Pyramids are also used for segmentation. OpenCV provides a function to compute mean-shift pyramids based on the first step of the mean-shift segmentation algorithm:

- `void pyrMeanShiftFiltering(InputArray src, OutputArray dst, double sp, double sr, int maxLevel = 1, TermCriteria termcrit = TermCriteria (TermCriteria::MAX_ITER + TermCriteria::EPS, 5, 1))`: This implements the filtering stage of the mean-shift segmentation, obtaining an image, `dst`, with color gradients and fine-grain texture flattened. The `sp` and `sr` parameters indicate the spatial window and the color window radii.

More information about the mean-shift segmentation can be found at `http://docs.opencv.org/trunk/doc/py_tutorials/py_video/py_meanshift/py_meanshift.html?highlight=meanshift`.

Laplacian pyramids

Laplacian pyramids do not have a specific function implementation in OpenCV, but they are formed from the Gaussian pyramids. Laplacian pyramids can be seen as border images where most of its elements are zeros. The i^{th} level in the Laplacian pyramid is the difference between the i^{th} level in the Gaussian pyramid and the expanded version of the $i^{th}+1$ level in the Gaussian pyramid.

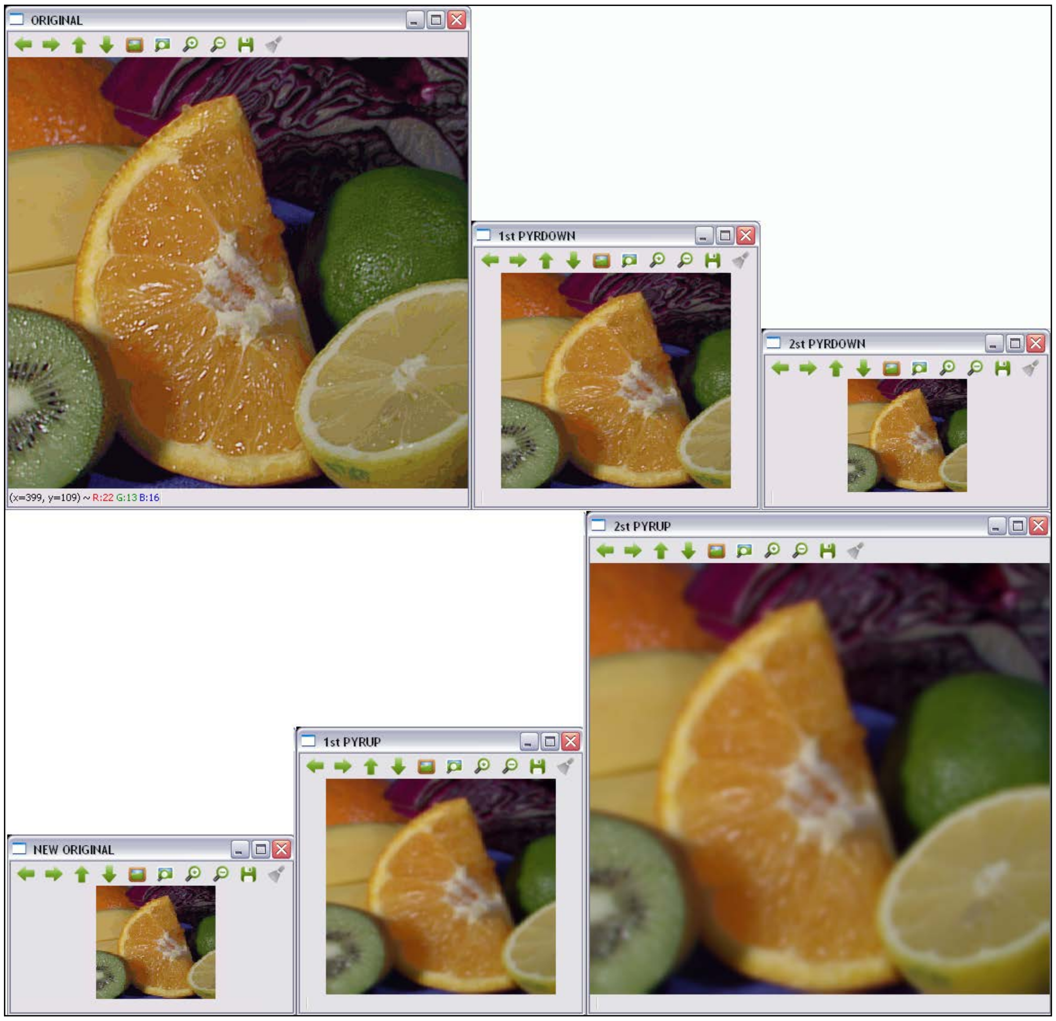
The example code

The following **Pyramids** example shows you how to obtain two levels from a Gaussian pyramid through the `pyrDown` function and the opposite operation through `pyrUp`. Notice that the original image cannot be obtained after using `pyrUp`:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
```

```
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply two times pyrDown
    Mat dst, dst2;
    pyrDown(src, dst);
    pyrDown(dst, dst2);

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " 1st PYRDOWN ", WINDOW_AUTOSIZE );
    imshow( " 1st PYRDOWN ", dst );
    namedWindow( " 2st PYRDOWN ", WINDOW_AUTOSIZE );
    imshow( " 2st PYRDOWN ", dst2 );

    // Apply two times pyrUp
    pyrUp(dst2, dst);
    pyrUp(dst, src);

    // Show the results
    namedWindow( " NEW ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " NEW ORIGINAL ", dst2 );
    namedWindow( " 1st PYRUP ", WINDOW_AUTOSIZE );
    imshow( " 1st PYRUP ", dst );
    namedWindow( " 2st PYRUP ", WINDOW_AUTOSIZE );
    imshow( " 2st PYRUP ", src );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:

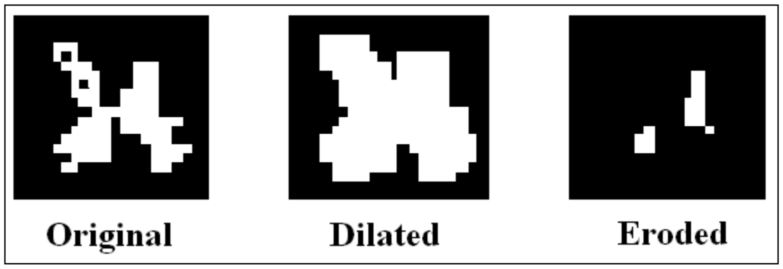


The original and two levels of the Gaussian pyramid

Morphological operations

Morphological operations process images according to shapes. They apply a defined "structuring element" to an image, obtaining a new image where the pixels at positions (x_i,y_j) are computed by comparing the input pixel values at positions (x_i,y_j) and their neighborhoods. Depending on the structuring element selected, a morphological operation is more sensitive to one specific shape or the other.

The two basic morphological operations are dilation and erosion. Dilation adds pixels from the background to the boundaries of the objects in an image, while erosion removes pixels. Here is where the structuring element is taken into account to select the pixels that are to be added or deleted. In dilation, the value of the output pixel is the maximum of all the pixels in the neighborhood. Using erosion, the value of the output pixel is the minimum value of all the pixels in the neighborhood.



An example of dilation and erosion

Other image-processing operations can be defined by combining dilation and erosion, such as the opening and closing operations, and the morphological gradient. The opening operation is defined as erosion, followed by dilation, while closing is its reverse operation—dilation followed by erosion. Therefore, opening removes small objects from an image while preserving the larger ones and closing is used to remove small holes while preserving the larger ones in a manner similar to opening. The morphological gradient is defined as the difference between the dilation and the erosion of an image. Furthermore, two more operations are defined using opening and closing: top-hat and black-hat operations. They are defined as the difference between the source image and its opening in the case of top hat and the difference between the closing of an image and the source image in the case of black hat. All the operations are applied with the same structuring element.

In OpenCV, it is possible to apply dilation, erosion, opening, and closing through the following functions:

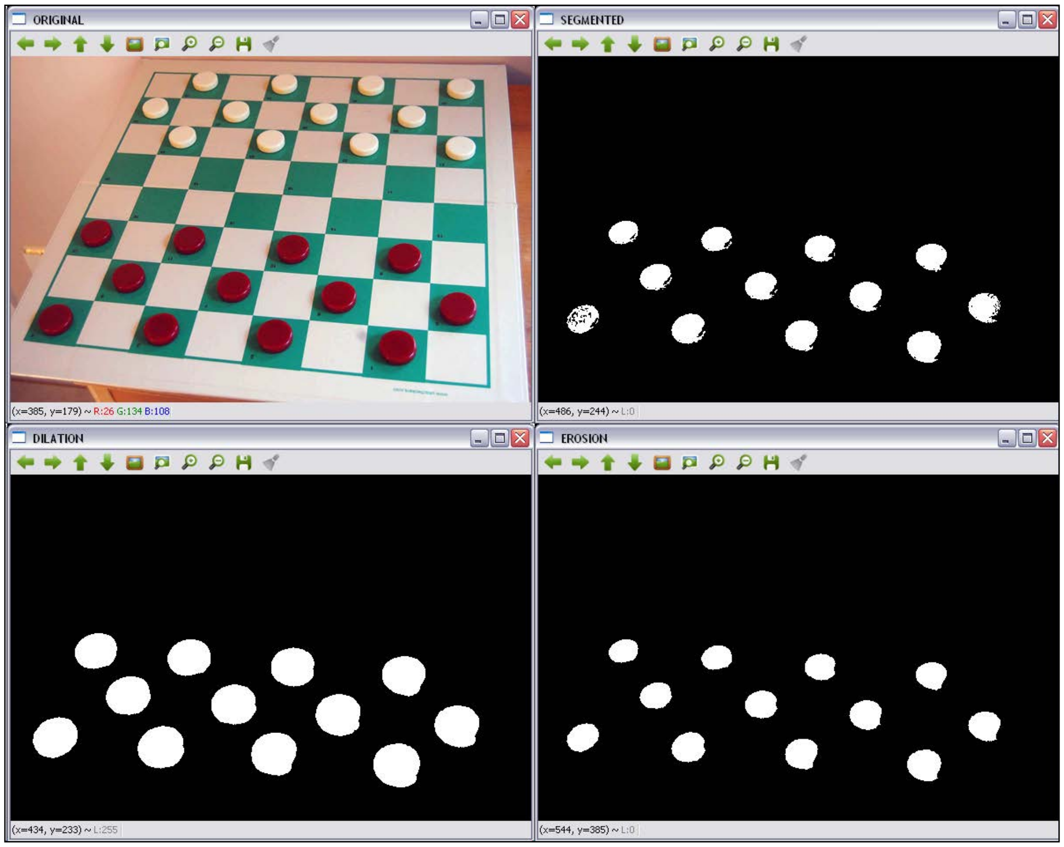
- `void dilate(InputArray src, OutputArray dst, InputArray kernel, Point anchor = Point(-1,-1), int iterations = 1, int borderType = BORDER_CONSTANT, const Scalar& borderValue = morphologyDefaultBorderValue())`: This dilates an image stored in `src` using a specific structuring element, saving the result in `dst`. The `kernel` parameter is the structuring element used. The `anchor` point indicates the position of anchor pixel. The (-1, -1) value means that the anchor is at the center. The operation can be applied several times using `iterations`. The border-type treatment is indicated in the `borderType` parameter and is the same as in other filters from previous sections. Finally, a constant is indicated in `borderValue` if the `BORDER_CONSTANT` border type is used.
- `void erode(InputArray src, OutputArray dst, InputArray kernel, Point anchor = Point(-1,-1), int iterations = 1, int borderType = BORDER_CONSTANT, const Scalar& borderValue = morphologyDefaultBorderValue())`: This erodes an image using a specific structuring element. Its parameters are the same as that in `dilate`.
- `void morphologyEx(InputArray src, OutputArray dst, int op, InputArray kernel, Point anchor = Point(-1,-1), int iterations = 1, int borderType = BORDER_CONSTANT, const Scalar& borderValue = morphologyDefaultBorderValue())`: This performs advanced morphological operations defined using the `op` parameter. Possible `op` values are `MORPH_OPEN`, `MORPH_CLOSE`, `MORPH_GRADIENT`, `MORPH_TOPHAT`, and `MORPH_BLACKHAT`.
- `Mat getStructuringElement(int shape, Size ksize, Point anchor = Point(-1,-1))`: This returns a structuring element of the specified size and shape for morphological operations. Supported types are `MORPH_RECT`, `MORPH_ELLIPSE`, and `MORPH_CROSS`.

The example code

The following **Morphological** example shows you how to segment red checkers in a checkerboard, applying a binary threshold (the `inRange` function) and then refining the results with dilation and erosion operations (through `dilate` and `erode` functions). The structure used is a circle of 15 x 15 pixels. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply the filters
    Mat dst, dst2, dst3;
    inRange(src, Scalar(0, 0, 100), Scalar(40, 30, 255), dst);

    Mat element = getStructuringElement(MORPH_ELLIPSE,Size(15,15));
    dilate(dst, dst2, element);
    erode(dst2, dst3, element);

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " SEGMENTED ", WINDOW_AUTOSIZE );
    imshow( " SEGMENTED ", dst );
    namedWindow( " DILATION ", WINDOW_AUTOSIZE );
    imshow( " DILATION ", dst2 );
    namedWindow( " EROSION ", WINDOW_AUTOSIZE );
    imshow( " EROSION ", dst3 );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:



Original, red color segmentation, dilation, and erosion

LUTs

Look-up tables (**LUTs**) are very common in custom filters in which two pixels with the same value in the input involves the same value in the output too. An LUT transformation assigns a new pixel value to each pixel in the input image according to the values given by a table. In this table, the index represents the input intensity value and the content of the cell given by the index represents the corresponding output value. As the transformation is actually computed for each possible intensity value, this results in a reduction in the time needed to apply the transformation over an image (images typically have more pixels than the number of intensity values).

The `LUT(InputArray src, InputArray lut, OutputArray dst, int interpolation = 0)` OpenCV function applies a look-up table transformation over an 8-bit signed or an `src` unsigned image. Thus, the table given in the `lut` parameter contains 256 elements. The number of channels in `lut` is either 1 or `src.channels`. If `src` has more than one channel but `lut` has a single one, the same `lut` channel is applied to all the image channels.

The example code

The following **LUT** example shows you how to divide (by two) the intensity of the pixels from an image using a look-up table. The LUT needs to be initialized before using it with this code:

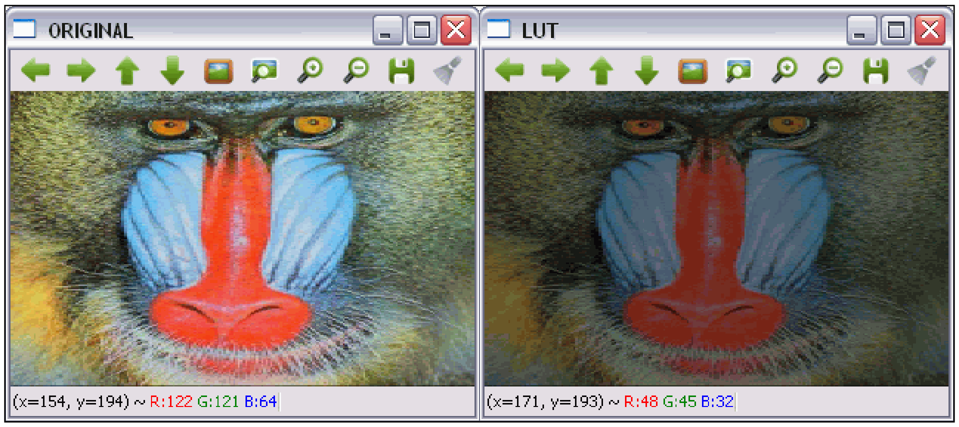
```
uchar * M = (uchar*)malloc(256*sizeof(uchar));
for(int i=0; i<256; i++){
    M[i] = i*0.5; //The result is rounded to an integer value
}
Mat lut(1, 256, CV_8UC1, M);
```

A `Mat` object is created where each cell contains the new value. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Create the LUT
    uchar * M = (uchar*)malloc(256*sizeof(uchar));
    for(int i=0; i<256; i++){
        M[i] = i*0.5;
    }
    Mat lut(1, 256, CV_8UC1, M);

    // Apply the LUT
    Mat dst;
```

```
        LUT(src,lut,dst);

        // Show the results
        namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
        imshow( " ORIGINAL ", src );
        namedWindow( " LUT ", WINDOW_AUTOSIZE );
        imshow( " LUT ", dst );

        waitKey();
        return 0;
    }
```

The following figure shows you the output of the code:



```
(x=154, y=194) ~ R:122 G:121 B:64
(x=171, y=193) ~ R:48 G:45 B:32
```

The original and LUT-transformed images

Geometrical transformations

Geometrical transformations do not change the image content but instead deform the image by deforming their grid. In this case, output image pixel values are computed by first obtaining the coordinates of the appropriate input pixels by applying the corresponding mapping functions and copying the original pixel values from the obtained positions to the new ones:

$$O(x,y) = I\left(f_x(x,y), f_y(x,y)\right)$$

This type of operation has two problems:

- **Extrapolation**: $f_x(x,y)$ and $f_y(x,y)$ could obtain values that indicate a pixel outside the image boundary. The extrapolation methods used in geometrical transformations are the same as the ones used in image filtering plus another one called `BORDER_TRANSPARENT`.
- **Interpolation**: $f_x(x,y)$ and $f_y(x,y)$ are usually floating-point numbers. In OpenCV, it is possible to select between nearest-neighbor and polynomial interpolation methods. Nearest-neighbor interpolation consists of rounding the floating-point coordinate to the nearest integer. The supported interpolation methods are:
    - `INTER_NEAREST`: This is the nearest-neighbor interpolation explained previously.
    - `INTER_LINEAR`: This is a bilinear interpolation method. It is used by default.
    - `INTER_AREA`: This resamples using pixel area relation.
    - `INTER_CUBIC`: This is bicubic interpolation method over a 4 x 4 pixel neighborhood.
    - `INTER_LANCZOS4`: This is the Lanczos interpolation method over an 8 x 8 pixel neighborhood.

The geometrical transformations supported in OpenCV include affine (scaling, translation, rotation, and so on) and perspective transformations.

Affine transformation

An affine transformation is a geometric transformation that preserves all points from an initial line on a line after applying it. Furthermore, distance ratios from each of these points to the ends of the lines are also preserved. On the other hand, affine transformations don't necessarily preserve angles and lengths.

Geometric transformations such as scaling, translation, rotation, skewing, and reflection are all affine transformations.

Scaling

Scaling an image is resizing it by shrinking or zooming. The function in OpenCV for this purpose is `void resize(InputArray src, OutputArray dst, Size dsize, double fx = 0, double fy = 0, int interpolation = INTER_LINEAR)`. Apart from `src` and `dst`, the input and output images, it has some parameters to specify the size to which the image is to be rescaled. If the new image size is specified by setting `dsize` to a value different from 0, the scaled factor parameters, `fx` and `fy`, are both 0 and `fx` and `fy` are calculated from `dsize` and the original size of the input image. If `fx` and `fy` are different from 0 and `dsize` equals 0, `dsize` is calculated from the other parameters. A scale operation could be represented by its transformation matrix:

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

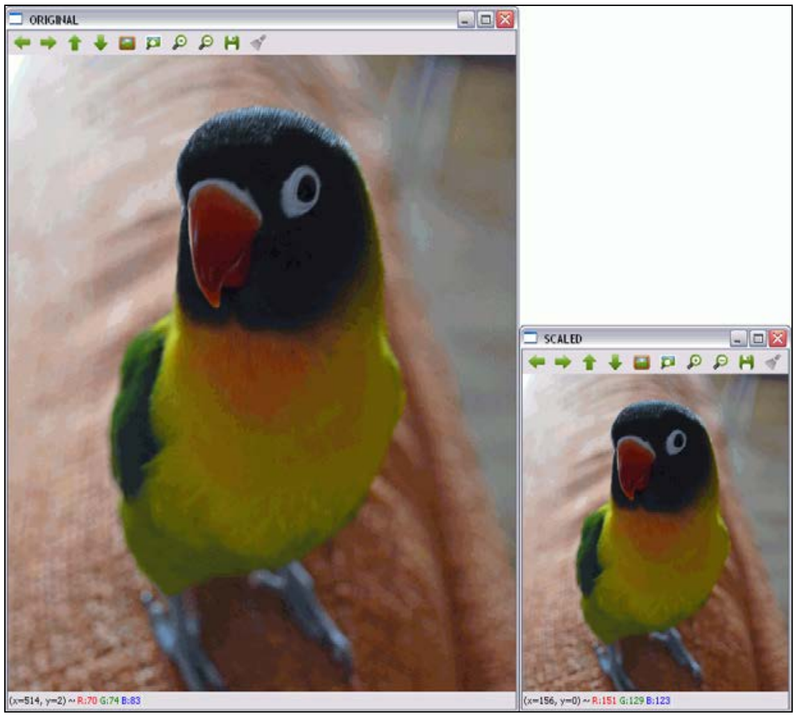
Here, s_x and s_y are the scale factors in the x and y axis.

The example code

The following **Scale** example shows you how to scale an image through the `resize` function. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply the scale
    Mat dst;
    resize(src, dst, Size(0,0), 0.5, 0.5);

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " SCALED ", WINDOW_AUTOSIZE );
    imshow( " SCALED ", dst );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:


ORIGINAL
SCALED
(x=514, y=2) ~ R:70 G:74 B:83
(x=156, y=0) ~ R:151 G:129 B:123

Original and scaled images; fx and fy are both 0.5

Translation

Translation is simply moving the image along a specific direction and distance. Thus, a translation could be represented by means of a vector, (t_x, t_y), or its transformation matrix:

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$$

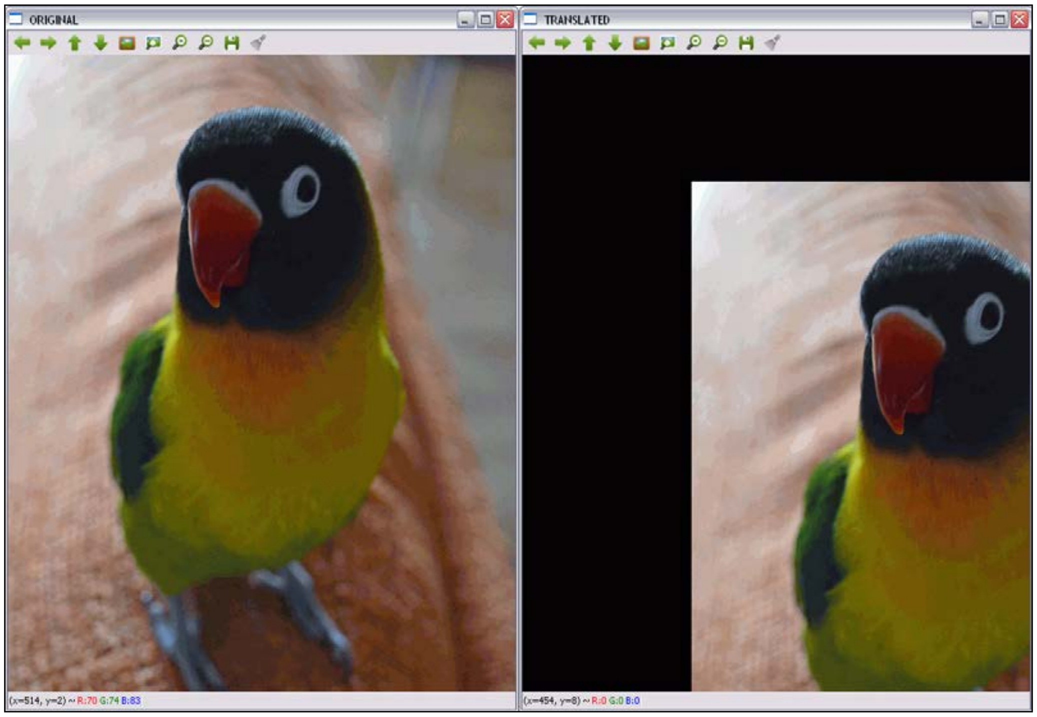
In OpenCV, it is possible to apply translations using the `void warpAffine( InputArray src, OutputArray dst, InputArray M, Size dsize, int flags = INTER_LINEAR, int borderMode = BORDER_CONSTANT, const Scalar& borderValue = Scalar())` function. The `M` parameter is the transformation matrix that converts `src` into `dst`. The interpolation method is specified using the `flags` parameter, which also supports the `WARP_INVERSE_MAP` value, which means that `M` is the inverse transformation. The `borderMode` parameter is the extrapolation method, and `borderValue` is used when `borderMode` is `BORDER_CONSTANT`.

The example code

The **Translation** example shows you how to use the `warpAffine` function to translate an image. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply translation
    Mat dst;
    Mat M = (Mat_<double>(2,3) << 1, 0, 200, 0, 1, 150);
    warpAffine(src,dst,M,src.size());

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " TRANSLATED ", WINDOW_AUTOSIZE );
    imshow( " TRANSLATED ", dst );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:



Original and displaced images. The horizontal displacement is 200 and vertical displacement is 150.

Image rotation

Image rotation involves a specific angle, θ. OpenCV supports scaled rotations in a specific location using a transformation matrix defined as follows:

$$M = \begin{bmatrix} sf \cdot \cos(\theta) & sf \cdot \sin(\theta) & (1 - sf \cdot \cos(\theta)) \cdot x - sf \cdot \sin(\theta) \cdot y \\ -sf \cdot \sin(\theta) & sf \cdot \cos(\theta) & sf \cdot \sin(\theta) \cdot x + (1 - sf \cdot \cos(\theta)) \cdot y \end{bmatrix}$$

Here, *x* and *y* are the coordinates of the rotation point and *sf* is the scale factor.

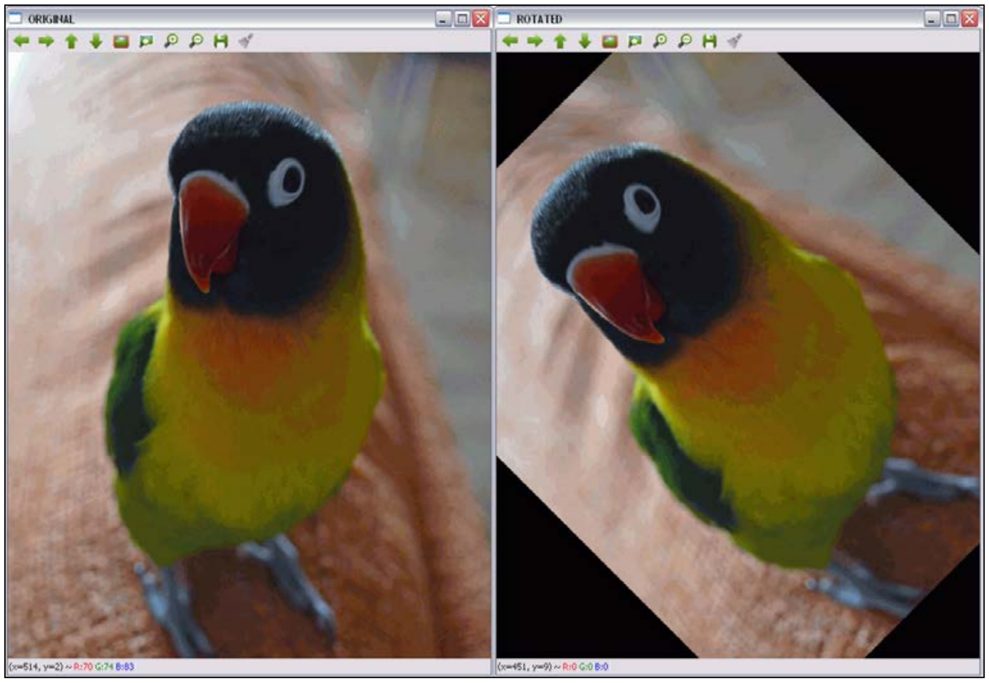
Rotations are applied like translations by means of the `warpAffine` function but using the `Mat getRotationMatrix2D(Point2f center, double angle, double scale)` function to create the rotation transformation matrix. The `M` parameter is the transformation matrix that converts `src` into `dst`. As the names of the parameters indicate, `center` is the center point of the rotation, `angle` is the rotation angle (in a counter-clockwise direction), and `scale` is the scale factor.

The example code

The following **Rotate** example shows you how to use the `warpAffine` function to rotate an image. A 45-degree centered rotation matrix is firstly obtained by `getRotationMatrix2D( Point2f( src.cols/2, src.rows/2 ), 45, 1 )`. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply the rotation
    Mat dst;
    Mat M = getRotationMatrix2D(Point2f(src.cols/2,src.rows/2),45,1);
    warpAffine(src,dst,M,src.size());

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " ROTATED ", WINDOW_AUTOSIZE );
    imshow( " ROTATED ", dst );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:

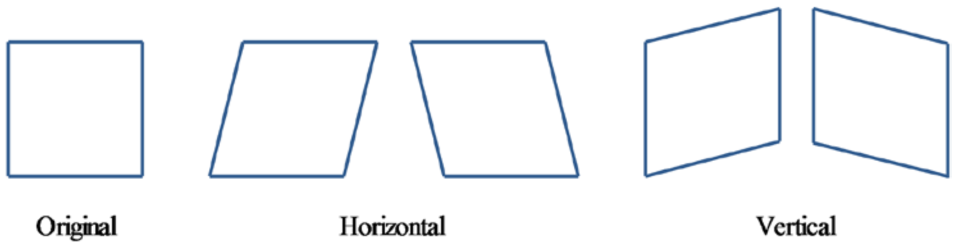


The original image and the image after a 45-degree centered rotation is applied

Skewing

A skewing transformation displaces each point in a fixed direction by an amount proportional to its signed distance from a line that is parallel to that direction. Therefore, it will usually distort the shape of a geometric figure, for example, turning squares into non-square parallelograms and circles into ellipses. However, a skewing preserves the area of geometric figures, the alignment, and relative distances of collinear points. A skewing mapping is the main difference between upright and slanted (or italic) styles of letters.

Skewing can also be defined by its angle, θ.



The original and its rotated 45 degrees from center image

Using the skewing angle, the transformation matrices for horizontal and vertical skewing are:

$$T_H = \begin{bmatrix} 1 & \cot(\theta) & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad T_V = \begin{bmatrix} 1 & 0 & 0 \\ \cot(\theta) & 1 & 0 \end{bmatrix}$$

Due to the similarities with previous transformations, the function used to apply skewing is `warpAffine`.

On most occasions, it will be necessary to add some size to the output image and/or apply translation (changing the last column on the shear transformation matrix) in order to display the output image completely and in a centered manner.

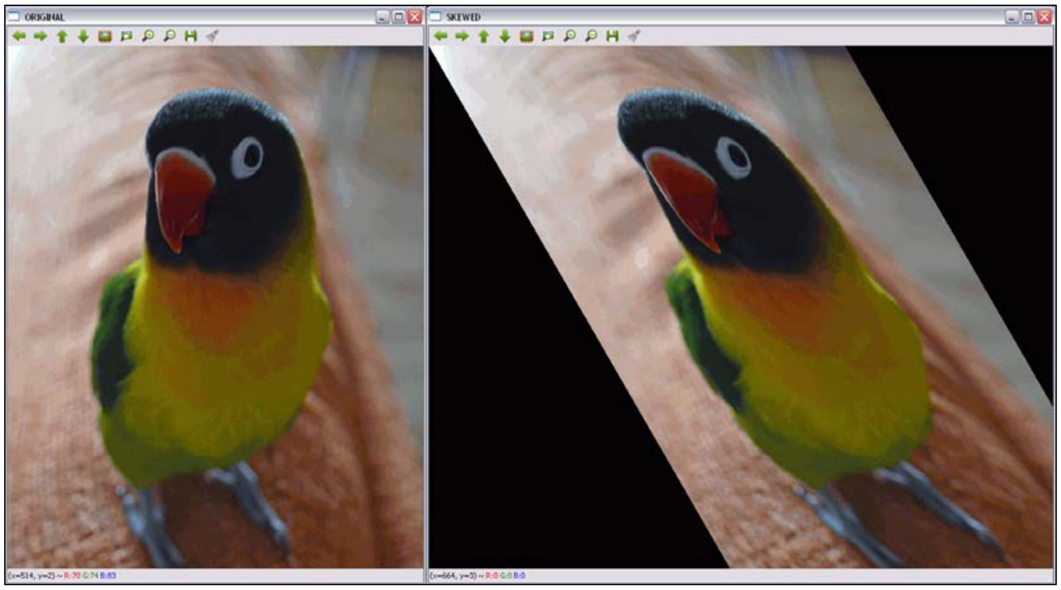
The example code

The following **Skew** example shows you how to use the `warpAffine` function to skew θ = π/3 horizontally in an image. The example code is:

```
#include "opencv2/opencv.hpp"
#include <math.h>

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply skew
    Mat dst;
    double m = 1/tan(M_PI/3);
    Mat M = (Mat_<double>(2,3) << 1, m, 0, 0, 1, 0);
    warpAffine(src,dst,M,Size(src.cols+0.5*src.cols,src.rows));

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
```

```
        namedWindow( " SKEWED ", WINDOW_AUTOSIZE );
        imshow( " SKEWED ", dst );

        waitKey();
        return 0;
    }
```

The following figure shows you the output of the code:



The original image and the image when skewed horizontally

Reflection

As reflection is done over the *x* and *y* axes by default, it is necessary to apply translation (the last column of the transformation matrix). Then, the reflection matrix is:

$$T_H = \begin{bmatrix} -1 & 0 & t_x \\ 0 & 1 & 0 \end{bmatrix} \quad T_V = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & t_y \end{bmatrix} \quad T = \begin{bmatrix} -1 & 0 & t_x \\ 0 & -1 & t_y \end{bmatrix}$$

Here, t_x is the number of image columns and t_y is the number of image rows.

As with previous transformations, the function used to apply reflection is `warpAffine`.

> Other affine transformations can be applied using the `warpAffine` function with their corresponding transformation matrices.

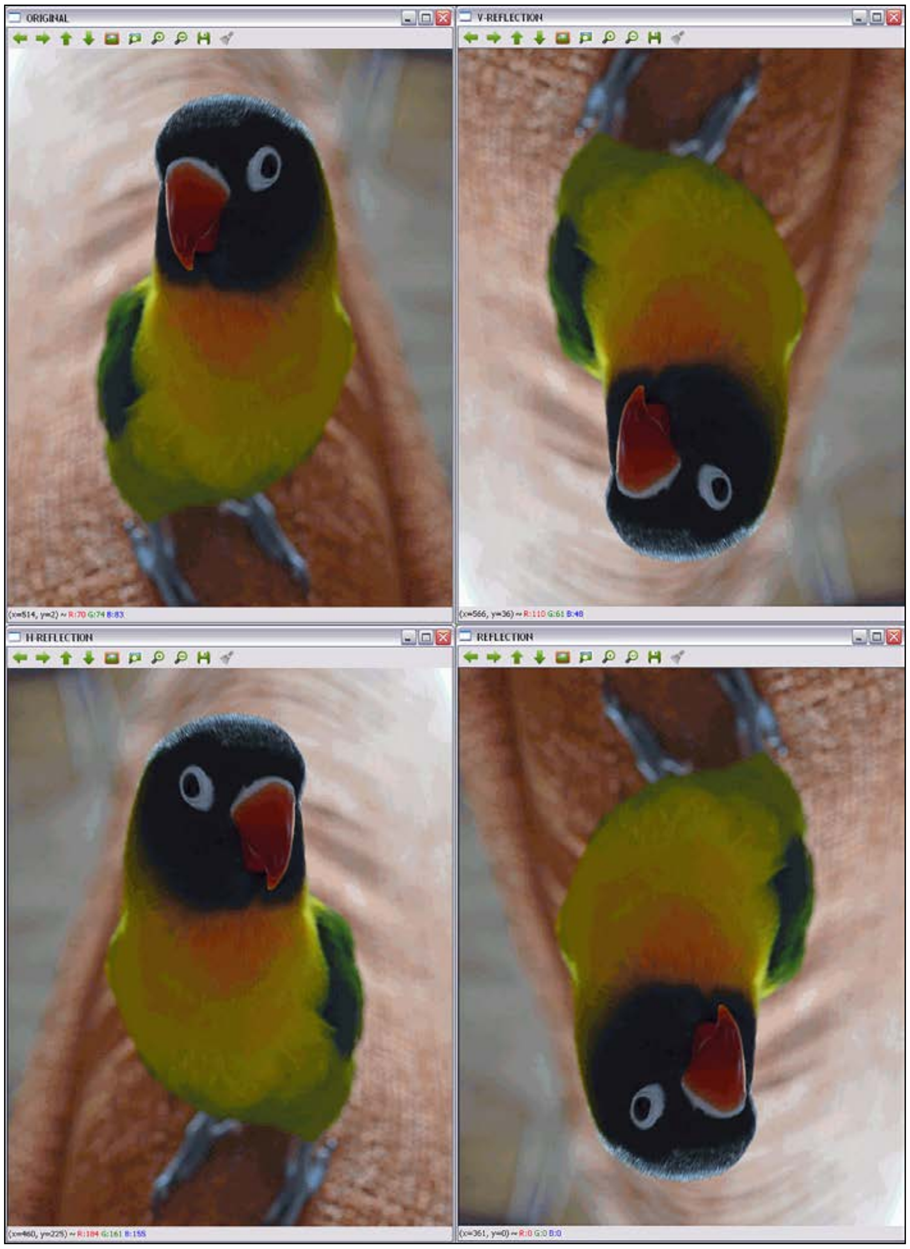
The example code

The following **Reflect** example shows you an example of horizontal, vertical, and combined reflection of an image using the `warpAffine` function. The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Apply the reflections
    Mat dsth, dstv, dst;
    Mat Mh  = (Mat_<double>(2,3) << -1, 0, src.cols, 0, 1, 0
    Mat Mv  = (Mat_<double>(2,3) << 1, 0, 0, 0, -1, src.rows);
    Mat M  = (Mat_<double>(2,3) << -1, 0, src.cols, 0, -1, src.rows);
    warpAffine(src,dsth,Mh,src.size());
    warpAffine(src,dstv,Mv,src.size());
    warpAffine(src,dst,M,src.size());

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " H-REFLECTION ", WINDOW_AUTOSIZE );
    imshow( " H-REFLECTION ", dsth );
    namedWindow( " V-REFLECTION ", WINDOW_AUTOSIZE );
    imshow( " V-REFLECTION ", dstv );
    namedWindow( " REFLECTION ", WINDOW_AUTOSIZE );
    imshow( " REFLECTION ", dst );

    waitKey();
```

```
        return 0;
    }
```

The following figure shows you the output for the code:



The original and rotated images in X, Y, and both axes

Perspective transformation

For perspective transformation, a 3 x 3 transformation matrix is needed, although the work is performed over two-dimensional images. Straight lines remain straight in the output image, but in this case, proportions change. Finding the transformation matrix is more complex than with affine transformations. When working with perspective, the coordinates of four points of the input image matrix and their corresponding coordinates on the output image matrix are used to perform this operation.

With these points and the `getPerspectiveTransform` OpenCV function, it is possible to find the perspective transformation matrix. After obtaining the matrix, `warpPerspective` is applied to obtain the output of the perspective transformation. The two functions are explained in detail here:

- `Mat getPerspectiveTransform(InputArray src, InputArray dst)` and `Mat getPerspectiveTransform(const Point2f src[], const Point2f dst[])`: This returns the perspective transformation matrix calculated from `src` and `dst`.
- `void warpPerspective(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar())`: This applies an `M` affine transformation to an `src` image, obtaining the new `dst` image. The rest of the parameters are the same as in other geometrical transformations discussed.

The example code

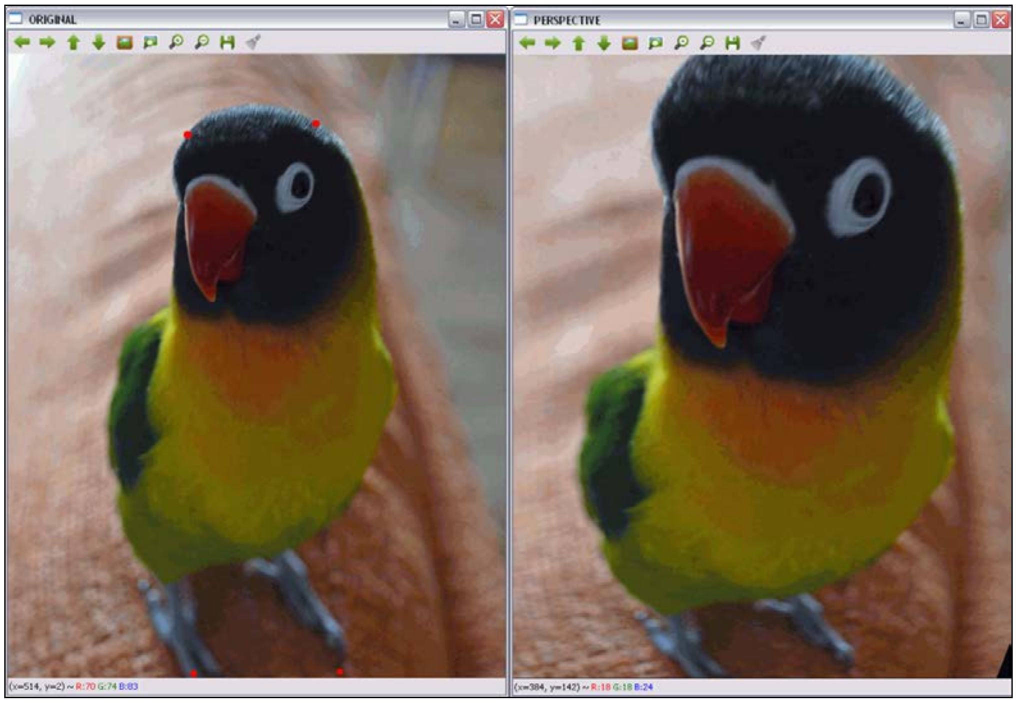
The following **Perspective** example shows you an example of how to change the perspective of an image using the `warpPerspective` function. In this case, it is necessary to indicate the coordinates of four points from the first image and another four from the output to calculate the perspective transformation matrix through `getPerspectiveTransform`. The selected points are:

```
Point2f src_verts[4];
src_verts[2] = Point(195, 140);
src_verts[3] = Point(410, 120);
src_verts[1] = Point(220, 750);
src_verts[0] = Point(400, 750);
Point2f dst_verts[4];
dst_verts[2] = Point(160, 100);
dst_verts[3] = Point(530, 120);
dst_verts[1] = Point(220, 750);
dst_verts[0] = Point(400, 750);
```

The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    Mat dst;
    Point2f src_verts[4];
    src_verts[2] = Point(195, 140);
    src_verts[3] = Point(410, 120);
    src_verts[1] = Point(220, 750);
    src_verts[0] = Point(400, 750);
    Point2f dst_verts[4];
    dst_verts[2] = Point(160, 100);
    dst_verts[3] = Point(530, 120);
    dst_verts[1] = Point(220, 750);
    dst_verts[0] = Point(400, 750);

    // Obtain and Apply the perspective transformation
    Mat M = getPerspectiveTransform(src_verts,dst_verts);
    warpPerspective(src,dst,M,src.size());

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " PERSPECTIVE ", WINDOW_AUTOSIZE );
    imshow( " PERSPECTIVE ", dst );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:


ORIGINAL
PERSPECTIVE
(x=514, y=2) ~ R:70 G:74 B:83
(x=384, y=142) ~ R:18 G:18 B:24

Perspective results with the points marked in the original image

Inpainting

Inpainting is the process of reconstructing damaged parts of images and videos. This process is also known as image or video interpolation. The basic idea is to simulate the process done by restorers with antiques. Nowadays, with the wide use of digital cameras, inpainting has become an automatic process that is used not only for image restoration by deleting scratches, but also for other tasks, such as object or text removal.

OpenCV supports an inpainting algorithm as of Version 2.4. The function for this purpose is:

- `void inpaint(InputArray src, InputArray inpaintMask, OutputArray dst, double inpaintRadius, int flags)`: This restores the areas indicated with non-zero values by the `inpaintMask` parameter in the source (`src`) image. The `inpaintRadius` parameter indicates the neighborhood to be used by the algorithm specified by `flags`. Two methods could be used in OpenCV:
    - `INPAINT_NS`: This is the Navier-Stokes-based method
    - `INPAINT_TELEA`: This is the method proposed by Alexandru Telea

Finally, the restored image is stored in `dst`.

More details about the inpainting algorithms used in OpenCV can be found at `http://www.ifp.illinois.edu/~yuhuang/inpainting.html`

For video inpainting, consider the video as a sequence of images and apply the algorithm over all of them.

The example code

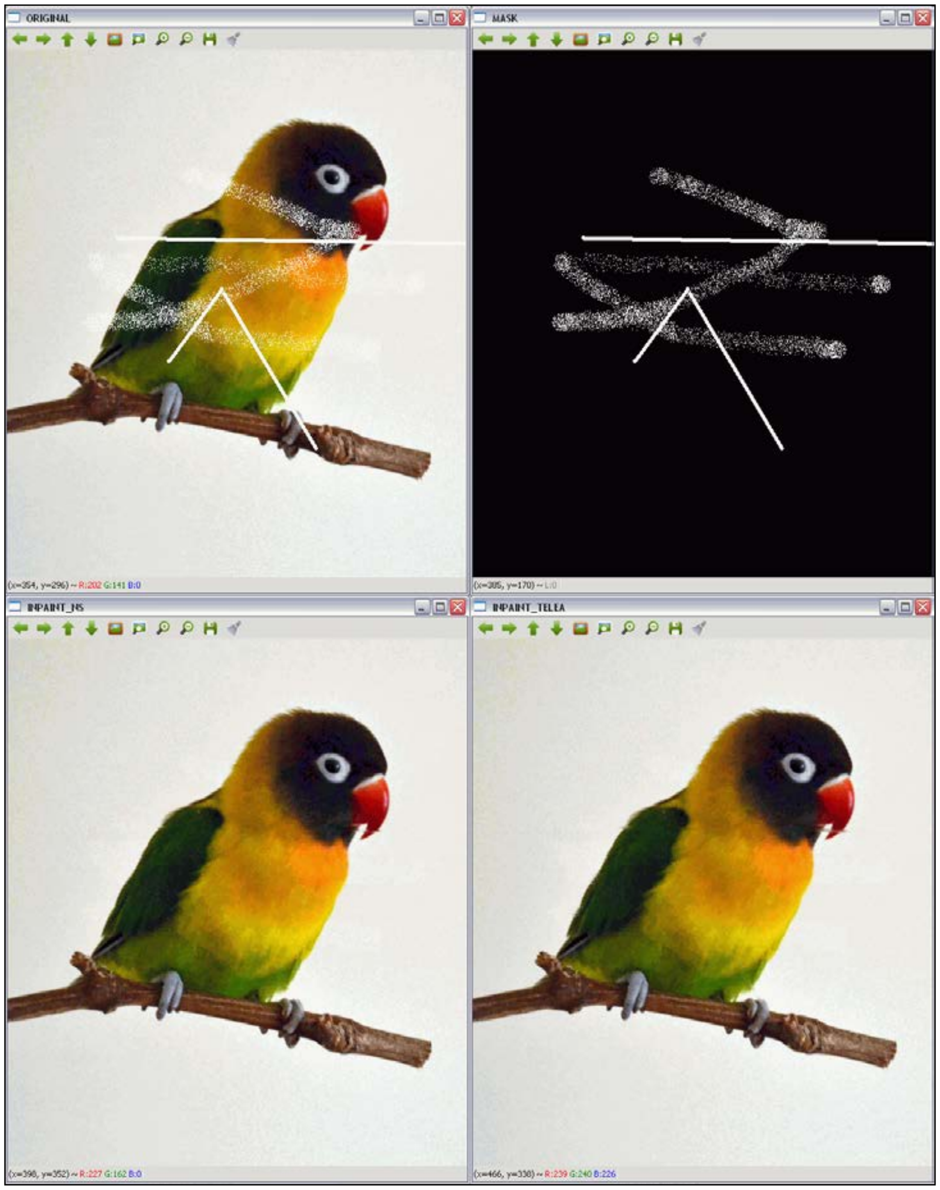
The following **inpainting** example shows you how to use the `inpaint` function to inpaint the areas of an image specified in an image mask.

The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
```

```
        src = imread(argv[1]);

        // Read the mask file
        Mat mask;
        mask = imread(argv[2]);
        cvtColor(mask, mask, COLOR_RGB2GRAY);

        // Apply the inpainting algorithms
        Mat dst, dst2;
        inpaint(src, mask, dst, 10, INPAINT_TELEA);
        inpaint(src, mask, dst2, 10, INPAINT_NS);

        // Show the results
        namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
        imshow( " ORIGINAL ", src );
        namedWindow( " MASK ", WINDOW_AUTOSIZE );
        imshow( " MASK ", mask );
        namedWindow(" INPAINT_TELEA ", WINDOW_AUTOSIZE );
        imshow( " INPAINT_TELEA ", dst );
        namedWindow(" INPAINT_NS ", WINDOW_AUTOSIZE );
        imshow( " INPAINT_NS ", dst2 );

        waitKey();
        return 0;
    }
```

The following figure shows you the output of the code:



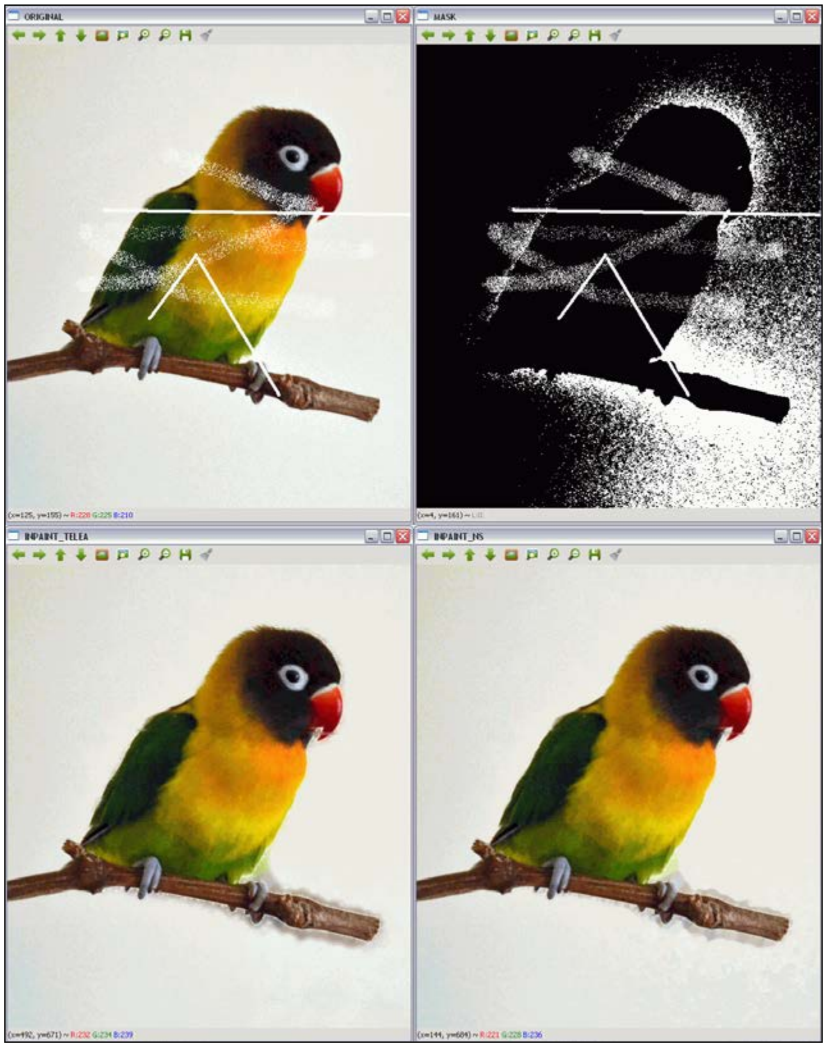
Results of applying inpainting

The first row contains the original image and the mask used. The second row contains the results from the inpainting proposed by Telea on the left-hand side and the Navier-Stokes-based method on the right-hand side.

Getting the inpainting mask is not an easy task. The **inpainting2** example code shows you an example of how we can obtain the mask from the source image using binary thresholding through `threshold(mask, mask, 235, 255, THRESH_BINARY)`:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Create the mask
    Mat mask;
    cvtColor(src, mask, COLOR_RGB2GRAY);
    threshold(mask, mask, 235, 255, THRESH_BINARY);

    // Apply the inpainting algorithms
    Mat dst, dst2;
    inpaint(src, mask, dst, 10, INPAINT_TELEA);
    inpaint(src, mask, dst2, 10, INPAINT_NS);

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " MASK ", WINDOW_AUTOSIZE );
    imshow( " MASK ", mask );
    namedWindow(" INPAINT_TELEA ", WINDOW_AUTOSIZE );
    imshow( " INPAINT_TELEA ", dst );
    namedWindow(" INPAINT_NS ", WINDOW_AUTOSIZE );
    imshow( " INPAINT_NS ", dst2 );

    waitKey();
    return 0;
}
```

The following figure shows you the output of the code:



Results of applying inpainting algorithms without knowing the mask

The first row contains the original image and the extracted mask. The second row contains the results from the inpainting proposed by Telea on the left-hand side and the Navier-Stokes-based method on the right-hand side.

The results from this example show you that obtaining a perfect mask is not always possible. Some other parts of the image, such as the background or noise, are included sometimes. However, the inpainting results remain acceptable as the resulting images are close to the ones obtained in the other case.

Denoising

Denoising or noise reduction is the process of removing noise from signals obtained from analog or digital devices. This section focuses its attention on reducing noise from digital images and videos.

Although smoothing and median filtering are good options to denoise an image, OpenCV provides other algorithms to perform this task. These are the nonlocal means and the **TVL1** (**Total Variation L^1**) algorithms. The basic idea of the nonlocal means algorithm is to replace the color of a pixel with an average of the colors from several image sub-windows that are similar to the one that comprises the pixel neighborhood. On the other hand, the TVL1 variational denoising model, which is implemented with the primal-dual optimization algorithm, considers the image-denoising process a variational problem.

> More information about the nonlocal means and the TVL1 denoising algorithms can be found at `http://www.ipol.im/pub/art/2011/bcm_nlm` and `http://znah.net/rof-and-tv-l1-denoising-with-primal-dual-algorithm.html`, respectively.

OpenCV provides four functions to denoise color and grayscale images following the nonlocal means approach. For the TVL1 model, one function is provided. These functions are:

- `void fastNlMeansDenoising(InputArray src, OutputArray dst, float h = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: This denoises a single grayscale image loaded in `src`. The `templateWindowSize` and `searchWindowSize` parameters are the sizes in pixels of the template patch that is used to compute weights and the window that is used to compute the weighted average for the given pixel. These should be odd and their recommended values are 7 and 21 pixels, respectively. The `h` parameter regulates the effect of the algorithm. Larger `h` values remove more noise defects but with the drawback of removing more image details. The output is stored in `dst`.
- `void fastNlMeansDenoisingColored(InputArray src, OutputArray dst, float h = 3, float hForColorComponents = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: This is a modification of the previous function for colored images. It converts the `src` image to the CIELAB color space and then separately denoises the L and AB components with the `fastNlMeansDenoising` function.

- `void fastNlMeansDenoisingMulti(InputArrayOfArrays srcImgs, OutputArray dst, int imgToDenoiseIndex, int temporalWindowSize, float h = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: This uses an image sequence to obtain a denoised image. Two more parameters are needed in this case: `imgToDenoiseIndex` and `temporalWindowSize`. The value of `imgToDenoiseIndex` is the target image index in `srcImgs` to be denoised. Finally, `temporalWindowSize` is used to establish the number of surrounding images to be used for denoising. This should be odd.
- `void fastNlMeansDenoisingColoredMulti(InputArrayOfArrays srcImgs, OutputArray dst, int imgToDenoiseIndex, int temporalWindowSize, float h = 3, float hForColorComponents = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: This is based on the `fastNlMeansDenoisingColored` and `fastNlMeansDenoisingMulti` functions. The parameters are explained in the rest of the functions.
- `void denoise_TVL1(const std::vector<Mat>& observations, Mat& result, double lambda, int niters)`: This obtains a denoised image in `result` from one or more noisy images stored in `observations`. The `lambda` and `niters` parameters control the strength and the number of iterations of the algorithm.

The example code

The following **denoising** example shows you how to use one of the denoising functions for noise reduction over a colored image (`fastNlMeansDenoisingColored`). As the example uses an image without noise, something needs to be added. For this purpose, the following lines of code are used:

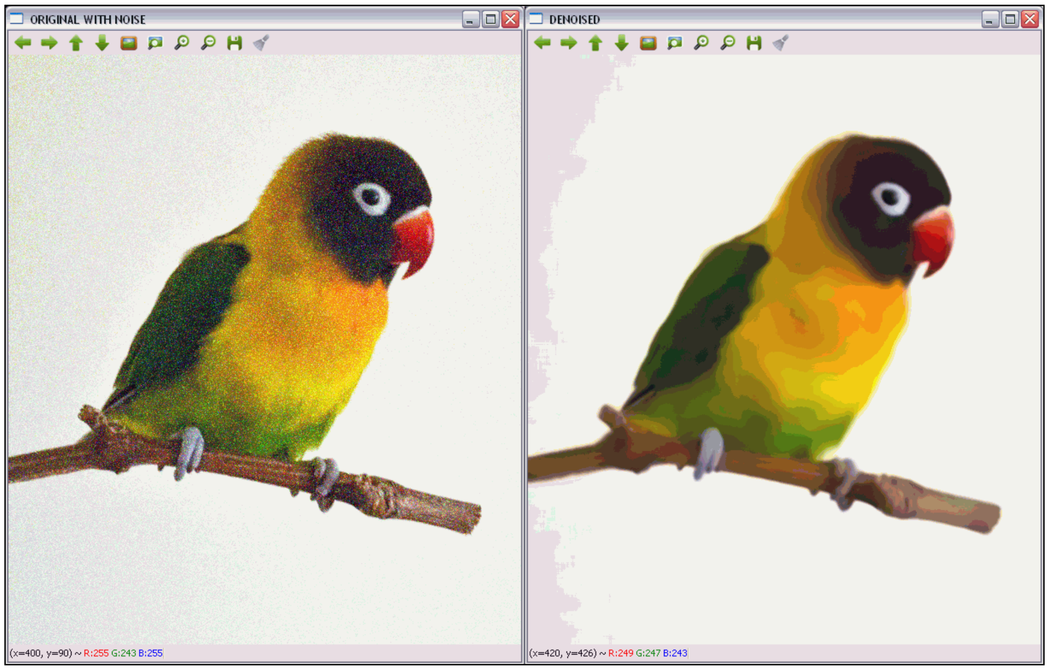
```
Mat noisy = src.clone();
Mat noise(src.size(), src.type());
randn(noise, 0, 50);
noisy += noise;
```

A `Mat` element is created with the same size and type of the original image to store noise generated by the `randn` function on it. Finally, the noise is added to the cloned image to obtain the noisy image.

The example code is:

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
    // Read the source file
    Mat src;
    src = imread(argv[1]);

    // Add some noise
    Mat noisy = src.clone();
    Mat noise(src.size(), src.type());
    randn(noise, 0, 50);
    noisy += noise;

    // Apply the denoising algorithm
    Mat dst;
    fastNlMeansDenoisingColored(noisy, dst,30,30,7,21);

    // Show the results
    namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL ", src );
    namedWindow( " ORIGINAL WITH NOISE ", WINDOW_AUTOSIZE );
    imshow( " ORIGINAL WITH NOISE ", noisy );
    namedWindow(" DENOISED ", WINDOW_AUTOSIZE );
    imshow( " DENOISED ", dst );

    waitKey();
    return 0;
}
```

The following figure shows you noisy and denoised images from executing the previous code:


ORIGINAL WITH NOISE
DENOISED
(x=400, y=90) ~ R:255 G:243 B:255
(x=420, y=426) ~ R:249 G:247 B:243

Results from applying denoising

Summary

In this chapter, we explained methods for image enhancement and correction, including noise reduction, edge enhancement, morphological operations, geometrical transformations, and the restoration of damaged images. Different options have been presented in each case to provide the reader with all the options that can be used in OpenCV.

The next chapter will cover color spaces and how to convert them. In addition, color-space-based segmentation and color-transfer methods will be explained.

4
Processing Color

Color is a perceptual result created in response to the excitation of our visual system by light incident upon the retina in the visible region of the spectrum. The color of an image may contain a great deal of information, which can be used for simplifying image analysis, object identification, and extraction based on color. These procedures are usually carried out considering the pixel values in the color space in which it is defined. In this chapter, the following topics will be covered:

- The color spaces used in OpenCV and how to convert an image from one color model to another
- An example of how to segment a picture considering the color space in which it is defined
- How to transfer the appearance of an image to another using the color transfer method

Color spaces

The human visual system is able to distinguish hundreds of thousands of colors. To obtain this information, the human retina has three types of color photoreceptor cone cells, which respond to incident radiation. Because of this, most human color perceptions can be generated with three numerical components called primaries.

To specify a color in terms of three or more particular characteristics, there are a number of methods called **color spaces** or **color models**. Selecting between them to represent an image depends on the operations to be performed, because some are more appropriate according to the required application. For example, in some color spaces such as RGB, the brightness affects the three channels, a fact that could be unfavorable for some image-processing operations. The next section explains color spaces used in OpenCV and how to convert a picture from one color model to another.

Conversion between color spaces (cvtColor)

There are more than 150 color-space conversion methods available in OpenCV. The function provided by OpenCV in the `imgproc` module is `void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0)`. The arguments of this function are:

- `src`: This is an input image 8-bit unsigned, 16-bit unsigned (CV_16UC), or single-precision floating-point.
- `dst`: This is the output image of the same size and depth as `src`.
- `code`: This is the color space conversion code. The structure of this parameter is `COLOR_SPACEsrc2SPACEdst`. Some example values are `COLOR_BGR2GRAY` and `COLOR_YCrCb2BGR`.
- `dstCn`: This is the number of channels in the destination image. If this parameter is 0 or omitted, the number of the channels is derived automatically from `src` and `code`.

Examples of this function will be described in the upcoming sections.

[The `cvtColor` function can only convert from RGB to another color space or from another color space to RGB, so if the reader wants to convert between two color spaces other than RGB, a first conversion to RGB must be done.]

Various color spaces in OpenCV are discussed in the upcoming sections.

RGB

RGB is an additive model in which an image consists of three independent image planes or channels: red, green, and blue (and optionally, a fourth channel for the transparency, sometimes called alpha channel). To specify a particular color, each value indicates the amount of each of the components present on each pixel, with higher values corresponding to brighter pixels. This color space is widely used because it corresponds to the three photoreceptors of the human eye.

[The default color format in OpenCV is often referred to as RGB but it is actually stored as BGR (the channels are reversed).]

The example code

The following **BGRsplit** example shows you how to load an RGB image, splitting and showing each particular channel in gray and in a color. The first part of the code is used to load and show the picture:

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

using namespace std;
using namespace cv;

vector<Mat> showSeparatedChannels(vector<Mat> channels);

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("BGR.png");
    imshow("Picture",image);
```

The next part of the code splits the picture into each channel and shows it:

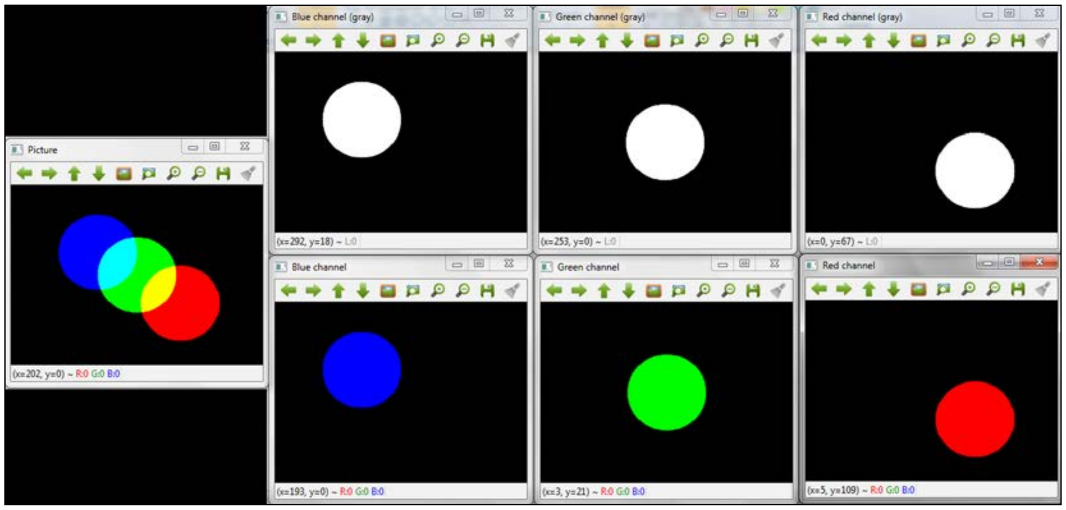
```
    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("Blue channel (gray)", WINDOW_AUTOSIZE );
    imshow("Blue channel (gray)",channels[0]);
    namedWindow("Green channel (gray)", WINDOW_AUTOSIZE );
    imshow("Green channel (gray)",channels[1]);
    namedWindow("Red channel (gray)", WINDOW_AUTOSIZE );
    imshow("Red channel (gray)",channels[2]);

    //show channels in BGR
    vector<Mat> separatedChannels=showSeparatedChannels(channels);

    namedWindow("Blue channel", WINDOW_AUTOSIZE );
    imshow("Blue channel",separatedChannels[0]);
    namedWindow("Green channel", WINDOW_AUTOSIZE );
    imshow("Green channel",separatedChannels[1]);
    namedWindow("Red channel", WINDOW_AUTOSIZE );
```

```
        imshow("Red channel",separatedChannels[2]);

        waitKey(0);

        return 0;
    }
```

It is worth noting the use of the `void split(InputArray m, OutputArrayOfArrays mv)` OpenCV function to split the image `m` in its three channels and save it in a vector of `Mat` called `mv`. On the contrary, the `void merge( InputArrayOfArrays mv, OutputArray dst)` function is used to merge all the `mv` channels in one `dst` image. Furthermore, a function denominated as `showSeparatedChannels` is used to create three color images representing each of the channels. For each channel, the function generates `vector<Mat> aux` composed by the channel itself and two auxiliary channels ordered with all their values set to 0, which represent the other two channels of the color model. Finally, the aux picture is merged, generating an image with only one channel fulfilled. This function code, which will also be used in other examples of this chapter, is as follows:

```
vector<Mat> showSeparatedChannels(vector<Mat> channels){
    vector<Mat> separatedChannels;
    //create each image for each channel
    for ( int i = 0 ; i < 3 ; i++){
        Mat zer=Mat::zeros( channels[0].rows, channels[0].cols,
channels[0].type());
        vector<Mat> aux;
        for (int j=0; j < 3 ; j++){
            if(j==i)
                aux.push_back(channels[i]);
            else
                aux.push_back(zer);
        }

        Mat chann;
        merge(aux,chann);

        separatedChannels.push_back(chann);
    }
    return separatedChannels;
}
```

The following figure shows you the output of the example:



The original RGB image and channel splitting

Grayscale

In grayscale, the value of each pixel is represented as a single value carrying only the intensity information, composing an image exclusively formed from different shades of gray. The color space conversion code to convert between RGB and grayscale (Y) in OpenCV using `cvtColor` is `COLOR_BGR2GRAY`, `COLOR_RGB2GRAY`, `COLOR_GRAY2BGR`, and `COLOR_GRAY2RGB`. These transformations are internally computed as follows:

$$RGB[A]\, to\, Gray:\; Y = 0.299 * R + 0.587 * G + 0.114 * B$$

$$Gray\, to\, RGB[A]:\; R = Y, G = Y, B = Y, A = \max(ChannelRange)$$

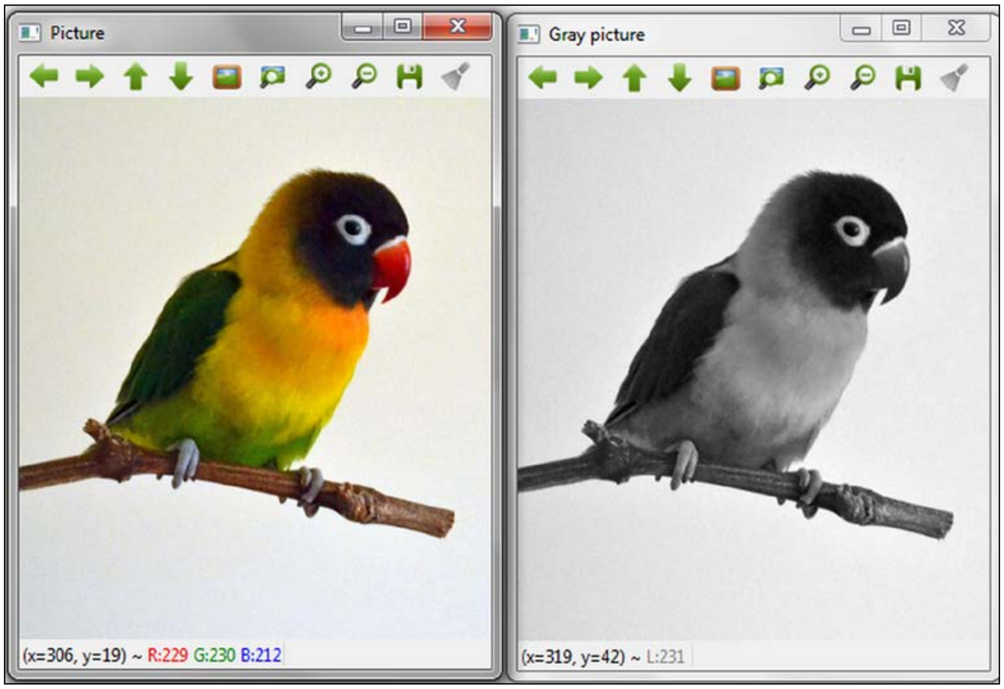
Note from the preceding formula that it is not possible to retrieve colors directly from a grayscale image.

Example code

The following **Gray** example shows you how to convert an RGB image to grayscale, showing the two pictures. The example code is:

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

using namespace cv;

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    namedWindow("Picture", WINDOW_AUTOSIZE );
    imshow("Picture",image);

    Mat imageGray;
    cvtColor(image, imageGray, COLOR_BGR2GRAY);

    namedWindow( "Gray picture", WINDOW_AUTOSIZE );
    imshow("Gray picture",imageGray);

    waitKey(0);
    return 0;
}
```

The following figure shows you the output of the code:



The original RGB image and the grayscale conversion

This method to convert from RGB to grayscale has the disadvantage of losing the contrast of the original image. *Chapter 6*, *Computational Photography*, of this book describes the decolorization process, which makes this same conversion while overcoming this issue.

CIE XYZ

The CIE XYZ system describes color with a luminance component Y, which is related to the brightness sensitivity of human vision and two additional channels, X and Z, standardized by the **Commission Internationale de L'Éclairage** (**CIE**) using statistics from experiments with several human observers. This color space is used to report color from measuring instruments, such as a colorimeter or a spectrophotometer, and it is useful when a consistent color representation across different devices is needed. The main problem of this color space is that the colors are scaled in a non-uniform manner. This fact caused the CIE to adopt the CIE L*a*b* and CIE L*u*v* color models.

The color space conversion code to convert between RGB and CIE XYZ in OpenCV using `cvtColor` is `COLOR_BGR2XYZ`, `COLOR_RGB2XYZ`, `COLOR_XYZ2BGR`, and `COLOR_XYZ2RGB`. These transformations are computed as follows:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} * \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

The example code

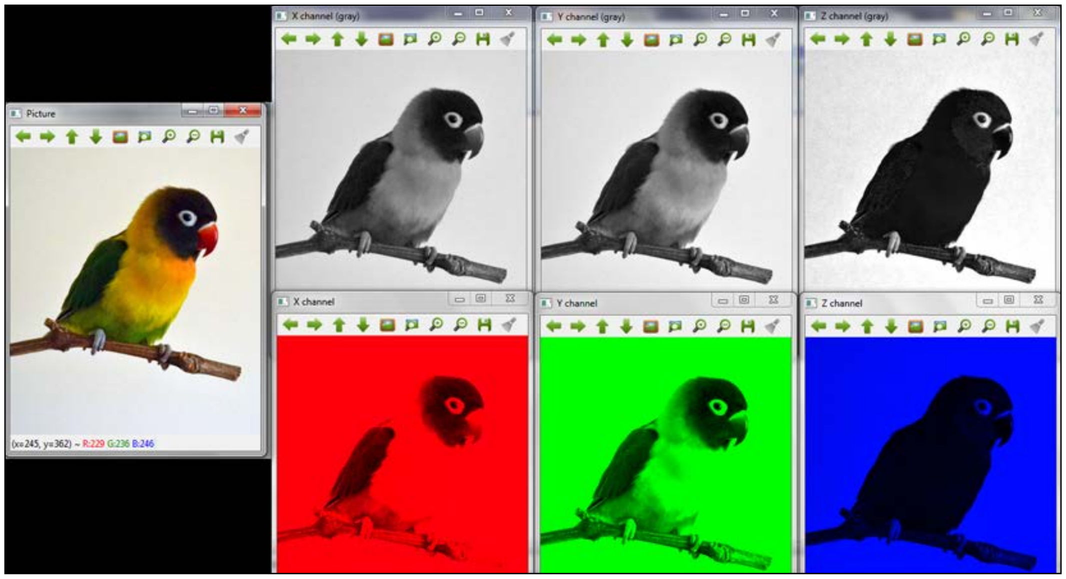
The following **CIExyz** example shows you how to convert an RGB image to the CIE XYZ color space, splitting and showing each particular channel in gray and in a color. The first part of the code is used to load and convert the picture:

```
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
using namespace std;
using namespace cv;

vector<Mat> showSeparatedChannels(vector<Mat> channels);

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to CIEXYZ
    cvtColor(image,image,COLOR_BGR2XYZ);
```

The next part of the code splits the picture in each of the CIE XYZ channels and shows them:

```
    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("X channel (gray)", WINDOW_AUTOSIZE );
    imshow("X channel (gray)",channels[0]);
    namedWindow("Y channel (gray)", WINDOW_AUTOSIZE );
    imshow("Y channel (gray)",channels[1]);
    namedWindow("Z channel (gray)", WINDOW_AUTOSIZE );
    imshow("Z channel (gray)",channels[2]);

    //show channels in BGR
    vector<Mat> separatedChannels=showSeparatedChannels(channels);

    for (int i=0;i<3;i++){        cvtColor(separatedChannels[i],separate
dChannels[i],COLOR_XYZ2BGR);
    }
    namedWindow("X channel", WINDOW_AUTOSIZE );
    imshow("X channel",separatedChannels[0]);
    namedWindow("Y channel", WINDOW_AUTOSIZE );
    imshow("Y channel",separatedChannels[1]);
    namedWindow("Z channel", WINDOW_AUTOSIZE );
    imshow("Z channel",separatedChannels[2]);

    waitKey(0);

    return 0;
}
```

The following figure shows you the output of the code:



The original RGB image and the CIE XYZ channel splitting

YCrCb

This color space is widely used in video- and image-compression schemes, and it is not an absolute color space because it is a way to encode the RGB color space. The Y channel represents luminance, while Cr and Cb represent red-difference (the difference between the R channel in the RGB colorspace and Y) and blue-difference (the difference between the B channel in the RGB colorspace and Y) chroma components, respectively. It is used widely in video- and image-compression schemes, such as MPEG and JPEG.

The color space conversion code to convert between RGB and YCrCb in OpenCV using `cvtColor` is `COLOR_BGR2YCrCb`, `COLOR_RGB2YCrCb`, `COLOR_YCrCb2BGR`, and `COLOR_YCrCb2RGB`. These transformations are computed as follows:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

$$Cr = (R - Y) * 0.713 + delta$$

$$Cb = (B - Y) * 0.564 + delta$$

$$R = Y + 1.403 * (Cr - delta)$$

$$G = Y - 0.714 * (Cr - delta) - 0.344 * (Cb - delta)$$

$$B = Y + 1.773 * (Cb - delta)$$

Then, take a look at the following:

$$delta = \begin{cases} 128 & for\,8-bit\;images \\ 32768 & for\,16-bit\;images \\ 0.5 & for\;floating-point\;images \end{cases}$$

The example code

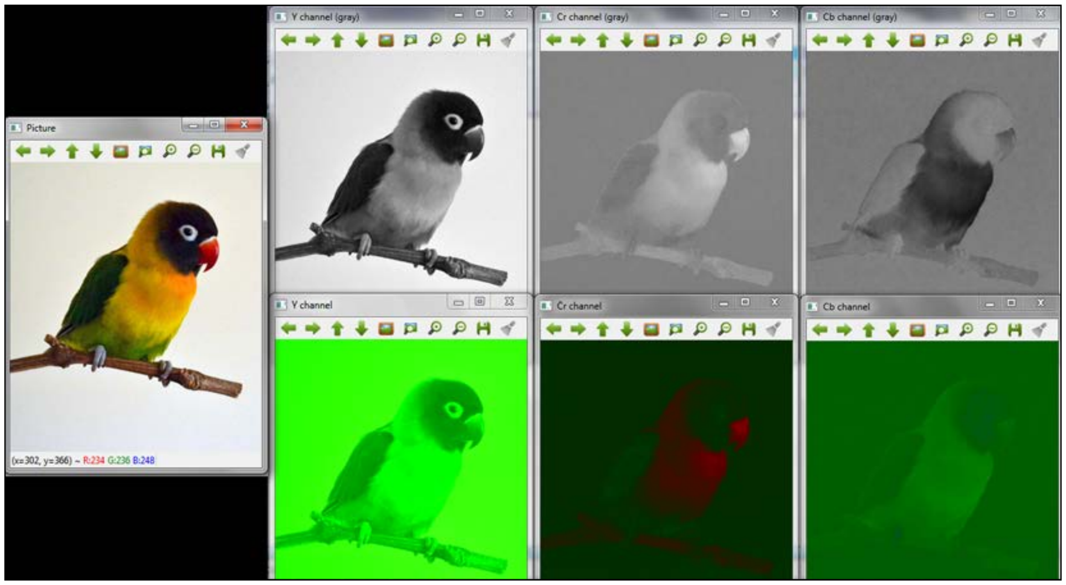
The following **YCrCb color** example shows you how to convert an RGB image to the YCrCb color space, splitting and showing each particular channel in gray and in a color. The first part of the code is used to load and convert the picture:

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

using namespace std;
using namespace cv;

vector<Mat> showSeparatedChannels(vector<Mat> channels);

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to YCrCb
    cvtColor(image,image,COLOR_BGR2YCrCb);
```

The next part of the code splits the picture in to each of the YCrCb channels and shows them:

```
    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("Y channel (gray)", WINDOW_AUTOSIZE );
    imshow("Y channel (gray)",channels[0]);
    namedWindow("Cr channel (gray)", WINDOW_AUTOSIZE );
    imshow("Cr channel (gray)",channels[1]);
    namedWindow("Cb channel (gray)", WINDOW_AUTOSIZE );
    imshow("Cb channel (gray)",channels[2]);

    //show channels in BGR
    vector<Mat> separatedChannels=showSeparatedChannels(channels);

    for (int i=0;i<3;i++){
        cvtColor(separatedChannels[i],separatedChannels[i],COLOR_
YCrCb2BGR);
    }
    namedWindow("Y channel", WINDOW_AUTOSIZE );
    imshow("Y channel",separatedChannels[0]);
    namedWindow("Cr channel", WINDOW_AUTOSIZE );
    imshow("Cr channel",separatedChannels[1]);
    namedWindow("Cb channel", WINDOW_AUTOSIZE );
    imshow("Cb channel",separatedChannels[2]);

    waitKey(0);

    return 0;
}
```

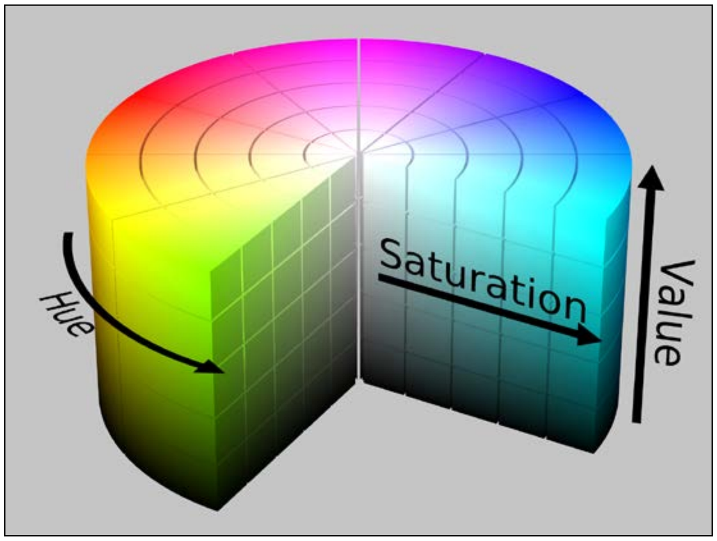
The following figure shows you the output of the code:



The original RGB image and the YCrCb channel splitting

HSV

The HSV color space belongs to the group of the so-called hue-oriented color-coordinate systems. This type of color model closely emulates models of human color perception. While in other color models, such as RGB, an image is treated as an additive result of three base colors, the three channels of **HSV** represent **hue** (H gives a measure of the spectral composition of a color), **saturation** (S gives the proportion of pure light of the dominant wavelength, which indicates how far a color is from a gray of equal brightness), and **value** (V gives the brightness relative to the brightness of a similarly illuminated white color) corresponding to the intuitive appeal of tint, shade, and tone. HSV is widely used to make a comparison of colors because H is almost independent light variations. The following figure shows you this color model representing each of the channels as a part of a cylinder:



The color space conversion code to convert between RGB and HSV in OpenCV using `cvtColor` is `COLOR_BGR2HSV`, `COLOR_RGB2HSV`, `COLOR_HSV2BGR`, and `COLOR_HSV2RGB`. In this case, it is worth noting that if the `src` image format is 8-bit or 16-bit, `cvtColor` first converts it to a floating-point format, scaling the values between 0 and 1. After that, the transformations are computed as follows:

$$V = \max(R,G,B)$$

$$S = \begin{cases} \dfrac{V - \min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H = \begin{cases} \dfrac{60(G-B)}{V - \min(R,G,B)} & \text{if } V = R \\ 120 + \dfrac{60(B-R)}{V - \min(R,G,B)} & \text{if } V = G \\ 240 + \dfrac{60(R-G)}{V - \min(R,G,B)} & \text{if } V = B \end{cases}$$

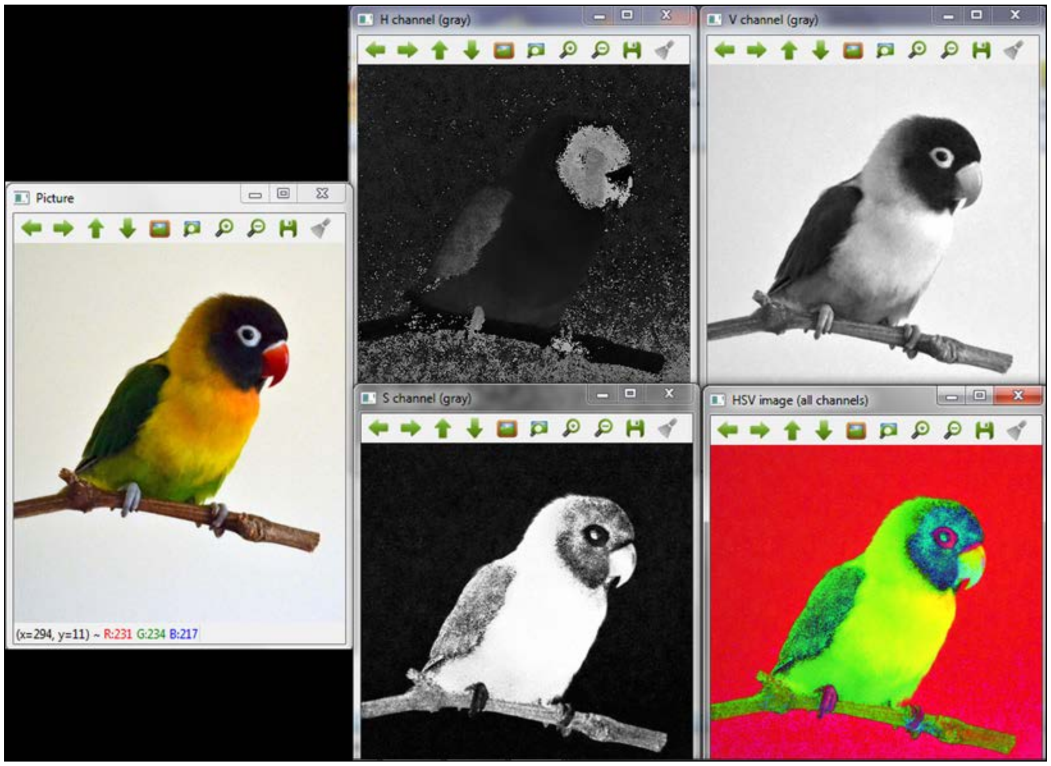
If $H<0$, then $H = H + 360$. Finally, the values are reconverted to the destination data type.

The example code

The following **HSVcolor** example shows you how to convert an RGB image to the HSV color space, splitting and showing each particular channel in grayscale and the HSV image. The example code is:

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to HSV
    cvtColor(image,image,COLOR_BGR2HSV);

    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("H channel (gray)", WINDOW_AUTOSIZE );
    imshow("H channel (gray)",channels[0]);
    namedWindow("S channel (gray)", WINDOW_AUTOSIZE );
    imshow("S channel (gray)",channels[1]);
    namedWindow("V channel (gray)", WINDOW_AUTOSIZE );
    imshow("V channel (gray)",channels[2]);

    namedWindow("HSV image (all channels)", WINDOW_AUTOSIZE );
    imshow("HSV image (all channels)",image);

    waitKey(0);

    return 0;
}
```

The following figure shows you the output of the code:



The original RGB image, HSV conversion, and channel splitting

The `imshow` function of OpenCV assumes that the color of the image to be shown is RGB, so it displays it incorrectly. If you have an image in another color space and you want to display it correctly, you first have to convert it back to RGB.

HLS

The HLS color space belongs to the group of hue-oriented color-coordinate systems, such as the HSV color model explained previously. This model was developed to specify the values of hue, lightness, and saturation of a color in each channel. The difference with respect to the HSV color model is that the lightness of a pure color defined by HLS is equal to the lightness of a medium gray, while the brightness of a pure color defined by HSV is equal to the brightness of white.

The color space conversion code to convert between RGB and HLS in OpenCV using `cvtColor` is `COLOR_BGR2HLS`, `COLOR_RGB2HLS`, `COLOR_HLS2BGR`, and `COLOR_HLS2RGB`. In this case, as with HSV, if the `src` image format is 8-bit or 16-bit, `cvtColor` first converts it to a floating-point format, scaling the values between 0 and 1. After that, the transformations are computed as follows:

$$Vmax = \max(R,G,B)$$

$$Vmin = \min(R,G,B)$$

$$L = \frac{Vmax + Vmin}{2}$$

$$S = \begin{cases} \dfrac{Vmax - Vmin}{Vmax + Vmin} & if\ L < 0.5 \\ \dfrac{Vmax - Vmin}{2 - (Vmax + Vmin)} & if\ L \geq 0.5 \end{cases}$$

$$H = \begin{cases} 60(G-B)/S & if\ Vmax = R \\ 120 + 60(B-R)/S & if\ Vmax = G \\ 240 + 60(R-G)/S & if\ Vmax = B \end{cases}$$

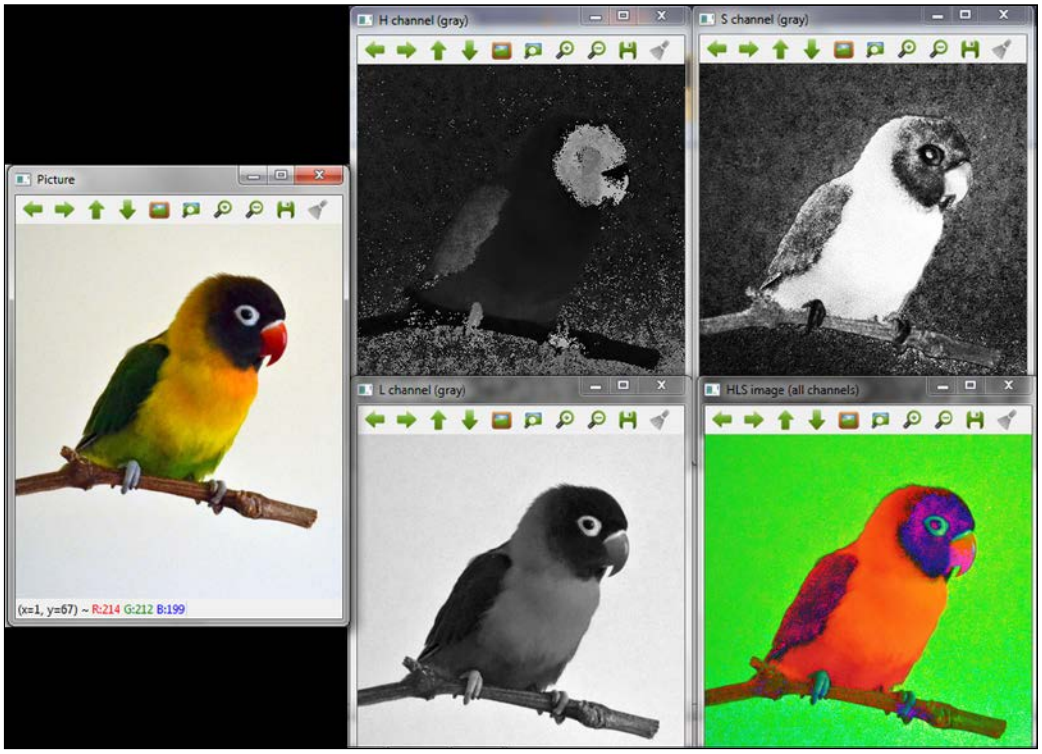
If *H<0*, then *H* = *H* + *360*. Finally, the values are reconverted to the destination data type.

The example code

The following **HLScolor** example shows you how to convert an RGB image to HLS color space, splitting and showing each particular channel in grayscale and the HLS image. The example code is:

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to HSV
    cvtColor(image,image,COLOR_BGR2HLS);

    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("H channel (gray)", WINDOW_AUTOSIZE );
    imshow("H channel (gray)",channels[0]);
    namedWindow("L channel (gray)", WINDOW_AUTOSIZE );
    imshow("L channel (gray)",channels[1]);
    namedWindow("S channel (gray)", WINDOW_AUTOSIZE );
    imshow("S channel (gray)",channels[2]);

    namedWindow("HLS image (all channels)", WINDOW_AUTOSIZE );
    imshow("HLS image (all channels)",image);

    waitKey(0);

    return 0;
}
```

The following figure shows you the output of the code:



The original RGB image, HLS conversion, and channel splitting

CIE L*a*b*

The CIE L*a*b* color space is the second uniform color space standardized by CIE after CIE L*u*v*, which is derived based on the CIE XYZ space and white reference point. Actually, it is the most complete color space specified by CIE and was created to be device-independent, like the CYE XYZ model, and to be used as a reference. It is able to describe the colors visible to the human eye. The three channels represent the lightness of the color (L*), its position between magenta and green (a*), and its position between yellow and blue (b*).

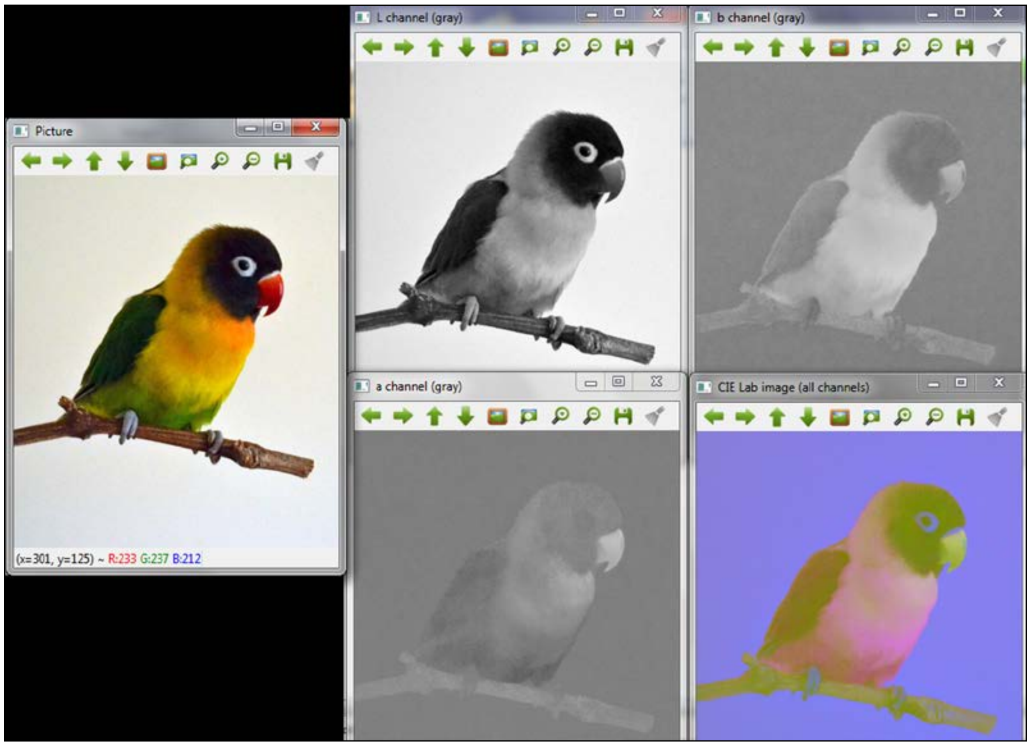
The color space conversion code to convert between RGB and CIE L*a*b* in OpenCV using `cvtColor` is `COLOR_BGR2Lab`, `COLOR_RGB2Lab`, `COLOR_Lab2BGR`, and `COLOR_Lab2RGB`. The procedure used to compute these transformations is explained at `http://docs-hoffmann.de/cielab03022003.pdf`.

The example code

The following **CIElab** example shows you how to convert an RGB image to the CIE L*a*b* color space, splitting and showing each particular channel in grayscale and the CIE L*a*b* image. The example code is:

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to CIE Lab
    cvtColor(image,image,COLOR_BGR2Lab);

    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("L channel (gray)", WINDOW_AUTOSIZE );
    imshow("L channel (gray)",channels[0]);
    namedWindow("a channel (gray)", WINDOW_AUTOSIZE );
    imshow("a channel (gray)",channels[1]);
    namedWindow("b channel (gray)", WINDOW_AUTOSIZE );
    imshow("b channel (gray)",channels[2]);

    namedWindow("CIE Lab image (all channels)", WINDOW_AUTOSIZE );
    imshow("CIE Lab image (all channels)",image);

    waitKey(0);

    return 0;
}
```

The following figure shows you the output of the code:


Picture
L channel (gray)
b channel (gray)
a channel (gray)
CIE Lab image (all channels)
(x=301, y=125) ~ R:233 G:237 B:212

The original RGB image, CIE L*a*b* conversion, and channel splitting

CIE L*u*v*

The CIE L*u*v* color space is the first uniform color space standardized by CIE. It is a simple-to-compute transformation of the CIE XYZ space and white reference point, which attempts perceptual uniformity. Like the CIE L*a*b* color space, it was created to be device-independent. The three channels represent the lightness of the color (L*) and its position between green and red (u*), and the last one represents mostly blue and purple type colors (v*). This color model is useful for additive mixtures of lights due to its linear addition properties.

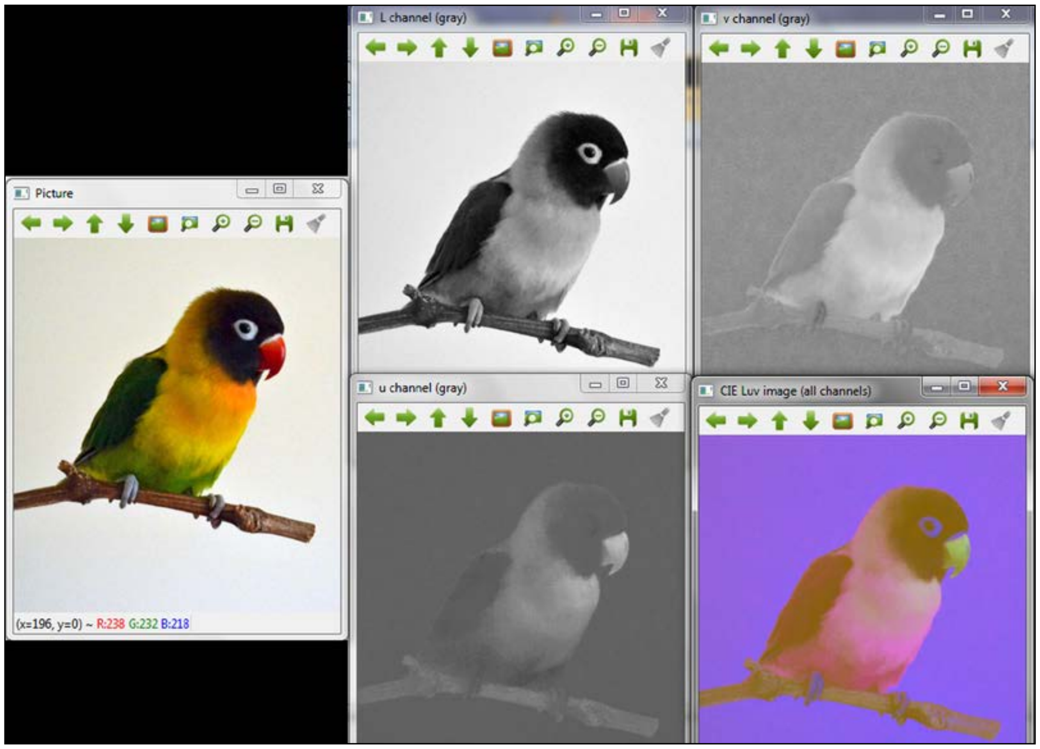
The color space conversion code to convert between RGB and CIE L*u*v* in OpenCV using `cvtColor` is `COLOR_BGR2Luv`, `COLOR_RGB2Luv`, `COLOR_Luv2BGR`, and `COLOR_Luv2RGB`. The procedure used to compute these transformations can be seen at `http://docs.opencv.org/trunk/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor`.

The example code

The following **CIELuvcolor** example shows you how to convert an RGB image to the CIE L*u*v* color space, splitting and showing each particular channel in grayscale and the CIE L*u*v* image. The example code is:

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
    //Load the image
    Mat image = imread("Lovebird.jpg");
    imshow("Picture",image);

    //transform to CIE Luv
    cvtColor(image,image,COLOR_BGR2Luv);

    vector<Mat> channels;

    split( image, channels );

    //show channels in gray scale
    namedWindow("L channel (gray)", WINDOW_AUTOSIZE );
    imshow("L channel (gray)",channels[0]);
    namedWindow("u channel (gray)", WINDOW_AUTOSIZE );
    imshow("u channel (gray)",channels[1]);
    namedWindow("v channel (gray)", WINDOW_AUTOSIZE );
    imshow("v channel (gray)",channels[2]);

    namedWindow("CIE Luv image (all channels)", WINDOW_AUTOSIZE );
    imshow("CIE Luv image (all channels)",image);

    waitKey(0);

    return 0;
}
```

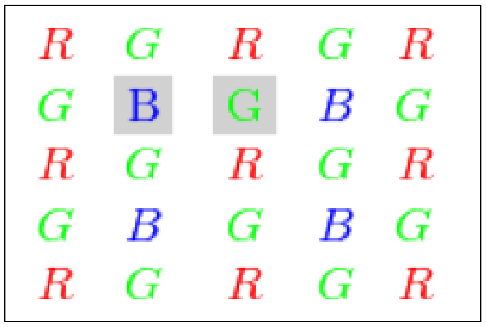
The following figure shows you the output of the code:


L channel (gray)
v channel (gray)
Picture
u channel (gray)
CIE Luv image (all channels)
(x=196, y=0) ~ R:238 G:232 B:218

Original RGB image, CIE L*u*v* conversion, and channel splitting

Bayer

The Bayer pixel-space composition is widely used in digital cameras with only one image sensor. Unlike cameras with three sensors (one per RGB channel, which is able to obtain all the information of a particular component), in one sensor camera, every pixel is covered by a different color filter, so each pixel is only measured in this color. The missing color information is extrapolated from its neighbors using the Bayer method. It allows you to get complete color pictures from a single plane where the pixels are interleaved as follows:



A Bayer pattern example

Note that the Bayer pattern is represented by more G pixels than R and B because the human eye is more sensitive to green frequencies.

There are several modifications of the shown pattern obtained by shifting the pattern by one pixel in any direction. The color space conversion code to convert from Bayer to RGB in OpenCV is defined considering the components of the second and third columns of the second row (`X` and `Y`, respectively) as `COLOR_BayerXY2BGR`. For example, the pattern of the previous picture has a "BG" type, so its conversion code is `COLOR_BayerBG2BGR`.

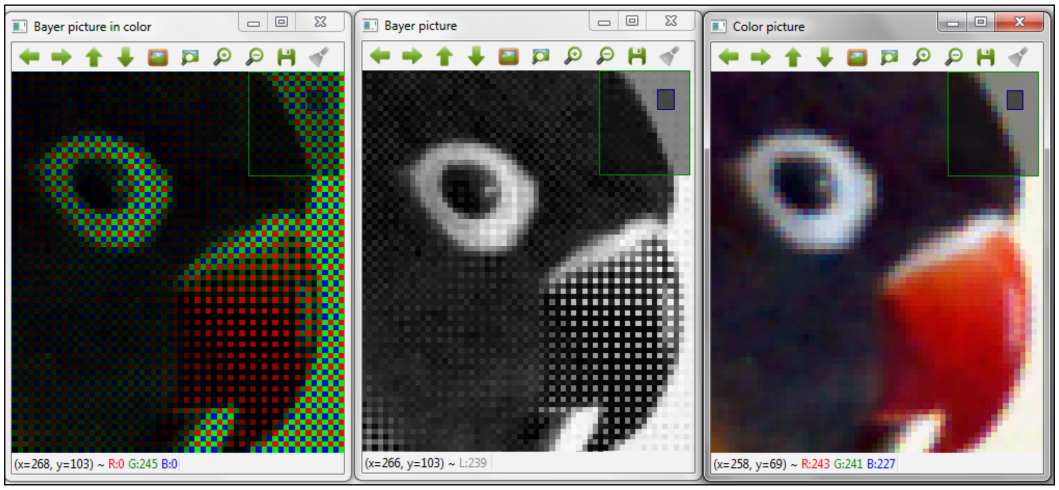
The example code

The following **Bayer** example shows you how to convert a picture defined by an RG Bayer pattern obtained from an image sensor to an RGB image. The example code is:

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

using namespace cv;

int main(int argc, const char** argv)
{
    //Show bayered image in color
    Mat bayer_color = imread("Lovebird_bayer_color.jpg");
    namedWindow("Bayer picture in color", WINDOW_AUTOSIZE );
    imshow("Bayer picture in color",bayer_color);

    //Load bayered image
    Mat bayer = imread("Lovebird_bayer.jpg",CV_8UC3);
```

```
        namedWindow("Bayer picture ", WINDOW_AUTOSIZE );
        imshow("Bayer picture",bayer);

        Mat imageColor;
        cvtColor(bayer, imageColor, COLOR_BayerRG2BGR);

        namedWindow( "Color picture", WINDOW_AUTOSIZE );
        imshow("Color picture",imageColor);

        waitKey(0);
        return 0;
    }
```

The following figure shows you the output of the code:



The Bayer pattern image and RGB conversion

Color-space-based segmentation

Each color space represents an image indicating the numeric value of the specific characteristic measured by each channel on each pixel. Considering these characteristics, it is possible to partition the color space using linear boundaries (for example, planes in three-dimensional spaces and one space per channel), allowing you to classify each pixel according to the partition it lies in, therefore allowing you to select a set of pixels with predefined characteristics. This idea can be used to segment objects of an image we are interested in.

OpenCV provides the `void inRange(InputArray src, InputArray lowerb, InputArray upperb, OutputArray dst)` function to check whether an array of elements lie between the elements of two other arrays. With respect to color-space-based segmentation, this function allows you to obtain the set of pixels of an `src` image, the values of whose channels lie between the `lowerb` lower boundaries and `upperb` upper boundaries, obtaining the `dst` image.

> The `lowerb` and `upperb` boundaries are usually defined as `Scalar(x, y, z)`, where `x`, `y`, and `z` are the numerical values of each channel defined as lower or upper boundaries.

The following examples show you how to detect pixels that can be considered to be skin. It has been observed that skin color differs more in intensity than chrominance, so normally, the luminance component is not considered for skin detection. This fact makes it difficult to detect skin in a picture represented in RGB because of the dependence of this color space on luminance, so HSV and YCrCb color models are used. It is worth noting that for this type of segmentation, it is necessary to know or obtain the values of the boundaries per channel.

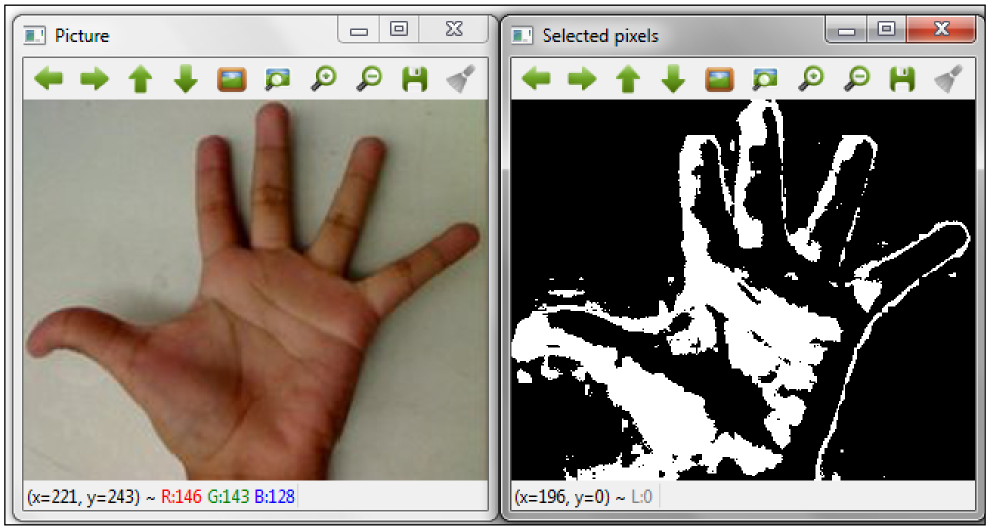
HSV segmentation

As stated previously, HSV is widely used to make a comparison of colors because H is almost independent of light variations, so it is useful in skin detection. In this example, the lower boundaries (0, 10, 60) and the upper boundaries (20, 150, 255) are selected to detect the skin in each pixel. The example code is:

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

using namespace std;
using namespace cv;

int main()
{
    //Load the image
    Mat image = imread("hand.jpg");
    namedWindow("Picture", WINDOW_AUTOSIZE );
    imshow("Picture",image);

    Mat hsv;
```

```
        cvtColor(image, hsv, COLOR_BGR2HSV);

        //select pixels
        Mat bw;
        inRange(hsv, Scalar(0, 10, 60), Scalar(20, 150, 255), bw);

        namedWindow("Selected pixels", WINDOW_AUTOSIZE );
        imshow("Selected pixels", bw);

        waitKey(0);
        return 0;
    }
```

The following figure shows you the output of the code:



Skin detection using the HSV color space

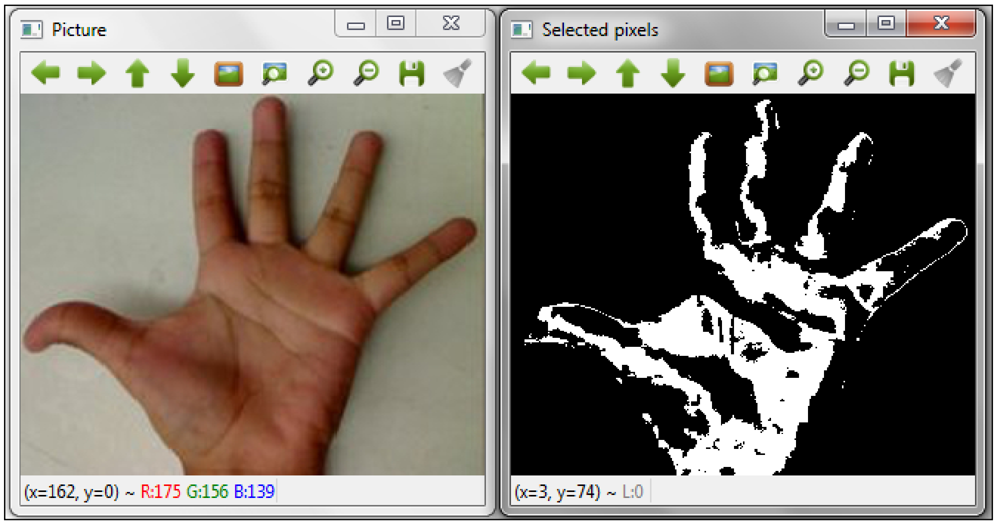
YCrCb segmentation

The YCrCb color space reduces the redundancy of RGB color channels and represents the color with independent components. Considering that the luminance and chrominance components are separated, this space is a good choice for skin detection.

The following example uses the YCrCb color space for skin detection using the lower boundaries (0, 133, 77) and the upper boundaries (255, 173, 177) in each pixel. The example code is:

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

using namespace std;
using namespace cv;

int main()
{
    //Load the image
    Mat image = imread("hand.jpg");
    namedWindow("Picture", WINDOW_AUTOSIZE );
    imshow("Picture",image);

    Mat ycrcb;
    cvtColor(image, ycrcb, COLOR_BGR2HSV);

    //select pixels
    Mat bw;
    inRange(ycrcb, Scalar(0, 133, 77), Scalar(255, 173, 177), bw);

    namedWindow("Selected pixels", WINDOW_AUTOSIZE );
    imshow("Selected pixels", bw);

    waitKey(0);
    return 0;
}
```

The following figure shows you the output of the code:



Skin detection using the YCrCb color space

[For more image-segmentation methods, refer to Chapter 4 of *OpenCV Essentials* by Packt Publishing.]

Color transfer

Another task commonly carried out in image processing is to modify the color of an image, specifically in cases where it is necessary to remove a dominant or undesirable color cast. One of these methods is called color transfer, which carries out a set of color corrections that borrow one source image's color characteristics, and transfer the appearance of the source image to the target image.

The example code

The following **colorTransfer** example shows you how to transfer the color from a source to target image. This method first converts the image color space to CIE L*a*b*. Next, it splits the channels for source and target images. After that, it fits the channel distribution from one image to another using the mean and the standard deviation. Finally, the channels are merged back together and converted to RGB.

> For full theoretical details of the transformation used in the example, refer to *Color Transfer between Images* at `http://www.cs.tau.ac.il/~turkel/imagepapers/ColorTransfer.pdf`.

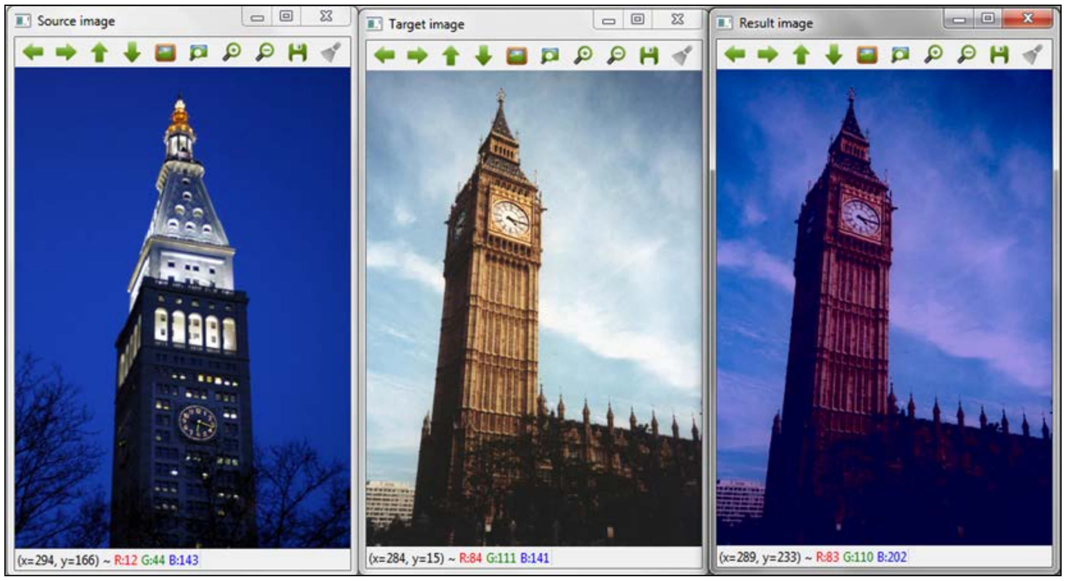
The first part of the code converts the image to the CIE L*a*b* color space, while also changing the type of the image to `CV_32FC1`:

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
    //Load the images
    Mat src = imread("clock_tower.jpg");
    Mat tar = imread("big_ben.jpg");

    //Convert to Lab space and CV_32F1
    Mat src_lab, tar_lab;

    cvtColor(src, src_lab, COLOR_BGR2Lab );
    cvtColor(tar, tar_lab, COLOR_BGR2Lab );

    src_lab.convertTo(src_lab,CV_32FC1);
    tar_lab.convertTo(tar_lab,CV_32FC1);
```

The next part of the code performs the color transfer as stated previously:

```
    //Find mean and std of each channel for each image
    Mat mean_src, mean_tar, stdd_src, stdd_tar;
    meanStdDev(src_lab, mean_src, stdd_src);
    meanStdDev(tar_lab, mean_tar, stdd_src);

    // Split into individual channels
    vector<Mat> src_chan, tar_chan;

    split( src_lab, src_chan );
    split( tar_lab, tar_chan );

    // For each channel calculate the color distribution
    for( int i = 0; i < 3; i++ ) {
        tar_chan[i] -= mean_tar.at<double>(i);
        tar_chan[i] *= (stdd_src.at<double>(i) / stdd_src.
at<double>(i));
        tar_chan[i] += mean_src.at<double>(i);
    }

    //Merge the channels, convert to CV_8UC1 each channel and convert
to BGR
    Mat output;
    merge(tar_chan, output);
    output.convertTo(output,CV_8UC1);
    cvtColor(output, output, COLOR_Lab2BGR );

    //show pictures
    namedWindow("Source image", WINDOW_AUTOSIZE );
    imshow("Source image",src);
    namedWindow("Target image", WINDOW_AUTOSIZE );
    imshow("Target image",tar);
    namedWindow("Result image", WINDOW_AUTOSIZE );
    imshow("Result image",output);

    waitKey(0);

    return 0;
}
```

The following figure shows you the output of the code:


Source image
Target image
Result image
(x=294, y=166) ~ R:12 G:44 B:143
(x=284, y=15) ~ R:84 G:111 B:141
(x=289, y=233) ~ R:83 G:110 B:202

A night appearance color-transfer example

Summary

In this chapter, we provided a deeper view of the color spaces used in OpenCV and showed you how to convert between them using the `cvtColor` function. Furthermore, the possibilities of image processing using different color models and the importance of selecting the correct color space considering the operations we need to make was highlighted. To this end, color-space-based segmentation and color-transfer methods were implemented.

The next chapter will cover image-processing techniques used for video or a sequence of images. We will see how to implement video stabilization, superresolution, and stitching algorithms with OpenCV.

5
Image Processing for Video

This chapter shows you different techniques related to image processing for video. While most classical image processing deals with static images, video-based processing is becoming popular and affordable.

This chapter covers the following topics:

- Video stabilization
- The video superresolution process
- Image stitching

In this chapter, we will work with a video sequence or a live camera directly. The output of image processing may be either a set of modified images or useful high-level information. Most image-processing techniques consider images as a two-dimensional digital signal and apply different techniques to it. In this chapter, a sequence of images from a video or live camera will be used to make or improve a new enhanced sequence with different high-level techniques. Thus, more useful information is obtained, that is, a third time dimension is incorporated.

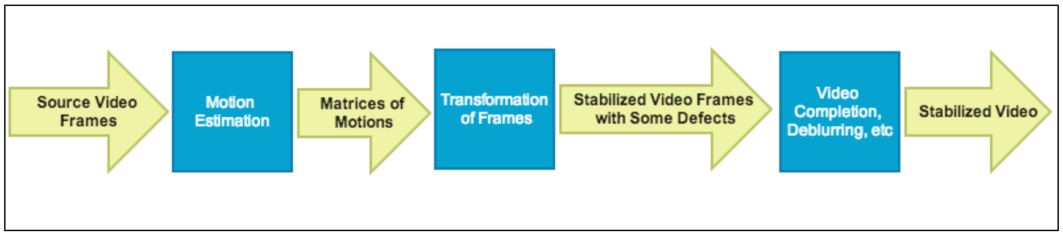
Video stabilization

Video stabilization refers to a family of methods used to reduce the blurring associated with the motion of the camera. In other words, it compensates for any angular movement, equivalent to yaw, pitch, roll, and x and y translations of the camera. The first image stabilizers appeared in the early 60s. These systems were able to slightly compensate for camera shakes and involuntary movements. They were controlled by gyroscopes and accelerometers based on mechanisms that could cancel or reduce unwanted movement by changing the position of a lens. Currently, these methods are widely used in binoculars, video cameras, and telescopes.

There are various methods for image or video stabilization, and this chapter focuses on the most extended families of methods:

- **Mechanical stabilization systems**: These systems use a mechanical system on the camera lens so that when the camera is moved, motion is detected by accelerometers and gyroscopes, and the system generates a movement on the lens. These systems will not be considered here.
- **Digital stabilization systems**: These are normally used in video and they act directly on the image obtained from the camera. In these systems, the surface of the stabilized image is slightly smaller than the source image's surface. When the camera is moved, the captured image is moved to compensate this movement. Although these techniques effectively work to cancel movement by reducing the usable area of movement sensor, resolution and image clarity are sacrificed.

Video-stabilization algorithms usually encompass the following steps:



General steps of Video-Stabilization algorithms

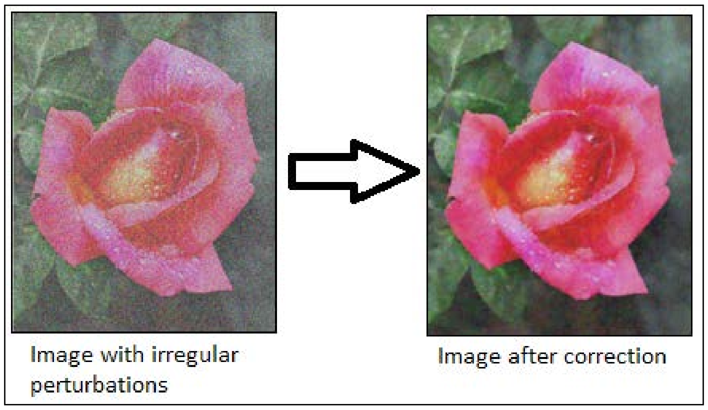
This chapter focuses on the `videostab` module in OpenCV 3.0 Alpha, which contains a set of functions and classes that can be used to solve the video-stabilization problem.

Let's explore the general process in more detail. Video stabilization is achieved by a first estimation of the inter-frame motion between consecutive frames using the RANSAC method. At the end of this step, an array of 3 x 3 matrices is obtained, and each of them describes the motion of the two pairs of consecutive frames. Global motion estimation is very important for this step and it affects the accuracy of the stabilized final sequence.

> You can find more detailed information about the RANSAC method at `http://en.wikipedia.org/wiki/RANSAC`.

The second step generates a new sequence of frames based on the estimated motion. Additional processing is performed, such as smoothing, deblurring, border extrapolation, and so on, to improve the quality of stabilization.

The third step removes the annoying irregular perturbations—refer to the following figure. There are approaches that assume a camera-motion model, which work well when some assumptions can be made about the actual camera motion.



Removing the irregular perturbations

In the OpenCV examples (`[opencv_source_code]/samples/cpp/videostab.cpp`), a video-stabilization program example can be found. For the following videoStabilizer example, the `videoStabilizer.pro` project needs these libraries: `lopencv_core300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300`, and `lopencv_videostab300`.

The following **videoStabilizer** example has been created using the `videostab` module of OpenCV 3.0 Alpha:

```
#include <string>
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/videostab.hpp>

using namespace std;
using namespace cv;
using namespace cv::videostab;

void processing(Ptr<IFrameSource> stabilizedFrames, string
outputPath);

int main(int argc, const char **argv)
{
Ptr<IFrameSource> stabilizedFrames;
    try
    {

// 1-Prepare the input video and check it
        string inputPath;
       string outputPath;
        if (argc > 1)
            inputPath = argv[1];
        else
            inputPath = ".\\cube4.avi";

  if (argc > 2)
            outputPath = argv[2];
        else
            outputPath = ".\\cube4_stabilized.avi";

Ptr<VideoFileSource> source = makePtr<VideoFileSource>(inputPath);
        cout << "frame count (rough): " << source->count() << endl;

// 2-Prepare the motion estimator
// first, prepare the motion the estimation builder, RANSAC L2
        double min_inlier_ratio = 0.1;
Ptr<MotionEstimatorRansacL2> est = makePtr<MotionEstimatorRansacL2>(
MM_AFFINE);
RansacParams ransac = est->ransacParams();
ransac.size = 3;
        ransac.thresh = 5;
        ransac.eps = 0.5;
        est->setRansacParams(ransac);
        est->setMinInlierRatio(min_inlier_ratio);

    // second, create a feature detector
int nkps = 1000;
Ptr<GoodFeaturesToTrackDetector> feature_detector = makePtr<GoodFeatur
esToTrackDetector>(nkps);

// third, create the motion estimator
Ptr<KeypointBasedMotionEstimator> motionEstBuilder = makePtr<KeypointB
asedMotionEstimator>(est);
        motionEstBuilder->setDetector(feature_detector);
Ptr<IOutlierRejector> outlierRejector =
makePtr<NullOutlierRejector>();
        motionEstBuilder->setOutlierRejector(outlierRejector);

// 3-Prepare the stabilizer
StabilizerBase *stabilizer = 0;

// first, prepare the one or two pass stabilizer
        bool isTwoPass = 1;
        int radius_pass = 15;
        if (isTwoPass)
        {
            // with a two pass stabilizer
            bool est_trim = true;

TwoPassStabilizer *twoPassStabilizer = new TwoPassStabilizer();
```

```
            twoPassStabilizer->setEstimateTrimRatio(est_trim);
            twoPassStabilizer->setMotionStabilizer(makePtr<GaussianMot
ionFilter>(radius_pass));

            stabilizer = twoPassStabilizer;
        }
        else
        {
            // with an one pass stabilizer
OnePassStabilizer *onePassStabilizer = new OnePassStabilizer();
            onePassStabilizer->setMotionFilter(makePtr<GaussianMotionF
ilter>(radius_pass));

            stabilizer = onePassStabilizer;
        }

        // second, set up the parameters
        int radius = 15;
        double trim_ratio = 0.1;
        bool incl_constr = false;
stabilizer->setFrameSource(source);
stabilizer->setMotionEstimator(motionEstBuilder);
        stabilizer->setRadius(radius);
        stabilizer->setTrimRatio(trim_ratio);
        stabilizer->setCorrectionForInclusion(incl_constr);
        stabilizer->setBorderMode(BORDER_REPLICATE);


        // cast stabilizer to simple frame source interface to read
stabilized frames
stabilizedFrames.reset(dynamic_cast<IFrameSource*>(stabilizer));

// 4-Processing the stabilized frames. The results are showed and
saved.
processing(stabilizedFrames, outputPath);
    }
    catch (const exception &e)
    {
        cout << "error: " << e.what() << endl;
        stabilizedFrames.release();
        return -1;
    }
    stabilizedFrames.release();
    return 0;
}

void processing(Ptr<IFrameSource> stabilizedFrames, string outputPath)
{
    VideoWriter writer;
    Mat stabilizedFrame;
    int nframes = 0;
double outputFps = 25;

    // for each stabilized frame
while (!(stabilizedFrame = stabilizedFrames->nextFrame()).empty())
    {
        nframes++;

        // init writer (once) and save stabilized frame
        if (!outputPath.empty())
        {
            if (!writer.isOpened())                writer.open(outputP
ath,VideoWriter::fourcc('X','V','I','D'),
outputFps, stabilizedFrame.size());
writer << stabilizedFrame;
        }

imshow("stabilizedFrame", stabilizedFrame);
        char key = static_cast<char>(waitKey(3));
        if (key == 27) { cout << endl; break;}

    }
    cout << "processed frames: " << nframes << endl;
    cout << "finished " << endl;
}
```

This example accepts the name of an input video file as a default video filename (`.\cube4.avi`). The resulting video will be displayed and then saved as `.\cube4_stabilized.avi`. Note how the `videostab.hpp` header is included and the `cv::videostab` namespace is used. The example takes four important steps. The first step prepares the input video path, and this example uses the standard command-line input arguments (`inputPath = argv[1]`) to select the video file. If it does not have an input video file, then it uses the default video file (`.\cube4.avi`).

The second step builds a motion estimator. A robust RANSAC-based global 2D method is created for the motion estimator using a smart pointer (`Ptr<object>`) of OpenCV (`Ptr<MotionEstimatorRansacL2> est = makePtr <MotionEstimatorRansacL2> (MM_AFFINE)`). There are different motion models to stabilize the video:

- MM_TRANSLATION = 0
- MM_TRANSLATION_AND_SCALE = 1
- MM_ROTATIO = 2
- MM_RIGID = 3
- MM_SIMILARITY = 4
- MM_AFFINE = 5
- MM_HOMOGRAPHY = 6
- MM_UNKNOWN = 7

There is a trade-off between accuracy to stabilize the video and computational time. The more basic motion models have worse accuracy and better computational time; however, the more complex models have better accuracy and worse computational time.

The RANSAC object is now created (`RansacParams ransac = est->ransacParams()`) and their parameters are set (`ransac.size`, `ransac.thresh` and `ransac.eps`). A feature detector is also needed to estimate the movement between each consecutive frame that will be used by the stabilizer. This example uses the `GoodFeaturesToTrackDetector` method to detect (`nkps = 1000`) salient features in each frame. Then, it uses the robust RANSAC and feature-detector methods to create the motion estimator using the `Ptr<KeypointBasedMotionEstimator> motionEstBuilder = makePtr<KeypointBasedMotionEstimator>(est)` class and setting the feature detector with `motionEstBuilder->setDetector (feature_detector)`.

RANSAC parameters	
Size	The subset size
Thresh	The maximum error to classify as inliers
Eps	The maximum outliers ratio
Prob	The probability of success

The third step creates a stabilizer that needs the previous motion estimator. You can select (`isTwoPass = 1`) a one- or two-pass stabilizer. If you use the two-pass stabilizer (`TwoPassStabilizer *twoPassStabilizer = new TwoPassStabilizer()`), the results are usually better. However, in this example, this is computationally slower. If you use the other option, one-pass stabilizer (`OnePassStabilizer *onePassStabilizer = new OnePassStabilizer()`), the results are worse but the response is faster. The stabilizer needs to set other options to work correctly, such as the source video file (`stabilizer->setFrameSource(source)`) and the motion estimator (`stabilizer->setMotionEstimator(motionEstBuilder)`). It also needs to cast the stabilizer to simple frame source video to read stabilized frames (`stabilizedFrames.reset(dynamic_cast<IFrameSource*>(stabilizer))`).

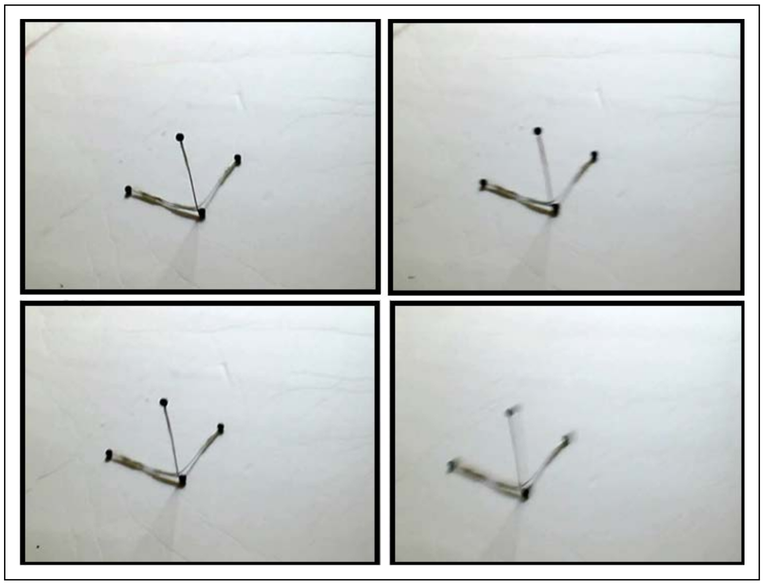
The last step stabilizes the video using the created stabilizer. The `processing(Ptr<IFrameSource> stabilizedFrames)` function is created to process and stabilize each frame. This function needs to introduce a path to save the resulting video (`string outputPath = ".//stabilizedVideo.avi"`) and set the playback speed (`double outputFps = 25`). Afterwards, this function calculates each stabilized frame until there are no more frames (`stabilizedFrame = stabilizedFrames-> nextFrame().empty()`). Internally, the stabilizer first estimates the motion of every frame. This function creates a video writer (`writer.open(outputPath,VideoWriter::fourcc('X','V','I','D'), outputFps, stabilizedFrame.size())`) to store each frame in the **XVID** format. Finally, it saves and shows each stabilized frame until the user presses the *Esc* key.

To demonstrate how to stabilize a video with OpenCV, the previous **videoStabilizer** example is used. The example is executed from the command line as follows:

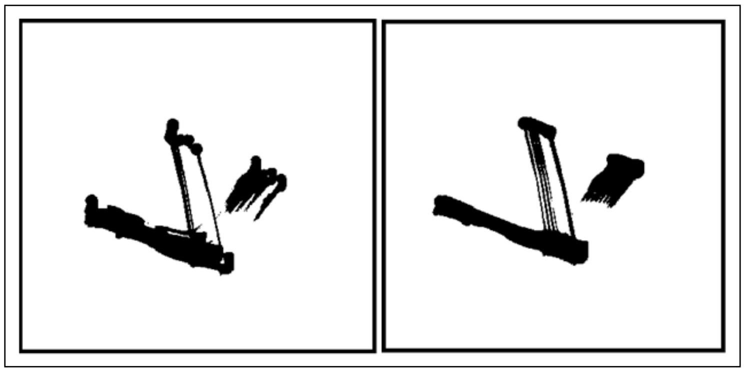
```
<bin_dir>\videoStabilizer.exe .\cube4.avi .\cube4_stabilized.avi
```

[This `cube4.avi` video can be found in the OpenCV samples folder. It also has a great deal of camera movement, which is perfect for this example.]

To show the stabilization results, first, see four frames of `cube4.avi` in the following figure. The figure following these frames shows the first 10 frames of `cube4.avi` and `cube4_stabilized.avi` superimposed without (left-hand side of the figure) and with (right-hand side of the figure) stabilization.



The four consecutive frames of `cube4.avi` video that are camera movements



10 superimposed frames of `cube4.avi` and `cube4_stabilizated` videos without and with stabilization

Looking at the preceding figure, on the right-hand side, you can see that the vibrations produced by the camera movement have been reduced due to the stabilization.

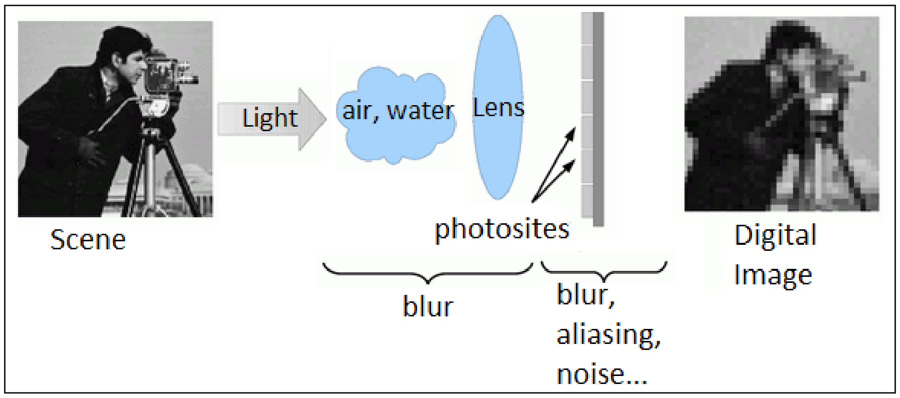
Superresolution

Superresolution refers to the techniques or algorithms designed to increase the spatial resolution of an image or video, usually from a sequence of images of lower resolution. It differs from the traditional techniques of image scaling, which use a single image to increase the resolution, keeping the sharp edges. In contrast, superresolution merges information from multiple images taken from the same scene to represent details that were not initially captured in the original images.

The process of capturing an image or video from a real-life scene requires the following steps:

- **Sampling**: This is the transformation of the continuous system from the scene of an ideal discrete system without aliasing
- **Geometric transformation**: This refers to applying a set of transformations, such as translation or rotation, due to the camera position and lens system to infer, ideally, details of the scene that arrive at each sensor
- **Blur**: This happens due to the lens system or the existing motion in the scene during the integration time
- **Subsampling**: With this, the sensor only integrates the number of pixels at its disposal (photosites)

You can see this process of image capturing in the following figure:



The process of capturing an image from a real scene

During this capture process, the details of the scene are integrated by different sensors so that each pixel in each capture includes different information. Therefore, superresolution is based on trying to find the relationship between different captures that have obtained different details of the scene in order to create a new image with more information. Superresolution is, therefore, used to regenerate a discretized scene with a higher resolution.

Superresolution can be obtained by various techniques, ranging from the most intuitive in the spatial domain to techniques based on analyzing the frequency spectrum. Techniques are basically divided into optical (using lenses, zoom, and so on) or image-processing-based techniques. This chapter focuses on image-processing-based superresolution. These methods use other parts of the lower-resolution images, or other unrelated images, to infer what the high-resolution image should look like. These algorithms can be also divided into the frequency or spatial domain. Originally, superresolution methods only worked well on grayscale images, but new methods have been developed to adapt them to color images.

In general, superresolution is computationally demanding, both spatially and temporally, because the size of low-resolution and high-resolution images is high and hundreds of seconds may be needed to generate an image. To try to reduce the computational time, preconditioners are used currently for optimizers that are responsible for minimizing these functions. Another alternative is to use **GPU processing** to improve the computational time because the superresolution process is inherently parallelizable.

This chapter focuses on the `superres` module in OpenCV 3.0 Alpha, which contains a set of functions and classes that can be used to solve the problem of resolution enhancement. The module implements a number of methods based on image-processing superresolution. This chapter specifically focuses on the **Bilateral TV-L1 (BTVL1) superresolution method** implemented. A major difficulty of the superresolution process is to estimate the warping function to build the superresolution image. The Bilateral TV-L1 uses optical flow to estimate the warping function.

> You can find more detailed information about the Bilateral TV-L method at `http://www.ipol.im/pub/art/2013/26/` and optical flow at `http://en.wikipedia.org/wiki/Optical_flow`.

In the OpenCV examples (`[opencv_source_code]/samples/gpu/super_resolution.cpp`), a basic example of superresolution can be found.

> You can also download this example from the OpenCV GitHub repository at `https://github.com/Itseez/opencv/blob/master/samples/gpu/super_resolution.cpp`.

For the following **superresolution** example project, the `superresolution.pro` project file shall include these libraries: `lopencv_core300`, `lopencv_imgproc300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300`, and `lopencv_superres300` to work correctly:

```
#include <iostream>
#include <iomanip>
#include <string>

#include <opencv2/core.hpp>
#include <opencv2/core/utility.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/superres.hpp>
#include <opencv2/superres/optical_flow.hpp>
#include <opencv2/opencv_modules.hpp>

using namespace std;
using namespace cv;
using namespace cv::superres;

static Ptr<DenseOpticalFlowExt> createOptFlow(string name);

int main(int argc, char *argv[])
{
// 1-Initialize the initial parameters

// Input and output video
string inputVideoName;
    string outputVideoName;
    if (argc > 1)
        inputVideoName = argv[1];
    else
        inputVideoName = ".\\tree.avi";

    if (argc > 2)
        outputVideoName = argv[2];
    else
        outputVideoName = ".\\tree_superresolution.avi";

const int scale = 4;// Scale factor

const int iterations = 180;// Iterations count

const int temporalAreaRadius =4;// Radius of the temporal search area

string optFlow = "farneback";// Optical flow algorithm
       // optFlow = "farneback";
       // optFlow = "tvl1";
       // optFlow = "brox";
       // optFlow = "pyrlk";

double outputFps = 25.0;// Playback speed output

// 2- Create an optical flow method
Ptr<DenseOpticalFlowExt> optical_flow = createOptFlow(optFlow);

    if (optical_flow.empty()) return -1;

// 3- Create the superresolution method and set its parameters
Ptr<SuperResolution> superRes;
    superRes = createSuperResolution_BTVL1();

superRes->set("opticalFlow", optical_flow);
    superRes->set("scale", scale);
    superRes->set("iterations", iterations);
    superRes->set("temporalAreaRadius", temporalAreaRadius);

Ptr<FrameSource> frameSource;
    frameSource = createFrameSource_Video(inputVideoName);

superRes->setInput(frameSource);

    // Not use the first frame
    Mat frame;
    frameSource->nextFrame(frame);

// 4- Processing the input video with the superresolution
    // Show the initial options
    cout << "Input                  : " << inputVideoName << " " <<
frame.size() << endl;
    cout << "Output                 : " << outputVideoName << endl;
    cout << "Playback speed output  : " << outputFps << endl;
    cout << "Scale factor           : " << scale << endl;
    cout << "Iterations             : " << iterations << endl;
    cout << "Temporal radius        : " << temporalAreaRadius << endl;
    cout << "Optical Flow           : " << optFlow << endl;
    cout << endl;

    VideoWriter writer;
    double start_time,finish_time;

    for (int i = 0;; ++i)
    {
        cout << '[' << setw(3) << i << "] : ";
        Mat result;

        // Calculate the processing time
        start_time = getTickCount();
superRes->nextFrame(result);
        finish_time = getTickCount();
        cout << (finish_time - start_time)/getTickFrequency() << "
secs, Size: " << result.size() << endl;

        if (result.empty()) break;

        // Show the result
imshow("Super Resolution", result);

        if (waitKey(1000) > 0) break;

        // Save the result on output file
        if (!outputVideoName.empty())
        {
```

```
if (!writer.isOpened())
                writer.open(outputVideoName, VideoWriter::fourcc('X',
'V', 'I', 'D'), outputFps, result.size());
            writer << result;
        }
    }
    writer.release();
    return 0;
}

static Ptr<DenseOpticalFlowExt> createOptFlow(string name)
{
    if (name == "farneback")
return createOptFlow_Farneback();

    else if (name == "tvl1")
return createOptFlow_DualTVL1();

    else if (name == "brox")
        return createOptFlow_Brox_CUDA();

    else if (name == "pyrlk")
        return createOptFlow_PyrLK_CUDA();

    else
        cerr << "Incorrect Optical Flow algorithm - " << name << endl;

    return Ptr<DenseOpticalFlowExt>();
}
```

This example creates a program (**superresolution**) to obtain videos with superresolution. It takes the path of an input video or uses a default video path (`.\tree.avi`). The resulting video is displayed and saved as `.\tree_superresolution.avi`. In the first place, the `superres.hpp` and `superres/optical_flow.hpp` headers are included and the `cv::superres` namespace is used. The example follows four important steps.

The first step sets the initial parameters. It is the input video path that uses the standard input (`inputVideoName = argv[1]`) to select the video file, and if it does not have an input video file, then it uses a default video file. The output video path also uses the input standard (`outputVideoName = argv[2]`) to select the output video file, and if it does not have an output video file, then it uses the default output video file (`.\tree_superresolution`) and the output playback speed is also set (`double outputFps = 25.0`). Other important parameters of the `superresolution` method are the scale factor (`const int scale = 4`), the iteration `count(const int iterations = 100)`, the radius of the temporal search area (`const int temporalAreaRadius = 4`), and the optical-flow algorithm (`string optFlow = "farneback"`).

The second step creates an optical-flow method to detect salient features and track them for each video frame. A new method (`static Ptr<DenseOpticalFlowExt> createOptFlow(string name)`) has been created to select between the different optical-flow methods. You can select between **farneback**, **tvl1**, **brox**, and **pyrlk** optical-flow methods. A new method (`static Ptr<DenseOpticalFlowExt> createOptFlow(string name)`) is written to create an optical;-flow method to track features. The two most important methods are Farneback (`createOptFlow_Farneback()`) and TV-L1 (`createOptFlow_DualTVL1()`). The first method is based on Gunner Farneback's algorithm that computes the optical flow for all points in the frame. The second method calculates the optical flow between two image frames that are based on dual formulation on the TV energy and employs an efficient point-wise thresholding step. This second method is computationally more efficient.

Comparison between the different optical flow methods		
Method	**Complexity**	**Parallelizable**
Farneback	Quadratic	No
TV-L1	Lineal	Yes
Brox	Lineal	Yes
PyrLK	Lineal	No

You can also learn more about the Farneback optical-flow method used at `http://www.diva-portal.org/smash/get/diva2:273847/FULLTEXT01.pdf`.

The third step creates and sets the `superresolution` method. An instance of this method is created (`Ptr<SuperResolution> superRes`), which uses the **Bilateral TV-L1** algorithm (`superRes = createSuperResolution_BTVL1()`). This method has the following parameters for the algorithm:

- `scale`: This is the scale factor
- `iterations`: This is the iteration count
- `tau`: This is an asymptotic value of the steepest descent method
- `lambda`: This is the weight parameter to balance the data term and smoothness term
- `alpha`: This is a parameter of spatial distribution in Bilateral-TV
- `btvKernelSize`: This is the kernel size of the Bilateral-TV filter
- `blurKernelSize`: This is the Gaussian blur kernel size
- `blurSigma`: This is the Gaussian blur sigma
- `temporalAreaRadius`: This is the radius of the temporal search area
- `opticalFlow`: This is a dense optical-flow algorithm

These parameters are set as follows:

```
superRes->set("parameter", value);
```

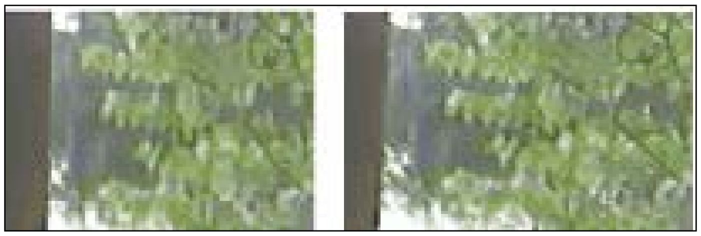
Only the following parameters are set; the other parameters use their default values:

```
superRes->set("opticalFlow", optical_flow);
superRes->set("scale", scale);
superRes->set("iterations", iterations);
superRes->set("temporalAreaRadius", temporalAreaRadius);
```

Afterwards, the input video frame is selected (`superRes->setInput(frameSource)`).

The last step processes the input video to compute the superresolution. For each video frame, the superresolution is calculated (`superRes->nextFrame(result)`); this calculation is computationally very slow, thus the processing time is estimated to show progress. Finally, each result frame is shown (`imshow("Super Resolution", result)`) and saved (`writer << result`).

To show the superresolution results, a small part of the first frame of the `tree.avi` and `tree_superresolution.avi` videos are compared with and without superresolution:

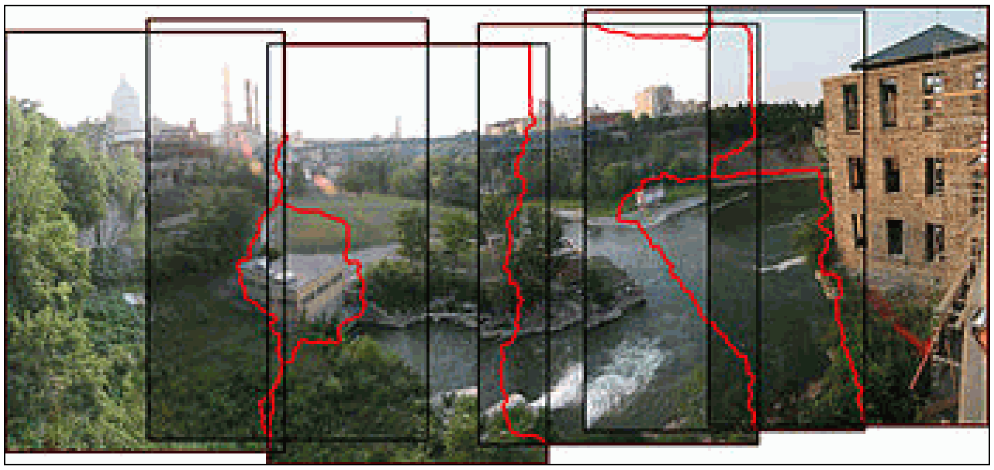


Part of the first frame of `tree.avi` and `tree_superresolution.avi` videos without and with the superresolution process

In the right-hand-side section of the preceding figure, you can observe more details in the leaves and branches of the tree due to the superresolution process.

Stitching

Image **stitching**, or photo stitching, can discover the correspondence relationship between images with some degree of overlap. This process combines a set of images with overlapping fields of view to produce a panorama or higher-resolution image. Most of the techniques for image stitching need nearly exact overlaps between the images to produce seamless results. Some digital cameras can internally stitch a set of images to build a panorama image. An example is shown in the following figure:



A panorama image created with stitching

The preceding image example and more information about image stitching can be found at `http://en.wikipedia.org/wiki/Image_stitching`.

Stitching can normally be divided into three important steps:

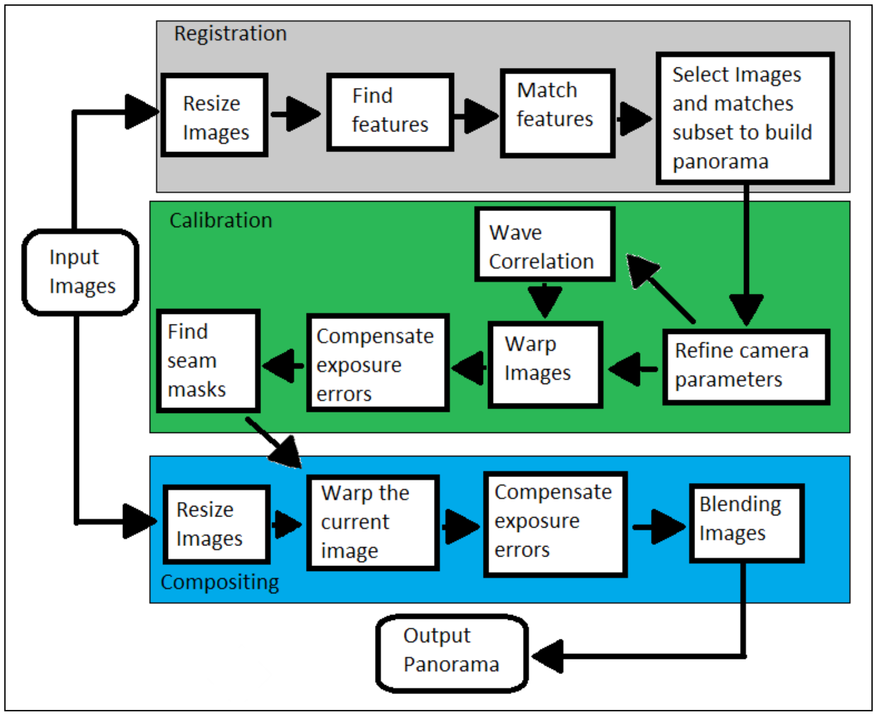
- **Registration** (images) implies matching features in a set of images to search for a displacement that minimizes the sum of absolute values in the differences between overlapping pixels. Direct-alignment methods could be used to get better results. The user could also add a rough model of the panorama to help the feature matching stage, in which case, the results are typically more accurate and computationally faster.
- **Calibration** (images) focuses on minimizing differences between an ideal model and the camera-lens system: different camera positions and optical defects such as distortions, exposure, chromatic aberrations, and so on.
- **Compositing** (images) uses the results of the previous step, calibration, combined with the remapping of the images to an output projection. Colors are also adjusted between images to compensate for exposure differences. Images are blended together and seam-line adjustment is done to minimize the visibility of seams between images.

When there are image segments that have been taken from the same point in space, stitching can be performed using one of various map projections. The most important map projections are shown as follows:

- **Rectilinear projection**: Here, the stitching image is viewed on a two-dimensional plane intersecting the panorama sphere in a single point. Lines that are straight in reality are shown similar regardless of their direction on the image. When there are wide views (around 120 degrees), images are distorted near the borders.
- **Cylindrical projection**: Here, the stitching image shows a 360-degree horizontal field of view and a limited vertical field of view. This projection is meant to be viewed as though the image is wrapped into a cylinder and viewed from within. When viewed on a 2D plane, horizontal lines appear curved, while vertical lines remain straight.
- **Spherical projection**: Here, the stitching image shows a 360-degree horizontal field of view and 180-degree vertical field of view, that is, the whole sphere. Panorama images with this projection are meant to be viewed as though the image is wrapped into a sphere and viewed from within. When viewed on a 2D plane, horizontal lines appear curved as in a cylindrical projection, while vertical lines remain vertical.

- **Stereographic projection or fisheye projection**: This can be used to form a little planet panorama by pointing the virtual camera straight down and setting the field of view large enough to show the whole ground and some of the areas above it; pointing the virtual camera upwards creates a tunnel effect.
- **Panini projection**: This has specialized projections that may have more aesthetically pleasing advantages over normal cartography projections. This projection combines different projections in the same image to fine-tune the final look of the output panorama image.

This chapter focuses on the `stitching` module and the `detail` submodule in OpenCV 3.0 Alpha, which contains a set of functions and classes that implement a **stitcher**. Using these modules, it is possible to configure or skip some steps. The implemented stitching example has the following general diagram:



In the OpenCV examples, there are two basic examples of stitching, which can be found at (`[opencv_source_code]/samples/cpp/stitching.cpp]`) and (`[opencv_source_code]/samples/cpp/stitching_detailed.cpp]`).

For the following, more advanced **stitchingAdvanced** example, the `stitchingAdvanced.pro` project file must include the following libraries to work correctly: `lopencv_core300`, `lopencv_imgproc300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300`, `lopencv_imgcodecs300`, and `lopencv_stitching300`:

```
#include <iostream>
#include <string>
#include <opencv2/opencv_modules.hpp>
#include <opencv2/core/utility.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/features2d.hpp>
#include <opencv2/stitching/detail/blenders.hpp>
#include <opencv2/stitching/detail/camera.hpp>
#include <opencv2/stitching/detail/exposure_compensate.hpp>
#include <opencv2/stitching/detail/matchers.hpp>
#include <opencv2/stitching/detail/motion_estimators.hpp>
#include <opencv2/stitching/detail/seam_finders.hpp>
#include <opencv2/stitching/detail/util.hpp>
#include <opencv2/stitching/detail/warpers.hpp>
#include <opencv2/stitching/warpers.hpp>

using namespace std;
using namespace cv;
using namespace cv::detail;

int main(int argc, char* argv[])
{
    // Default parameters
    vector<String> img_names;
double scale = 1;
string features_type = "orb";//"surf" or "orb" features type
float match_conf = 0.3f;
float conf_thresh = 1.f;
string adjuster_method = "ray";//"reproj" or "ray" adjuster method
bool do_wave_correct = true;
WaveCorrectKind wave_correct_type = WAVE_CORRECT_HORIZ;
string warp_type = "spherical";
int expos_comp_type = ExposureCompensator::GAIN_BLOCKS;
string seam_find_type = "gc_color";
```

```
float blend_strength = 5;
int blend_type = Blender::MULTI_BAND;
string result_name = "panorama_result.jpg";

    double start_time = getTickCount();


// 1-Input images
    if(argc > 1)
    {
        for(int i=1; i < argc; i++)
img_names.push_back(argv[i]);
    }
    else
    {
img_names.push_back("./panorama_image1.jpg");
        img_names.push_back("./panorama_image2.jpg");
    }
    // Check if have enough images
    int num_images = static_cast<int>(img_names.size());
if (num_images < 2) {cout << "Need more images" << endl; return -1; }

// 2- Resize images and find features steps
    cout << "Finding features..." << endl;
    double t = getTickCount();

    Ptr<FeaturesFinder> finder;
    if (features_type == "surf")
finder = makePtr<SurfFeaturesFinder>();

    else if (features_type == "orb")
finder = makePtr<OrbFeaturesFinder>();

    else {cout << "Unknown 2D features type: '" << features_type <<
endl; return -1; }

    Mat full_img, img;
    vector<ImageFeatures> features(num_images);
    vector<Mat> images(num_images);
    vector<Size> full_img_sizes(num_images);

    for (int i = 0; i < num_images; ++i)
    {
        full_img = imread(img_names[i]);
        full_img_sizes[i] = full_img.size();

        if (full_img.empty()) {cout << "Can't open image " << img_
names[i] << endl; return -1; }

resize(full_img, img, Size(), scale, scale);
        images[i] = img.clone();

(*finder)(img, features[i]);
        features[i].img_idx = i;
        cout << "Features in image #" << i+1 << " are : " <<
features[i].keypoints.size() << endl;
    }
    finder->collectGarbage();
    full_img.release();
    img.release();
    cout << "Finding features, time: " << ((getTickCount() - t) /
getTickFrequency()) << " sec" << endl;

// 3- Match features
    cout << "Pairwise matching" << endl;
    t = getTickCount();
    vector<MatchesInfo> pairwise_matches;
BestOf2NearestMatcher matcher(false, match_conf);
    matcher(features, pairwise_matches);
    matcher.collectGarbage();
    cout << "Pairwise matching, time: " << ((getTickCount() - t) /
getTickFrequency()) << " sec" << endl;

// 4- Select images and matches subset to build panorama
vector<int> indices = leaveBiggestComponent(features, pairwise_
matches, conf_thresh);
    vector<Mat> img_subset;
    vector<String> img_names_subset;
    vector<Size> full_img_sizes_subset;
    for (size_t i = 0; i < indices.size(); ++i)
    {
        img_names_subset.push_back(img_names[indices[i]]);
        img_subset.push_back(images[indices[i]]);
        full_img_sizes_subset.push_back(full_img_sizes[indices[i]]);
    }
    images = img_subset;
    img_names = img_names_subset;
    full_img_sizes = full_img_sizes_subset;


    // Estimate camera parameters rough
    HomographyBasedEstimator estimator;
    vector<CameraParams> cameras;
    if (!estimator(features, pairwise_matches, cameras)){cout <<
"Homography estimation failed." << endl; return -1; }

    for (size_t i = 0; i < cameras.size(); ++i)
    {
        Mat R;
        cameras[i].R.convertTo(R, CV_32F);
        cameras[i].R = R;
        cout << "Initial intrinsic #" << indices[i]+1 << ":\n" <<
cameras[i].K() << endl;
    }

// 5- Refine camera parameters globally
Ptr<BundleAdjusterBase> adjuster;
    if (adjuster_method == "reproj")
        // "reproj" method
adjuster = makePtr<BundleAdjusterReproj>();
    else // "ray" method
adjuster = makePtr<BundleAdjusterRay>();

    adjuster->setConfThresh(conf_thresh);
    if (!(*adjuster)(features, pairwise_matches, cameras)) {cout <<
"Camera parameters adjusting failed." << endl; return -1; }

    // Find median focal length
    vector<double> focals;
```

```
        for (size_t i = 0; i < cameras.size(); ++i)
        {
            cout << "Camera #" << indices[i]+1 << ":\n" << cameras[i].K()
    << endl;
            focals.push_back(cameras[i].focal);
        }
        sort(focals.begin(), focals.end());
        float warped_image_scale;
        if (focals.size() % 2 == 1)
            warped_image_scale = static_cast<float>(focals[focals.size() /
    2]);
        else
            warped_image_scale = static_cast<float>(focals[focals.size() /
    2 - 1] + focals[focals.size() / 2]) * 0.5f;


    // 6- Wave correlation (optional)
        if (do_wave_correct)
        {
            vector<Mat> rmats;
            for (size_t i = 0; i < cameras.size(); ++i)
                rmats.push_back(cameras[i].R.clone());

    waveCorrect(rmats, wave_correct_type);
            for (size_t i = 0; i < cameras.size(); ++i)
                cameras[i].R = rmats[i];
        }

    // 7- Warp images
        cout << "Warping images (auxiliary)... " << endl;
        t = getTickCount();
        vector<Point> corners(num_images);
        vector<UMat> masks_warped(num_images);
        vector<UMat> images_warped(num_images);
        vector<Size> sizes(num_images);
        vector<UMat> masks(num_images);

        // Prepare images masks
        for (int i = 0; i < num_images; ++i)
        {
```

```
        masks[i].create(images[i].size(), CV_8U);
        masks[i].setTo(Scalar::all(255));
    }

    // Map projections
    Ptr<WarperCreator> warper_creator;
    if (warp_type == "rectilinear")
warper_creator = makePtr<cv::CompressedRectilinearWarper>(2.0f, 1.0f);

    else if (warp_type == "cylindrical")
warper_creator = makePtr<cv::CylindricalWarper>();

    else if (warp_type == "spherical")
warper_creator = makePtr<cv::SphericalWarper>();

    else if (warp_type == "stereographic")
warper_creator = makePtr<cv::StereographicWarper>();

    else if (warp_type == "panini")
warper_creator = makePtr<cv::PaniniWarper>(2.0f, 1.0f);

    if (!warper_creator){ cout << "Can't create the following warper
'" << warp_type << endl; return 1; }

Ptr<RotationWarper> warper = warper_creator->create(static_
cast<float>(warped_image_scale * scale));

    for (int i = 0; i < num_images; ++i)
    {
        Mat_<float> K;
        cameras[i].K().convertTo(K, CV_32F);
        float swa = (float)scale;
        K(0,0) *= swa; K(0,2) *= swa;
        K(1,1) *= swa; K(1,2) *= swa;

        corners[i] = warper->warp(images[i], K, cameras[i].R, INTER_
LINEAR, BORDER_REFLECT, images_warped[i]);
```

```
        sizes[i] = images_warped[i].size();

warper->warp(masks[i], K, cameras[i].R, INTER_NEAREST, BORDER_
CONSTANT, masks_warped[i]);
    }

    vector<UMat> images_warped_f(num_images);
    for (int i = 0; i < num_images; ++i)
        images_warped[i].convertTo(images_warped_f[i], CV_32F);

    cout << "Warping images, time: " << ((getTickCount() - t) /
getTickFrequency()) << " sec" << endl;

// 8- Compensate exposure errors
Ptr<ExposureCompensator> compensator = ExposureCompensator::createDefa
ult(expos_comp_type);
    compensator->feed(corners, images_warped, masks_warped);

// 9- Find seam masks
    Ptr<SeamFinder> seam_finder;
    if (seam_find_type == "no")
seam_finder = makePtr<NoSeamFinder>();

    else if (seam_find_type == "voronoi")
seam_finder = makePtr<VoronoiSeamFinder>();

    else if (seam_find_type == "gc_color")
        seam_finder = makePtr<GraphCutSeamFinder>(GraphCutSeamFinderBa
se::COST_COLOR);

    else if (seam_find_type == "gc_colorgrad")
        seam_finder = makePtr<GraphCutSeamFinder>(GraphCutSeamFinderBa
se::COST_COLOR_GRAD);

    else if (seam_find_type == "dp_color")
seam_finder = makePtr<DpSeamFinder>(DpSeamFinder::COLOR);

    else if (seam_find_type == "dp_colorgrad")
seam_finder = makePtr<DpSeamFinder>(DpSeamFinder::COLOR_GRAD);

    if (!seam_finder){cout << "Can't create the following seam finder
'" << seam_find_type << endl; return 1; }
```

```
    seam_finder->find(images_warped_f, corners, masks_warped);

    // Release unused memory
    images.clear();
    images_warped.clear();
    images_warped_f.clear();
    masks.clear();

// 10- Create a blender
Ptr<Blender> blender = Blender::createDefault(blend_type, false);
    Size dst_sz = resultRoi(corners, sizes).size();
    float blend_width = sqrt(static_cast<float>(dst_sz.area())) *
blend_strength / 100.f;
    if (blend_width < 1.f)
blender = Blender::createDefault(Blender::NO, false);

    else if (blend_type == Blender::MULTI_BAND)
    {
MultiBandBlender* mb = dynamic_cast<MultiBandBlender*>(blender.get());
        mb->setNumBands(static_cast<int>(ceil(log(blend_width)/
log(2.)) - 1.));
        cout << "Multi-band blender, number of bands: " << mb-
>numBands() << endl;
    }
    else if (blend_type == Blender::FEATHER)
    {
FeatherBlender* fb = dynamic_cast<FeatherBlender*>(blender.get());
        fb->setSharpness(1.f/blend_width);
        cout << "Feather blender, sharpness: " << fb->sharpness() <<
endl;
    }
blender->prepare(corners, sizes);

    // 11- Compositing step
    cout << "Compositing..." << endl;
    t = getTickCount();
    Mat img_warped, img_warped_s;
    Mat dilated_mask, seam_mask, mask, mask_warped;

    for (int img_idx = 0; img_idx < num_images; ++img_idx)
    {
        cout << "Compositing image #" << indices[img_idx]+1  << endl;

        // 11.1- Read image and resize it if necessary
full_img = imread(img_names[img_idx]);

        if (abs(scale - 1) > 1e-1)
resize(full_img, img, Size(), scale, scale);
        else
            img = full_img;

        full_img.release();
        Size img_size = img.size();

        Mat K;
        cameras[img_idx].K().convertTo(K, CV_32F);

        // 11.2- Warp the current image
warper->warp(img, K, cameras[img_idx].R, INTER_LINEAR, BORDER_REFLECT,
img_warped);

        // Warp the current image mask
        mask.create(img_size, CV_8U);
        mask.setTo(Scalar::all(255));
        warper->warp(mask, K, cameras[img_idx].R, INTER_NEAREST,
BORDER_CONSTANT, mask_warped);

        // 11.3- Compensate exposure error step
compensator->apply(img_idx, corners[img_idx], img_warped, mask_
warped);

        img_warped.convertTo(img_warped_s, CV_16S);
        img_warped.release();
        img.release();
        mask.release();

        dilate(masks_warped[img_idx], dilated_mask, Mat());
        resize(dilated_mask, seam_mask, mask_warped.size());
        mask_warped = seam_mask & mask_warped;

        // 11.4- Blending images step
blender->feed(img_warped_s, mask_warped, corners[img_idx]);
    }
    Mat result, result_mask;
    blender->blend(result, result_mask);

    cout << "Compositing, time: " << ((getTickCount() - t) / 
getTickFrequency()) << " sec" << endl;

    imwrite(result_name, result);

    cout << "Finished, total time: " << ((getTickCount() - start_time) 
/ getTickFrequency()) << " sec" << endl;
    return 0;
}
```

This example creates a program to stitch images using OpenCV steps. It takes an input path to select the different input images or uses default input images (`.\panorama_image1.jpg` and `panorama_image2.jpg`), which are shown later. Finally, the resulting image is shown and saved as `.\panorama_result.jpg`. In the first place, the `stitching.hpp` and `detail` headers are included and the `cv::detail` namespace is used. The more important parameters are also set, and you can configure the stitching process with these parameters. If you need to use a custom configuration, it is very useful to understand the general diagram of the stitching process (the previous figure). This advanced example has 11 important steps. The first step reads and checks the input images. This example needs two or more images to work.

The second step resizes the input images using the `double scale = 1` parameter and finds the features on each image; you can select between the **Surf** (`finder = makePtr<SurfFeaturesFinder>()`) or **Orb** (`finder = makePtr<OrbFeaturesFinder>()`) feature finders using the `string features_type = "orb"` parameter. Afterwards, this step resizes the input images (`resize(full_img, img, Size(), scale, scale)`) and finds the features (`(*finder)(img, features[i])`).

For more information about SURF and ORB descriptors, refer to Chapter 5 of *OpenCV Essentials* by Packt Publishing.

The third step matches the features that have been found previously. A matcher is created (`BestOf2NearestMatcher matcher(false, match_conf)`) with the `float match_conf = 0.3f` parameter.

The fourth step selects images and matches subsets to build the panorama. Then, the best features are selected and matched using the `vector<int> indices = leaveBiggestComponent(features, pairwise_matches, conf_thresh)` function. With these features, a new subset is created to be used.

The fifth step refines parameters globally using **bundle** adjustment to build an adjuster (`Ptr<BundleAdjusterBase> adjuster`). Given a set of images depicting a number of 2D or 3D points from different viewpoints, bundle adjustment can be defined as the problem of simultaneously refining the 2D or 3D coordinates, describing the scene geometry as well as the parameters of the relative motion and optical characteristics of the cameras employed to acquire the images according to an optimality criterion involving the corresponding image projections of all points. There are two methods to calculate this bundle adjustment, `reproj` (`adjuster = makePtr<BundleAdjusterReproj>()`) or `ray` (`adjuster = makePtr<BundleAdjusterRay>()`), which are selected with the `string adjuster_method = "ray"` parameter. Finally this bundle adjustment is used as `(*adjuster)(features, pairwise_matches, cameras)`.

The sixth step is an optional step (`bool do_wave_correct = true`) that calculates the wave correlation to improve the camera setting. The type of wave correlation is selected with the `WaveCorrectKind wave_correct_type = WAVE_CORRECT_HORIZ` parameter and calculated as `waveCorrect(rmats, wave_correct_type)`.

The seventh step creates a warper image that needs a map projection. The map projections have been described previously, and they can be *rectilinear*, *cylindrical*, *spherical*, *stereographic*, or *panini*. There are actually more map projections implemented in OpenCV. The map projections can be selected with the `string warp_type = "spherical"` parameter. Afterwards, a warper is created (`Ptr<RotationWarper> warper = warper_creator-> create(static_cast<float>(warped_image_scale * scale))`) and each image is warped (`warper->warp(masks[i], K, cameras[i].R, INTER_NEAREST, BORDER_CONSTANT, masks_warped[i])`).

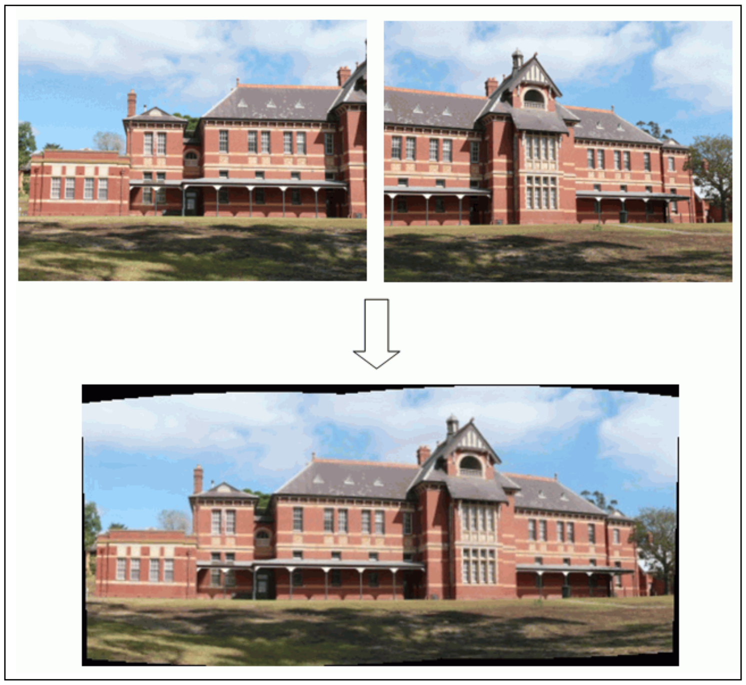
The eighth step compensates exposure errors by creating a compensator (`Ptr<ExposureCompensator> compensator = ExposureCompensator::createDefault(expos_comp_type)`) and it is applied to each warped image (`compensator->feed(corners, images_warped, masks_warped)`).

The ninth step finds seam masks. This process searches for the best areas of attachment for each panorama image. There are some methods implemented in OpenCV to perform this task, and this example uses the `string seam_find_type = "gc_color"` parameter to select them. These methods are `NoSeamFinder` (there's no use of this method), `VoronoiSeamFinder`, `GraphCutSeamFinderBase::COST_COLOR`, `GraphCutSeamFinderBase::COST_COLOR_GRAD`, `DpSeamFinder::COLOR`, and `DpSeamFinder::COLOR_GRAD`.

The tenth step creates a blender to combine each image to build the panorama. There are two types of blenders implemented in OpenCV, `MultiBandBlender* mb = dynamic_cast<MultiBandBlender*>(blender.get())` and `FeatherBlender* fb = dynamic_cast<FeatherBlender*>(blender.get())`, which can be selected with the `int blend_type = Blender::MULTI_BAND` parameter. Finally, the blender is prepared (`blender->prepare(corners, sizes)`).

The last step composites the final panorama. This step needs what the previous steps have done to configure the stitching. Four sub-steps are performed to calculate the final panorama. First, each input image is read (`full_img = imread(img_names[img_idx])`) and, if necessary, resized (`resize(full_img, img, Size(), scale, scale)`). Second, these images are warped with the created warper (`warper->warp(img, K, cameras[img_idx].R, INTER_LINEAR, BORDER_REFLECT, img_warped)`). Third, these images are compensated for exposure errors with the created compensator (`compensator->apply(img_idx, corners[img_idx], img_warped, mask_warped)`). Finally, these images are blended using the created blender. The final result panorama is now saved in the `string result_name = "panorama_result.jpg"` file.

To show you the **stitchingAdvanced** results, two input images are stitched and the resulting panorama is shown as follows:



Summary

In this chapter, you learned how to use three important modules of OpenCV that handle image processing in video. These modules are **video stabilization**, **superresolution**, and **stitching**. Some of the theoretical underpinnings have also been explained for each module.

In each section of this chapter, a complete example, developed in C++, is explained. An image result was also shown for each module, showing the main effect.

The next chapter introduces high-dynamic-range images and shows you how to handle them with OpenCV. High-dynamic-range imaging is typically considered within what is now called computational photography. Roughly speaking, computational photography refers to techniques that allow you to extend the typical capabilities of digital photography. This may include hardware add-ons or modifications, but it mostly refers to software-based techniques. These techniques may produce output images that cannot be obtained with a "traditional" digital camera.

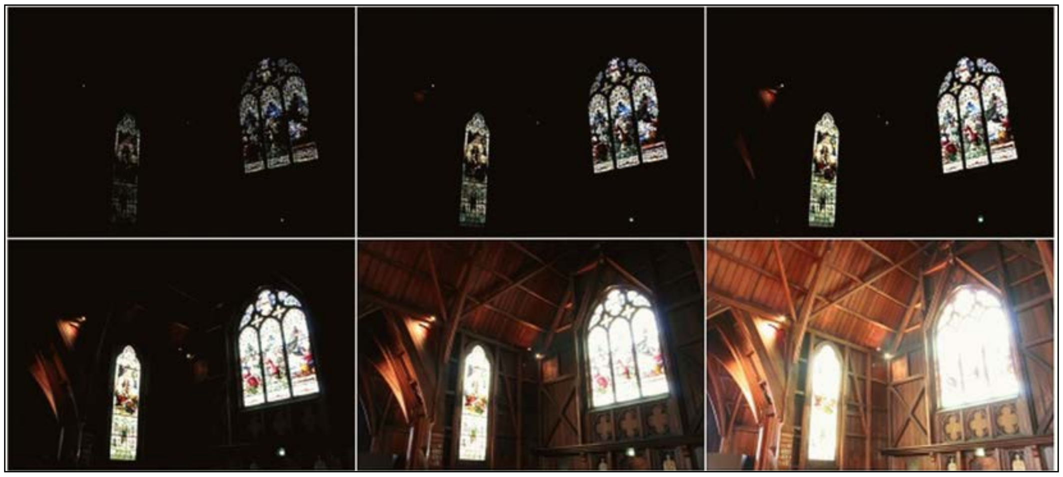
6
Computational Photography

Computational photography refers to techniques that allow you to extend the typical capabilities of digital photography. This may include hardware add-ons or modifications, but it mostly refers to software-based techniques. These techniques may produce output images that cannot be obtained with a "traditional" digital camera. This chapter introduces some of the lesser-known techniques available in OpenCV for computational photography: high-dynamic-range imaging, seamless cloning, decolorization, and non-photorealistic rendering. These three are inside the `photo` module of the library. Note that other techniques inside this module (inpainting and denoising) have been already considered in previous chapters.

High-dynamic-range images

The typical images we process have 8 **bits per pixel** (**bpp**). Color images also use 8 bits to represent the value of each channel, that is, red, green, and blue. This means that only 256 different intensity values are used. This 8 bpp limit has prevailed throughout the history of digital imaging. However, it is obvious that light in nature does not have only 256 different levels. We should, therefore, consider whether this discretization is desirable or even sufficient. The human eye, for example, is known to capture a much higher dynamic range (the number of light levels between the dimmest and brightest levels), estimated at between 1 and 100 million light levels. With only 256 light levels, there are cases where bright lights appear overexposed or saturated, while dark scenes are simply captured as black.

There are cameras that can capture more than 8 bpp. However, the most common way to create high-dynamic-range images is to use an 8 bpp camera and take images with different exposure values. When we do this, problems of a limited dynamic range are evident. Consider, for example, the following figure:



A scene captured with six different exposure values

The top-left image is mostly black, but window details are visible. Conversely, the bottom-right image shows details of the room, but the window details are barely visible.

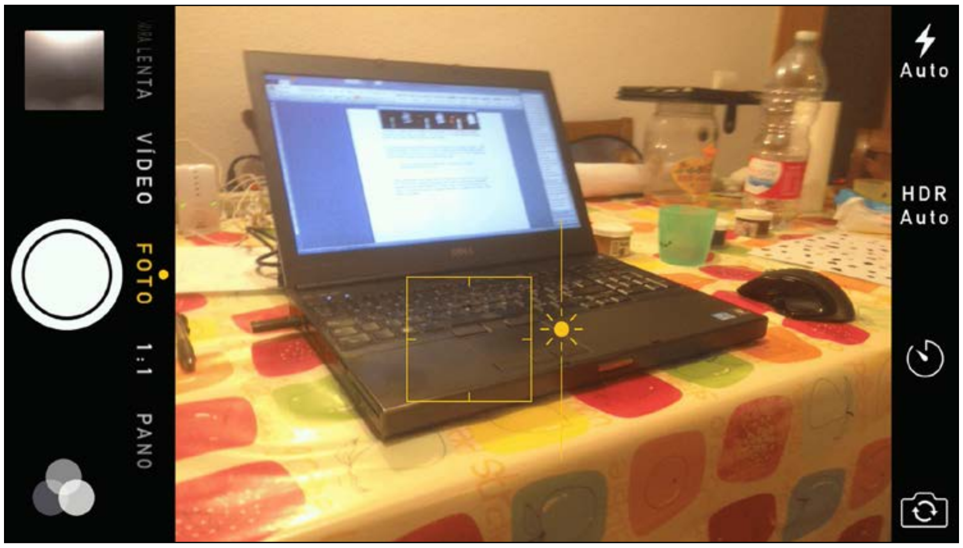
We can take pictures with different exposure levels using modern smartphone cameras. With iPhone and iPads, for example, as of iOS 8, it is very easy to change the exposure with the native camera app. By touching the screen, a yellow box appears with a small sun on its side. Swiping up or down can then change the exposure (see the following screenshot).

The range of exposure levels is quite large, so we may have to repeat the swiping gesture a number of times.

If you use previous versions of iOS, you can download camera apps such as *Camera+* that allow you to focus on a specific point and change exposure.

For Android, tons of camera apps are available on Google Play that can adjust the exposure. One example is *Camera FV-5*, which has both free and paid versions.

[If you use a handheld device to capture the images, make sure the device is static. In fact, you may well use a tripod. Otherwise, images with different exposures will not be aligned. Also, moving subjects will inevitably produce ghost artifacts. Three images are sufficient for most cases, with low, medium, and high exposure levels.]



The exposure control using the native camera app in an iPhone 5S

Smartphones and tables are handy to capture a number of images with different exposures. To create HDR images, we need to know the exposure (or shutter) time for each captured image (see the following section for the reason). Not all apps allow you to control (or even see) this manually (the iOS 8 native app doesn't). At the time of writing this, at least two free apps allow this for iOS: *Manually* and *ManualShot!* In Android, the free *Camera FV-5* allows you to control and see exposure times. Note that F/Stop and ISO are two other parameters that control the exposure.

Images that are captured can be transferred to the development computer and used to create the HDR image.

[As of iOS 7, the native camera app has an HDR mode that automatically captures three images in a rapid sequence, each with different exposure. These images are also automatically combined into a single (sometimes better) image.]

Creating HDR images

How do we combine multiple (three, for example) exposure images into an HDR image? If we consider only one of the channels and a given pixel, the three pixel values (one for each exposure level) must be mapped to a single value in the larger output range (say, 16 bpp). This mapping is not easy. First of all, we have to consider that pixel intensities are a (rough) measure of sensor irradiance (the amount of light incident on the camera sensor). Digital cameras measure irradiance but in a nonlinear way. Cameras have a nonlinear response function that translates irradiance to pixel intensity values in the range of 0 to 255]. In order to map these values to a larger set of discrete values, we must estimate the camera response function (that is, the response within the 0 to 255 range).

How do we estimate the camera response function? We do that from the pixels themselves! The response function is an S-shaped curve for each color channel, and it can be estimated from the pixels (with three exposures of a pixel, we have three points on the curve for each color channel). As this is very time consuming, usually, a set of random pixels is chosen.

There's only one thing left. We previously talked about estimating the relationship between irradiance and pixel intensity. How do we know irradiance? Sensor irradiance is directly proportional to the exposure time (or equivalently, the shutter speed). This is the reason why we need exposure time!

Finally, the HDR image is computed as a weighted sum of the recovered irradiance values from the pixels of each exposure. Note that this image cannot be displayed on conventional screens, which also have a limited range.

[A good book on high-dynamic-range imaging is *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting* by Reinhard et al, *Morgan Kaufmann Pub*. The book is accompanied by a DVD containing images in different HDR formats.]

Example

OpenCV (as of 3.0 only) provides functions to create HDR images from a set of images taken with different exposures. There's even a tutorial example called *hdr_imaging*, which reads a list of image files and exposure times (from a text file) and creates the HDR image.

In order to run the hdr_imaging tutorial, you will need to download the required image files and text files with the list. You can download them from `https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/hdr`.

The `CalibrateDebevec` and `MergeDebevec` classes implement Debevec's method to estimate the camera response function and merge the exposures into an HDR image, respectively. The following **createHDR** example shows you how to use both classes:

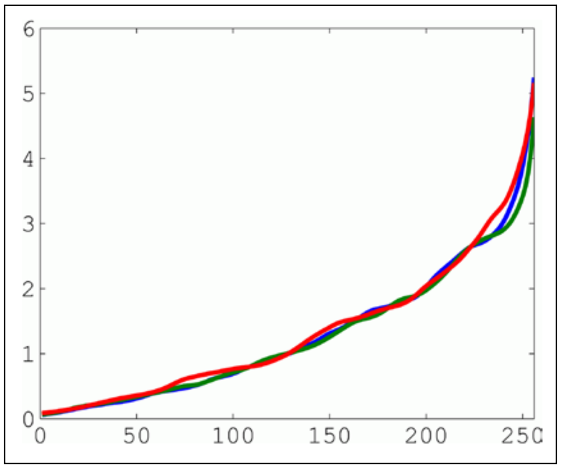
```
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    vector<Mat> images;
    vector<float> times;

    // Load images and exposures...
    Mat img1 = imread("1div66.jpg");
    if (img1.empty())
    {
        cout << "Error! Input image cannot be read...\n";
        return -1;
    }
    Mat img2 = imread("1div32.jpg");
    Mat img3 = imread("1div12.jpg");
    images.push_back(img1);
    images.push_back(img2);
    images.push_back(img3);
    times.push_back((float)1/66);
```

```
        times.push_back((float)1/32);
        times.push_back((float)1/12);

        // Estimate camera response...
        Mat response;
        Ptr<CalibrateDebevec> calibrate = createCalibrateDebevec();
        calibrate->process(images, response, times);

        // Show the estimated camera response function...
        cout << response;

        // Create and write the HDR image...
        Mat hdr;
        Ptr<MergeDebevec> merge_debevec = createMergeDebevec();
        merge_debevec->process(images, hdr, times, response);
        imwrite("hdr.hdr", hdr);

        cout << "\nDone. Press any key to exit...\n";
        waitKey(); // Wait for key press
        return 0;
    }
```

The example uses three images of a cup (the images are available along with the code accompanying this book). The images were taken with the *ManualShot!* app mentioned previously, using exposures of 1/66, 1/32, and 1/12 seconds; refer to the following figure:

The three images used in the example as inputs

Note that the `createCalibrateDebevec` method expects the images and exposure times in an STL vector (STL is a kind of library of useful common functions and data structures available in standard C++). The camera response function is given as a 256 real-valued vector. This represents the mapping between the pixel value and irradiance. Actually, it is a 256 x 3 matrix (one column per each of the three color channels). The following figure shows you the response given by the example:



The estimated RGB camera response functions

The `cout` part of code displays the matrix in the format used by MATLAB and Octave, two widely used packages for numerical computation. It is straightforward to copy the matrix in the output and paste it in MATLAB/Octave in order to display it.

The resulting HDR image is stored in the lossless RGBE format. This image format uses one byte per color channel plus one byte as a shared exponent. The format uses the same principle as the one used in the floating-point number representation: the shared exponent allows you to represent a much wider range of values. RGBE images use the `.hdr` extension. Note that as it is a lossless image format, `.hdr` files are relatively large. In this example, the RGB input images are 1224 x 1632 each (100 to 200 KB each), while the output `.hdr` file occupies 5.9 MB.

The example uses Debevec and Malik's method, but OpenCV also provides another calibration function based on Robertson's method. Both calibration and merge functions are available, that is, `createCalibrateRobertson` and `MergeRobertson`.

For more information on the other functions and the theory behind them, refer to `http://docs.opencv.org/trunk/modules/photo/doc/hdr_imaging.html`.

Finally, note that the example does not display the resulting image. The HDR image cannot be displayed in conventional screens, so we need to perform another step called tone mapping.

Tone mapping

When high-dynamic-range images are to be displayed, information can be lost. This is due to the fact that computer screens also have a limited contrast ratio, and printed material is also typically limited to 256 tones. When we have a high-dynamic-range image, it is necessary to map the intensities to a limited set of values. This is called tone mapping.

Simply scaling the HDR image values to the reduced range of the display device is not sufficient in order to provide a realistic output. Scaling typically produces images that appear as lacking detail (contrast), eliminating the original scene content. Ultimately, tone-mapping algorithms aim at providing outputs that appear visually similar to the original scene (that is, similar to what a human would see when viewing the scene). Various tone-mapping algorithms have been proposed and it is still a matter of extensive research. The following lines of code can apply tone mapping to the HDR image obtained in the previous example:

```
Mat ldr;
Ptr<TonemapDurand> tonemap = createTonemapDurand(2.2f);
tonemap->process(hdr, ldr); // ldr is a floating point image with
ldr=ldr*255;      //  values in interval [0..1]
imshow("LDR", ldr);
```

The method was proposed by Durand and Dorsey in 2002. The constructor actually accepts a number of parameters that affect the output. The following figure shows you the output. Note how this image is not necessarily *better* than any of the three original images:

The tone-mapped output

Three other tone-mapping algorithms are available in OpenCV: `createTonemapDrago`, `createTonemapReinhard`, and `createTonemapMantiuk`.

An HDR image (the RGBE format, that is, files with the `.hdr` extension) can be displayed using MATLAB. All it takes is three lines of code:

```
hdr=hdrread('hdr.hdr');
rgb=tonemap(hdr);
imshow(rgb);
```

pfstools is an open source suite of command-line tools to read, write, and render HDR images. The suite, which can read `.hdr` and other formats, includes a number of camera calibration and tone-mapping algorithms. Luminance HDR is free GUI software based on pfstools.

Alignment

The scene that will be captured with multiple exposure images must be static. The camera must also be static. Even if the two conditions met, it is advisable to perform an alignment procedure.

OpenCV provides an algorithm for image alignment proposed by G. Ward in 2003. The main function, `createAlignMTB`, takes an input parameter that defines the maximum shift (actually, a logarithm the base two of the maximum shift in each dimension). The following lines should be inserted right before estimating the camera response function in the previous example:

```
vector<Mat> images_(images);
Ptr<AlignMTB> align=createAlignMTB(4);// 4=max 16 pixel shift
align->process(images_, images);
```

Exposure fusion

We can also combine images with multiple exposures with neither camera response calibration (that is, exposure times) nor intermediate HDR image. This is called exposure fusion. The method was proposed by Mertens et al in 2007. The following lines perform exposure fusion (`images` is the STL vector of input images; refer to the previous example):

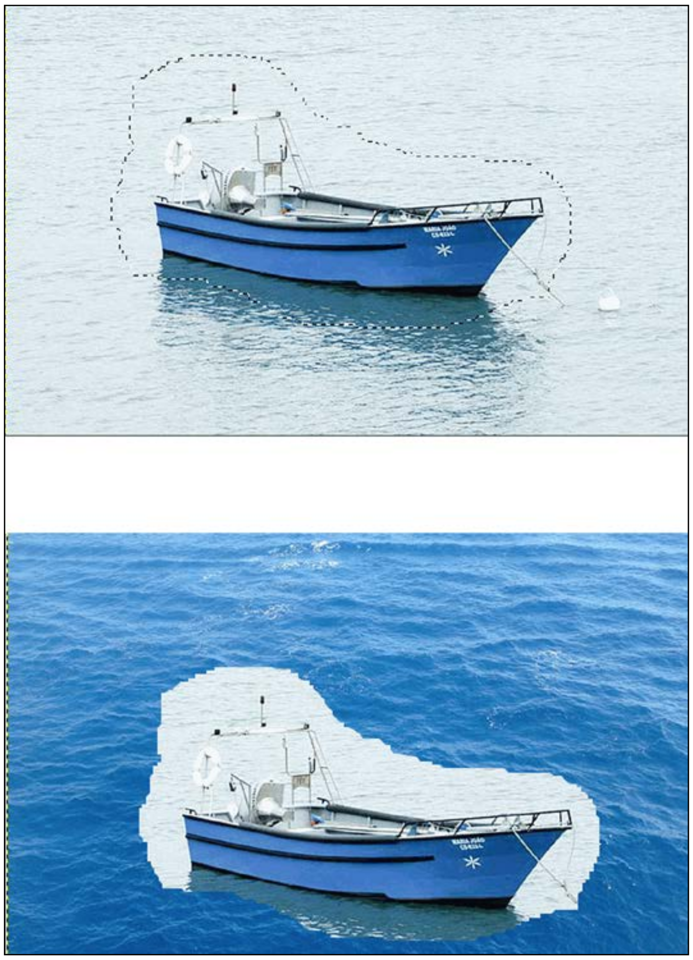
```
    Mat fusion;
    Ptr<MergeMertens> merge_mertens = createMergeMertens();
    merge_mertens->process(images, fusion); // fusion is a
fusion=fusion*255; // float. point image w. values in [0..1]
imwrite("fusion.png", fusion);
```

The following figure shows you the result:

Exposure fusion

Seamless cloning

In photomontages, we typically want to cut an object/person in a source image and insert it into a target image. Of course, this can be done in a straightforward way by simply pasting the object. However, this would not produce a realistic effect. See, for example, the following figure, in which we wanted to insert the boat in the top half of the image into the sea at the bottom half of the image:

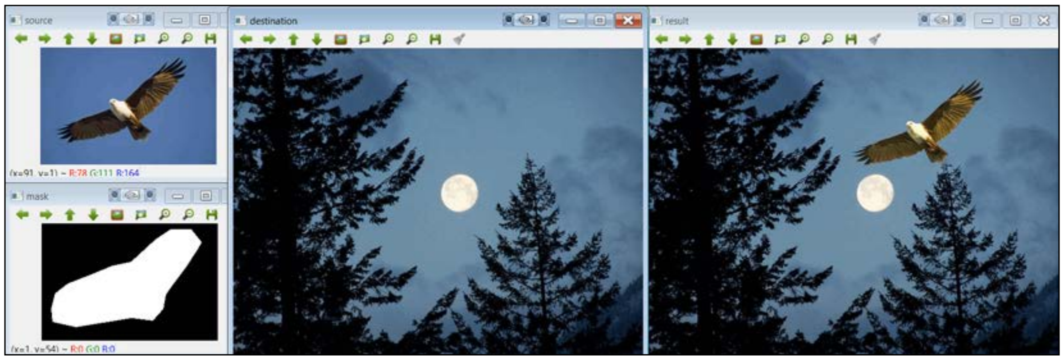


Cloning

As of OpenCV 3, there are seamless cloning functions available in which the result is more realistic. This function is called `seamlessClone` and it uses a method proposed by Perez and Gangnet in 2003. The following **seamlessCloning** example shows you how it can be used:

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

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    // Load and show images...
    Mat source = imread("source1.png", IMREAD_COLOR);
    Mat destination = imread("destination1.png", IMREAD_COLOR);
    Mat mask = imread("mask.png", IMREAD_COLOR);
    imshow("source", source);
    imshow("mask", mask);
    imshow("destination", destination);

    Mat result;
    Point p;    // p will be near top right corner
    p.x = (float)2*destination.size().width/3;
    p.y = (float)destination.size().height/4;
    seamlessClone(source, destination, mask, p, result, NORMAL_CLONE);
    imshow("result", result);

    cout << "\nDone. Press any key to exit...\n";
    waitKey(); // Wait for key press
    return 0;
}
```

The example is straightforward. The `seamlessClone` function takes the source, destination, and mask images and a point in the destination image in which the cropped object will be inserted (these three images can be downloaded from `https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/cloning/Normal_Cloning`). See the result in the following figure:



Seamless cloning

The last parameter of `seamlessClone` represents the exact method to be used (there are three methods available that produce a different final effect). On the other hand, the library provides the following related functions:

Function	**Effect**
`colorChange`	Multiplies each of the three color channels of the source image by a factor, applying the multiplication only in the region given by the mask
`illuminationChange`	Changes illumination of the source image, only in the region given by the mask
`textureFlattening`	Washes out textures in the source image, only in the region given by the mask

As opposed to `seamlessClone`, these three functions only accept source and mask images.

Decolorization

Decolorization is the process of converting a color image to grayscale. Given this definition, the reader may well ask, don't we already have grayscale conversion? Yes, grayscale conversion is a basic routine in OpenCV and any image-processing library. The standard conversion is based on a linear combination of the R, G, and B channels. The problem is that such a conversion may produce images in which contrast in the original image is lost. The reason is that two different colors (which are perceived as contrasts in the original image) may end up being mapped to the same grayscale value. Consider the conversion of two colors, A and B, to grayscale. Let's suppose that B is a variation of A in the R and G channels:

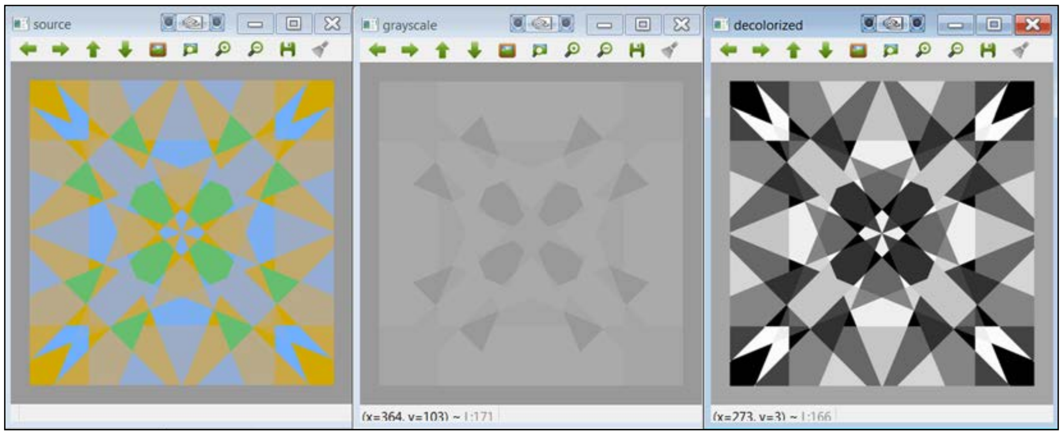
$A = (R,G,B) \quad \Rightarrow \quad G = (R+G+B)/3$

$B = (R-x,G+x,B) \quad \Rightarrow \quad G = (R-x+G+x+B)/3 = (R+G+B)/3$

Even though they are perceived as distinct, the two colors A and B are mapped to the same grayscale value! The images from the following **decolorization** example show this:

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

using namespace cv;
using namespace std;

int main(int, char** argv)
{
    // Load and show images...
    Mat source = imread("color_image_3.png", IMREAD_COLOR);
    imshow("source", source);

    // first compute and show standard grayscale conversion...
    Mat grayscale = Mat(source.size(),CV_8UC1);
    cvtColor(source, grayscale, COLOR_BGR2GRAY);
    imshow("grayscale",grayscale);

    // now compute and show decolorization...
    Mat decolorized = Mat(source.size(),CV_8UC1);
```

```
        Mat dummy = Mat(source.size(),CV_8UC3);
        decolor(source,decolorized,dummy);
        imshow("decolorized",decolorized);

        cout << "\nDone. Press any key to exit...\n";
        waitKey(); // Wait for key press
        return 0;
    }
```



Decolorization example output

The example is straightforward. After reading the image and showing the result of a standard grayscale conversion, it uses the `decolor` function to perform the decolorization. The image used (the `color_image_3.png` file) is included in the opencv_extra repository at `https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/decolor`.

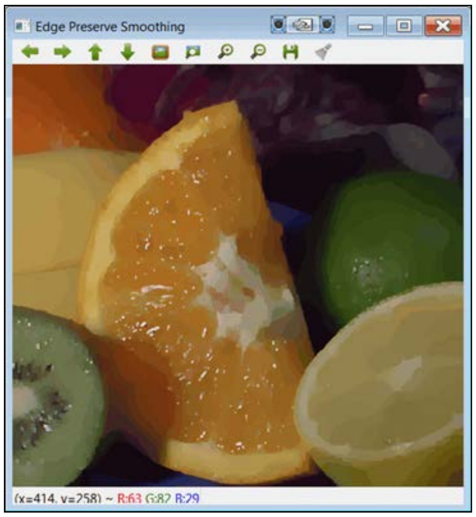
[The image used in the example is actually an extreme case. Its colors have been chosen so that the standard grayscale output is fairly homogeneous.]

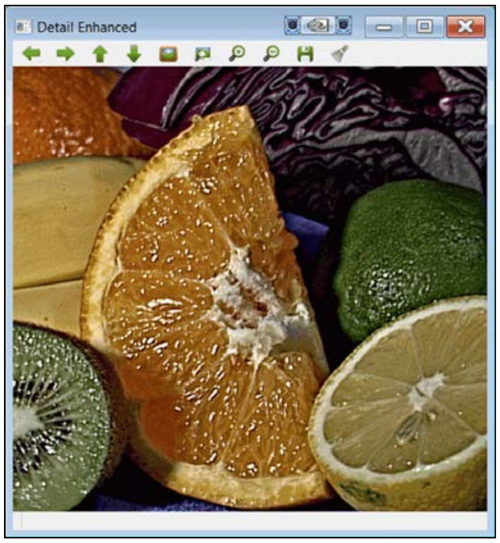
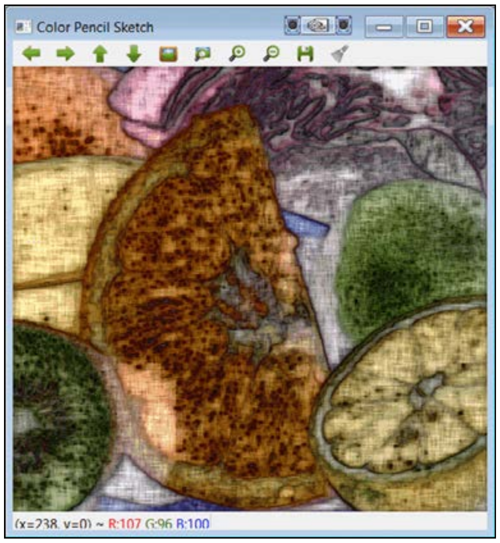
Non-photorealistic rendering

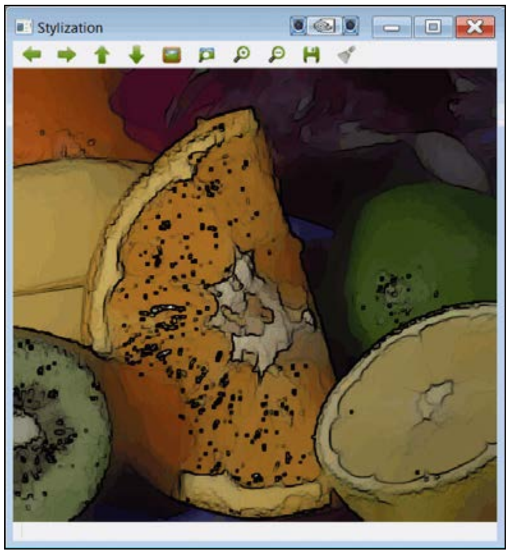
As part of the `photo` module, four functions are available that transform an input image in a way that produces a non-realistic but still artistic output. The functions are very easy to use and a nice example is included with OpenCV (`npr_demo`). For illustrative purposes, here we show you a table that allows you to grasp the effect of each function. Take a look at the following `fruits.jpg` input image, included with OpenCV:

The input reference image

The effects are:

Function	**Effect**
`edgePreservingFilter`	Smoothing is a handy and frequently used filter. This function performs smoothing while preserving object edge details.
 (x=414, y=258) ~ R:63 G:82 B:29	

Function	Effect
`detailEnhance`	Enhances details in the image
 Detail Enhanced	
`pencilSketch`	A pencil-like line drawing version of the input image
 Color Pencil Sketch (x=238, y=0) ~ R:107 G:96 B:100	

Function	**Effect**
`stylization`	Watercolor effect
	

Summary

In this chapter, you learned what computational photography is and the related functions available in OpenCV 3. We explained the most important functions within the `photo` module, but note that other functions of this module (inpainting and noise reduction) were also considered in previous chapters. Computational photography is a rapidly expanding field, with strong ties to computer graphics. Therefore, this module of OpenCV is expected to grow in future versions.

The next chapter will be devoted to an important aspect that we have not yet considered: time. Many of the functions explained take a significant time to compute the results. The next chapter will show you how to deal with that using modern hardware.

7
Accelerating Image Processing

This chapter deals with the acceleration of image processing tasks using General Purpose Graphics Processing Units (**GPGPUs**) or, in short, **GPU**s with parallel processing. A GPU is essentially a coprocessor dedicated to graphics processing or floating point operations, aimed at improving performance on applications such as video games and interactive 3D graphics. While the graphics processing is executed in the GPU, the CPU can be dedicated to other calculations (such as the artificial intelligence part in games). Every GPU is equipped with hundreds of simple processing cores that allow massive parallel execution on hundreds of "simple" mathematical operations on (normally) floating point numbers.

CPUs seem to have reached their speed and thermal power limits. Building a computer with several CPUs has become a complex problem. This is where GPUs come into play. GPU processing is a new computing paradigm that uses the GPU to improve the computational performance. GPUs initially implemented certain parallel operations called graphics primitives that were optimized for graphics processing. One of the most common primitives for 3D graphics processing is antialiasing, which makes the edges of the figures have a more realistic appearance. Other primitives are drawings of rectangles, triangles, circles, and arcs. GPUs currently include hundreds of general-purpose processing functions that can do much more than rendering graphics. Particularly, they are very valuable in tasks that can be parallelized, which is the case for many computer vision algorithms.

OpenCV libraries include support for the OpenCL and CUDA GPU architectures. CUDA implements many algorithms; however, it only works with NVIDIA graphic cards. CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by the GPUs that they produce. This chapter focuses on the OpenCL architecture, as it is supported by more devices and is even included in some NVIDIA graphic cards.

The **Open Computing Language** (**OpenCL**) is a framework that writes programs that can be executed on CPUs or GPUs attached to a host processor (a CPU). It defines a C-like language to write functions, called kernels, which are executed on the computing devices. Using OpenCL, kernels can be run on all or many of the individual processing elements (PEs) in parallel to the CPUs or GPUs.

In addition, OpenCL defines an **Application Programming Interface** (**API**) that allows programs running on the host (the CPU) to launch kernels on the computer devices and manage their device memories, which are (at least conceptually) separated from the host memory. OpenCL programs are intended to be compiled at runtime so that applications that use OpenCL are portable between implementations of various host computer devices. OpenCL is also an open standard maintained by the nonprofit technology consortium Khronos Group (`https://www.khronos.org/`).

OpenCV contains a set of classes and functions that implement and accelerate the OpenCV functionality using OpenCL. OpenCV currently provides a transparent API that enables the unification of its original API with OpenCL-accelerated programming. Therefore, you only need to write your code once. There is a new unified data structure (**UMat**) that handles data transfers to the GPUs when it is needed and possible.

Support for OpenCL in OpenCV has been designed for ease of use and does not require any knowledge of OpenCL. At a minimum level, it can be viewed as a set of accelerations, which can take advantage of the high computing power when using modern CPU and GPU devices.

To correctly run OpenCL programs, the OpenCL runtime should be provided by the device vendor, typically in the form of a device driver. Also, to use OpenCV with OpenCL, a compatible SDK is needed. Currently, there are five available OpenCL SDKs:

- **AMD APP SDK**: This SDK supports OpenCL on CPUs and GPUs, such as X86+SSE2 (or higher) CPUs and AMD Fusion, AMD Radeon, AMD Mobility, and ATI FirePro GPUs.
- **Intel SDK**: This SDK supports OpenCL on Intel Core processors and Intel HD GPUs, such as Intel+SSE4.1, SSE4.2 or AVX, Intel Core i7, i5 and i3 (1st, 2nd, and 3rd Generation), Intel HD Graphics, Intel Core 2 Solo (Duo Quad and Extreme), and Intel Xeon CPUs.
- **IBM OpenCL Development Kit**: This SDK supports OpenCL on AMD servers such as IBM Power, IBM PERCS, and IBM BladeCenter.

- **IBM OpenCL Common Runtime**: This SDK supports OpenCV on CPUs and GPUs, such as X86+SSE2 (or higher) CPUs and AMD Fusion and Raedon, NVIDIA Ion, NVIDIA GeForce, and NVIDIA Quadro GPUs.
- **Nvidia OpenCL Driver and Tools**: This SDK supports OpenCL on some Nvidia graphic devices such as NVIDIA Tesla, NVIDIA GeForce, NVIDIA Ion, and NVIDIA Quadro GPUs.

OpenCV with the OpenCL installation

The installation steps already presented in *Chapter 1*, *Handling Image and Video Files*, need some additional steps to include OpenCL. The newly required software is explained in the following section.

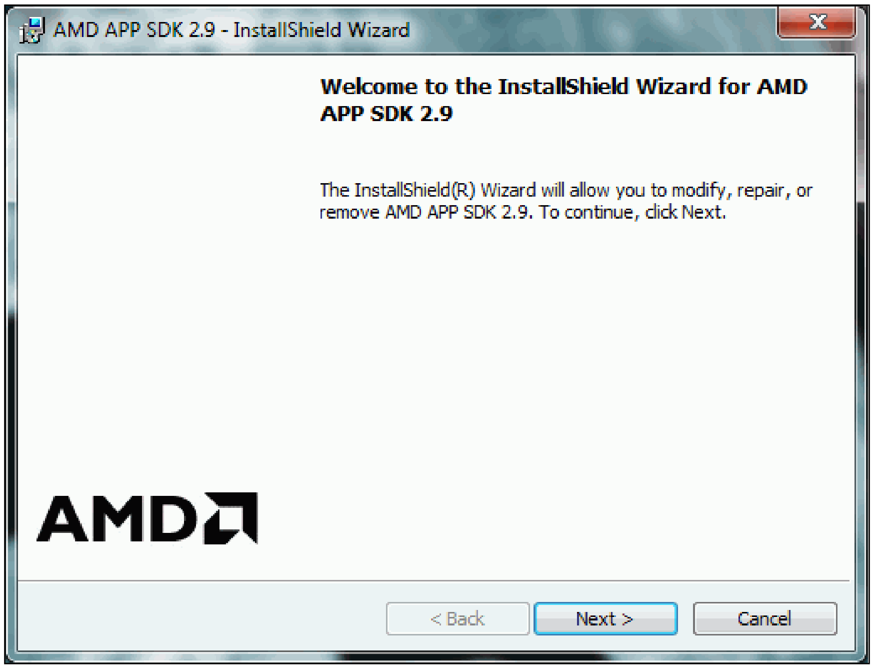
There are new requirements to compile and install OpenCV with OpenCL on Windows:

- **OpenCL-capable GPU or CPU**: This is the most important requirement. Note that OpenCL supports many computing devices but not all. You can check whether your graphic cards or processors are compatible with OpenCL. This chapter uses the AMD APP SDK for an AMD FirePro W5000 GPU to execute the examples.

> There is a list with the supported computer devices for this SDK at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/system-requirements-driver-compatibility/`.There, you can also consult the minimum SDK version that you need.

- **Compilers**: OpenCV with OpenCL is compatible with Microsoft and MinGW compilers. It is possible to install the free Visual Studio Express edition. However, if you choose Microsoft to compile OpenCV, at least Visual Studio 2012 is recommended. However, the MinGW compiler is used in this chapter.
- **AMD APP SDK**: This SDK is a set of advanced software technologies that enable us to use compatible computing devices to execute and accelerate many applications beyond just graphics. This SDK is available at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`.This chapter uses Version 2.9 of the SDK (for 64 bit Windows); you can see the installation progress in the following screenshot.

If this step fails, maybe you might need to update the controller of your graphic card. The AMD controllers are available at `http://www.amd.com/en-us/innovations/software-technologies`.

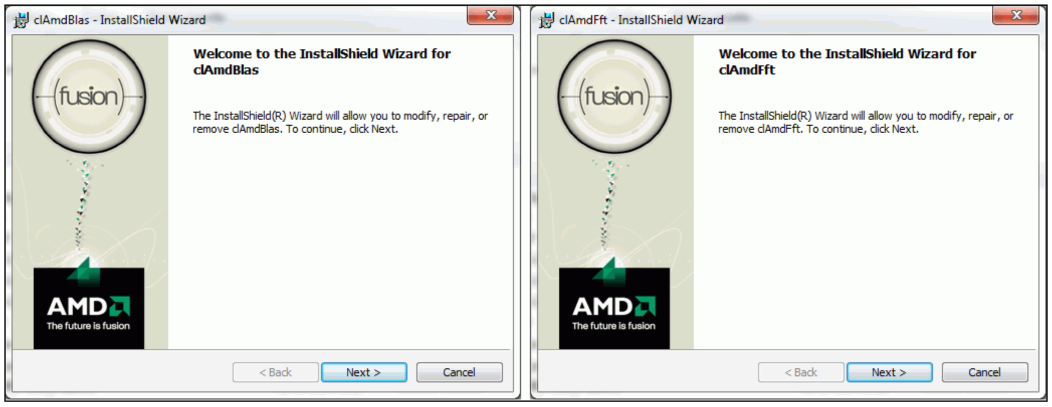


Welcome to the InstallShield Wizard for AMD APP SDK 2.9

The InstallShield(R) Wizard will allow you to modify, repair, or remove AMD APP SDK 2.9. To continue, click Next.

Installing the AMD APP SDK

- **OpenCL BLAS**: **Basic Linear Algebra Subroutines** (**BLAS**) is a set of open source math libraries for parallel processing on AMD devices. It can be downloaded from `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-math-libraries/`. This chapter uses the 1.1 BLAS version for Windows 32/64 bits, and you can see the installation progress in the following screenshot (the left-hand side).
- **OpenCL FFT**: **Fast Fourier Transform** (**FFT**) is a very useful function that many algorithms of image processing need. Therefore, this function is implemented for parallel processing on AMD devices. It can be downloaded from the same URL as given previously. This chapter uses the 1.1 FFT version for Windows 32/64 bits, and you can see the installation progress in the following screenshot (the right-hand side):


clAmdBlas - InstallShield Wizard

Welcome to the InstallShield Wizard for clAmdBlas

The InstallShield(R) Wizard will allow you to modify, repair, or remove clAmdBlas. To continue, click Next.

clAmdFft - InstallShield Wizard

Welcome to the InstallShield Wizard for clAmdFft

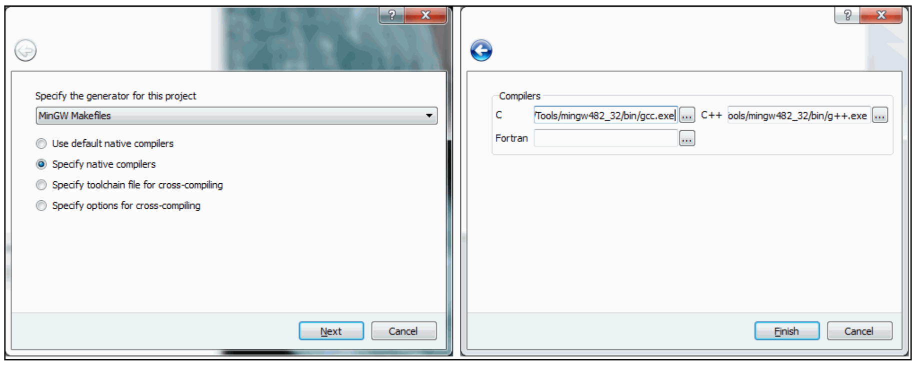
The InstallShield(R) Wizard will allow you to modify, repair, or remove clAmdFft. To continue, click Next.

Installing BLAS and FFT for OpenCL

- **Qt libraries for C++ compiler**: In this chapter, the MinGW binaries of Qt libraries are used to compile OpenCV with OpenCL. The other alternative is to install the latest version of Qt and use the Visual C++ compiler. You can choose the Qt version and used compiler. The package manager, by means of the `MaintenanceTool.exe` application located at `C:\Qt\Qt5.3.1`, can be used to download other Qt versions. This chapter uses Qt (5.3.1) and MinGW (4.8.2) 32 bits to compile OpenCV with OpenCL.

When the previous requirements are met, you can generate a new build configuration with CMake. This process differs in some points from the typical installation that was explained in the first chapter. The differences are explained in this list:

- When selecting the generator for the project, you can choose the compiler version corresponding with the installed environment in your machine. This chapter uses MinGW to compile OpenCV with OpenCL, and then the **MinGW Makefiles** option is selected, specifying the native compilers. The following screenshot shows this selection:


Specify the generator for this project

MinGW Makefiles

- ○ Use default native compilers
- ● Specify native compilers
- ○ Specify toolchain file for cross-compiling
- ○ Specify options for cross-compiling

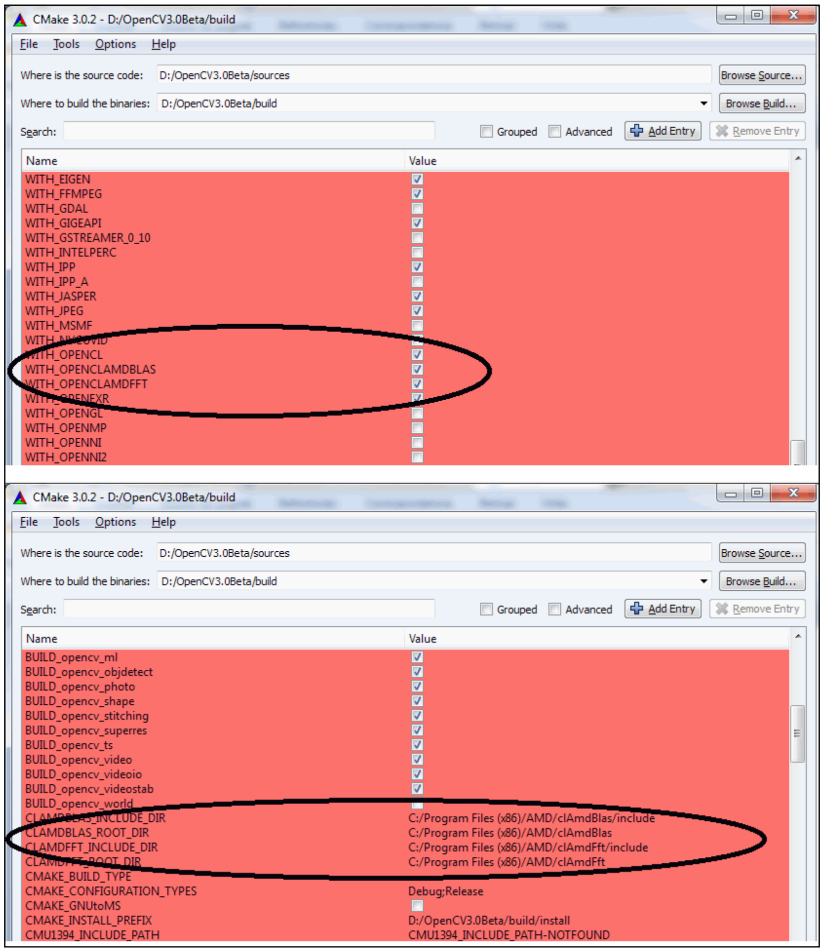
Compilers

C: Tools/mingw482_32/bin/gcc.exe C++: ools/mingw482_32/bin/g++.exe

Fortran:

CMake selecting the generator project

- The options shown in the following screenshot are needed to build the OpenCV with OpenCL project. The **WITH_OPENCL**, **WITH_OPENCLAMDBLAS**, and **WITH_OPENCLAMDFFT** options must be enabled. The BLAS and FFT paths must be introduced on **CLAMDBLAS_INCLUDE_DIR**, **CLAMDBLAS_ROOT_DIR**, **CLAMDFFT_INCLUDE_DIR**, and **CLAMDFFT_ROOT_DIR**. In addition, as shown in *Chapter 1, Handling Image and Video Files*, you will need to enable **WITH_QT** and disable the **WITH_IPP** option as well. It is also advisable to enable **BUILD_EXAMPLES**. The following screenshot shows you the main options selected in the build configuration:



Where is the source code: D:/OpenCV3.0Beta/sources

Where to build the binaries: D:/OpenCV3.0Beta/build

Search:

Name	Value
WITH_EIGEN	✓
WITH_FFMPEG	✓
WITH_GDAL	☐
WITH_GIGEAPI	✓
WITH_GSTREAMER_0_10	☐
WITH_INTELPERC	☐
WITH_IPP	✓
WITH_IPP_A	☐
WITH_JASPER	✓
WITH_JPEG	✓
WITH_MSMF	☐
WITH_NVCUVID	☐
WITH_OPENCL	✓
WITH_OPENCLAMDBLAS	✓
WITH_OPENCLAMDFFT	✓
WITH_OPENEXR	✓
WITH_OPENGL	☐
WITH_OPENMP	☐
WITH_OPENNI	☐
WITH_OPENNI2	☐

Where is the source code: D:/OpenCV3.0Beta/sources

Where to build the binaries: D:/OpenCV3.0Beta/build

Search:

Name	Value
BUILD_opencv_ml	✓
BUILD_opencv_objdetect	✓
BUILD_opencv_photo	✓
BUILD_opencv_shape	✓
BUILD_opencv_stitching	✓
BUILD_opencv_superres	✓
BUILD_opencv_ts	✓
BUILD_opencv_video	✓
BUILD_opencv_videoio	✓
BUILD_opencv_videostab	✓
BUILD_opencv_world	☐
CLAMDBLAS_INCLUDE_DIR	C:/Program Files (x86)/AMD/clAmdBlas/include
CLAMDBLAS_ROOT_DIR	C:/Program Files (x86)/AMD/clAmdBlas
CLAMDFFT_INCLUDE_DIR	C:/Program Files (x86)/AMD/clAmdFft/include
CLAMDFFT_ROOT_DIR	C:/Program Files (x86)/AMD/clAmdFft
CMAKE_BUILD_TYPE	
CMAKE_CONFIGURATION_TYPES	Debug;Release
CMAKE_GNUtoMS	☐
CMAKE_INSTALL_PREFIX	D:/OpenCV3.0Beta/build/install
CMU1394_INCLUDE_PATH	CMU1394_INCLUDE_PATH-NOTFOUND

CMake selecting the main options

Finally, to build the OpenCV with OpenCL project, the CMake project that was previously generated must be compiled. The project was generated for MinGW, and therefore, the MinGW compiler is needed to build this project. Firstly, the `[opencv_build]/` folder is selected with **Windows Console**, and we execute this:

```
./mingw32-make.exe -j 4 install
```

The `-j 4` parameter is the number of the core CPUs of the system that we want to use for the parallelization of the compilation.

Now the OpenCV with OpenCL project is ready to be used. The path of the new binaries files must be added to the system path, in this case, `[opencv_build]/install/x64/mingw/bin`.

[Do not forget to remove the old binary files of OpenCV from the path environment variable.]

A quick recipe to install OpenCV with OpenCL

The installation process can be summarized in the following steps:

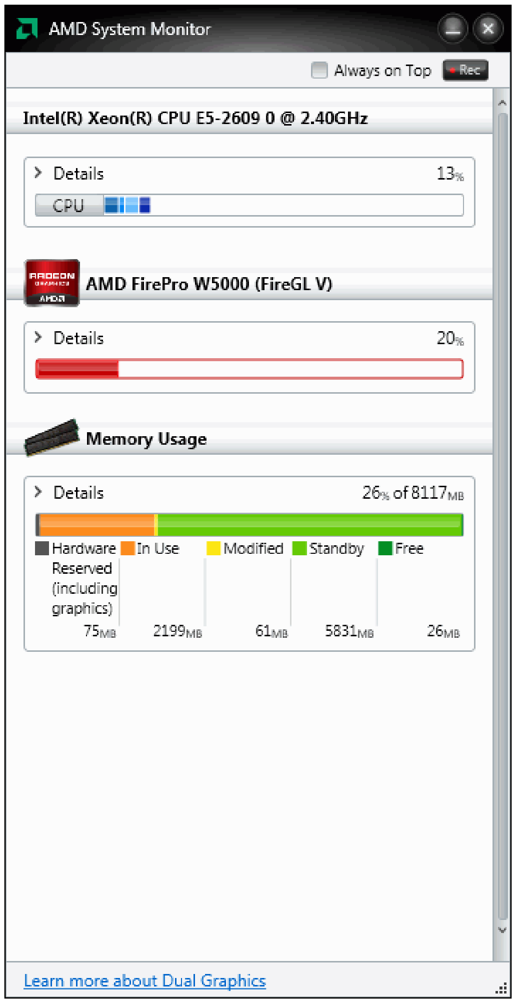
1. Download and install AMD APP SDK, which is available at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk`.
2. Download and install BLAS and FFT AMD, which are available at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-math-libraries`.
3. Configure the OpenCV build with CMake. Enable the **WITH_OPENCL, WITH_OPENCLAMDBLAS, WITH_QT**, and **Build_EXAMPLESWITH_OPENCLAMDFFT** options. Disable the **WITH_IPP** option. Finally, introduce the BLAS and FFT paths on **CLAMDBLAS_INCLUDE_DIR**, **CLAMDBLAS_ROOT_DIR**, **CLAMDFFT_INCLUDE_DIR**, and **CLAMDFFT_ROOT_DIR**.
4. Compile the OpenCV project with `mingw32-make.exe`.
5. Finally, modify the path environment variable to update the OpenCV bin directory (for example, `[opencv_build]/install/x64/mingw/bin`).

Check the GPU usage

When the GPU is being used on a Windows platform, there is no application to measure its usage. The GPU usage is useful for two reasons:

- It is possible to know whether you are using the GPU correctly
- You can monitor the GPU usage percentage

There are some applications in the market for this purpose. This chapter uses **AMD System Monitor** to check the GPU usage. This application monitors the CPU, memory RAM, and GPU usage. Refer to the following screenshot for this:



AMD System Monitor monitors the CPU, GPU, and Memory RAM usages

[The AMD System Monitor can be downloaded from `http://support.amd.com/es-xl/kb-articles/Pages/AMDSystemMonitor.aspx` for Microsoft Windows (32 or 64 bits).]

Accelerating your own functions

In this section, there are three examples of using OpenCV with OpenCL. The first example allows you to check whether the installed SDK is available and obtain useful information about the computing devices that support OpenCL. The second example shows you two versions of the same program using CPU and GPU programming, respectively. The last example is a complete program to detect and mark faces. In addition, a computational comparative is performed.

Checking your OpenCL

The following is a simple program that is shown to check your SDK and the available computing devices. This example is called **checkOpenCL.** It allows you to display the computer devices using the OCL module of OpenCV:

```
#include <opencv2/opencv.hpp>
#include <opencv2/core/ocl.hpp>

using namespace std;
using namespace cv;
using namespace cv::ocl;

int main()
{
vector<ocl::PlatformInfo> info;
    getPlatfomsInfo(info);
PlatformInfo sdk = info.at(0);

    if (sdk.deviceNumber()<1)
        return -1;

    cout << "******SDK*******" << endl;
    cout << "Name: " <<sdk.name()<< endl;
cout << "Vendor: " <<sdk.vendor()<< endl;
cout << "Version: " <<sdk.version()<< endl;
    cout << "Number of devices: " <<sdk.deviceNumber()<< endl;
    for (int i=0; i<sdk.deviceNumber(); i++){
Device device;
          sdk.getDevice(device, i);
          cout << "\n\n*********************\n Device " << i+1 <<
endl;

        cout << "Vendor ID: " <<device.vendorID()<< endl;
        cout << "Vendor name: " <<device.vendorName()<< endl;
        cout << "Name: " <<device.name()<< endl;
        cout << "Driver version: " <<device.driverVersion()<< endl;
        if (device.isAMD()) cout << "Is an AMD device" << endl;
        if (device.isIntel()) cout << "Is a Intel device" << endl;
        cout << "Global Memory size: " <<device.globalMemSize()<<
endl;
        cout << "Memory cache size: " <<device.globalMemCacheSize()<<
endl;
        cout << "Memory cache type: " <<device.globalMemCacheType()<<
endl;
        cout << "Local Memory size: " <<device.localMemSize()<< endl;
        cout << "Local Memory type: " <<device.localMemType()<< endl;
        cout << "Max Clock frequency: " <<device.maxClockFrequency()<<
endl;
    }

    return 0;
}
```

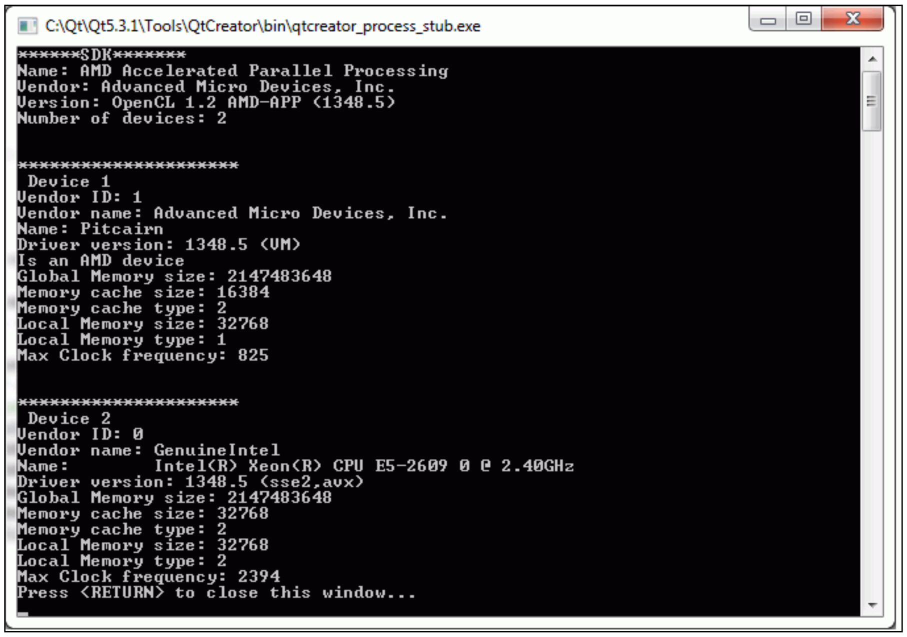
The code explanation

This example displays the installed SDK and the available computing devices that are compatible with OpenCL. Firstly, the `core/ocl.hpp` header is included and the `cv::ocl` namespace is declared.

The information about the available SDK in your computer is obtained using the `getPlatfomsInfo(info)` method. This information is stored in the `vector<ocl::PlatformInfo> info` vector and selected with `PlatformInfo sdk = info.at(0)`. Afterwards, the main information about your SDK is shown, such as the name, vendor, SDK version, and the number of computing devices compatible with OpenCL.

Finally, for each compatible device, its information is obtained with the `sdk.getDevice(device, i)` method. Now different information about each computing device can be shown, such as the vendor ID, vendor name, driver version, global memory size, memory cache size, and so on.

The following screenshot shows you the results of this example for the computer used:



C:\Qt\Qt5.3.1\Tools\QtCreator\bin\qtcreator_process_stub.exe

```
******SDK*******
Name: AMD Accelerated Parallel Processing
Vendor: Advanced Micro Devices, Inc.
Version: OpenCL 1.2 AMD-APP (1348.5)
Number of devices: 2

*********************
 Device 1
Vendor ID: 1
Vendor name: Advanced Micro Devices, Inc.
Name: Pitcairn
Driver version: 1348.5 (VM)
Is an AMD device
Global Memory size: 2147483648
Memory cache size: 16384
Memory cache type: 2
Local Memory size: 32768
Local Memory type: 1
Max Clock frequency: 825

*********************
 Device 2
Vendor ID: 0
Vendor name: GenuineIntel
Name:          Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz
Driver version: 1348.5 (sse2,avx)
Global Memory size: 2147483648
Memory cache size: 32768
Memory cache type: 2
Local Memory size: 32768
Local Memory type: 2
Max Clock frequency: 2394
Press <RETURN> to close this window...
```

Information about the SDK used and compatible computing devices

Your first GPU-based program

In the following code, two versions of the same program are shown: one only uses the CPU (native) to perform the computations and the other uses the GPU (with OpenCL). These two examples are called **calculateEdgesCPU** and **calculateEdgesGPU**, respectively, and allow you to observe the differences between CPU and GPU versions.

The **calculateEdgesCPU** example is shown in the first place:

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

using namespace std;
using namespace cv;

int main(int argc, char * argv[])
{
    if (argc < 2)
    {
        cout << "./calculateEdgesCPU <image>" << endl;
        return -1;
    }

Mat cpuFrame = imread(argv[1]);
    Mat cpuBW, cpuBlur, cpuEdges;

    namedWindow("Canny Edges CPU",1);

cvtColor(cpuFrame, cpuBW, COLOR_BGR2GRAY);
    GaussianBlur(cpuBW, cpuBlur, Size(1,1), 1.5, 1.5);
    Canny(cpuBlur, cpuEdges, 50, 100, 3);

imshow("Canny Edges CPU", cpuEdges);
    waitKey();

    return 0;
}
```

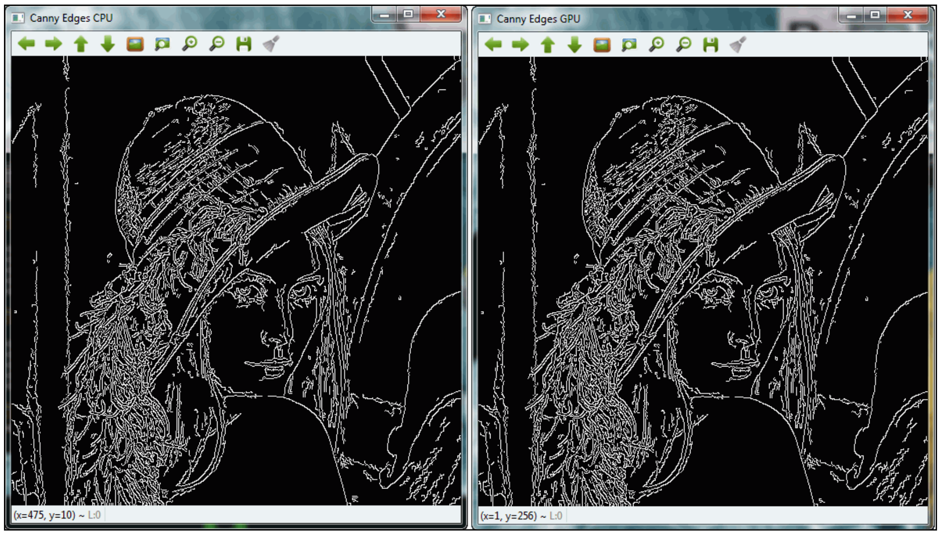
Now, the **calculateEdgesGPU** example is shown:

```
    #include "opencv2/opencv.hpp"
    #include "opencv2/core/ocl.hpp"

    using namespace std;
    using namespace cv;
    using namespace cv::ocl;

    int main(int argc, char * argv[])
    {
        if (argc < 2)
        {
            cout << "./calculateEdgesGPU <image>" << endl;
            return -1;
        }

    setUseOpenCL(true);

    Mat cpuFrame = imread(argv[1]);
    UMat gpuFrame, gpuBW, gpuBlur, gpuEdges;

    cpuFrame.copyTo(gpuFrame);

        namedWindow("Canny Edges GPU",1);
        cvtColor(gpuFrame, gpuBW, COLOR_BGR2GRAY);
        GaussianBlur(gpuBW, gpuBlur, Size(1,1), 1.5, 1.5);
        Canny(gpuBlur, gpuEdges, 50, 100, 3);

    imshow("Canny Edges GPU", gpuEdges);
        waitKey();

        return 0;
    }
```

The code explanation

These two examples obtain the same result, as shown in the following screenshot. They read an image from the standard command-line input arguments. Afterwards, the image is converted to gray scale and the Gaussian Blur and the Canny filter functions are applied.

In the second example, there are some differences that are required to work with the GPU. First, OpenCL must be activated with the `setUseOpenCL(true)` method. Second, **Unified Mats** (**UMat**) are used to allocate memory in the GPU (`UMat gpuFrame, gpuBW, gpuBlur, gpuEdges`). Third, the input image is copied from the RAM to GPU memory with the `cpuFrame.copyTo(gpuFrame)` method. Now, when the functions are used, if they have an OpenCL implementation, then these functions will be executed on the GPU. If some of these functions do not have an OpenCL implementation, the normal function will be executed on the CPU. In this example, the time elapse using the GPU programming (second example) is 10 times better:



Results of the preceding two examples

Going real time

One of the main advantages of GPU processing is to perform computations in a much faster way. This increase in speed allows you to execute heavy computational algorithms in real-time applications, such as stereo vision, pedestrian detection, optical flow, or face detection. The following **detectFaces** example shows you an application to detect faces from a video camera. This example also allows you to select between the **CPU** or **GPU processing** in order to compare the computational time.

In the OpenCV examples (`[opencv_source_code]/samples/cpp/facedetect.cpp`), a related face detector example can be found. For the following **detectFaces** example, the `detectFace.pro` project needs these libraries: `-lopencv_core300`, `-opencv_imgproc300`, `-lopencv_highgui300`, `-lopencv_videoio300`, and `lopencv_objdetct300`.

The **detectFaces** example uses the `ocl` module of OpenCV:

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

#include <iostream>
#include <stdio.h>

using namespace std;
using namespace cv;
using namespace cv::ocl;

int main(int argc, char * argv[])
{
    // 1- Set the initial parameters
    // Vector to store the faces
    vector<Rect> faces;
    CascadeClassifier face_cascade;
String face_cascade_name = argv[2];
    int face_size = 30;
    double scale_factor = 1.1;
    int min_neighbours = 2;
```

```
VideoCapture cap(0);
UMat frame, frameGray;
    bool finish = false;


    // 2- Load the file xml to use the classifier
    if (!face_cascade.load(face_cascade_name))
    {
        cout << "Cannot load the face xml!" << endl;
        return -1;
    }

    namedWindow("Video Capture");

    // 3- Select between the CPU or GPU processing
    if (argc < 2)
    {
          cout << "./detectFaces [CPU/GPU | C/G]" << endl;
          cout << "Trying to use GPU..." << endl;
setUseOpenCL(true);
    }
    else
    {
        cout << "./detectFaces trying to use " << argv[1] << endl;
        if(argv[1][0] == 'C')
            // Trying to use the CPU processing
            setUseOpenCL(false);
        else
            // Trying to use the GPU processing
            setUseOpenCL(true);
    }

    Rect r;
 double start_time, finish_time, start_total_time,       finish_total_
time;
    int counter = 0;


    // 4- Detect the faces for each image capture
    start_total_time = getTickCount();
    while (!finish)
    {
        start_time = getTickCount();
cap >> frame;
        if (frame.empty())
        {
            cout << "No capture frame --> finish" << endl;
            break;
        }

cvtColor(frame, frameGray, COLOR_BGR2GRAY);
equalizeHist(frameGray,frameGray);

        // Detect the faces
face_cascade.detectMultiScale(frameGray, faces, scale_factor, min_
neighbours, 0|CASCADE_SCALE_IMAGE, Size(face_size,face_size));

        // For each detected face
        for (int f = 0; f <faces.size(); f++)
        {
            r = faces[f];
            // Draw a rectangle over the face
rectangle(frame, Point(r.x, r.y), Point(r.x + r.width, r.y +
r.height), Scalar(0,255,0), 3);
        }

        // Show the results
imshow("Video Capture",frame);

        // Calculate the time processing
        finish_time = getTickCount();
cout << "Time per frame: " << (finish_time - start_time)/
getTickFrequency() << " secs" << endl;

        counter++;

        // Press Esc key to finish
        if(waitKey(1) == 27) finish = true;
    }

    finish_total_time = getTickCount();
cout << "Average time per frame: " << ((finish_total_time - start_
total_time)/getTickFrequency())/counter << " secs" << endl;

    return 0;
}
```

The code explanation

The first step sets the initial parameters, such as the xml file (`String face_cascade_name argv[2]`) that uses the classifier to detect the faces, the minimum size of each detected face (`face_size=30`), the scale factor (`scale_factor = 1.1`), and the minimum number of neighbors (`min_neighbours = 2`) to find a trade-off between true positive and false positive detections. You can also see the more important differences between the CPU and GPU source codes; you only need to use the **Unified Mat** (`UMat frame, frameGray`).

[There are other available xml files in the `[opencv_source_code]/data/haarcascades/` folder to detect different body parts such as eyes, lower bodies, smiles, and so on.]

The second step creates a detector using the preceding xml file to detect faces. This detector is based on a **Haar feature-based classifier** that is an effective object-detection method proposed by Paul Viola and Michael Jones. This classifier has a high accuracy of detecting faces. This step loads the xml file with the `face_cascade.load( face_cascade_name)` method.

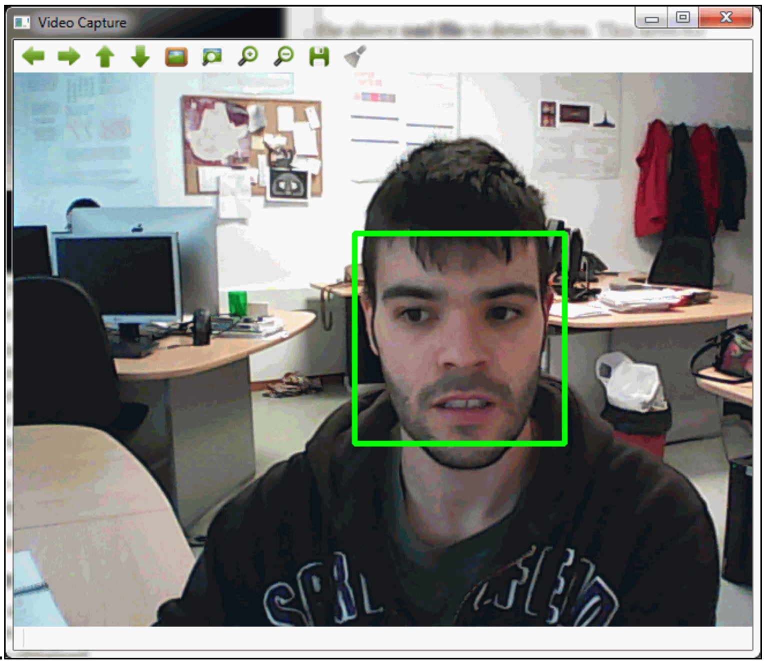
[You can find more detailed information about Paul Viola and Michael Jones method at `http://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework`.]

The third step allows you to select between the CPU or GPU processing (`setUseOpenCL(false)` or `setUseOpenCL(true)`, respectively). This example uses the standard command-line input arguments (`argv[1]`) to select this. The user can execute the following from the Windows console to select between CPU or GPU processing, respectively, and the classifier path:

```
<bin_dir>/detectFaces CPU pathClassifier
<bin_dir>/detectFaces GPU pathClassifier
```

If the user does not introduce an input argument, then GPU processing is used.

The fourth step detects faces for each image captured from the video camera. Before that, each captured image is converted to gray scale (`cvtColor(frame, frameGray, COLOR_BGR2GRAY)`) and its histogram is equalized (`equalizeHist(frameGray, frameGray)`). Afterwards, using the created face detector, the different faces are searched over the current frame using the `face_cascade.detectMultiScale(frameGray, faces, scale_factor, min_neighbours, 0|CASCADE_SCALE_IMAGE, Size(face_size,face_size))` multiscale detection method. Finally, a green rectangle is drawn over each detected face, and then it is displayed. A screenshot of this example running is shown in the following screenshot:



The preceding example detecting a face

The performance

In the preceding example, the computational time is calculated to compare the CPU and GPU processing. The average processing time per frame is obtained.

A big advantage of choosing GPU programming is performance. Therefore, the previous example calculates the time measurements to compare the speedups obtained with respect to the CPU version. The time is stored at the beginning of the program using the `getTickCount()` method. Afterwards, at the end of the program, the same function to estimate the time is used again. A counter is stored to know the number of iterations as well. Finally, the average processing time per frame is calculated. The preceding example has an average processing time per frame of 0.057 seconds per frame (or 17.5 FPS) using the GPU, whereas the same example using the CPU has an average processing time per frame of 0.335 seconds per frame (or 2.9 FPS). In conclusion, there is *a speed increment of 6x*. This increment is significant, especially when you only need to change a few lines of the code. However, it is possible to achieve a much higher rate of speed increments, which is related to the problem and even with how the kernels are designed.

Summary

In this chapter, you learned how to install OpenCV with OpenCL on your computer and develop applications using your computer devices, which are compatible with OpenCL, with the last version of OpenCV.

The first section explains what OpenCL is and the available SDKs. Remember that depending your computing devices, you will need a specific SDK to work correctly with OpenCL. In the second section, the installation process to install OpenCV with OpenCL is explained, and the AMD APP SDK has been used. In the last section, there are three examples using GPU programming (the second example also has a CPU version in order to compare them). In addition, in the last section, there is a computational comparative between CPU and GPU processing, where the GPU is shown to be six times faster than the CPU version.

Index