Глория Буэно Гарсия, Оскар Дениз Суарес,
Хосе Луис Эспиноса Аранда, Хесус Салидо Терсеро,
Исмаэль Серрано Грасиа, Ноэлия Валлез Энано

Обработка изображений с помощью OpenCV

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

Обработка изображений с помощью OpenCV

На простых и понятных примерах узнайте, как с помощью поразительных возможностей OpenCV создавать мощные приложения для обработки изображений

Глория Буэно Гарсия
Оскар Дениз Суарес
Хосе Луис Эспиноса Аранда
Хесус Салидо Терсеро
Исмаэль Серрано Грасиа
Ноэлия Валлез Энано

ДМК Пресс
издательство
Москва, 2016

УДК 004.932OpenCV
ББК 32.972.13
Г20

Г20 Глория Буэно Гарсия, Оскар Дениз Суарес,
Хосе Луис Эспиноса Аранда, Хесус Салидо Терсеро,
Исмаэль Серрано Грасиа, Ноэлия Валлез Энано

Обработка изображений с помощью OpenCV/ пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 210 с.: ил.

ISBN 978-5-97060-387-1

OpenCV является наиболее широко распространенной библиотекой компьютерного зрения. Она включает сотни готовых функций обработки изображений и используется как в академических учреждениях, так и в промышленности.

В этой книге на примерах демонстрируются основные алгоритмы обработки изображений, реализованные в OpenCV. Сначала рассказывается об установке библиотеки, описывается ее общая структура и приводятся простые примеры чтения и записи изображений и видео. Далее рассматривается фильтрация изображений и манипуляции с цветом. Вы узнаете о таких методах обработки, как ретуширование, очистка от шумов и создание HDR-изображений. В последней главе речь пойдет о повышении быстродействия за счет использования графических процессоров. Все рассмотренные темы иллюстрируются примерами.

Издание предназначено программистам, знакомым с языком C++ и желающим изучить методы обработки изображений с помощью библиотеки OpenCV.

ISBN 978-1-78328-765-9 (англ.)
ISBN 978-5-97060-387-1 (рус.)

ОГЛАВЛЕНИЕ

ОБ АВТОРАХ

Глория Буэно Гарсия имеет степень доктора по машинному зрению, присвоенную университетом Ковентри, Великобритания. Она работала в должности старшего научного сотрудника в нескольких исследовательских центрах, в частности в подразделении UMR 7005 Национального научно-исследовательского центра при университете Луи Пастера в Страсбурге (Франция), в компании Gilbert Gilkes & Gordon Technology (Великобритания) и в центре научно-технических исследований CEIT San Sebastian (Испания). Она автор двух патентов, одного зарегистрированного типа программ и более 100 рецензированных публикаций. Ее научные интересы лежат в области двумерной и трехмерной мультимодальной обработки изображений и искусственного интеллекта. Возглавляет исследовательскую группу VISILAB в университете Кастилия–Ла-Манча. Является соавтором книги по программированию для мобильных устройств с применением OpenCV «OpenCV Essentials», вышедшей в издательстве Packt Publishing.

Посвящается нашим сыновьям в качестве компенсации за то время, когда мы не могли поиграть с ними, и нашим родителям за постоянную поддержку безо всяких условий. Спасибо от Глории и Оскара.

Научные интересы **Оскара Дениза Суареса** сосредоточены в основном в области машинного зрения и распознавания образов. Он автор более 50 работ в рецензируемых журналах и на конференциях. Занял второе место на конкурсе работ докторов наук по машинному зрению и распознаванию образов, проводимом AERFAI (Испанская ассоциация по распознаванию образов и анализу изображений), а также получил награду от компании Innocentive Inc. за лучшую программу по обработке и переформатированию файлов изображений. Его работы используются такими передовыми компаниями, как Existor, Gliif, Tapmedia, E-Twenty и другими. Он внес вклад в разра-

ботку OpenCV. В настоящее время работает доцентом в университете Кастилия–Ла-Манча и сотрудничает с VISILAB. Является старшим членом IEEE и участвует в организациях AAAI, SIANI, CEA-IFAC, AEPIA и AERFAI-IAPR. Исполняет функции академического редактора журнала PLoS ONE. По приглашению занимался исследовательской работой в университете Карнеги-Меллон, Имперском колледже Лондона и в компании Leica Biosystems. Соавтор двух книг по OpenCV.

Хосе Луис Эспиноса Аранда имеет степень доктора информатики, присвоенную университетом Кастилия–Ла-Манча. Он вышел в финал испанского конкурса Certamen Universitario Arquímedes de Introducción a la Investigación científica 2009 года со своим дипломным проектом. В сферу его научных интересов входят машинное зрение, эвристические алгоритмы и исследование операций. В настоящее время работает в группе VISILAB младшим научным сотрудником, занимаясь разработками в области машинного зрения.

Посвящаю своим родителям и братьям.

Хесус Салидо Терсеро получил степень доктора электротехники в 1996 году в Мадридском политехническом университете (Испания). Затем два года работал приглашенным научным сотрудником в Институте робототехники (университет Карнеги-Меллон, Питтсбург, США), занимаясь кооперативными системами из нескольких роботов. После возвращения в испанский университет Кастилия–Ла-Манча ведет курсы по робототехнике и промышленной информатике, а также занимается исследованиями в области интеллектуальных систем машинного зрения. Последние три года направлял основные усилия на разработку приложений машинного зрения для мобильных устройств. Соавтор книги по программированию мобильных устройств с применением OpenCV.

Посвящаю тем, кому обязан всем в своей жизни: родителям, Саграрио и Марии.

Исмаэль Серрано Грасиа получил ученую степень по информатике в 2012 году в университете Кастилия–Ла-Манча. Удостоен выс-

шей оценки за дипломный проект, посвященный распознаванию людей. В этом приложении используются камеры глубины, программируемые с помощью библиотеки OpenCV. В настоящее время трудится над докторской диссертацией, получив грант от испанского Министерства науки и исследований. По совместительству работает младшим научным сотрудником в группе VISILAB, занимаясь различными вопросами машинного зрения.

Посвящаю родителям, которые дали мне возможность получить образование и поддерживают на протяжении всей жизни. А также своему научному руководителю, доктору Оскару Денизу, моему другу, наставнику и помощнику. И еще друзьям и своей девушке, которая всегда помогала мне и верила, что я смогу это сделать.

Ноэлия Валлез Энано с детства обожала компьютеры, но свою первую машину получила только в пятнадцать лет. В 2009 году она с высшими баллами окончила курс информатики в университете Кастилия–Ла-Манча. Начав работать в группе VISILAB, занималась проектом в области CAD-систем для маммографии и обработкой медицинских электронных данных. Затем получила степень магистра физико-математических наук и приступила к работе над докторской диссертацией, посвященной методам обработки изображений и распознавания образов. Также любит преподавать и заниматься различными вопросами искусственного интеллекта.

О РЕЦЕНЗЕНТАХ

Вальтер Лучетти, известный в Интернете под ником Myzhar, – инженер-компьютерщик из Италии, специализируется в робототехнике и робототехническом восприятии. Получил степень лауреата в 2005 года за написанную в Пизанском исследовательском центре «E.Piaggio» диссертацию по 3D-маппингу с помощью двумерного лазера, управляемого серводвигателем, когда на трехмерную поверхность проецируются RGB-данные. В ходе работы над диссертацией он впервые познакомился с OpenCV – это было в начале 2004 года, когда библиотека еще находилась в зачаточном состоянии. После защиты занимался разработкой низкоуровневых встраиваемых систем и высокоуровневых систем для настольных компьютеров. Знания в области машинного зрения и машинного обучения он серьезно обогатил, работая инженером-исследователем в центре передовой робототехники Густаво Стефанини в Ла Специи (Италия), филиале лаборатории PERCRO Пизанской высшей школы Sant'Anna.

В настоящее время работает в программной индустрии, пишет прошивки для встраиваемых ARM-систем, программы для настольных компьютеров на базе библиотеки Qt и разрабатывает интеллектуальные алгоритмы для систем видеонаблюдения на основе OpenCV и CUDA. Также работает над личным робототехническим проектом: MyzharBot. Это гусеничный наземный подвижный робот, который пользуется машинным зрением для распознавания препятствий, анализа и исследования окружающей среды. Управление роботом осуществляется с помощью алгоритмов, основанных на ROS, CUDA и OpenCV. Познакомиться с проектом можно на сайте `http://myzharbot.robot-home.it`.

Андре де Суза Морейра получил степень магистра информатики со специализацией в компьютерной графике от папского католического университета Рио де Жанейро (Бразилия).

Закончил федеральный университет штата Мараньян (Бразилия) со степенью бакалавра информатики. Во время работы над дипломом входил в состав исследовательской группы Labmint и занимался об-

работкой медицинских изображений, а конкретно распознаванием и диагностикой рака молочной железы.

В настоящее время работает исследователем и системным аналитиком в институте Текграф (Instituto Tecgraf), одном из главных научно-исследовательских центров компьютерной графики в Бразилии. С 2007 года активно использует PHP, HTML и CSS, а сейчас применяет для разработки C++11/C++14, SQLite, Qt, Boost, и OpenGL. Дополнительные сведения можно найти на его персональном сайте по адресу `www.andredsm.com`.

Марвин Смит в настоящее время работает инженером-программистом в оборонной промышленности и специализируется в области фотограмметрии и дистанционного зондирования. Степень бакалавра информатики получил в университете Невады в Рино. В область его технических интересов входят высокопроизводительные вычисления, распределенная обработка изображений и многоспектральные формирователи изображений. До перехода в оборонную отрасль Марвин проходил практику в группе интеллектуальной робототехники в исследовательском центре НАСА в Амесе и в центре ходовых испытаний в Неваде.

ПРЕДИСЛОВИЕ

OpenCV является, наверное, самой широко распространенной библиотекой компьютерного зрения. Она включает сотни готовых функций обработки изображений и используется как в академических учреждениях, так и в промышленности. Камеры становятся все дешевле, а спрос на обработку изображений растет, поэтому спектр приложений OpenCV как на настольных, так и на мобильных платформах постоянно расширяется.

В этой книге на примерах демонстрируются основные алгоритмы обработки изображений, реализованные в OpenCV. Есть книги, где объясняется теория, лежащая в основе OpenCV. Есть и такие, где приводятся примеры больших, почти законченных приложений. Мы же ориентируемся на читателей, которым нужны – и притом быстро – простые для понимания работающие примеры, на которые можно навешивать дополнительные функции.

В первой, вступительной, части объясняется, как установить библиотеку, описывается ее общая структура и приводятся простые примеры чтения и записи изображений и видео. Далее рассматриваются следующие темы: работа с изображениями и видео, основные инструменты обработки изображений, коррекция и улучшение изображений, цвет, обработка видео, вычислительная фотография. И напоследок обсуждаются более сложные темы, в частности, повышение быстродействия за счет использования графических процессоров. На протяжении всей книги описываются новые функции и методы, появившиеся в последней основной версии, OpenCV 3.

Структура книги

В главе 1 «Работа с файлами изображений и видео» описывается, как читать такие файлы. Демонстрируются также основные средства взаимодействия с пользователем, которые позволяют изменять значения параметров, выбирать интересующие области и т. д.

В главе 2 «Инструменты обработки изображений» описываются основные структуры данных и процедуры, которые понадобятся в последующих главах.

В главе 3 «Коррекция и улучшение изображений» речь пойдет о преобразованиях, которые обычно применяются для исправления дефектов изображения. Здесь рассматриваются фильтры, точечные преобразования с использованием справочных таблиц, геометрические преобразования и алгоритмы ретуширования и очистки от шумов.

Глава 4 «Работа с цветом» посвящена месту цвета в обработке изображений. Здесь объясняется, как устроены различные цветовые пространства и как осуществляется цветоперенос с одного изображения на другое.

В главе 5 «Обработка видео» рассматриваются методы работы с последовательностями изображений, т. е. видео. Здесь будут описаны реализации алгоритмов стабилизации видео, сверхвысокого разрешения и сшивки.

В главе 6 «Вычислительная фотография» объясняется, как читать HDR-изображения и выполнять для них тональную компрессию.

Глава 7 «Ускорение обработки изображений» посвящена очень важному для обработки изображений вопросу: быстродействию. Современные графические процессоры (GPU) – наилучшая на сегодняшний день технология ускорения длительных операций над изображениями.

Что необходимо для чтения этой книги

Цель этой книги – научить применению OpenCV для обработки изображений на ряде практических примеров. Используется последняя версия – OpenCV 3.0.

Каждая глава содержит несколько готовых к работе примеров, иллюстрирующих излагаемый материал. Таким образом, мы стремимся как можно быстрее предоставить код, над которым можно надстраивать дополнительный функционал.

Для проработки примеров нужно только бесплатное ПО. Все примеры разрабатывались и тестировались в бесплатной интегрированной среде Qt Creator с использованием компилятора GNU/GCC. Для конфигурирования процесса сборки библиотеки OpenCV используется платформенно-независимая программа CMake. Для примеров

применения ускорителей в главе 7 понадобится еще бесплатный комплект средств разработки OpenCL SDK.

Предполагаемая аудитория

Эта книга рассчитана на читателей, уже знакомых с языком C++ и желающих научиться обработке изображений с помощью библиотеки OpenCV. Предполагается, что у читателя имеются минимальные теоретические познания в области обработки изображений. Мы не рассматриваем вопросы, в большей степени относящиеся к компьютерному зрению, например: выделение признаков, обнаружение объектов, трассировка или машинное обучение.

Обозначения и графические выделения

В этой книге для выделения семантически различной информации применяются различные стили. Ниже приведены примеры стилей с пояснениями.

Фрагменты кода в тексте, имена папок и файлов, имена системных переменных, URL-адреса и данные, вводимые пользователем, выглядят следующим образом: «С каждым модулем ассоциирован файл-заголовок (например, `core.hpp`)».

Отдельно стоящие фрагменты кода набраны так:

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

int main(int argc, char *argv[])
{
  Mat frame; // контейнер фрейма
```

Чтобы привлечь внимание к части кода, строки или отдельные слова выделяются полужирным шрифтом:

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

using namespace std;
```

```
using namespace cv;

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

Текст, который вводится на консоли или выводится на консоль, напечатан следующим образом:

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

Новые термины и **важные слова** набраны полужирным шрифтом. Также выделяются элементы интерфейса: «Для перехода к следующему экрану нажмите кнопку **Next»**.

Предупреждения и важные замечания оформлены так.

Советы и рекомендации выглядят так.

Отзывы

Мы всегда рады отзывам читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или, быть может, не понравилось. Читательские отзывы важны для нас, так как помогают выпускать книги, из которых вы черпаете максимум полезного для себя.

Чтобы отправить обычный отзыв, просто пошлите письмо на адрес `feedback@packtpub.com`, указав название книги в качестве темы.

Если вы являетесь специалистом в некоторой области и хотели бы стать автором или соавтором книги, познакомьтесь с инструкциями для авторов по адресу `www.packtpub.com/authors`.

Поддержка клиентов

Счастливым обладателям книг Packt мы можем предложить ряд услуг, которые позволят извлечь из приобретения максимум пользы.

Загрузка кода примеров

Вы можете скачать код примеров ко всем книгам издательства Packt, купленным на сайте `http://www.PacktPub.com`. Если книга была

куплена в другом месте, зайдите на страницу http://www.PacktPub.com/support, зарегистрируйтесь, и мы отправим файлы по электронной почте.

Загрузка цветных иллюстраций

Мы также предлагаем PDF-файл с цветными изображениями снимков экрана и диаграмм, встречающихся в книге. Имея цветные изображения, вы сможете полнее оценить, как меняется вывод. Загрузить файл можно со страницы https://www.packtpub.com/sites/default/files/downloads/ImageProcessingwithOpenCV_Graphics.pdf.

Опечатки

Мы проверяли содержимое книги со всем тщанием, но какие-то ошибки все же могли проскользнуть. Если вы найдете в нашей книге ошибку, в тексте или в коде, пожалуйста, сообщите нам о ней. Так вы избавите других читателей от разочарования и поможете нам сделать следующие издания книги лучше. При обнаружении опечатки просьба зайти на страницу http://www.packtpub.com/ submit-errata, выбрать книгу, щелкнуть по ссылке **Errata Submission Form** и ввести информацию об опечатке. Проверив ваше сообщение, мы поместим информацию об опечатке на нашем сайте или добавим ее в список замеченных опечаток в разделе Errata для данной книги. Список подтвержденных опечаток можно просмотреть, введя название книги в поле поиска на странице http://www.packtpub.com/books/content/support. Информация появится в разделе **Errata**.

Нарушение авторских прав

Незаконное размещение защищенного авторским правом материала в Интернете – проблема для всех носителей информации. В издательстве Packt мы относимся к защите прав интеллектуальной собственности и лицензированию очень серьезно. Если вы обнаружите незаконные копии наших изданий в любой форме в Интернете, пожалуйста, незамедлительно сообщите нам адрес или название веб-сайта, чтобы мы могли предпринять соответствующие меры.

Просим отправить ссылку на вызывающий подозрение в пиратстве материал по адресу copyright@packtpub.com.

Мы будем признательны за помощь в защите прав наших авторов и содействие в наших стараниях предоставлять читателям полезные сведения.

Вопросы

Если вас смущает что-то в этой книге, вы можете связаться с нами по адресу `questions@packtpub.com`, и мы сделаем все возможное для решения проблемы.

ГЛАВА 1. Работа с файлами изображений и видео

В этой главе мы познакомимся с установкой OpenCV и напишем первые, очень простые программы. Будут рассмотрены следующие вопросы:

- краткое введение в OpenCV для начинающих, сопровождаемое пошаговой инструкцией по установке библиотеки;
- краткий обзор структуры каталогов OpenCV после ее установки на локальный диск;
- рекомендации по созданию проектов в некоторых распространенных средах программирования;
- как пользоваться функциями чтения и записи изображений и видео;
- описание библиотечных функций для реализации интерфейса приложения, в том числе взаимодействия с помощью мыши, рисования примитивов и поддержки библиотеки Qt.

Введение в OpenCV

Библиотека **OpenCV (Open Source Computer Vision)**, первоначально разработанная компанией Intel, теперь является бесплатной, кроссплатформенной библиотекой обработки изображений в реальном времени и стала стандартом де факто для всего, что связано с компьютерным зрением. Ее первая версия была выпущена в 2000 году на условиях **лицензии BSD**, но с тех пор функциональность существенно обогатилась благодаря усилиям научного сообщества. В 2012 году некоммерческий фонд OpenCV.org взял на себя задачу ведения сайта для поддержки разработчиков и пользователей.

На момент написания этой книги вышла новая основная версия OpenCV 3.0, но пока она находится на стадии бета-тестирования. В этой книге мы расскажем о самых существенных изменениях в этой версии.

OpenCV реализована для большинства популярных операционных систем, в т. ч. GNU/Linux, OS X, Windows, Android, iOS и некоторых других. Первая реализация была написана на языке **C**, но популярность библиотеки значительно возросла с выходом версии 2.0, переписанной на C++. Новые функции пишутся только на C++. Впрочем, в настоящее время имеются полнофункциональные интерфейсы между OpenCV и другими языками программирования, в частности Java, Python и MATLAB/Octave. Разработаны также обертки для таких языков, как C#, Ruby и Perl, чтобы популяризировать библиотеку среди программистов.

Стремясь добиться максимальной производительности при решении счетных (потребляющих много процессорного времени) задач машинного зрения, авторы OpenCV включили поддержку следующих возможностей:

- многопоточность при работе на многоядерных компьютерах за счет использования шаблонной библиотеки **Threading Building Blocks (TBB)**, разработанной Intel;
- подмножество библиотеки **Integrated Performance Primitives (IPP)** на процессорах Intel для резкого повышения производительности. Благодаря любезности Intel эти примитивы бесплатно включены в бета-версию 3.0;
- интерфейсы для работы с **графическими процессорами (GPU)** с применением архитектуры **Compute Unified Device Architecture (CUDA)** и языка **Open Computing Language (OpenCL)**.

OpenCV применяется в таких областях, как сегментация, выделение признаков в двумерных и трехмерных изображениях, идентификация объектов, распознавание лиц, трассировка движения, распознавание жестов, сшивка изображений, обработка изображений с широким динамическим диапазоном (**high dynamic range**, **HDR**), дополненная реальность и т. д. Кроме того, для поддержки некоторых из вышеперечисленных задач включен модуль, содержащий функции статистического машинного обучения.

Загрузка и установка OpenCV

OpenCV можно бесплатно скачать с сайта `http://opencv.org`, где имеется как последняя версия (в настоящее время 3.0 beta), так и предыдущие.

Скачивая нестабильную версию, например 3.0 beta, всегда нужно помнить о возможности ошибок в программе.

На странице `http://opencv.org/downloads.html` имеются версии OpenCV для всех поддерживаемых платформ. Код библиотеки и информацию о нем можно получить из различных репозиториев (в зависимости от поставленной цели).

- Основной репозиторий (`http://sourceforge.net/projects/opencvlibrary`), ориентированный на конечных пользователей. Содержит двоичные версии библиотеки и готовые к компиляции исходные коды для конечной платформы.
- Репозиторий тестовых данных (`https://github.com/itseez/opencv_extra`), содержащий наборы данных для тестирования некоторых библиотечных модулей.
- Репозиторий дополнений (`http://github.com/itseez/opencv_contrib`), содержащий исходный код дополнений и самых передовых возможностей, добавленный сторонними разработчиками. Он не так хорошо протестирован, как код в стволовой ветви, и может содержать ошибки.

В последней версии OpenCV 3.0 дополнительные модули в основной пакет не включены. Их следует скачивать отдельно и явно включать в процесс компиляции с помощью соответствующих флагов. Будьте внимательны, потому что некоторые модули зависят от сторонних программ, не входящих в состав OpenCV.

- Сайт документации (`http://docs.opencv.org/master/`) по каждому модулю, включая дополнительные.
- Репозиторий разработчиков (`https://github.com/Itseez/opencv`), содержащий текущую разрабатываемую версию би-

блиотеки. Предназначен для разработчиков основной функциональности библиотеки и «нетерпеливых» пользователей, которым хочется попробовать последние обновления до их официального выпуска.

В отличие от платформ GNU/Linux и OS X, на которых OpenCV распространяется только в исходном виде, для Windows существуют откомпилированные (с помощью компиляторов Microsoft Visual C++ v10, v11 и v12) версии библиотеки. Откомпилированные версии совместимы с компиляторами Microsoft. Но если требуется разрабатывать проект в другой среде компиляции (например, GNU GCC), то библиотеку придется перекомпилировать для нее.

Самый быстрый способ начать работу с OpenCV – воспользоваться откомпилированной версией, включенной в состав дистрибутива. Но лучше бы самостоятельно собрать библиотеку с настройками, оптимальными для конкретной платформы и планируемого использования. В этой главе приведены инструкции по сборке и установке OpenCV для Windows. Дополнительные сведения о настройке библиотеки в Linux можно найти на страницах `http://docs.opencv.org/doc/tutorials/introduction/linux_install` и `https://help.ubuntu.com/community/OpenCV`.

Получение компилятора и настройка CMake

Для кросс-платформенной разработки с применением OpenCV лучше всего использовать **комплект инструментов GNU** (включающий gmake, g++ и gdb). Его нетрудно получить для популярных операционных систем. Мы предпочитаем среду разработки, состоящую из комплекта инструментов GNU и кросс-платформенного **каркаса Qt**, в который входит библиотека Qt и **интегрированная среда разработки (IDE)** Qt Creator. Каркас Qt можно бесплатно скачать с сайта `http://qt-project.org/`.

После установки компилятора в Windows не забудьте добавить в переменную окружения `PATH` путь к исполняемому файлу компилятора, например `C:\Qt\Qt5.2.1\5.2.1\mingw48_32\bin` в случае каркаса Qt. Для Windows имеется средство **Rapid Environment Editor** (на сай-

те `http://www.rapidee.com`), которое позволяет удобно изменять `PATH` и другие переменные окружения.

Для управления процессом сборки OpenCV способом, не зависящим от компилятора, рекомендуется система CMake, бесплатная кроссплатформенная программа с открытым исходным кодом, доступная на сайте `http://www.cmake.org/`.

Настройка OpenCV с помощью CMake

После копирования исходного кода библиотеки на локальный диск необходимо настроить make-файлы для ее компиляции. CMake – основной инструмент конфигурирования процесса установки OpenCV. Его можно использовать как из командной строки, так и из графического интерфейса.

Ниже перечислены шаги конфигурирования OpenCV с помощью CMake:

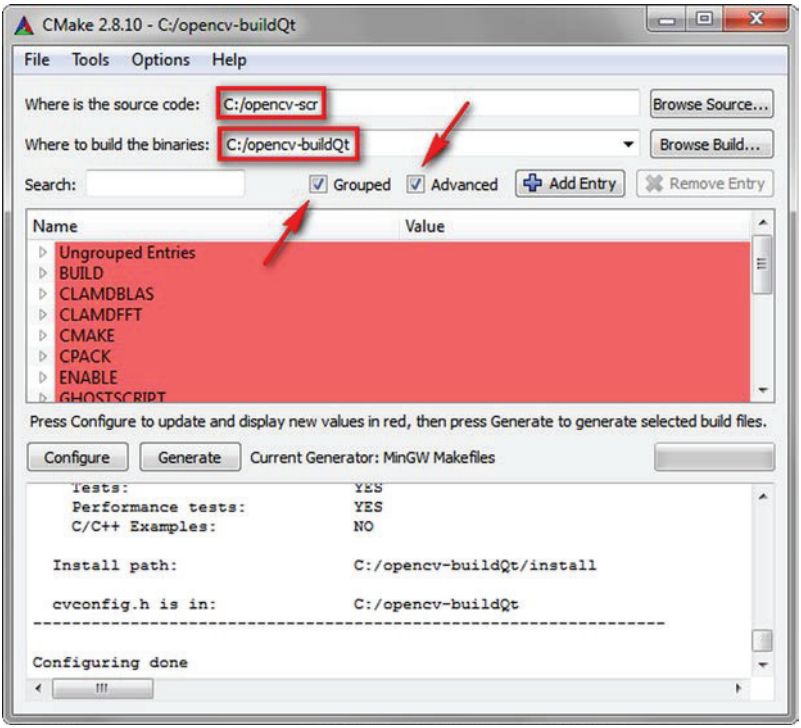
1. Задать начальный (ниже он обозначается `OPENCV_SRC`) и конечный (`OPENCV_BUILD`) каталог. В конечный каталог будут помещены откомпилированные двоичные файлы.
2. Отметить флажки **Grouped** и **Advanced** и нажать кнопку **Configure**.
3. Выбрать компилятор (например, предлагаемый по умолчанию компилятор GNU, MSVC и т. д.)
4. Включить нужные и отключить ненужные возможности.
5. Нажать кнопку **Configure** и повторять шаги 4 и 5, пока не перестанут выдаваться сообщения об ошибках.
6. Нажать кнопку **Generate** и выйти из CMake.

На рисунке ниже показано главное окно CMake, в котором заданы начальный и конечный каталог и отмечены флажки для группировки доступных функций.

Для краткости мы обозначаем начальный и конечный каталоги OpenCV в процедуре установки `OPENCV_SRC` и `OPENCV_BUILD` соответственно. Но помните, что все каталоги должны соответствовать вашей машине.

До начала конфигурирования CMake находит установленные на машине компиляторы и определяет другие свойства локального окружения, существенные для сборки OpenCV. На рисунке ниже по-

казано главное окно **CMake** по завершении этапа предварительного конфигурирования, а красным цветом выделены сгруппированные параметры.



Главное окно CMake до начала конфигурирования

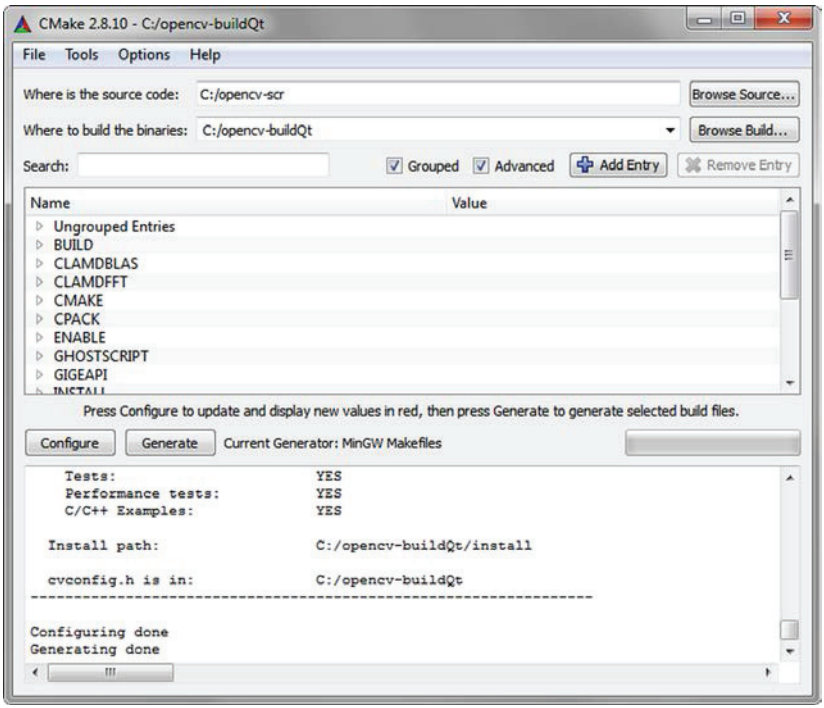
Можно вообще не менять параметры по умолчанию и продолжить процесс конфигурирования. Однако лучше бы для удобства установить некоторые параметры.

- `BUILD_EXAMPLES`: собирать примеры использования OpenCV.
- `BUILD_opencv_<module_name>`: включить в процесс сборки модуль с указанным именем.
- `OPENCV_EXTRA_MODULES_PATH`: используется, если нужен какой-то дополнительный модуль; задайте путь к исходному коду дополнительных модулей (например, `C:/opencv_contrib-master/modules`).
- `WITH_QT`: включить в библиотеку функциональность Qt.

- `WITH_IPP`: этот параметр по умолчанию включен. В текущую версию OpenCV 3.0 включено подмножество библиотеки **Intel Integrated Performance Primitives (IPP)**, ускоряющее работу библиотеки.

При компиляции новой версии OpenCV 3.0 (beta) будьте внимательны – сообщалось о неожиданных ошибках, связанных с включением IPP (т. е. при компиляции с параметрами по умолчанию). Мы рекомендуем выключить параметр `WITH_IPP`.

Если на этапе конфигурирования (шаги 4 и 5) CMake не выдает ошибок, то можно приступать к генерации окончательных make-файлов для процесса сборки. На рисунке ниже показано главное окно CMake после завершения шага генерации без ошибок.



Главное окно CMake после завершения конфигурирования

Компиляция и установка библиотеки

Следующий шаг после генерации make-файлов – компиляция библиотеки с помощью подходящей программы. Обычно эта программа запускается из командной строки, когда текущим является конечный каталог (заданный на этапе конфигурирования в CMake). Например, в Windows компиляция из командной строки запускается так:

```
OPENCV_BUILD>mingw32-make
```

При этом начинается процесс сборки с использованием make-файлов, сгенерированных CMake. Процесс компиляции обычно занимает несколько минут. Если компиляция завершится без ошибок, то можно переходить к установке:

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

Эта команда копирует исполняемые файлы OpenCV в каталог `OPENCV_BUILD\install`.

Если во время компиляции что-то пойдет не так, то нужно перезапустить CMake и изменить выбранные ранее параметры, а затем заново сгенерировать make-файлы.

По завершении установки нужно добавить путь к каталогу двоичных файлов библиотеки (например, в Windows созданные DLL-файлы находятся в каталоге `OPENCV_BUILD\install\x64\mingw\bin`) в переменную окружения **PATH**. Иначе при попытке выполнить любой исполняемый файл, собранный с OpenCV, будет выдано сообщение о том, что не найдена библиотека.

Чтобы проверить, успешно ли завершился процесс установки, можно попробовать выполнить какой-нибудь пример, откомпилированный вместе с библиотекой (если в CMake был задан параметр `BUILD_EXAMPLES`). Код примеров (на C++) находится в каталоге `OPENCV_BUILD\install\x64\mingw\samples\cpp`.

Краткие инструкции по установке OpenCV относятся к Windows. Подробное описание необходимых условий для установки в Linux можно найти на странице `http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html`. И хотя это пособие относится к OpenCV 2.0, почти все написанное в нем справедливо и для версии 3.0.

Структура каталогов OpenCV

После установки OpenCV в каталоге `OPENCV_BUILD\install` будут находиться файлы трех типов.

- **Заголовки.** Они находятся в подкаталоге `OPENCV_BUILD\install\include` и используются при разработке новых проектов.
- **Двоичные файлы библиотеки.** Это статические или динамические библиотеки (в зависимости от режима сборки, заданного в CMake), содержащие функционал отдельных модулей. Они находятся в подкаталоге `bin` (например, `x64\mingw\bin`, если использовался компилятор GNU).
- **Двоичные файлы примеров.** Это исполняемые файлы примеров использования библиотеки. Их исходный код можно найти в соответствующем каталоге (например, `OPENCV_SRC\sources\samples`).

OpenCV имеет модульную структуру, т. е. пакет включает статическую или динамическую (DLL) библиотеку для каждого модуля. Официальную документацию по модулям можно найти по адресу `http://docs.opencv.org/master/`. Перечислим основные модули.

- `core`: базовые функции, используемые всеми остальными модулями, а также фундаментальные структуры данных, включая тип многомерного массива `Mat`.
- `highgui`: простые средства организации **пользовательского интерфейса**. При сборке библиотеки с поддержкой Qt (параметр `WITH_QT`) обеспечивается совместимость с этим каркасом.
- `imgproc`: функции обработки изображений, в том числе фильтрация (линейная и нелинейная), геометрические преобразования, преобразования цветового пространства, вычисление гистограмм и т. д.
- `imgcodecs`: простой интерфейс для чтения и записи изображений.

Обращайте внимание на изменения, внесенные в версии OpenCV 3.0, т. к. функциональность частично перемещена из одного модуля в другой (например, функции чтения и записи изображений теперь находятся не в `highgui`, а в `imgcodecs`).

- `photo`: включает средства вычислительной фотографии, в т. ч. ретуширование, очистку от шумов, обработку изображений с **широким динамическим диапазоном (HDR)** и другие.
- `stitching`: нужен для сшивки изображений.
- `videoio`: содержит простой интерфейс для захвата видео и видеокодеки.
- `video`: функциональность для анализа видео (оценка движения, выделение заднего плана, трассировка объектов).
- `features2d`: функции для обнаружения признаков (углов и плоскостных объектов), их описания, сопоставления и т. д.
- `objdetect`: функции для обнаружения объектов и некоторые предопределенные детекторы (например, лиц, глаз, улыбок, людей, автомобилей и т. д.).

Назовем еще некоторые модули: `calib3d` (калибровка камеры), `flann` (кластеризация и поиск), `ml` (машинное обучение), `shape` (вычисление расстояния между фигурами и сопоставление фигур), `superres` (сверхвысокое разрешение), `video` (анализ видео), `videostab` (стабилизация видео).

Начиная с версии 3.0 beta, новые дополнительные модули распространяются в отдельном пакете (`opencv_contrib-master.zip`), который можно скачать по адресу `https://github.com/itseez/opencv_contrib`. Чтобы работать с функциями, которые находятся в этих модулях, нужно отчетливо понимать, что они делают. Краткий обзор новой функциональности, включенной в версию OpenCV 3.0, имеется в документе `http://opencv.org/opencv-3-0-beta.html`.

Создание проекта, включающего OpenCV

В этой книге предполагается, что основным языком для программирования приложений для обработки изображений является C++, хотя существуют интерфейсы и к другим языкам (например, Python, Java, MATLAB/Octave и т. д.).

В этом разделе объясняется, как разрабатывать приложения с использованием C++ API в простой и удобной кросс-платформенной среде.

Общие замечания об использовании библиотеки

Для разработки приложения OpenCV на **C++** необходимо:

- включить в программу заголовочные файлы, содержащие объявления;
- скомпоновать с двоичными библиотеками OpenCV для получения конечного исполняемого файла.

Заголовочные файлы OpenCV находятся в каталоге `OPENCV_BUILD\install\include\opencv2`, где каждому модулю соответствует отдельный заголовок (с расширением `.hpp`). Для включения заголовочного файла служит директива `#include`:

```
#include <opencv2/<module_name>/<module_name>.hpp>
// Включить заголовочный файл для каждого используемого в программе модуля
```

Эту директиву можно использовать для включения всех заголовков, необходимых программе. С другой стороны, если включен заголовок `opencv.hpp`, то все остальные заголовки будут включены автоматически:

```
#include <opencv2/opencv.hpp>
// Включение в программу сразу всех заголовочных файлов OpenCV
```

Напомним, что все установленные локально модули определены в заголовочном файле `OPENCV_BUILD\install\include\opencv2\opencv_modules.hpp`, который автоматически генерируется в процессе сборки OpenCV.

Использование директивы `#include` еще не дает гарантии правильного включения заголовочных файлов, потому что компилятору необходимо сообщить, где эти файлы искать. Для этого компилятору передается специальный аргумент с указанием местоположения файлов (для компиляторов GNU он называется `-I <location>`).

Компоновщику необходимо указать, в каких библиотеках (статических или динамических) находятся функции OpenCV. Обычно для этого служат два разных аргумента: местоположение библиотек (`-L<location>` в случае компиляторов GNU) и имя каждой библиотеки (`-l<module_name>`).

Полный список аргументов можно найти в онлайновой документации по GNU GCC и Make по адресам `https://gcc.gnu.org/onlinedocs/` и `https://www.gnu.org/software/make/manual/`.

Средства для разработки новых проектов

Для создания собственных приложений OpenCV на **C++** необходимы:

- **заголовочные файлы и двоичные библиотеки OpenCV:** разумеется, мы должны предварительно откомпилировать OpenCV и все библиотеки, от которых она зависит, причем тем же компилятором, который будет использоваться для создания пользовательского приложения;
- **компилятор C++:** хорошо бы иметь ряд дополнительных инструментов, например: редактор кода, отладчик, менеджер проектов, менеджер процесса сборки (CMake или нечто подобное), систему управления версиями (Git, Mercurial, SVN и т. п.), инспектор классов. Обычно всё это входит в состав так называемой интегрированной среды разработки (IDE);
- **вспомогательные библиотеки:** другие библиотеки, необходимые для работы приложения, например графическая, статистическая и т. д.

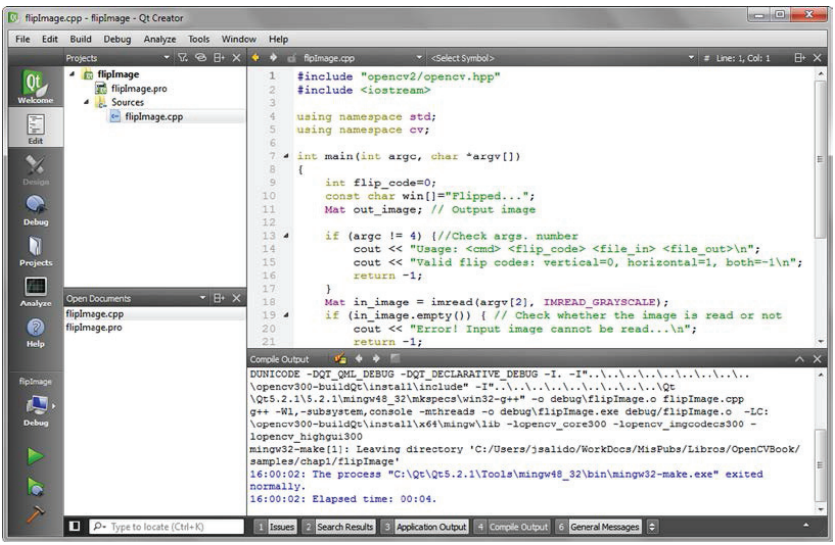
Чаще всего для создания приложений OpenCV на C++ используются следующие пакеты программ:

- **Microsoft Visual C (MSVC):** он поддерживается только в Windows и очень хорошо интегрирован со средой разработки Visual Studio, хотя может быть интегрирован также с другими кросс-платформенными IDE, например Qt Creator или Eclipse. С последней версией OpenCV совместимы следующие версии MSVC: VC 10, VC 11, VC 12 (Visual Studio 2010, 2012, 2013);
- **набор компиляторов GNU GCC:** это кросс-платформенная система компиляции, разработанная в рамках проекта GNU. Для Windows она известна под названием MinGW (Minimal GNU GCC). С текущей версией OpenCV совместима версия GNU GCC 4.8. Этот комплект можно использовать с несколькими IDE, в т. ч. Qt Creator, Code::Blocks и Eclipse.

Для компиляции примеров из этой книги мы использовали MinGW 4.8 для Windows в сочетании с библиотекой Qt 5.2.1 и Qt Creator IDE (3.0.1). Кросс-платформенная библиотека Qt необходима для компиляции OpenCV с новыми возможностями пользовательского интерфейса.

Для Windows весь комплект Qt (включающий библиотеку Qt, а также Qt Creator и компилятор MinGW) можно скачать с сайта `http://qt-project.org/`. Он занимает приблизительно 700 МБ.

На рисунке ниже показано главное окно Qt Creator с различными панелями и представлениями.



Главное окно Qt Creator с некоторыми представлениями для разработки приложения OpenCV на C++

Создание приложения OpenCV на C++ в Qt Creator

Далее мы объясним, как создать проект в интегрированной среде Qt Creator. И для примера создадим небольшое приложение OpenCV.

Чтобы создать проект, нужно выбрать из меню команду **File | New File** или **File | Project...** и перейти в раздел **Non-Qt Project | Plain C++ Project**. Затем задаем имя проекта и место, где он будет храниться. На следующем шаге выбираем комплект инструментов (т. е. компилятор) – в нашем случае Desktop Qt 5.2.1 MinGW 32 bit – и место для сгенерированных двоичных файлов. Обычно используются две конфигурации сборки, или профиля: отладочный и выпускной. В профиле задаются флаги, необходимые для сборки и выполнения двоичных файлов.

При использовании Qt Creator создаются два специальных файла (с расширениями `.pro` и `.pro.user`), которые служат для настройки процессов сборки и выполнения. Процесс сборки определяется комплектом инструментов, выбранным при создании проекта. Если был выбран комплект **Desktop Qt 5.2.1 MinGW 32 bit**, то для сборки будут использоваться программы qmake и mingw32-make. qmake читает данные из файла `*.pro` и генерирует make-файл, управляющий сборкой для каждого профиля (т. е. `release` и `debug`). Программа qmake используется в среде Qt Creator как альтернатива CMake, чтобы упростить сборку проектов. Для автоматизации процесса генерации make-файлов ей достаточно нескольких строчек настроечной информации.

Ниже приведен пример файла `*.pro` (скажем, `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
```

Здесь показано, какие параметры нужны qmake для генерации make-файлов, управляющих процессом построения исполняемых программ. В начале каждой строки задается метка, определяющая параметр (`TARGET`, `CONFIG`, `SOURCES`, `INCLUDEPATH`, `LIBS`), а за ней – оператор добавления (`+=`) или исключения (`-=`) значения параметра. В нашем примере создается консольное приложение без участия Qt. Исполняемый файл называется `showImage.exe` (`TARGET`), а исходный –

showImage.cpp (SOURCES). Поскольку в этом проекте используется библиотека OpenCV, в последних двух метках указано местоположение заголовочных файлов (INCLUDEPATH) и необходимых библиотек OpenCV (LIBS): core, imgcodecs, highgui и imgproc. Отметим, что обратная косая черта в конце строки означает, что значение продолжается на следующей строке.

Подробное описание всех инструментов, используемых при разработке приложений Qt (включая Qt Creator и qmake), см. на сайте http://doc.qt.io/.

Чтение и запись файлов изображений

Для обработки изображения (например, фотографии или видеокадра) его нужно сначала получить, а затем «поиграться» с ним, применяя различные методы обработки сигналов для получения желаемого результата. В этом разделе мы покажем, как читать изображения из файлов с помощью функций OpenCV.

Основные элементы API

Класс Mat – основная структура данных в OpenCV для хранения и манипулирования изображениями. Он определен в модуле core. В OpenCV реализованы механизмы автоматического выделения и освобождения памяти под эту структуру. Однако от программиста все же требуется внимание, если структуры данных разделяют общий буфер памяти. Например, оператор присваивания не копирует содержимое памяти из одного объекта (Mat A) в другой (Mat B), а копирует лишь ссылку (адрес содержимого в памяти). Следовательно, изменение в каком-то одном объекте (A или B) затрагивает оба. Для дублирования содержимого памяти объекта Mat следует использовать функцию-член Mat::clone().

Многие функции OpenCV обрабатывают плотные одноканальные или многоканальные массивы, обычно представленные классом Mat. Но в некоторых случаях удобнее другие типы данных, например: std::vector<>, Matx<>, Vec<> или Scalar. Для этой цели OpenCV предоставляет прокси-клас-

сы InputArray и OutputArray, которые позволяют использовать любой из вышеперечисленных типов в параметрах функций.

Класс Mat применяется для представления плотных *n*-мерных одноканальных или многоканальных массивов. В нем можно хранить вещественные или комплексные векторы и матрицы, цветные или полутоновые изображения, гистограммы, облака точек и т. д.

Есть много способов создать объект Mat, но чаще всего используется конструктор, в котором задается размер и тип массива:

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

Начальные значения элементов массива можно задавать с помощью класса Scalar в виде стандартного вектора с четырьмя элементами (три компоненты RGB и канал прозрачности), если в массиве хранится изображение. Ниже показан пример использования класса Mat:

```
Mat img_A(4, 4, CV_8U, Scalar(255));
// Белое изображение:
// Одноканальный массив 4 x 4, содержащий 8-разрядные целые без
// знака (каждый элемент может принимать одно из 255 значений, что
// подходит для полутонового изображения; например, 255=белый)
```

Класс DataType определяет примитивные типы данных для OpenCV. Примитивным типом может быть bool, unsigned char, signed char, unsigned short, signed short, int, float, double или кортеж значений одного из примитивных типов. Любой примитивный тип можно определить идентификатором следующего вида:

```
CV_<bit depth>{U|S|F}C(<число каналов>)
```

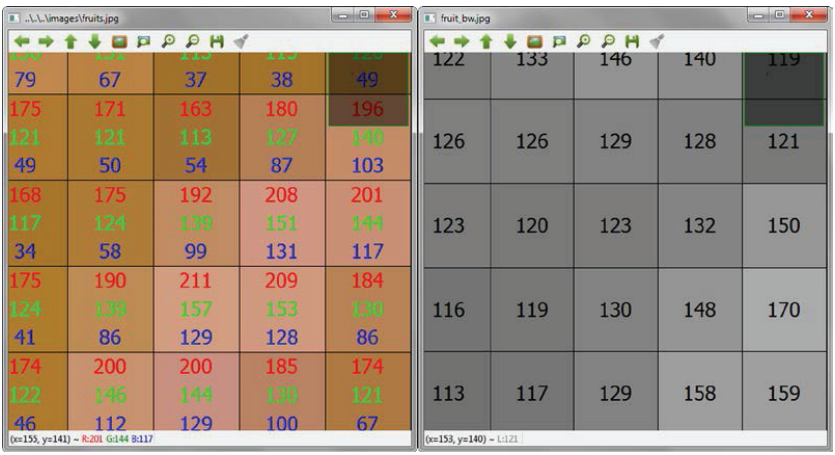
Здесь U, S, F обозначают unsigned, signed и float соответственно. Для одноканальных массивов применяется следующее перечисление, описывающее типы данных:

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

Отметим, что следующие три объявления эквивалентны: CV_8U, CV_8UC1, CV_8UC(1). Одноканальное объявление хорошо подходит для массивов целых чисел, описывающих полутоновые изображения, а трехканальное – для описания изображений с тремя цветовыми компонентами (например,

RGB, BRG, HSV и т. д.). Для операций линейной алгебры можно использовать массивы типа `float` (F).

Все перечисленные выше типы данных можно определить и для многоканальных массивов (до 512 каналов). На рисунке ниже показано внутреннее представление изображения с одним каналом (`CV_8U`, `grayscale`) и того же изображения с тремя каналами (`CV_8UC3`, `RGB`). Эти снимки экрана получены путем увеличения изображения, отображаемого в окне исполняемой программы, входящей в дистрибутив OpenCV (пример **showImage**):



8-разрядное представление цветного (в формате RGB) и полутонового изображения

Важно отметить, что для правильного сохранения RGB-изображения с помощью функций OpenCV оно должно храниться в памяти так, чтобы каналы следовали в порядке BGR. Аналогично после считывания RGB-изображения из файла каналы в памяти будут представлены в порядке BGR. Кроме того, необходим дополнительный четвертый канал (альфа), описывающий прозрачность. В случае RGB-изображений чем больше значение целого числа, тем ярче пиксель или – если речь идет об альфа-канале – тем выше прозрачность.

Все классы и функции OpenCV находятся в пространстве имен `cv`, поэтому у нас есть две возможности:

- добавить объявление `using namespace cv` после всех заголовочных файлов (так мы и поступаем в примерах);

- добавлять префикс `cv::` при каждом использовании класса, функции или структуры данных OpenCV. Делать так рекомендуется, если внешние имена, объявленные в OpenCV, конфликтуют с именами из часто используемых библиотек: **стандартной библиотеки шаблонов (STL)** или других.

Поддерживаемые форматы графических файлов

OpenCV поддерживает большинство графических форматов. Но для некоторых нужны сторонние (бесплатные) библиотеки. Перечислим основные форматы, поддерживаемые OpenCV:

- растровые изображения в Windows (`*.bmp`, `*dib`);
- переносимые форматы файлов (`*.pbm`, `*.pgm`, `*.ppm`);
- растровые изображения в Sun (`*.sr`, `*.ras`).

А для следующих форматов нужны дополнительные библиотеки:

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

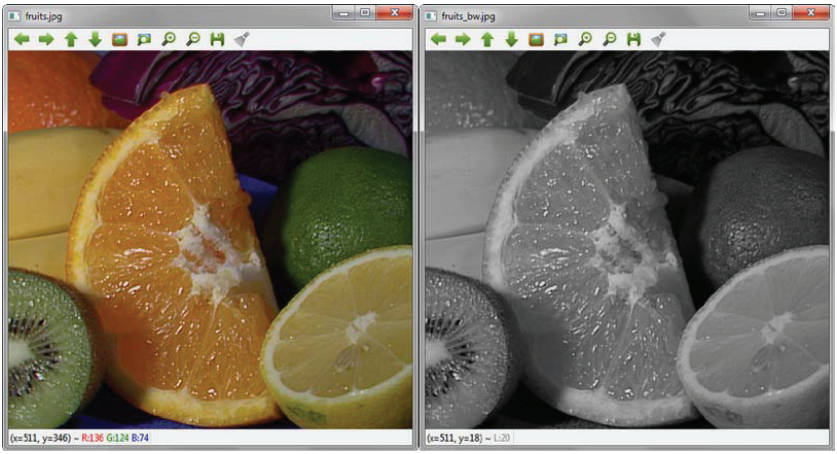
Помимо этого, в версию OpenCV 3.0 включен драйвер для форматов (**NITF**, **DTED**, **SRTM** и прочих), которые поддерживаются **библиотекой абстрагирования географических данных (Geographic Data Abstraction Library** – `GDAL`). Этот драйвер подключается, если в CMake задан параметр `WITH_GDAL`. Отметим, что на платформе Windows поддержка GDAL пока протестирована недостаточно полно. В Windows и OS X по умолчанию используются кодеки, поставляемые вместе с OpenCV (`libjpeg`, `libjasper`, `libpng` и `libtiff`). Поэтому в этих операционных системах можно читать файлы в форматах *JPEG*, *PNG* и *TIFF*. В Linux (и других ОС на основе Unix) OpenCV ищет кодеки, установленные в системе. Эти кодеки можно установить до OpenCV, а в противном случае библиотеки будут собраны при установке Open, если в CMake задан соответствующий параметр (точнее, `BUILD_JASPER`, `BUILD_JPEG`, `BUILD_PNG` и `BUILD_TIFF`).

Пример программы

Для иллюстрации чтения и записи файлов опишем программу **showImage**. Она запускается из командной строки следующим образом:

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

и открывает два окна:



Два окна программы **showImage**

Здесь в качестве аргументов указаны имена двух файлов. Первый содержит входное изображение, а второй – полутоновый вариант входного изображения, который нужно записать. Ниже приведен исходный код с пояснениями.

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

using namespace std;
using namespace cv;

int main(int, char *argv[])
{
  Mat in_image, out_image;
  // Порядок запуска: <cmd> <file_in> <file_out>
  // Читаем исходное изображение
  in_image = imread(argv[1], IMREAD_UNCHANGED);
  if (in_image.empty()) {
     // Проверяем, успешно ли прочитано изображение
     cout << "Ошибка! Не удалось прочитать исходное изображение...\n";
     return -1;
  }
  // Создаем два окна с именами изображений в заголовке
  namedWindow(argv[1], WINDOW_AUTOSIZE);
  namedWindow(argv[2], WINDOW_AUTOSIZE);

  // Показываем изображения в окнах
```

```
    imshow(argv[1], in_image);
    cvtColor(in_image, out_image, COLOR_BGR2GRAY);
    imshow(argv[2], in_image);
    cout << "Для выхода нажмите любую клавишу...\n";
    waitKey(); // Ждем нажатия клавиши

    // Записываем изображение
    imwrite(argv[2], in_image);
    return 0;
}
```

Здесь мы с помощью директивы `#include` включаем заголовочный файл `opencv.hpp`, а вместе с ними и все вообще заголовки OpenCV, так что больше никакие файлы включать не нужно. Далее мы объявляем пространство имен `cv`, поэтому добавлять префикс `cv::` к переменным и функциям, объявленным в этом пространстве, необязательно. Функция `main` первым делом проверяет, сколько аргументов ей передано в командной строке, и в случае ошибки печатает сообщение о порядке вызова программы.

Чтение файла изображения

Если число аргументов правильно, то файл изображения считывается в объект `Mat in_image` функцией `imread(argv[1], IMREAD_UNCHANGED)`. В первом параметре ей передается первый аргумент, заданный в командной строке (`argv[1]`), а во втором – флаг (`IMREAD_UNCHANGED`), который говорит, что изображение, сохраненное в памяти, изменять не следует. Функция `imread` определяет тип изображения (кодек) по содержимому файла, а не по расширению его имени.

Прототип функции `imread` выглядит следующим образом:

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

Флаг определяет цветность прочитанного изображения. Его значения определяются следующим перечислением, объявленным в заголовке `imgcodecs.hpp`:

```
enum { IMREAD_UNCHANGED = -1, // 8-разрядное, цветность не важна
       IMREAD_GRAYSCALE = 0, // 8- разрядное, полутоновое
       IMREAD_COLOR = 1,     // цветное, не изменять глубину
       IMREAD_ANYDEPTH = 2, // глубина любая, не изменять цветность
       IMREAD_ANYCOLOR = 4, // не изменять глубину, цветность любая
       IMREAD_LOAD_GDAL = 8 // использовать драйвер gdal
};
```

В версии OpenCV 3.0 функция `imread` находится в модуле `imgcodecs`, а не в `highgui`, как в OpenCV 2.x.

Поскольку в OpenCV 3.0 некоторые функции и объявления поменяли «место прописки», при сборке возможны ошибки, если компоновщик не найдет объявления каких-то символов. Чтобы узнать, в каком заголовке (hpp-файле) определен символ и с какой библиотекой следует компоновать программу, мы рекомендуем использовать в среде Qt Creator следующий прием:

Добавьте в программу объявление `#include <opencv2/opencv.hpp>`. Нажмите клавишу **F2**, наведя курсор на имя символа, в результате откроется `hpp`-файл, в котором этот символ определен.

Прочитав файл изображения, мы проверяем, успешно ли завершилась операция чтения. Для этого можно воспользоваться функцией-членом `in_image.empty()`. Если ошибок не было, то создаются два окна, в которых отображается входное и выходное изображение. Для создания окон применяется следующая функция:

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

Все функции OpenCV имеют понятные имена. Допустимые значения параметра `flags` определяются следующим перечислением, объявленным в заголовке `highgui.hpp`:

```
enum {
  WINDOW_NORMAL = 0x00000000,
  // пользователь может изменять размер окна, а также переходить
  // в полноэкранный режим и обратно

  WINDOW_AUTOSIZE = 0x00000001,
  // пользователь не может изменять размер окна,
  // его размер определяется загруженным изображением

  WINDOW_OPENGL = 0x00001000, // окно с поддержкой opengl
  WINDOW_FULLSCREEN = 1,
  WINDOW_FREERATIO = 0x00000100,
  // изображение занимает всю доступную площадь (на отношение
  // сторон не налагаются ограничения)

  WINDOW_KEEPRATIO = 0x00000000
  // отношение сторон сохраняется
};
```

Сама по себе операция создания окна ничего не выводит на экран. Для показа изображения служит функция `imshow` из модуля `highgui`:

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

Изображение (`mat`) показывается в оригинальном размере, если при создании окна с именем `winname` был задан флаг `WINDOW_AUTOSIZE`.

В примере **showImage** во втором окне показывается полутоновая копия входного изображения. Для преобразования цветного изображения в полутоновое применяется функция `cvtColor` из модуля `imgproc`. На самом деле, ее назначение – изменить цветовое пространство изображения.

Размер и положение любого созданного в программе окна можно изменить. Когда надобность в окне отпадает, его следует уничтожить, чтобы освободить ресурсы. Освобождение ресурсов неявно производится в конце программы, как в данном случае.

Обработка событий во внутреннем цикле

Если мы просто покажем изображение в окне и этим ограничимся, то, как ни странно, изображение не появится на экране. После вызова функции `imshow` нужно войти в цикл выборки и обработки событий взаимодействия с пользователем. Для этого служит следующая функция из модуля `highgui`:

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

Эта функция ожидает нажатия клавиши в течение заданного числа миллисекунд (`delay` > 0) и возвращает код нажатой клавиши или –1, если клавиша не была нажата. Если значение `delay` равно 0 или отрицательно, то функция не возвращает управление, пока не будет нажата клавиша.

Функция `waitKey` работает только в том случае, если существует хотя бы одно созданное и активное окно.

Запись файлов изображений

В модуле `imgcodecs` имеется также следующая функция:

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

Она сохраняет изображение (`img`) в файле (`filename`), а необязательный третий аргумент – вектор пар свойство–значение, описывающих параметры кодека (чтобы оставить значения по умолчанию, опустите этот параметр). Кодек определяется по расширению имени файла.

Полный список свойств кодека смотрите в заголовочном файле `imgcodecs.hpp` или в справочном руководстве по OpenCV API на странице `http://docs.opencv.org/master/modules/refman.html`.

Чтение и запись видеофайлов

В отличие от статических изображений, видео содержит изменяющиеся изображения. Источником видео может быть видеокамера, вебкамера, видеофайл или последовательность файлов изображений. В OpenCV имеются классы `VideoCapture` и `VideoWriter`, предоставляющие простой API для операций захвата и записи, участвующих в обработке видео.

Пример программы

В примере программы **recVideo** приведен короткий фрагмент кода, из которого понятно, как использовать камеру по умолчанию в качестве устройства для захвата кадров, обнаружения на них границ и сохранения преобразованного кадра в файле.

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

using namespace std;
using namespace cv;

int main(int, char **)
{
  Mat in_frame, out_frame;
  const char win1[]="Захват...", win2[]="Запись...";
  double fps=30; // Число кадров в секунду
  char file_out[]="recorded.avi";

  VideoCapture inVid(0); // Открыть камеру по умолчанию
  if (!inVid.isOpened()) { // Проверка ошибок
     cout << "Ошибка! Камера не готова...\n";
```

```
        return -1;
    }

    // Получаем ширину и высоту входного видео
    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 << "Ошибка! Видеофайл не открыт...\n";
        return -1;
    }

    // Создаем два окна: для исходного и конечного видео
    namedWindow(win1);
    namedWindow(win2);
    while (true) {
        // Читаем кадр с камеры (захват и декодирование)
        inVid >> in_frame;
        // Преобразуем кадр в полутоновый формат
        cvtColor(in_frame, out_frame, COLOR_BGR2GRAY);

        // Записываем кадр в видеофайл (кодирование и сохранение)
        recVid << out_frame;
        imshow(win1, in_frame);  // Показываем кадр в окне
        imshow(win2, out_frame); // Показываем кадр в окне
        if (waitKey(1000/fps) >= 0)
            break;
    }
    inVid.release(); // Закрываем камеру
    return 0;
}
```

Некоторых пояснений заслуживают следующие функции.

- `double VideoCapture::get(int propId)`: возвращает значение указанного свойства объекта `VideoCapture`. Полный список свойств основан на спецификации цифровых камер IEEE 1394 (документ DC1394) и включен в заголовочный файл `videoio.hpp`.
- `static int VideoWriter::fourcc(char c1, char c2, char c3, char c4)`: конкатенирует четыре символа, получая четырехзначный код **FourCC**. В нашем примере MSVC означает Microsoft Video (поддерживается только в Windows). Список допустимых FourCC-кодов опубликован на странице `http://www.fourcc.org/codecs.php`.

- `bool VideoWriter::isOpened()`: возвращает `true`, если объект для записи видео был успешно инициализирован. Например, попытка использовать неподходящий кодек заканчивается ошибкой.

Будьте осторожны; какие FourCC-коды допустимы в данной системе, зависит от установленных в ней кодеков. Чтобы узнать, какие кодеки установлены, рекомендуем воспользоваться программой с открытым исходным кодом MediaInfo, ее варианты для многих платформ можно скачать со страницы `http://mediaarea.net/en/MediaInfo`.

- `VideoCapture& VideoCapture::operator>>(Mat& image)`: захватывает, декодирует и возвращает следующий кадр. Этот метод эквивалентен функции `bool VideoCapture::read(OutputArray image)`. Его можно использовать вместо вызова функции `VideoCapture::grab()` с последующим вызовом `VideoCapture::retrieve()`.
- `VideoWriter& VideoWriter::operator<<(const Mat& image)`: записывает следующий кадр. Метод эквивалентен функции `void VideoWriter::write(const Mat& image)`.
- В нашем примере имеется также цикл чтения-записи, в котором выбираются и обрабатываются оконные события. За это отвечает вызов `waitKey(1000/fps)`, где `1000/fps` задает число миллисекунд ожидания до возврата во внешний цикл; `fps` – это приблизительная оценка числа кадров в секунду для записанного видео.
- `void VideoCapture::release()`: освобождает видеофайл или устройство сбора данных. В данном примере вызывать этот метод необязательно, но мы включили его для иллюстрации.

Средства взаимодействия с пользователем

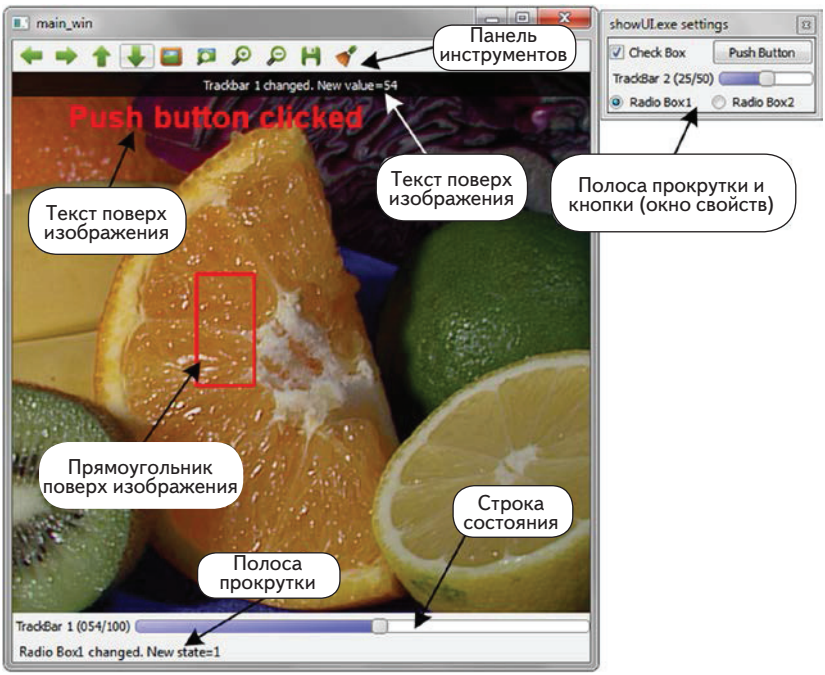
Выше мы объяснили, как создать (`namedWindow`) окно для показа (`imshow`) изображения, а затем выбирать и обрабатывать события (`waitKey`). Это был очень простой способ взаимодействия пользователя с приложением OpenCV посредством клавиатуры. Функция

`waitKey` возвращает код клавиши, нажатой до истечения заданного времени ожидания.

По счастью, OpenCV предоставляет и более гибкие средства взаимодействия, например полосы прокрутки и управление мышью. В сочетании с функциями рисования он позволяют построить гораздо более удобный интерфейс. Кроме того, если OpenCV откомпилирована с поддержкой Qt (в CMake был задан параметр `WITH_QT`), то появляется целый ряд дополнительных функций для конструирования развитых пользовательских интерфейсов.

В этом разделе мы дадим краткий обзор возможностей программирования пользовательских интерфейсов в приложениях OpenCV с поддержкой Qt. Для иллюстрации напишем программу **showUI**.

Она показывает в окне цветное изображение и несколько простых элементов для взаимодействия с пользователем.



Окно программы **showUI**

Ниже приведен исходный код программы **showUI** (без функций обратного вызова):

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

using namespace std;
using namespace cv;

// Объявления функций обратного вызова
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);

// Глобальные определения и переменные
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; // Начальное значение полосы прокрутки 1
  int tb2_value = 25; // Начальное значение полосы прокрутки 2

  orig_img = imread(argv[1]); // Открываем и читаем изображение
  if (orig_img.empty()) {
     cout << "Ошибка!!! Не удалось загрузить изображение..." << endl;
     return -1;
  }

  namedWindow(main_win); // Создаем главное окно
  // Создаем шрифт для добавления текста поверх изображения
  QtFont font = fontQt("Arial", 20, Scalar(255,0,0,0),
     QT_FONT_BLACK, QT_STYLE_NORMAL);

  // Задаем функции обратного вызова
  setMouseCallback(main_win, cbMouse, NULL);
  createTrackbar(track1, main_win, &tb1_value, 100, tb1_Callback);
  createButton(checkbox, checkboxCallBack, 0,  QT_CHECKBOX);

  // Передаем значение (шрифт) функции обратного вызова
  createButton(pushbutton, pushbuttonCallBack,
     (void *)&font, QT_PUSH_BUTTON);
  createTrackbar(track2, NULL, &tb2_value,
     50, tb2_Callback);
```

```
    // Передаем значения функциям обратного вызова
    createButton(radiobox1, radioboxCallBack,
       (void *)radiobox1, QT_RADIOBOX);
    createButton(radiobox2, radioboxCallBack,
       (void *)radiobox2, QT_RADIOBOX);

    imshow(main_win, orig_img); // показываем исходное изображение
    cout << "Для выхода нажмите любую клавишу..." << endl;
    waitKey(); // Бесконечный цикл обработки событий
    return 0;
}
```

Если OpenCV собрана с поддержкой Qt, то в любом созданном – с помощью функций из модуля `highgui` – окне по умолчанию будет присутствовать **панель инструментов** (см. рисунок выше) с командами (слева направо) панорамирования, изменения масштаба, сохранения и открытия окна свойств.

В следующих подразделах мы расскажем о других элементах интерфейса, присутствующих в этом примере, и о том, как они программируются.

Полосы прокрутки

Полоса прокрутки создается функцией `createTrackbar(const String& trackbarname, const String& winname, int* value, int count, TrackbarCallback onChange=0, void* userdata=0)`, которой передается окно (`winname`), ассоциированное целочисленное значение (`value`), максимальное значение (`count`), необязательная функция обратного вызова (`onChange`), которая вызывается при любом изменении положения ползунка, и аргумент (`userdata`), передаваемый функции обратного вызова. Сама функция обратного вызова получает два аргумента: значение `value` (выбранное с помощью ползунка) и указатель на пользовательские данные `userdata` (необязательный). Если поддерживается Qt, то в случае, когда окно не задано, полоса прокрутки создается в окне свойств. В программе **showUI** созданы две полосы прокрутки: одна в главном окне, а другая – в окне свойств. Ниже показан код функций обратного вызова для этих полос прокрутки:

```
void tb1_Callback(int value, void *) {
  sprintf(msg, "Полоса прокрутки 1 изменилась. Новое значение=%d", value);
  displayOverlay(main_win, msg);
  return;
}

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

```
    sprintf(msg, "Полоса прокрутки 2 изменилась. Новое значение=%d", value);
    displayStatusBar(main_win, msg, 1000);
    return;
}
```

Управление с помощью мыши

Пользователь взаимодействует с мышью, обрабатывая события от нее (перемещение и щелчок). Задав подходящий обработчик в виде функции обратного вызова, мы можем реализовать такие действия, как выбор, перетаскивание и т. д. Функция обратного вызова (`onMouse`) задается с помощью функции `setMouseCallback(const String& winname, MouseCallback onMouse, void* userdata=0)`, которой передается имя окна (`winname`) и необязательный аргумент (`userdata`).

Ниже приведен исходный код функции обратного вызова для обработки событий мыши.

```
void cbMouse(int event, int x, int y, int flags, void*) {
  // В статических переменных хранятся значения между вызовами
  static Point p1, p2;
  static bool p2set = false;

  // Нажата левая кнопка мыши
  if (event == EVENT_LBUTTONDOWN) {
     p1 = Point(x, y); // Точка, в которой произошло событие
     p2set = false;
  } else if (event == EVENT_MOUSEMOVE &&
             flags == EVENT_FLAG_LBUTTON) {
     // Перемещение мыши с нажатой левой кнопкой
     // Проверяем выход за границы
     if (x > orig_img.size().width)
        x = orig_img.size().width;
     else if (x < 0)
        x = 0;
     // Проверяем выход за границы
     if (y > orig_img.size().height)
        y = orig_img.size().height;
     else if (y < 0)
        y = 0;
     p2 = Point(x, y); // Устанавливаем конечную точку
     p2set = true;

     // Копируем исходное изображение во временное
     orig_img.copyTo(tmp_img);
     // Рисуем прямоугольник
     rectangle(tmp_img, p1, p2, Scalar(0, 0 ,255));
     // Показываем временное изображение с прямоугольником
     imshow(main_win, tmp_img);
```

```
    } else if (event == EVENT_LBUTTONUP && p2set) {
        // Левая кнопка отпущена в области окна
        // Подмассив исходного изображения с выбранным прямоугольником
        Mat submat = orig_img(Rect(p1, p2));

        // Здесь должен быть код обработки подматрицы
        //...
        // Отмечаем границы выбранного прямоугольника
        rectangle(orig_img, p1, p2, Scalar(0, 0, 255), 2);
        imshow("main_win", orig_img);
    }
    return;
}
```

В программе **showUI** события мыши используются для выбора прямоугольной области путем рисования прямоугольника. Этой цели служит функция обратного вызова `cbMouse`. В нашем примере функция объявлена как `void cbMouse(int event, int x, int y, int flags, void*)`, а ее аргументами являются позиция мыши (`x`, `y`), в которой произошло событие, условия в момент события (`flags`) и необязательные пользовательские данные `userdata`.

Допустимые события, флаги и соответствующие символы объявлены в заголовочном файле `highgui.hpp`.

Кнопки

OpenCV (с поддержкой Qt) позволяет создавать кнопки трех типов: **флажок** (`QT_CHECKBOX`), **переключатель** (`QT_RADIOBOX`) и **нажимаемая кнопка** (`QT_PUSH_BUTTON`). С их помощью можно задавать соответственно бинарные параметры, набор взаимоисключающих параметров и действия при нажатии. Кнопка любого вида создается функцией `createButton(const String& button_name, ButtonCallback on_change, void* userdata=0, int type=QT_PUSH_BUTTON, bool init_state=false)`. Все кнопки размещаются в окне свойств на панели кнопок, которая располагается вслед за последней созданной в этом окне полосой прокрутки. В качестве аргументов передаются имя кнопки (`button_name`), функция обратного вызова, вызываемая при изменении состояния (`on_change`) и необязательные пользовательские данные для этой функции (`userdata`), тип кнопки (`type`) и ее начальное состояние (`init_state`).

Ниже приведен код функций обратного вызова для кнопок.

```
void checkboxCallBack(int state, void *) {
  sprintf(msg, "Флажок изменился. Новое состояние=%d", state);
  displayStatusBar(main_win, msg);
  return;
}

void radioboxCallBack(int state, void *id) {
  // Ид переключателя передается функции обратного вызова
  sprintf(msg, "%s изменился. Новое состояние=%d",
     (char *)id, state);
  displayStatusBar(main_win, msg);
  return;
}

void pushbuttonCallBack(int, void *font) {
  // Добавить текст поверх изображения
  addText(orig_img, "Нажата кнопка",
     Point(50,50), *((QtFont *)font));
  imshow(main_win, orig_img); // Показываем исходное изображение
  return;
}
```

Функция обратного вызова для кнопки принимает два аргумента: состояние кнопки и необязательный указатель на пользовательские данные. В примере выше показано, как передать целое число (`radioboxCallBack(int state, void *id)`), идентифицирующее кнопку, и более сложный объект (`pushbuttonCallBack(int, void *font)`).

Рисование и отображение текста

Чрезвычайно эффективный способ сообщить пользователю о результатах обработки изображения – нарисовать на нем фигуры или текст. В модуле `imgproc` имеются вспомогательные функции для таких операций, как нанесение текста, рисование прямых, окружностей, эллипсов, прямоугольников, многоугольников и т. д. В примере **showUI** показано, как выбрать прямоугольную область и нарисовать ограничивающий ее прямоугольник. Приведенная ниже функция рисует поверх изображения `img` прямоугольник, определенный двумя точками (`p1`, `p2`) заданного цвета. В качестве необязательных аргументов задаются толщина рамки (отрицательное значение означает, что фигуру нужно залить) и тип линии:

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

Помимо рисования фигур, модуль `imgproc` содержит функцию для нанесения текста поверх изображения:

```
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 )
```

Доступные шрифтовые гарнитуры смотрите в заголовочном файле `core.hpp`.

При наличии поддержки Qt (модуль `highgui`) появляются дополнительные способы показа текста в главном окне приложения OpenCV:

- **Текст поверх изображения**: для этого служит функция `addText(const Mat& img, const String& text, Point org, const QtFont& font)`. Она позволяет задать начальную точку текста и шрифт, ранее созданный функцией `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)`. В программе **showUI** эта функция используется, чтобы нанести текст поверх изображения при нажатии кнопки – функция `addText` вызывается из функции обратного вызова.
- **Текст в строке состояния:** с помощью функции `displayStatusBar(const String& winname, const String& text, int delayms=0)` мы показываем в строке состояния текст, который будет виден в течение числа миллисекунд, заданного в последнем аргументе (`delayms`). В программе **showUI** эта функция вызывается (из функций обратного вызова), чтобы показать информационное сообщение при изменении состояния кнопок и полосы прокрутки в окне свойств.
- **Текст, наложенный на изображение:** с помощью функции `displayOverlay(const String& winname, const String& text, int delayms=0)` мы выводим поверх изображения текст, который будет виден в течение числа миллисекунд, заданного в последнем аргументе. В программе **showUI** эта функция вызывается (из функций обратного вызова), чтобы показать информационное сообщение при изменении состояния полосы прокрутки в главном окне.

Резюме

В этой главе мы привели краткий обзор назначения библиотеки OpenCV и ее модулей. Вы узнали, как откомпилировать, установить и начать пользоваться библиотекой для разработки приложений OpenCV с поддержкой Qt на языке **C++**. Мы рассказали о том, как приступить к работе в интегрированной среде Qt Creator с комплектом компиляторов GNU.

В этой главе был приведен полный код простых примеров, демонстрирующих чтение и запись изображений и видео. И напоследок мы показали, как пользоваться элементами интерфейса: полосой прокрутки, кнопками, текстом поверх изображения, рисованием фигур и т. д.

Следующая глава посвящена описанию основных инструментов обработки изображений и задачам, которые лягут в основу последующих глав.

ГЛАВА 2. Инструменты обработки изображений

В этой главе описываются основные структуры данных и процедуры, которыми мы будем пользоваться в последующих главах:

- типы изображений;
- доступ к пикселям;
- основные операции с изображениями;
- гистограммы.

Речь пойдет о наиболее часто выполняемых операциях над изображениями. Большая часть рассматриваемой здесь функциональности находится в модуле `core`.

Основные типы данных

Главным типом данных в OpenCV является класс `Mat`, который служит для хранения изображений. Изображение хранится в виде заголовка и области для пиксельных данных. Изображение состоит из нескольких каналов. У полутоновых изображений один канал, а у цветных обычно три: для красной, зеленой и синей составляющей (правда, в OpenCV принят обратный порядок: синяя, зеленая, красная). Можно использовать также четвертый канал (альфа), описывающий прозрачность. Количество каналов изображения `img` возвращает функция `img.channels()`.

Для хранения одного пикселя изображения используется определенное число битов, которое называется *глубиной* изображения. Для полутоновых изображений пиксель обычно представляется 8 битами, так что всего получается 256 уровней яркости (целые значения от 0 до 255). Пиксели цветного изображения представляются тремя байтами, по одному на каждый цветовой канал. Для некоторых операций пиксели необходимо хранить в формате с плавающей точкой. Глуби-

ну изображения можно получить с помощью функции `img.depth()`, которая может возвращать следующие значения:

- `CV_8U`, 8-разрядное целое без знака (0..255)
- `CV_8S`, 8-разрядное целое со знаком (–128..127)
- `CV_16U`, 16-разрядное целое без знака (0..65 535)
- `CV_16S`, 16-разрядное целое со знаком (–32 768..32 767)
- `CV_32S`, 32-разрядное целое без знака (–2 147 483 648..2 147 483 647)
- `CV_32F`, 32-разрядное с плавающей точкой
- `CV_64F`, 64-разрядное с плавающей точкой

Отметим, что и для полутоновых, и для цветных изображений чаще всего используется глубина CV_8U. Преобразовать глубину позволяет метод `convertTo`:

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

Нередко выполняются операции над изображениями, представленными числами с плавающей точкой (когда значения пикселей являются результатами математических операций). Если для показа такого изображения воспользоваться функцией `imshow()`, то получится бессмысленный результат. Во избежание этого необходимо преобразовать значения пикселей к целочисленному диапазону 0..255. Функция `convertTo` выполняет линейное преобразование и принимается два дополнительных параметра, *alpha* и *beta*, представляющие масштабный коэффициент и свободный член. Таким образом, пиксель *p* преобразуется по формуле:

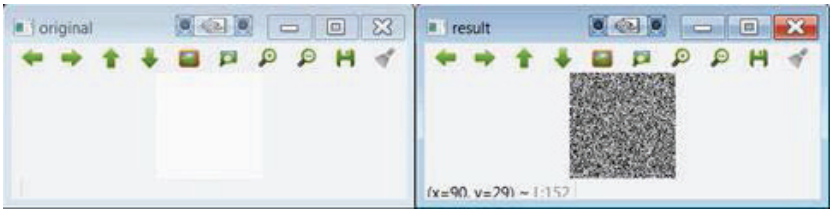
$$newp = alpha*p + beta$$

Эту функцию можно использовать для правильного показа изображений в формате с плавающей точкой. В предположении, что минимальное значение пикселя изображения равно `m`, а максимальное – `M` (как найти эти значения, показано в коде ниже), нужно написать такой код:

```
Mat m1 = Mat(100, 100, CV_32FC1);
randu(m1, 0, 1e6); // случайное значение от 0 до 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);
```

Этот код преобразует диапазон значений пикселей результирующего изображения к диапазону 0–255. На рисунке ниже показан результат его выполнения.



Результат работы метода convertTo
(обратите внимание, что на левом рисунке изображение выглядит белым)

Размер изображения можно узнать, опросив атрибуты `rows` и `cols`. Существует также атрибут `size`, содержащий сразу оба размера:

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

Помимо собственно изображения, часто употребляются и другие типы данных, перечисленные в следующей таблице.

Тип	Ключевое слово	Пример
Вектор (небольшой)	`VecAB`, где `A` может принимать значения 2,3,4,5,6, в `B` – значения `b,s,i,f,d`	`Vec3b rgb;` `rgb[0]=255;`

Скаляры (до 4)		`Scalar a;`
		`a[0]=0;`
		`a[1]=0;`

Точка		`Point3d p;`
		`p.x=0;`
		`p.y=0;`
		`p.z=0;`

Размер		`Size s;`
		`s.width=30;`
		`s.height=40;`

Прямоугольник		`Rect r;`
		`r.x=r.y=0;`
		`r.width=r.height=100;`

Для некоторых типов имеются дополнительные операции. Например, можно проверить, лежит ли точка внутри прямоугольника:

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

Аргументы `p` и `r` – это точка (на плоскости) и прямоугольник соответственно. Отметим, что эта таблица далеко не полна, в OpenCV есть еще немало структур, в том числе содержащих методы.

Доступ к пикселям

Для обработки изображений нужно уметь обращаться к отдельным пикселям. В OpenCV для этого есть много способов. В этом разделе мы рассмотрим только два: первый удобнее для программиста, второй – эффективнее.

В первом способе применяется шаблонная функция `at<>`. Для работы с ней мы должны задать тип элементов матрицы, как в примере ниже:

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

Здесь мы читаем одно и то же изображение сначала как полутоновое, а потом как цветное и обращаемся к первому пикселю в позиции (0,0). В первом случае пиксель будет иметь тип `unsigned char (that is, uchar)`, а во втором нужно использовать тип `Vec3b`, т. е. тройку значений типа `unsigned char`. Разумеется, функция `at<>` может встречаться и в левой части оператора присваивания, если нужно изменить значение пикселя.

В следующем фрагменте матрица чисел с плавающей точкой с помощью этого способа инициализируется значением Pi:

```
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;
```

Но метод `at<>` не очень эффективен, потому что должен вычислять адрес в памяти по строке и столбцу пикселя. Если попиксельно требуется обработать все изображение, то на это может уйти много

времени. Во втором способе используется функция `ptr`, которая возвращает указатель на строку изображения. В следующем фрагменте мы получаем значения каждого пикселя цветного изображения.

```
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];
  }
}
```

Здесь функция `ptr` возвращает указатель на первый пиксель очередной строки. Зная этот указатель, мы можем во внутреннем цикле обойти пиксели в каждом столбце.

Хронометраж

Обработка изображений занимает время (гораздо больше, чем требуется для обработки одномерных данных). Часто именно время определяет, пригоден конкретный алгоритм или нет. В OpenCV есть две функции для измерения времени работы: `getTickCount()` и `getTickFrequency()`. Используются они так:

```
double t0 = (double)getTickCount();
// здесь находится ваш код ...
elapsed = ((double)getTickCount() - t0)/getTickFrequency();
```

В данном случае затраченное время измеряется в секундах.

Типичные операции над изображениями

В таблице ниже перечислены наиболее часто встречающиеся операции над изображениями.

Операция	Примеры

Задать значения элементов матрицы	`img.setTo(0); // Для 1-канального изобр.`
	`img.setTo(Scalar(B,G,R); // 3-канальное изобр.`

Операция	Примеры

Инициализация матрицы в стиле MATLAB	`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;`

Инициализация случайными значениями	`Mat m1 = Mat(100, 100, CV_8UC1);` `randu(m1, 0, 255);`
Создание копии матрицы	`Mat img1 = img.clone();`
Создание копии матрицы (с маской)	`img.copy(img1, mask);`
Ссылка на подматрицу (данные не копируются)	`Mat img1 = img` `(Range(r1,r2),Range(c1,c2));`

Кадрирование изображения	`Rect roi(r1,c2, width, height);`
	`Mat img1 = img(roi).clone(); // данные`
	`// копируются`

Изменение размера	`resize(img, imag1, Size(), 0.5, 0.5);` `// уменьшить вдвое по обоим измерениям`

Зеркальное отражение	`flip(imgsrc, imgdst, code);`
	`// code=0 => отразить по вертикали`
	`// code>0 => отразить по горизонтали`
	`// code<0 => отразить по горизонтали и по`
	`//  вертикали`

Разделение каналов	`Mat channel[3];`
	`split(img, channel);`
	`imshow("B", channel[0]); // показать синий`

Объединение каналов	`merge(channel,img);`
Подсчет ненулевых пикселей	`int nz = countNonZero(img);`

Минимум и максимум	`double m,M;`
	`Point mLoc,MLoc;`
	`minMaxLoc(img,&m,&M,&mLoc,&MLoc);`

Среднее значение пикселей	`Scalar m, stdd;`
	`meanStdDev(img, m, stdd);`
	`uint mean_pxl = mean.val[0];`

Проверка пустоты изображения	`if (img.empty())` `cout << "не удалось загрузить изображение";`

Арифметические операции

Арифметические операторы перегружены. Это означает, что над объектами типа `Mat` можно выполнять такие действия:

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

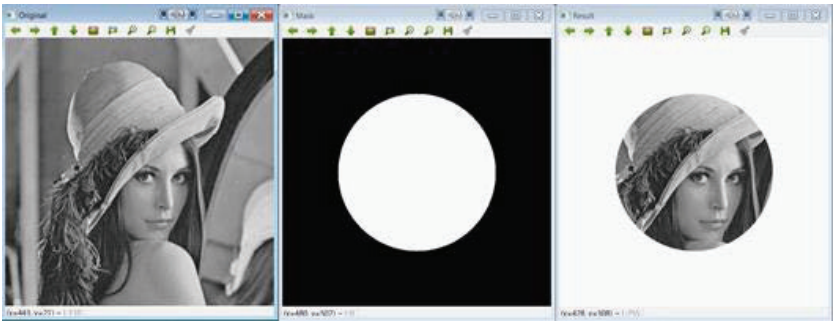
В OpenCV значение результата операции подчиняется правилам так называемой арифметики с насыщением (saturation arithmetic). Это означает, что значением является ближайшее целое в диапазоне 0–255.

При работе с масками очень полезны поразрядные операции `bitwise_and()`, `bitwise_or()`, `bitwise_xor()` и `bitwise_not()`. Маской называется бинарное изображение, которое показывает, над какими пикселями производить операцию. В примере ниже показано, как с помощью функции `bitwise_and` вырезать часть изображения:

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

using namespace cv;
using namespace std;

int main()
{
  Mat img1 = imread("lena.png", IMREAD_GRAYSCALE);
  if (img1.empty())
  {
     cout << "Не удалось загрузить изображение!" << endl;
     return -1;
  }

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

  // Создаем изображение-маску
  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);

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

  // Залить все остальное белым цветом
  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))
```

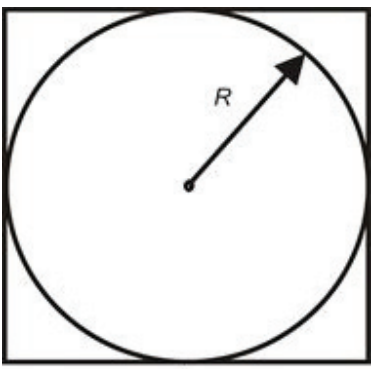
```
            r.at<uchar>(i,j)=white;

    imshow("Result",r);

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

Прочитав и показав изображение, мы создаем маску в виде залитого белого круга. Эта маска используется в логической операции AND, которая применяется только к тем пикселям, для которых значение маски не равно нулю; все остальные пиксели не изменяются. И напоследок мы заливаем часть изображения вне маски белым цветом. Для этого применяется один из описанных выше способов доступа к пикселям. На рисунке ниже показан результат.



Результат применения функции `bitwise_and`

Еще один интересный пример связан с оценкой значения числа Pi. Рассмотрим квадрат и вписанный в него круг:



Их площади вычисляются по формулам:

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

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

Отсюда имеем

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

Предположим, что имеется изображение квадрата с неизвестной длиной стороны и вписанного в него круга. Мы можем оценить площадь круга, нарисовав много пикселей в случайных позициях и подсчитав, сколько из них попали внутрь круга. С другой стороны, площадь квадрата оценивается как общее число нарисованных пикселей. Это позволит оценить значение Pi по формуле выше.

Описанную идею реализует следующий алгоритм.

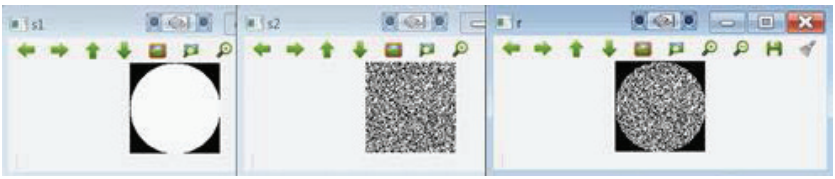
1. На изображении черного квадрата нарисовать вписанный в него сплошной белый круг.
2. На другом изображении черного квадрата (того же размера) нарисовать много пикселей в случайных позициях.
3. Выполнить операцию AND между двумя изображениями и подсчитать число ненулевых пикселей в результате.
4. Оценить Pi по приведенной выше формуле.

Ниже приведен код программы **estimatePi**:

```
#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 << "Оценка Pi: " << Pi << endl;

    waitKey();
    return 0;
}
```

Эта программа точно следует сформулированному выше алгоритму. Отметим, что для подсчета ненулевых (в данном случае белых) пикселей мы воспользовались функцией `countNonZero`. При `npixels=8000` получается оценка 3.125. Чем больше пикселей, тем точнее будет оценка.



Результат выполнения программы **estimatePi**

Сохранение данных

Помимо конкретных функций для чтения и записи изображений и видео, в OpenCV существует более общий способ сохранения и загрузки данных. Он так и называется «сохранение данных» (data persistence): значения объектов и переменных программы можно записать на диск (сериализовать). Это очень полезно для сохранения результатов и загрузки конфигурационных данных. Основной класс называется `FileStorage`, он представляет файл на диске. На самом деле, данные хранятся в формате XML или YAML.

Ниже описаны шаги записи данных.

1. Вызвать конструктор класса `FileStorage`, передав ему имя файла и флаг `FileStorage::WRITE`. Формат данных определяется расширением имени файла (`.xml`, `.yml` или `.yaml`).
2. Записать данные в файл с помощью оператора `<<`. Обычно данные записываются в виде пар строка-значение.
3. Закрыть файл, вызвав метод `release`.

Чтение данных производится в следующем порядке.

1. Вызвать конструктор класса `FileStorage`, передав ему имя файла и флаг `FileStorage::READ`.
2. Прочитать данные из файла с помощью оператора `[]` или `>>`.
3. Закрыть файл, вызвав метод `release`.

В следующем примере эта идея применяется для сохранения и загрузки состояния полосы прокрутки.

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

using namespace cv;
using namespace std;

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 << "Не удалось загрузить изображение!" << endl;
     return -1;
  }

  int tb1_value = 0;

  // загружаем значение полосы прокрутки
  FileStorage fs1("config.xml", FileStorage::READ);
  tb1_value=fs1["tb1_value"];    // способ 1
  fs1["tb1_value"] >> tb1_value; // способ 2
  fs1.release();

  // создаем полосу прокрутки
  namedWindow("main_win");
  createTrackbar("brightness", "main_win", &tb1_value,
```

```
                255, tb1_Callback);
    tb1_Callback(tb1_value, NULL);

    waitKey();

    // сохраняем значение полосы прокрутки при выходе
    FileStorage fs2("config.xml", FileStorage::WRITE);
    fs2 << "tb1_value" << tb1_value;
    fs2.release();

    return 0;
}
```

Если OpenCV была собрана с поддержкой Qt, то свойства окна, в том числе и состояние полосы прокрутки, можно сохранить с помощью функции `saveWindowParameters()`.

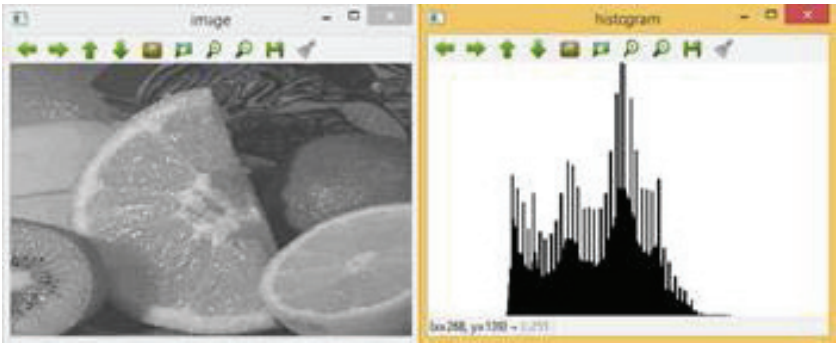
Если полоса прокрутки используется для управления целочисленным значением – яркостью, то она просто помещается на исходное изображение. Текущее значение читается в начале программы (при самом первом запуске оно будет равно 0) и сохраняется перед выходом. Обратите внимание на два эквивалентных способа прочитать значение переменной `tb1_value`. Ниже показано содержимое файла `config.xml`:

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

Гистограммы

Итак, мы умеем определять тип данных изображения и обращаться к отдельным пикселям, т. е. читать их уровни яркости. Теперь возникает желание получить функцию плотности вероятности уровней яркости, которая называется гистограммой. Гистограмма изображения описывает частоты различных уровней яркости. Путем моделирования гистограммы можно изменять уровень контрастности изображения – это называется **выравниванием гистограммы**. Моделирование гистограммы – эффективный способ улучшения изображения путем варьирования его контрастности. Благодаря выравниванию удается повысить контрастность низкоконтрастных участков. На рисунке

ниже показан пример выровненной гистограммы и соответствующего изображения.



Пример выровненной гистограммы изображения

В OpenCV гистограмму изображения можно вычислить с помощью функции `void calcHist`, а для выравнивания служит функция `void equalizeHist`.

Функция вычисления гистограммы принимает десять параметров:

```
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`: адрес первого изображения в коллекции. Позволяет обрабатывать последовательность изображений.
- `int nimages`: количество исходных изображений.
- `const int* channels`: список каналов, используемых для вычисления гистограммы. Число каналов изменяется от 0 до 2.
- `InputArray mask`: необязательная маска, показывающая, какие пиксели учитывать при вычислении гистограммы.
- `OutputArray hist`: результирующая гистограмма.
- `int dims`: позволяет задать размерность гистограммы.
- `const int* histSize`: массив размеров гистограмм по каждому измерению.
- `const float** ranges`: массив массивов, описывающих границы интервалов гистограммы по каждому измерению.
- `bool uniform=true`: по умолчанию этот параметр равен `true`. Он определяет, является ли гистограмма равномерной.

- `bool accumulate=false`: по умолчанию этот параметр равен `false`. Он определяет, является ли гистограмма кумулятивной.

Функция выравнивания гистограммы принимает всего два параметра: `void equalizeHist(InputArray src, OutputArray dst)`. Первый параметр – входное изображение, второй – выходное изображение с выровненной гистограммой.

Можно вычислить гистограммы сразу нескольких изображений. Это позволит сравнить гистограммы и вычислить совместную гистограмму. Для сравнения гистограмм двух изображений `histImage1` и `histImage2` служит функция `void compareHist(InputArray histImage1, InputArray histImage2, method)`. Метрика `method` нужна для того, чтобы вычислить соответствие гистограмм. В OpenCV определены четыре метрики: корреляция (`CV_COMP_CORREL`), хи-квадрат (`CV_COMP_CHISQR`), пересечение или минимальное расстояние (`CV_COMP_INTERSECT`) и расстояние Бхаттачария (`CV_COMP_BHATTACHARYYA`).

Третий параметр позволяет вычислить гистограммы нескольких каналов одного и того же изображения.

Ниже приведены два примера: вычисление гистограммы цветного изображения (**ColourImageEqualizeHist**) и сравнение гистограмм (**ColourImageComparison**). В программе **ColourImageEqualizeHist** показано также, как производится выравнивание гистограммы, а в примере **ColourImageComparison** – как вычисляется двумерная гистограмма двух каналов: оттенка (H) и насыщенности (S).

Пример программы

В примере **ColourImageEqualizeHist** показано, как выровнять цветное изображение и вывести одновременно гистограммы всех каналов. Вычисление гистограмм всех цветовых каналов RGB-изображения производится с помощью функции `histogramcalculation(InputArray Imagesrc, OutputArray histoImage)`. Для этой цели цветное изображение разделяется на три канала: R, G, B. Выравнивание гистограммы применяется к каждому каналу, а затем результаты объединяются в выровненное цветное изображение.

```
#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;

  // Задаем диапазоны (для 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 );

  // Вычисляем гистограммы
  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 );

  // Рисуем гистограммы для каналов B, G и 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) );

  // Нормируем результат на диапазон [ 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());

  // Рисуем для каждого канала
  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 );
```

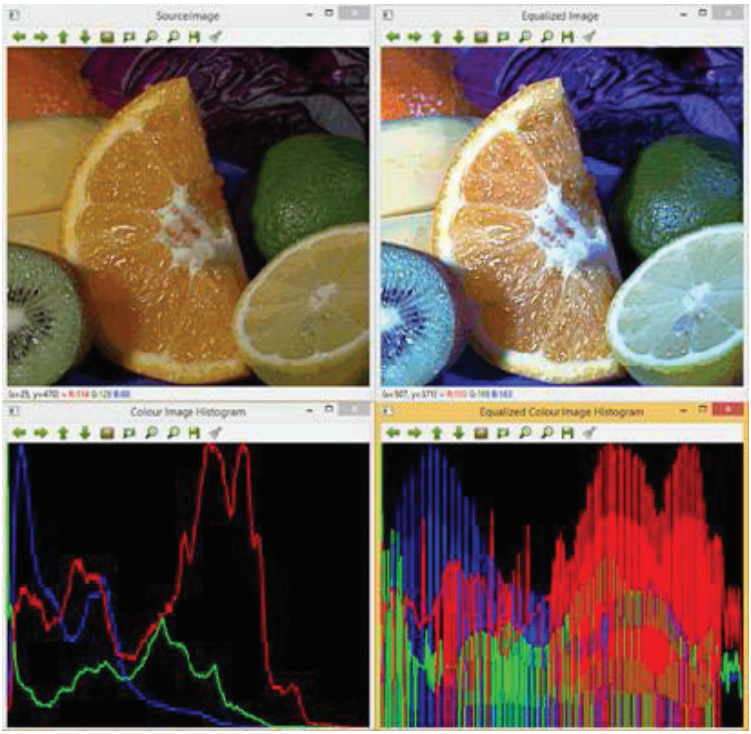
```
    }
    histoImage= histImage;
}

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

    // Читаем исходное изображение
    src = imread( "fruits.jpg");
    if(! src.data ) {
       printf("Ошибка при чтении изображения\n");
       exit(1);
    }

    // Разделяем изображение на 3 канала (B, G и R)
    vector<Mat> bgr_planes;
    split( src, bgr_planes );

    // Выводим результаты
    imshow( "Source image", src );

    // Вычисляем гистограммы каждого канала
    histogramcalculation(src, histImage);

    // Выводим гистограммы всех цветовых каналов
    imshow("Colour Image Histogram", histImage );

    // Выровненное изображение

    // Применяем выравнивание к гистограммам всех каналов
    equalizeHist(bgr_planes[0], bgr_planes[0]);
    equalizeHist(bgr_planes[1], bgr_planes[1]);
    equalizeHist(bgr_planes[2], bgr_planes[2]);

    // Объединяем выровненные каналы в выровненное цветное изображение
    merge(bgr_planes, imageq );

    // Выводим выровненное изображение
    imshow( "Equalized Image", imageq );

    // Вычисляем гистограммы каждого канала выровненного изображения
    histogramcalculation(imageq, histImage);

    // Отображаем гистограмму выровненного изображения
    imshow("Equalized Colour Image Histogram", histImage);

    // Ждем, когда пользователь завершит программу
    waitKey();
```

```
  return 0;
}
```

В этом примере создается четыре окна.

- **Исходное изображение:** показано на рисунке в левом верхнем углу.
- **Выровненное цветное изображение:** показано в правом верхнем углу.
- **Гистограмма трех каналов исходного изображения:** в данном случае R=красный, G=зеленый и B=синий. Показана в левом нижнем углу.
- **Гистограмма каналов RGB выровненного изображения:** показана в правом нижнем углу. На рисунке видно, как самые часто встречающиеся значения интенсивности R, G и B растянуты благодаря процессу выравнивания.

На рисунке ниже представлены результаты работы алгоритма.



Пример программы

В примере **ColourImageComparison** показано, как вычислить двумерную гистограмму двух каналов одного и того же цветного изображения. Здесь же производится сравнение исходного и выравненного изображения путем сопоставления гистограмм. Для сопоставления применяются четыре вышеупомянутых метрики: корреляция, хи-квадрат, минимальное расстояние и расстояние Бхаттачария. Вычисление двумерной гистограммы цветовых каналов H и S выполняет функция `histogram2Dcalculation(InputArray Imagesrc, OutputArray histo2D)`. Для сравнения гистограмм вычисляется нормированная одномерная гистограмма RGB-изображения, поскольку сравнивать можно только нормированные гистограммы. Это делает функция `histogramRGcalculation(InputArray Imagesrc, OutputArray histo)`:

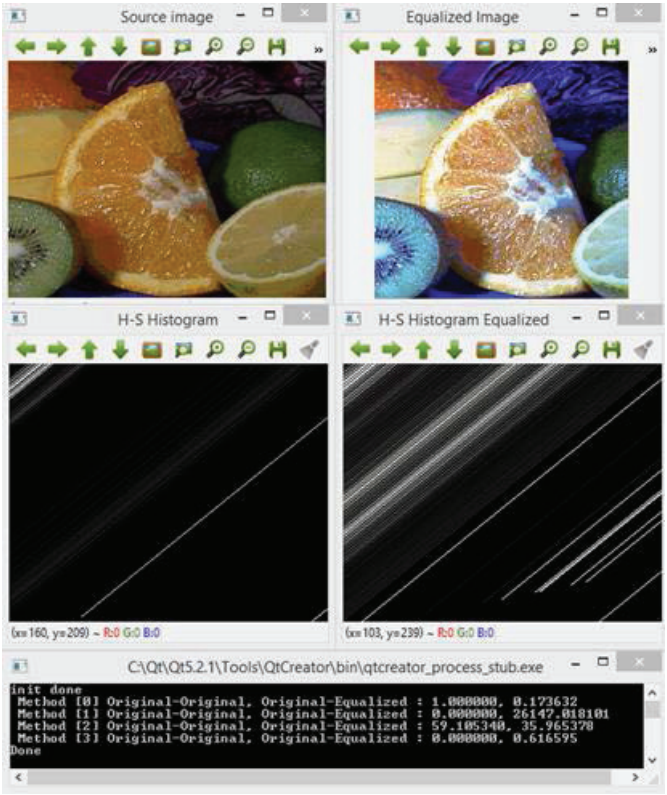
```
void histogram2Dcalculation(const Mat &src, Mat &histo2D)
{
   Mat hsv;

   cvtColor(src, hsv, CV_BGR2HSV);

   // Задаем 30-255 уровней квантования оттенка
   // и 32-255 уровней квантования насыщенности.
   int hbins = 255, sbins = 255;
   int histSize[] = {hbins, sbins};
   // оттенок изменяется от 0 до 179, см. cvtColor
   float hranges[] = { 0, 180 };
   // насыщенность изменяется от 0 (черный-серый-белый) до
   // 255 (спектр эталонных цветов)
   float sranges[] = { 0, 256 };
   const float* ranges[] = { hranges, sranges };
   MatND hist, hist2;
   // вычисляем гистограмму каналов 0 и 1
   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)
{
   // Используем 50 интервалов для красного и 60 для зеленого
   int r_bins = 50; int g_bins = 60;
   int histSize[] = { r_bins, g_bins };

   // Красный изменяется от 0 до 255, зеленый от 0 до 255
   float r_ranges[] = { 0, 255 };
   float g_ranges[] = { 0, 255 };

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

   // Используем нулевой и первый каналы
   int channels[] = { 0, 1 };

   // Гистограммы
   MatND hist_base;

   // Вычисляем гистограммы HSV-изображений
   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;

   // Читаем исходное изображение
   src = imread( "fruits.jpg");
   if(! src.data ) {
      printf("Ошибка при чтении изображения\n");
      exit(1);
   }

   // Разделяем изображение на 3 канала ( B, G, R )
   vector<Mat> bgr_planes;
   split( src, bgr_planes );

   // Выводим результаты
```

```
    namedWindow("Source image", 0 );
    imshow( "Source image", src );

    // Вычисляем гистограмму исходного изображения
    histogram2Dcalculation(src, histImg);

    // Выводим гистограммы каждого цветового канала
    imshow("H-S Histogram", histImg );

    // Выровненное изображение

    // Применяем выравнивание к гистограмме каждого канала
    equalizeHist(bgr_planes[0], bgr_planes[0] );
    equalizeHist(bgr_planes[1], bgr_planes[1] );
    equalizeHist(bgr_planes[2], bgr_planes[2] );

    // Объединяем выровненные каналы в выровненное цветное изображение
    merge(bgr_planes, imageq );

    // Выводим выровненное изображение
    namedWindow("Equalized Image", 0 );
    imshow("Equalized Image", imageq );

    // Вычисляем двумерную гистограмму каналов H и S
    histogram2Dcalculation(imageq, histImgeq);

    // Выводим двумерную гистограмму
    imshow( "H-S Histogram Equalized", histImgeq );
    histogramRGcalculation(src, histHSorg);
    histogramRGcalculation(imageq, histHSeq);

    /// Применяем методы сравнения гистограмм
    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("Метод [%d] Исходное-Исходное, Исходное-Выровненное: %f,%f\n",
              i, orig_orig, orig_equ );
    }

    printf("Готово\n");

    waitKey();
}
```

В этом примере создается четыре окна: исходное изображение, выровненное цветное изображение и двумерные гистограммы каналов H и S исходного и выровненного изображения. Выводятся также четыре результата численного сопоставления, полученные при сравнении гистограммы исходного RGB-изображения с собой и с

выровненным RGB-изображением. В случае корреляции и пересечения чем больше значение метрики, тем точнее сопоставление. Для хи-квадрат и расстояния Бхаттачария наоборот: чем меньше результат, тем лучше. На рисунке ниже показаны результаты работы программы **ColourImageComparison**.



В главе 3 также рассмотрены существенные аспекты этой широкой темы – улучшение изображения посредством моделирования гистограммы.

Дополнительные сведения см. в книге Deniz O., Fernández M. M., Vállez N., Bueno G., Serrano I., Patón .A., Salido J. «OpenCV Essentials», выпущенной издательством Packt Publishing (`https://www.packtpub.com/applicationdevelopment/opencv-essentials`).

Резюме

Обработка изображения часто является первым этапом приложений компьютерного зрения, а потому в этой главе рассмотрены основные элементы: фундаментальные типы данных, доступ на уровне пикселей, типичные операции над изображениями, арифметические операции, сохранение данных и вычисление гистограмм.

В главе 3 книги «OpenCV Essentials» издательства Packt Publishing описаны дополнительные аспекты этой широкой темы: улучшение изображений, восстановление изображений посредством фильтрации и геометрическая коррекция.

В следующей главе мы рассмотрим вопросы коррекции и улучшения изображений путем сглаживания, повышения резкости, анализа разрешения, морфологических и геометрических преобразований, ретуширования и очистки от шумов.

ГЛАВА 3. Коррекция и улучшение изображений

В этой главе описываются методы коррекции и улучшения изображений. Иногда бывает необходимо уменьшить уровень шума, подчеркнуть или убрать какие-то детали изображения. Обычно для выполнения этих действий модифицируются и подвергаются тем или иным операциям как значения отдельных пикселей, так и пикселей в ближайшей окрестности. По определению, операции улучшения используются для проявления важных деталей изображения. К их числу относятся: уменьшение шума, сглаживание и улучшение границ. С другой стороны, цель операций коррекции – восстановить поврежденное изображение. В OpenCV операции обработки изображений находятся в модуле `imgproc`.

В этой главе будут рассмотрены следующие вопросы.

- Фильтрация изображений. Сюда входят сглаживание, повышение резкости и работа с пирамидами изображений.
- Применение морфологических операций: наращивания, эрозии, размыкания и замыкания.
- Геометрические преобразования (аффинные и перспективные).
- Ретуширование, используемое для реконструкции поврежденных частей изображения.
- Очистка от шумов, необходимая для снижения уровня шумов, вносимых устройством захвата изображений.

Фильтрация изображений

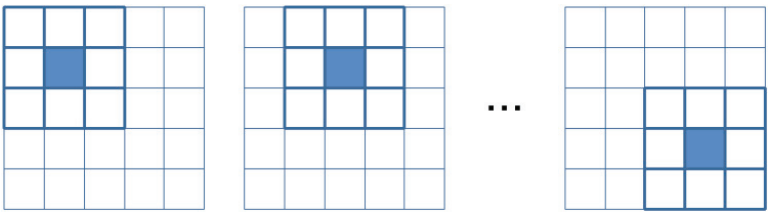
Фильтрацией называется процесс модификации или улучшения изображения. Подчеркивание одних частей и удаление других – примеры фильтрации. Фильтрация применяется к окрестностям пикселей.

Окрестностью называется множество пикселей, окружающих данный. В результате фильтрации определяется новое значение пикселя в позиции (x,y) путем применения тех или иных операций к значениям пикселей в его окрестности.

В OpenCV имеется несколько функций фильтрации, предназначенных для таких операций, как сглаживание или повышение резкости.

Сглаживание

Сглаживание, или размытие – операция, которая часто применяется для уменьшения шума, да и для других целей. Выполняется она путем применения линейных фильтров к изображению. Это означает, что новое значение пикселя (x_i,y_j) вычисляется как взвешенная сумма значений исходных пикселей в той же позиции и ее окрестности. Веса пикселей обычно хранятся в матрице, называемой **ядром**. Таким образом, фильтр можно представлять себе как скользящее окно коэффициентов.



Представление окрестности пикселя

Обозначим K – ядро, а I и O – входное и выходное изображение. Тогда новое значение пикселя в позиции (i,j) вычисляется по формуле:

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

В OpenCV для сглаживания чаще всего применяется медианный фильтр, фильтр Гаусса и двусторонний фильтр. Медианный фильтр очень хорош для устранения *зернистости* изображения («соль с перцем»), а фильтр Гаусса – в качестве предварительного шага для обнаружения границ. Что касается двусторонней фильтрации, то она применяется, в основном, для сглаживания изображений с сохранением резких границ.

Для этих целей в OpenCV включены следующие функции:

- `void boxFilter(InputArray src, OutputArray dst, int ddepth, Size ksize, Point anchor = Point(-1,-1), bool normalize = true, int borderType = BORDER_DEFAULT)`: это ящичный фильтр, т. е. все коэффициенты ядра равны. Если `normalize=true`, то каждый выходной пиксель вычисляется как среднее арифметическое соседей с коэффициентами ядра $1/n$, где n – число соседей. Если же `normalize=false`, то все коэффициенты равны 1. В аргументе `src` передается входное изображение, а в `dst` помещается отфильтрованное изображение. Параметр `ddepth` задает глубину выходного изображения, причем значение -1 означает, что нужно использовать такую же глубину, как у входного. Размер ядра задается параметром `ksize`. Параметр `anchor` определяет положение так называемого якорного пикселя. Подразумеваемое по умолчанию значение $(-1, -1)$ означает, что якорь находится в центре ядра. Наконец, параметр `borderType` определяет поведение на краях изображения.
- `void GaussianBlur(InputArray src, OutputArray dst, Size ksize, double sigmaX, double sigmaY = 0, int borderType=BORDER_DEFAULT)`: этот фильтр сворачивает каждую точку входного массива `src` с гауссовым ядром. Параметры `sigmaX` и `sigmaY` задают стандартные отклонения в направлении осей X и Y. Если `sigmaY` равно 0, он принимается равным `sigmaX`, а если нулю равны оба параметра, то они вычисляются по ширине и высоте, заданных в `ksize`.

Сверткой называется интеграл произведения функций $f(y)$ и $g(x-y)$ по dy.

- `void medianBlur(InputArray src, OutputArray dst, int ksize)`: этот фильтр обрабатывает каждый пиксель изображения, заменяя его медианой соседних пикселей.
- `void bilateralFilter(InputArray src, OutputArray dst, int d, double sigmaColor, double sigmaSpace, int borderType=BORDER_DEFAULT)`: аналогичен фильтру Гауссу в том смысле, что приписывает веса каждому соседнему пикселю, но вес состоит из двух компонент, одна из которых совпадает с используемой в фильтре Гаусса, в другая принимает во внимание разность интенсивностей обрабатываемого пикселя и его соседей.

Этой функции необходимо передать диаметр `d` окрестности пикселя, а также значения `sigmaColor` и `sigmaSpace`. Чем больше величина `sigmaColor`, тем более отдаленные цвета в окрестности пикселя будут смешиваться, т. е. будут порождаться более обширные области близкого цвета. А чем больше величина `sigmaSpace`, тем более далекие пиксели будут влиять друг на друга при условии, что их цвета достаточно близки.

- `void blur(InputArray src, OutputArray dst, Size ksize, Point anchor=Point(-1,-1), int borderType=BORDER_DEFAULT)`: размывает изображение нормированным ящичным фильтром. Эквивалентна функции `boxFilter` с параметром `normalize = true`. Используется такое ядро:

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

Функции `getGaussianKernel` и `getGaborKernel` можно использовать для создания пользовательских ядер, передаваемых функции `filter2D`.

В любом случае необходимо экстраполировать значения несуществующих пикселей за границей изображения. Для большинства фильтров OpenCV позволяет указать способ экстраполяции, а именно:

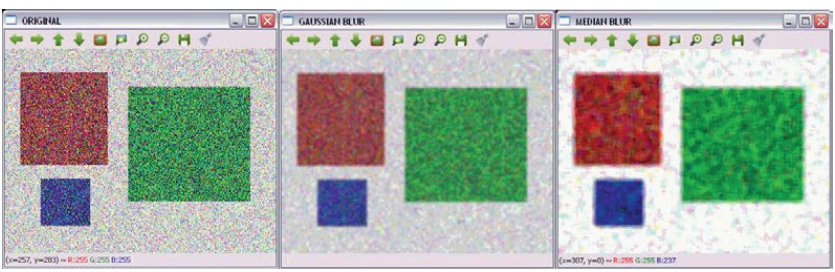
- `BORDER_REPLICATE`: повторять последнее известное значение пикселя: aaaaaa|abcdefgh|hhhhhhh;
- `BORDER_REFLECT`: отражать относительно границы изображения: fedcba|abcdefgh|hgfedcb;
- `BORDER_REFLECT_101`: отражать относительно границы изображения, не дублируя последний пиксель, примыкающий к границе: gfedcb|abcdefgh|gfedcba;
- `BORDER_WRAP`: добавлять значения, примыкающие к противоположной границе: cdefgh|abcdefgh|abcdefg;
- `BORDER_CONSTANT`: считать, что все пиксели за границей одинаковы.

Пример программы

В программе **Smooth** ниже показано, как загрузить изображение и применить к нему гауссово и медианное размытие, т. е. функции `GaussianBlur` и `medianBlur`:

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем фильтры
  Mat dst, dst2;
  GaussianBlur( src, dst, Size( 9, 9 ), 0, 0);
  medianBlur( src, dst2, 9);

  // Показываем результаты
  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;
}
```

Результат показан на рисунке ниже.



Исходное и размытое изображение после применения гауссова и медианного фильтра

Повышение резкости

Фильтры, повышающие резкость, используются, чтобы выделить границы и другие детали. Они основаны на вычислении первой и второй производной. Первая производная описывает градиент интенсивности изображения, а вторая определяется как дивергенция градиента. Поскольку при цифровой обработке изображений мы имеем дело с дискретными величинами (значениями пикселей), то для повышения резкости вычисляются дискретные производные.

Первые производные дают более жирные границы и широко применяются для выделения границ. Вторые же производные используются для улучшения изображения, потому что лучше реагируют на мелкие детали. Для вычисления производных чаще всего используются операторы Собеля и Лапласа.

Оператор Собеля вычисляет первую производную изображения I:

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

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

Модуль градиента тогда можно вычислить по формуле:

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

Дискретный оператор Лапласа (лапласиан) изображения вычисляется как свертка со следующим ядром:

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

Для этих целей в OpenCV имеются следующие функции:

- `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)`: вычисляет первую, вторую третью или смешанную производную изображения `src`,

применяя оператор Собеля. Параметр `ddepth` задает глубину выходного изображения, причем значение –1 означает, что нужно использовать такую же глубину, как у входного. Размер ядра задается параметром `ksize`, а порядок производных – параметрами `dx` и `dy`. Параметр `scale` позволяет задать коэффициент масштабирования вычисленных значений производной. Параметр `borderType` определяет поведение на краях изображения, а `delta` – значение, прибавляемое к результирующим пикселям перед сохранением в `dst`.

- `void Scharr(InputArray src, OutputArray dst, int ddepth, int dx, int dy, double scale = 1, double delta = 0, int borderType = BORDER_DEFAULT)`: более точно вычисляет производную для ядра размером 3×3. Вызов `Scharr(src, dst, ddepth, dx, dy, scale, delta, borderType)` эквивалентен вызову `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)`: вычисляет лапласиан изображения. Параметры такие же, как для операторов Собеля и Шарра, за исключением `ksize`. Если `ksize>1`, то лапласиан изображения `src` вычисляется путем сложения вторых производных по x и по y, вычисленных с помощью оператора Собеля. При `ksize=1` лапласиан вычисляется путем фильтрации изображения с ядром 3×3, которое содержит –4 в центре, 0 в углах, и 1 в остальных позициях.

Функцию `getDerivKernels` можно использовать для создания пользовательских ядер вычисления производных, которые передаются функции `sepFilter2D`.

Пример программы

В программе **Sharpen** ниже показано, как вычислять производные изображения с помощью операторов Собеля и Лапласа.

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

using namespace cv;

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

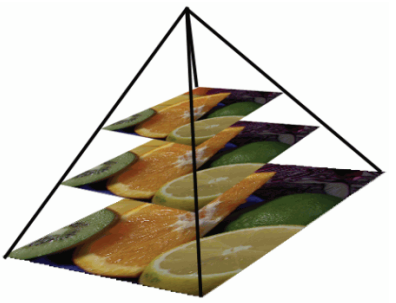
```
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем операторы Собеля и Лапласа
  Mat dst, dst2;
  Sobel(src, dst, -1, 1, 1 );
  Laplacian(src, dst2, -1 );

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " SOBEL ", WINDOW_AUTOSIZE );
  imshow( " SOBEL ", dst );
  namedWindow( " LAPLACIAN ", WINDOW_AUTOSIZE );
  imshow( " LAPLACIAN ", dst2 );

  waitKey();
  return 0;
}
```

Результат показан на рисунке ниже.

Контуры, полученные применением операторов Собеля и Лапласа

Работа с пирамидами изображений

Иногда работать с изображением фиксированного размера недостаточно, нужно иметь варианты оригинального изображения с разными разрешениями. Например, в задачах обнаружения объектов исследовать все изображение в поисках объектов слишком долго. В таком случае эффективнее производить поиск, начав с более низкого разрешения. Такой набор изображений называется **пирамидой** (или **mipmap**), поскольку изображения, упорядоченные от большего к меньшему и положенные друг на друга, напоминают пирамиду.



Пирамида Гаусса

Существуют пирамиды двух видов: Гаусса и Лапласа.

Пирамиды Гаусса

Пирамида Гаусса создается путем удаления каждой второй строки и столбца изображения нижнего уровня, а для получения значений пикселей на следующем уровне к изображению предыдущего уровня применяется фильтр Гаусса. На каждом шаге построения пирамиды ширина и высота изображения уменьшаются вдвое, а площадь – в четыре раза. В OpenCV для вычисления пирамиды Гаусса служат функции `pyrDown`, `pyrUp` и `buildPyramid`:

- `void pyrDown(InputArray src, OutputArray dst, const Size& dstsize = Size(), int borderType = BORDER_DEFAULT)`: производит выборку строк и столбцов и размытие изображения `src`, результат сохраняется в `dst`. Размер выходного изображения вычисляется как `Size((src.cols+1)/2, (src.rows+1)/2)`, если он не задан явно с помощью параметра `dstsize`.
- `void pyrUp(InputArray src, OutputArray dst, const Size& dstsize = Size(), int borderType = BORDER_DEFAULT)`: производит вычисление, обратное `pyrDown`.

- `void buildPyramid(InputArray src, OutputArrayOfArrays dst, int maxlevel, int borderType = BORDER_DEFAULT)`: строит пирамиду Гаусса для изображения `src`, создавая `maxlevel` новых изображений, которые сохраняются в массиве `dst` вслед за исходным изображением, которое помещается в элемент `dst[0]`. Таким образом, по завершении процедуры в массиве `dst` оказывается `maxlevel + 1` изображений.

Пирамиды используются также для сегментации. В OpenCV имеется функция для вычисления пирамид на первом шаге алгоритма сегментации со сдвигом среднего.

- `void pyrMeanShiftFiltering(InputArray src, OutputArray dst, double sp, double sr, int maxLevel = 1, TermCriteria termcrit = TermCriteria (TermCriteria::MAX_ITER + TermCriteria::EPS, 5, 1))`: реализует этап фильтрации в алгоритме сегментации методом сдвига среднего, получает изображение `dst`, содержащее линеаризованные градиенты и детальную текстуру. Параметры `sp` и `sr` задают радиусы пространственного и цветового окна соответственно.

Дополнительные сведения о сегментации методом сдвига среднего см. на странице `http://docs.opencv.org/trunk/doc/py_tutorials/py_video/py_meanshift/py_meanshift.html?highlight=meanshift`.

Пирамиды Лапласа

Для пирамид Лапласа в OpenCV нет отдельной реализации, они строятся по пирамидам Гаусса. Пирамиду Лапласа можно рассматривать как последовательность граничных изображений, в которых большая часть элементов равна нулю. *i*-ый уровень пирамиды Лапласа образуется как разность между *i*-ым уровнем пирамиды Гаусса и дополненным (*i*+1)-ым уровнем пирамиды Гаусса.

Пример программы

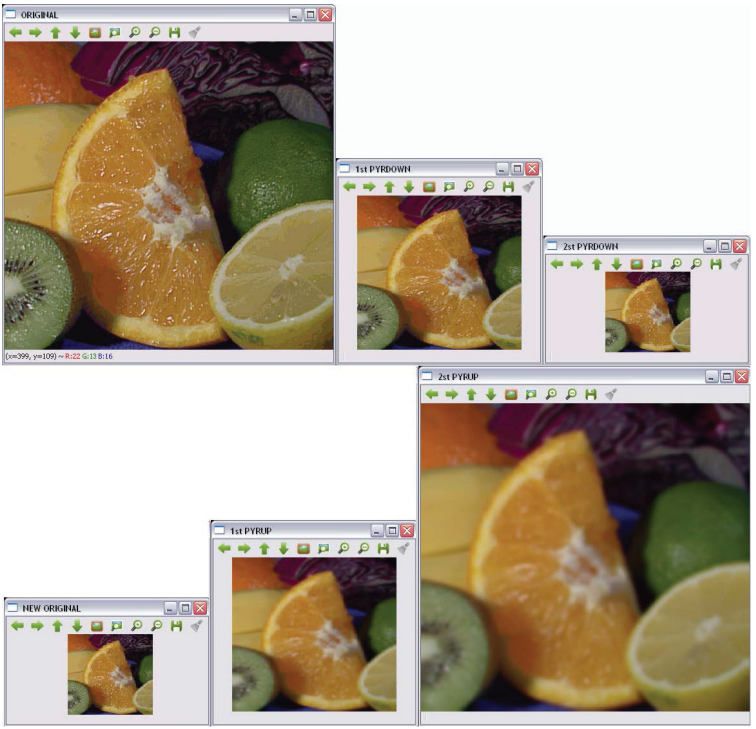
В программе **Pyramids** ниже показано, как получить два уровня пирамиды Гаусса с помощью функции `pyrDown` и выполнить обратную операцию с помощью функции `pyrUp`. Отметим, что получить исходное изображение с помощью `pyrUp` невозможно:

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Два раза применяем pyrDown
  Mat dst, dst2;
  pyrDown(src, dst);
  pyrDown(dst, dst2);

  // Показываем результаты
  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 );

  // Два раза применяем pyrUp
  pyrUp(dst2, dst);
  pyrUp(dst, src);

  // Показываем результаты
  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;
}
```

Результат показан на рисунке на следующей странице.

Морфологические операции

Морфологические операции применяют к изображению «структурный элемент», в результате чего формируется новое изображение, в котором пиксель в позиции (x_i,y_j) получается путем сравнения пикселя исходного изображения в той же позиции с его соседями. В зависимости от выбранного структурного элемента морфологическая операция может быть более чувствительной к одним формам и менее чувствительной к другим.



Исходное изображение и два уровня пирамиды Гаусса

Две основные морфологические операции – наращивание (dilation) и эрозия. В случае наращивания к границам объектов в изображении добавляются пиксели из фона, а в случае эрозии пиксели удаляются. Структурный элемент определяет, какие именно пиксели добавлять или удалять. При наращивании значение результирующего пикселя вычисляется как максимум, а при эрозии – как минимум всех пикселей в его окрестности.

Оригинал **Наращивание** **Эрозия**

Пример наращивания и эрозии

Путем комбинирования наращивания и эрозии можно определять и другие операции обработки изображений, например, размыкание, замыкание и морфологический градиент. Операция размыкания (открытия) определяется как эрозия, за которой следует наращивание, а операция замыкания (закрытия) наоборот – как наращивание с последующей эрозией. Таким образом, размыкание удаляет мелкие объекты, оставляя крупные, а замыкание удаляет небольшие дырки, оставляя более крупные. Морфологический градиент определяется как разность между наращиванием и эрозией изображения. На основе размыкания и замыкания определяются еще две операции: отбеливание (white top-hat) и зачернение (black top-hat). Операция отбеливания определяется как разность между исходным изображением и его размыканием, а операция зачернения – как разность между замыканием и исходным изображением. Все операции выполняются с одним и тем же структурным элементом.

В OpenCV наращивание, эрозия, размыкание и замыкание реализованы следующими функциями.

- `void dilate(InputArray src, OutputArray dst, InputArray kernel, Point anchor = Point(-1,-1), int iterations = 1, int borderType = BORDER_CONSTANT, const Scalar& borderValue = morphologyDefaultBorderValue())`: наращивает изображение `src` с помощью заданного структурного элемента и сохраняет результат в `dst`. Параметр `kernel` определяет структурный элемент, а `anchor` – позицию якорного пикселя в нем. Значение (–1, –1) означает, что якорь находится в центре. Операцию можно применять несколько раз, параметр `iterations` задает число повторений. Параметр `borderType` определяет поведение на краях – так же, как в описанных ранее фильтрах; если он не задан, подразумевается значение `BORDER_CONSTANT`.
- `void erode(InputArray src, OutputArray dst, InputArray kernel, Point anchor = Point(-1,-1), int iterations = 1, int borderType = BORDER_CONSTANT, const Scalar& borderValue = morphologyDefaultBorderValue())`: эродирует изображение с помощью заданного структурного элемента. Параметры такие же, как у функции `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())`: выполняет производные морфологические операции, определяемые параметром `op`, который может принимать следующие

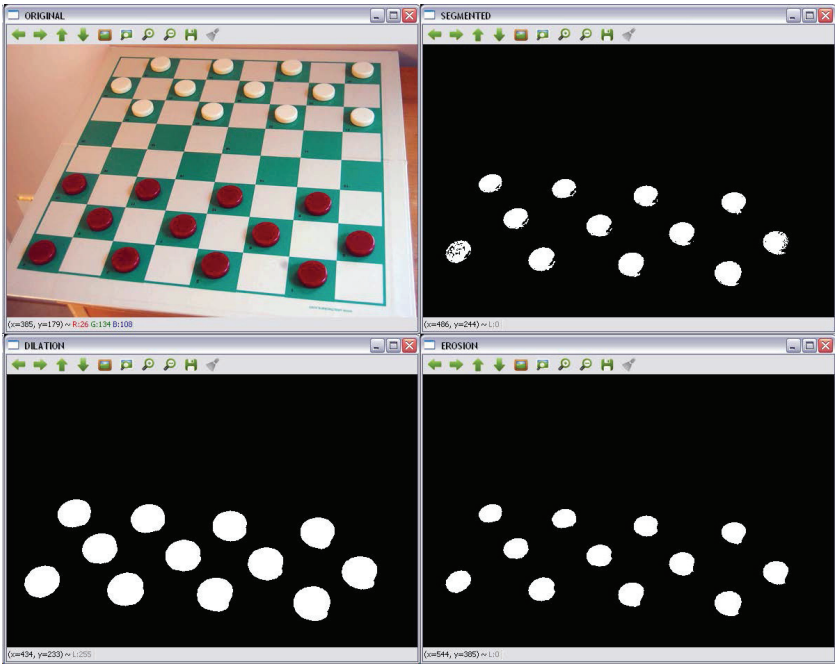
значения: MORPH_OPEN, MORPH_CLOSE, MORPH_GRADIENT, MORPH_TOPHAT и MORPH_BLACKHAT.

- Mat getStructuringElement(int shape, Size ksize, Point anchor = Point(-1,-1)): возвращает структурный элемент для морфологических операций указанного размера и формы. Допустимые значения: MORPH_RECT, MORPH_ELLIPSE и MORPH_CROSS.

Пример программы

В программе **Morphological** ниже показано, как можно сегментировать красные шашки на доске, применив бинаризацию (функция inRange) и уточнив результаты с помощью операций наращивания и эрозии. В качестве структурного элемента используется круг 15×15 пикселей.

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

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем фильтры
  Mat dst, dst2, dst3;
  inRange(src, Scalar(0, 0, 100), Scalar(40, 30, 255), dst);

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

  // Показываем результаты
  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;
}
```



Исходное изображение, сегментация по красному цвету, наращивание, эрозия

LUT-фильтры

Справочные таблицы (look-up table, LUT) часто применяются в пользовательских фильтрах, когда входные пиксели с одинаковыми значениями порождают одинаковые выходные пиксели. LUT-преобразование сопоставляет каждому входному пикселю значение, взятое из таблицы. Индексом в этой таблице является значение входной интенсивности, а в соответствующей ячейке находится значение выходной интенсивности. Поскольку преобразование определено для всех возможных значений интенсивности, время его применения к изображению значительно сокращается (обычно в изображениях пикселей гораздо больше, чем значений интенсивности).

Функция `LUT(InputArray src, InputArray lut, OutputArray dst, int interpolation = 0)` применяет LUT-преобразование к 8-разрядному изображению `src`. Следовательно, таблица, которую задает параметр `lut`, должна состоять из 256 элементов. Количество каналов

в массиве `lut` равно либо 1, либо or `src.channels`. Если `src` содержит более одного канала, а `lut` — только один канал, то один и тот же массив `lut` применяется ко всем каналам изображения.

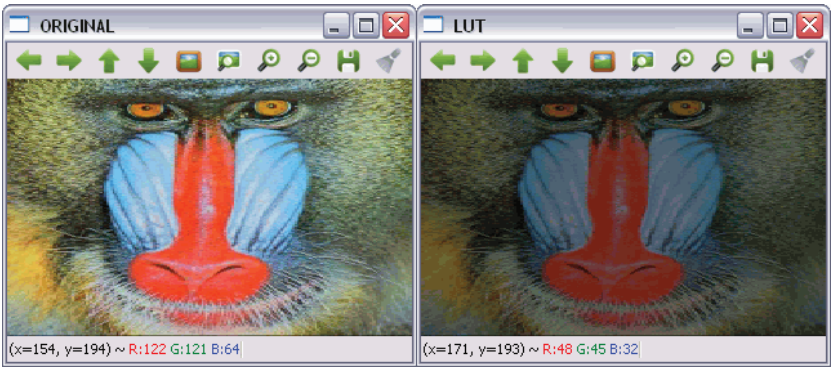
Пример программы

В программе **LUT** ниже показано, как вдвое уменьшить интенсивности пикселей изображения, пользуясь справочной таблицей. Таблицу необходимо предварительно инициализировать:

```
uchar * M = (uchar*)malloc(256*sizeof(uchar));
for(int i=0; i<256; i++){
  M[i] = i*0.5; // Результат округляется до целого
}
Mat lut(1, 256, CV_8UC1, M);
```

Здесь создается объект `Mat`, в элементах которого хранятся новые значения интенсивности. Ниже приведен полный код программы.

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Создаем справочную таблицу
  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);

  // Применяем функцию LUT
  Mat dst;
  LUT(src,lut,dst);

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " LUT ", WINDOW_AUTOSIZE );
  imshow( " LUT ", dst );

  waitKey();
  return 0;
}
```



Исходное изображение и оно же после применения LUT-преобразования

Геометрические преобразования

Геометрическое преобразование не изменяет содержание изображение, а деформирует его посредством изменения системы координат. То есть для каждого входного пикселя мы берем его координаты, применяем к ним преобразование и копируем пиксель, оказавшийся в точке с новыми координатами, в выходное изображение.

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

При этом может возникнуть две проблемы.

- **Экстраполяция:** значения $f_x(x, y)$ и $f_y(x, y)$ могут оказаться вне границ изображения. К геометрическим преобразованиям применяются те же режимы экстраполяции, что в случае фильтрации, плюс один дополнительный: `BORDER_TRANSPARENT`.
- **Интерполяция:** $f_x(x, y)$ и $f_y(x, y)$ обычно принимают вещественные значения. В OpenCV есть выбор между интерполяцией по ближайшему соседу и полиномиальной интерполяцией. В первом случае значение координаты с плавающей точкой округляется до ближайшего целого. Поддерживаются следующие методы интерполяции:
  - `INTER_NEAREST`: это описанная выше интерполяция по ближайшему соседу;
  - `INTER_LINEAR`: билинейная интерполяция, подразумевается по умолчанию;

- `INTER_AREA`: передискретизация по области пикселей;
- `INTER_CUBIC`: бикубическая интерполяция по окрестности 4×4;
- `INTER_LANCZOS4`: метод Ланцоша по окрестности 8×8.

OpenCV поддерживает аффинные преобразования (масштабирование, параллельный перенос, поворот и т. п.) и перспективные преобразования.

Аффинное преобразование

Аффинным называется геометрическое преобразование, при котором прямые переходят в прямые и дополнительно сохраняется отношение длин отрезков. Но углы и длины могут не сохраняться.

К аффинным преобразованиям относятся, в частности, масштабирование, параллельный перенос, поворот, скос и зеркальное отражение.

Масштабирование

Масштабированием изображения называется изменение его размера (увеличение или уменьшение). В OpenCV этой цели служит функция `void resize(InputArray src, OutputArray dst, Size dsize, double fx = 0, double fy = 0, int interpolation = INTER_LINEAR)`. Помимо входного (`src`) и выходного (`dst`) изображения, она принимает параметры, определяющие новый размер. Если новый размер, заданный параметром `dsize`, отличен от 0, то коэффициенты масштабирования `fx` и `fy` должны быть равны 0 и вычисляются по `dsize` и размеру исходного изображения. Если `fx` и `fy` отличны от 0, а `dsize` равно 0, то значение `dsize` вычисляется на основе прочих параметров. Операцию масштабирования можно представить такой матрицей преобразования:

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

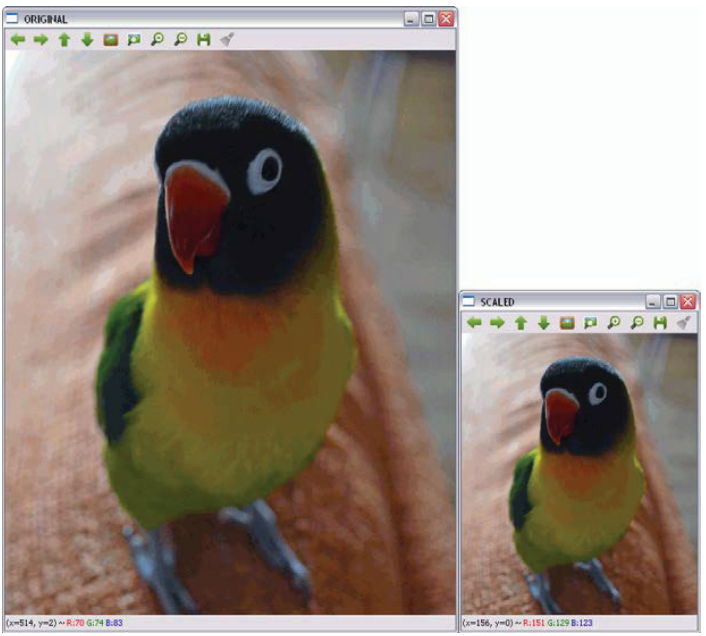
Здесь s_x и s_y – коэффициенты масштабирования по осям x и y.

Пример программы

В программе **Scale** ниже показано, как масштабировать изображение с помощью функции `resize`.

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем масштабирование
  Mat dst;
  resize(src, dst, Size(0,0), 0.5, 0.5);

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " SCALED ", WINDOW_AUTOSIZE );
  imshow( " SCALED ", dst );

  waitKey();
  return 0;
}
```

Результат показан на рисунке ниже.



Исходное и масштабированное изображение; `fx` и `fy` равны 0.5

Параллельный перенос

Параллельный перенос – это перемещение изображение в заданном направлении на заданное расстояние. Поэтому его можно представить вектором (t_x, t_y) или такой матрицей преобразования:

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

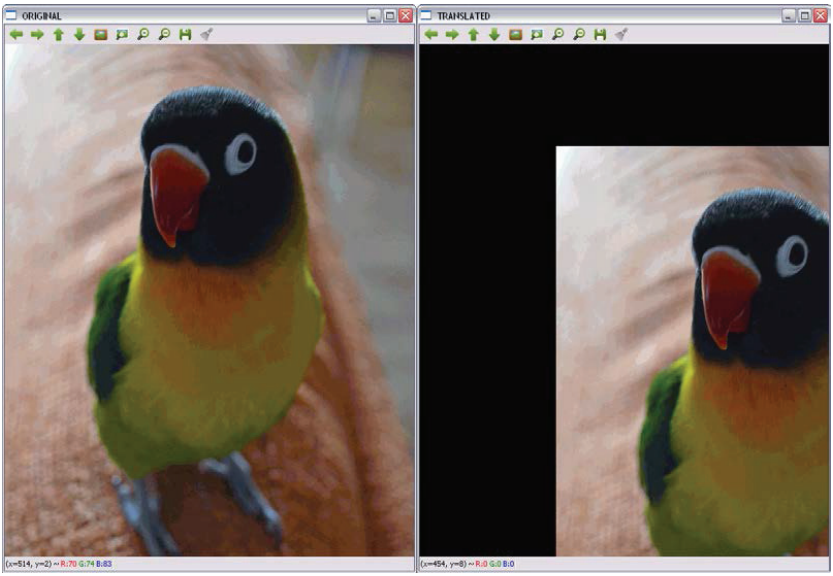
В OpenCV для применения параллельного переноса служит функция `void warpAffine(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags = INTER_LINEAR, int borderMode = BORDER_CONSTANT, const Scalar& borderValue = Scalar())`. Параметр `M` задает матрицу преобразования `src` в `dst`. Параметр `flags` определяет способ интерполяции и может комбинироваться с необязательным флагом `WARP_INVERSE_MAP`, означающим, что `M` – обратное преобразование (`dst` в `src`). Параметр `borderMode` определяет способ экстраполяции, а `borderValue` используется, когда `borderMode` равен `BORDER_CONSTANT`.

Пример программы

В программе **Translation** ниже показано, как параллельно перенести изображение с помощью функции `warpAffine`.

```
#include "opencv2/opencv.hpp"
using namespace cv;
int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

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

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " TRANSLATED ", WINDOW_AUTOSIZE );
  imshow( " TRANSLATED ", dst );

  waitKey();
  return 0;
}
```

Результат показан на рисунке ниже.



Исходное изображение и оно же, сдвинутое на 500 пикселей по горизонтали и на 150 по вертикали

Поворот изображения

Поворот определяет углом θ. OpenCV поддерживает поворот относительно заданной точки с масштабированием, это преобразование описывается матрицей вида:

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

Здесь x и y – координаты центра вращения, а sf – коэффициент масштабирования.

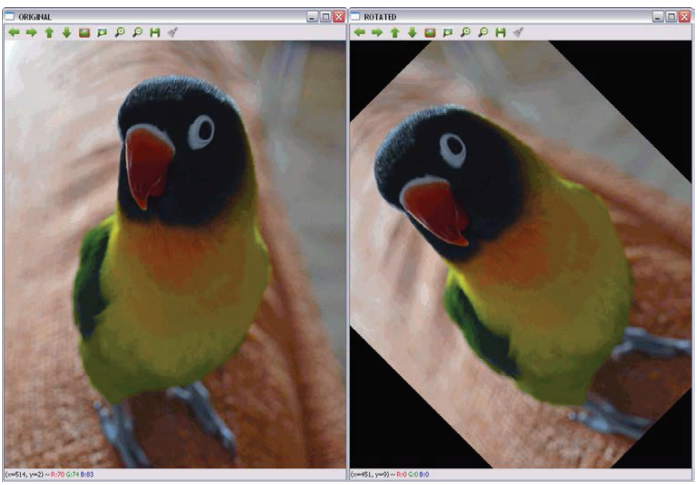
Поворот, как и параллельный перенос, реализуется функцией `warpAffine`, но для создания матрицы преобразования `M` используется функция `Mat getRotationMatrix2D(Point2f center, double angle, double scale)`. Параметр `center`, как легко догадаться, определяет центр вращения, `angle` – угол поворота (в направлении против часовой стрелки), а `scale` – коэффициент масштабирования.

Пример программы

В программе **Rotate** ниже показано, как повернуть изображение с помощью функции `warpAffine`. Матрицу поворота на 45 градусов возвращает вызов `getRotationMatrix2D(Point2f(src.cols/2, src.rows/2), 45, 1)`.

```
#include "opencv2/opencv.hpp"
using namespace cv;
int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

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

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " ROTATED ", WINDOW_AUTOSIZE );
  imshow( " ROTATED ", dst );

  waitKey();
  return 0;
}
```

Результат показан на рисунке ниже.

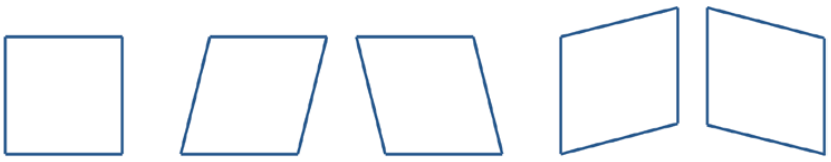


Исходное изображение и оно же после поворота на 45 градусов

Скос

Преобразование скоса сдвигает все точки в одном направлении на величину, пропорциональную взятому со знаком расстоянию от точки до прямой, параллельной этому направлению. Поэтому форма геометрической фигуры обычно искажается: квадраты преобразуется в параллелограммы, а окружности – в эллипсы. Однако скос сохраняет площадь фигуры, параллельность прямых и отношение расстояний между коллинеарными точками. Именно скос определяет основное различие между прямым и курсивным начертанием букв.

Скос определяется углом θ.



Оригинальное изображение и оно же, скошенное на 45 градусов

Матрицы скоса на угол θ по горизонтали и по вертикали выглядят так:

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

Поскольку они похожи на матрицы других преобразований, то для реализации скоса применяется все та же функция `warpAffine`.

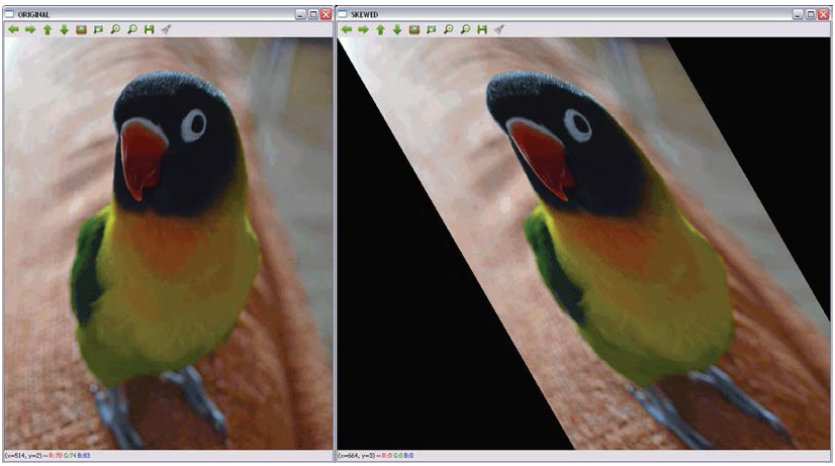
В большинстве случае необходимо изменить размер получившегося изображения и (или) применить к нему параллельный перенос (изменить последний столбец в матрице скоса), чтобы оно было видно целиком и располагалось в центре.

Пример программы

В программе **Skew** ниже показано, как выполнить скос изображения на угол θ = π/3 по горизонтали с помощью функции `warpAffine`.

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем скос
  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));

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " SKEWED ", WINDOW_AUTOSIZE );
  imshow( " SKEWED ", dst );

  waitKey();
  return 0;
}
```

Результат показан на рисунке ниже.



Исходное изображение и оно же после скоса по горизонтали

Зеркальное отражение

Поскольку по умолчанию отражение производится относительно осей x и y, после него необходимо выполнить параллельный перенос (изменить последний столбец матрицы преобразования). Таким образом, матрица отражения имеет вид:

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

Здесь t_x – число столбцов, а t_y – число строк изображения.

Как и предыдущие преобразования, отражение реализуется функцией `warpAffine`.

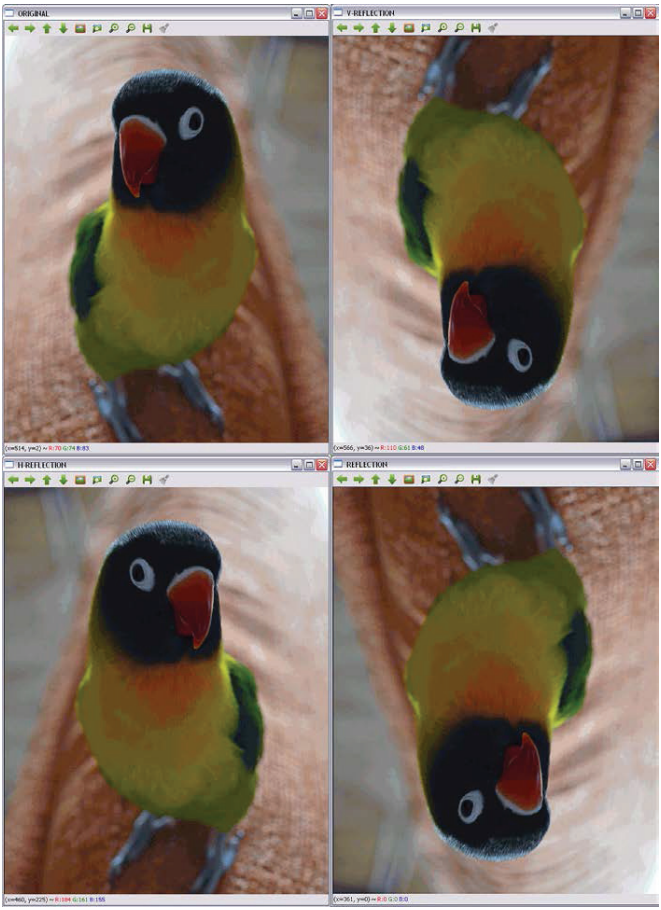
Функция `warpAffine` позволяет выполнить и другие аффинные преобразования, нужно лишь задать подходящую матрицу.

Пример программы

В программе **Reflect** ниже показано, как выполнить отражение относительно горизонтальной оси, вертикальной оси и обеих осей одновременно с помощью функции `warpAffine`.

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Применяем отражения
  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());

  // Показываем результаты
```

```
    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;
}
```



Исходное изображение и оно же после отражения относительно осей X, Y и обеих одновременно

Перспективное преобразование

Для перспективного преобразования необходима матрица 3 × 3, хотя оно и применяется к двумерным изображениям. Прямые линии после преобразования остаются прямыми, но пропорции изменяются. Найти нужную матрицу сложнее, чем в аффинном случае. Она определяется координатами четырех точек входного изображения и соответствующих им точек выходного изображения.

Зная четыре пары точек, мы можем найти матрицу перспективного преобразования с помощью функции `getPerspectiveTransform`. А получив матрицу, можно выполнить преобразование, передав ее функции `warpPerspective`.

Ниже подробно описаны обе функции.

- `Mat getPerspectiveTransform(InputArray src, InputArray dst)` и `Mat getPerspectiveTransform(const Point2f src[], const Point2f dst[])`: возвращает матрицу перспективного преобразования, вычисленную по массивам `src` и `dst`.
- `void warpPerspective(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar())`: применяет перспективное преобразование `M` к изображению `src` и получает новое изображение `dst`. Остальные параметры такие же, как в вышеописанных функциях геометрических преобразований.

Пример программы

В программе **Perspective** ниже показано, как применить перспективное преобразование к изображению с помощью функции `warpPerspective`. Для вычисления матрицы преобразования мы должны указать координаты четырех точек на первом изображении и четырех других на втором изображении. Выберем такие точки:

```
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);
```

Ниже приведен код программы.

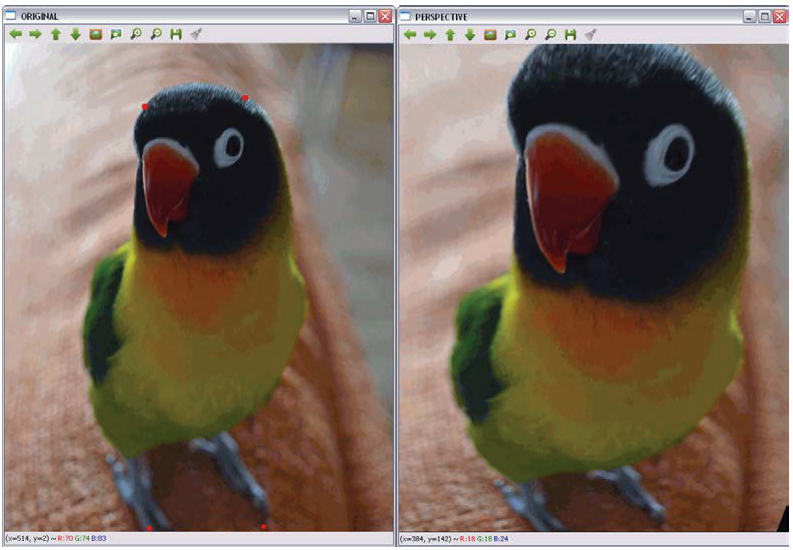
```
#include "opencv2/opencv.hpp"

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  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);

  // Получаем матрицу перспективного преобразования и применяем ее
  Mat M = getPerspectiveTransform(src_verts,dst_verts);
  warpPerspective(src,dst,M,src.size());

  // Показываем результаты
  namedWindow( " ORIGINAL ", WINDOW_AUTOSIZE );
  imshow( " ORIGINAL ", src );
  namedWindow( " PERSPECTIVE ", WINDOW_AUTOSIZE );
  imshow( " PERSPECTIVE ", dst );

  waitKey();
  return 0;
}
```

Результаты показаны на рисунке на следующей странице.

Ретуширование

Ретушированием называется восстановление поврежденных частей изображений или видео. Этот процесс известен также под названием интерполяции изображения или видео. Идея заключается в имитации действий реставраторов древностей. Сейчас, во времена широкого распространения цифровых камер ретуширование автоматизи-

ровано и применяется не только для замазывания царапин, но и для других целей, например удаления объектов или текста.



Результаты перспективного преобразования, определенного отмеченными точками

OpenCV поддерживает алгоритм ретуширования, начиная с версии 2.4. Для этой цели используется следующая функция:

- `void inpaint(InputArray src, InputArray inpaintMask, OutputArray dst, double inpaintRadius, int flags)`: восстанавливает области исходного изображения `src`, которым соответствуют ненулевые значения маски `inpaintMask`. Параметр `inpaintRadius` определяет окрестность, используемую в алгоритме. В OpenCV реализованы два алгоритма:
  - `INPAINT_NS`: основан на методе Навье-Стокса;
  - `INPAINT_TELEA`: этот метод предложил Александру Телеа.

Наконец, параметр `dst` – выходное изображение.

Дополнительные сведения об алгоритмах ретуширования, используемых в OpenCV, смотрите на странице `http://www.ifp.illinois.edu/~yuhuang/ inpainting.html`.

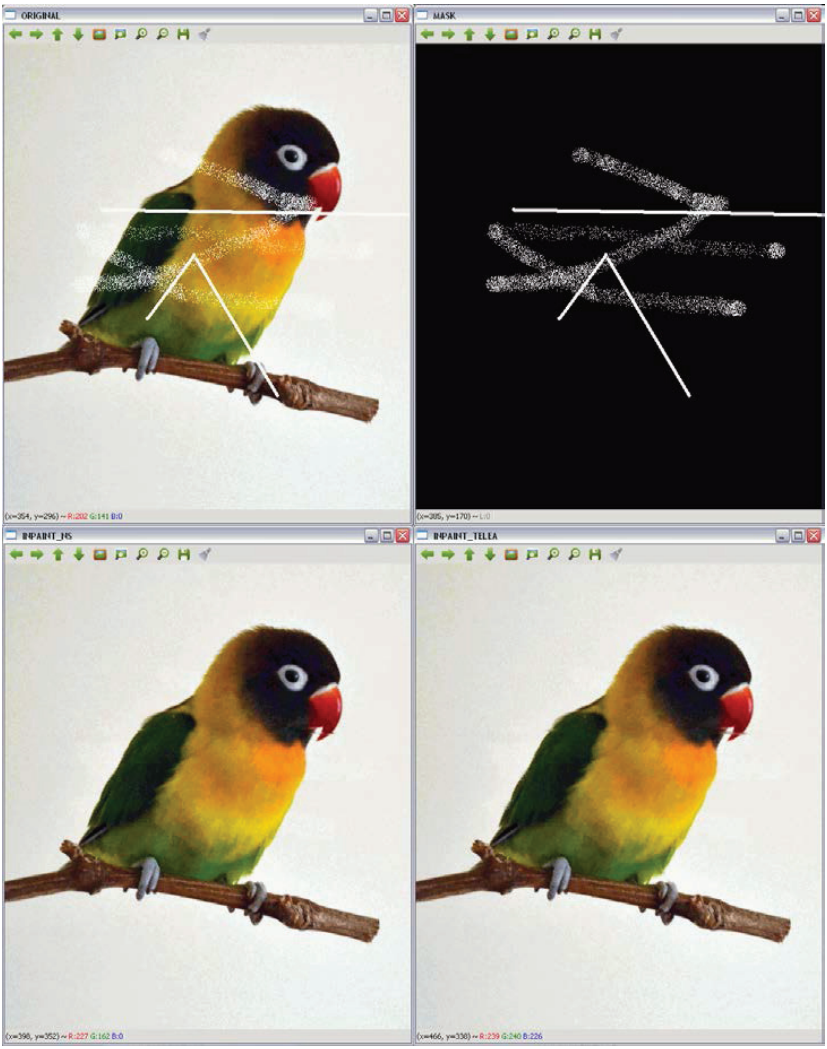
Для ретуширования видео его рассматривают как последовательность изображений и применяют алгоритм к каждому из них.

Пример программы

В программе **inpainting** ниже показано, как применить функцию `inpaint` для ретуширования областей изображения, заданных маской.

```
#include "opencv2/opencv.hpp"
using namespace cv;
int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Читаем маску
  Mat mask;
  mask = imread(argv[2]);
  cvtColor(mask, mask, COLOR_RGB2GRAY);

  // Применяем алгоритмы ретуширования
  Mat dst, dst2;
  inpaint(src, mask, dst, 10, INPAINT_TELEA);
  inpaint(src, mask, dst2, 10, INPAINT_NS);

  // Показываем результаты
  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;
}
```

Результат показан на рисунке ниже.



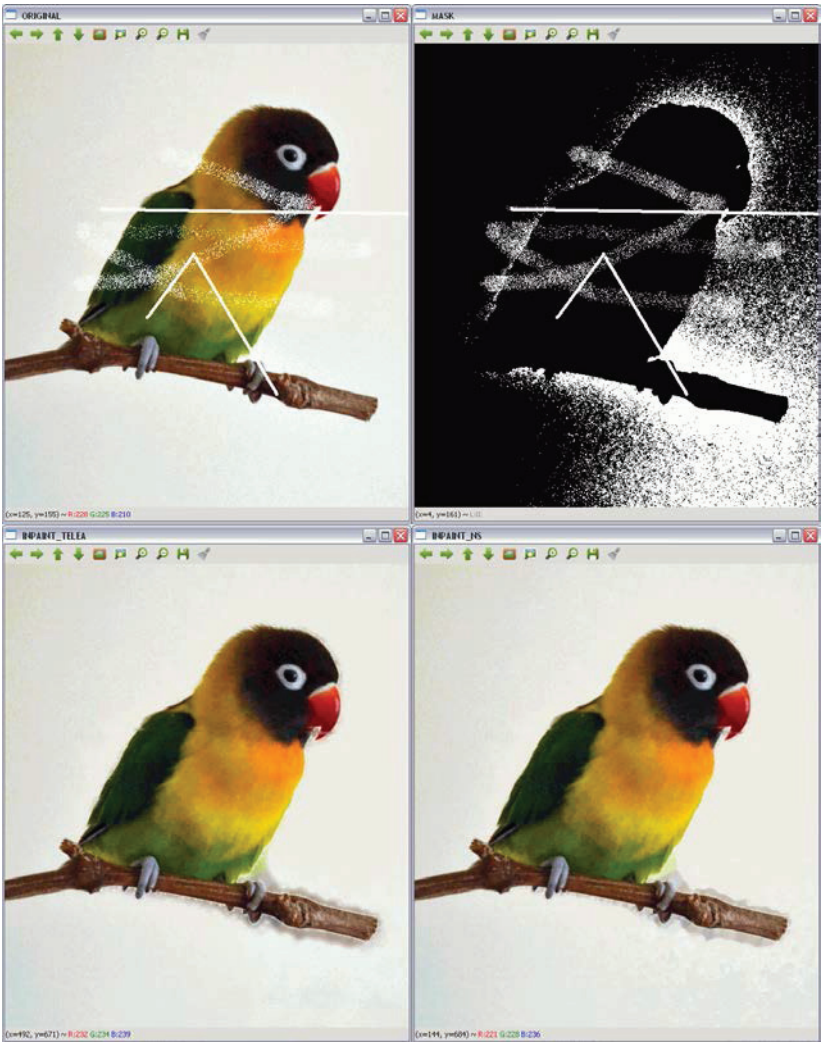
Результат ретуширования

В первом ряду показаны исходное изображение и маска. Во втором ряду слева – результат ретуширования методом Телеа, а справа – методом Навье-Стокса.

Получить маску ретуширования совсем не просто. В примере программы **inpainting2** показано, как можно получить маску путем бинаризации исходного изображения с помощью функции `threshold(mask, mask, 235, 255, THRESH_BINARY)`:

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Создаем маску
  Mat mask;
  cvtColor(src, mask, COLOR_RGB2GRAY);
  threshold(mask, mask, 235, 255, THRESH_BINARY);

  // Применяем алгоритмы ретуширования
  Mat dst, dst2;
  inpaint(src, mask, dst, 10, INPAINT_TELEA);
  inpaint(src, mask, dst2, 10, INPAINT_NS);

  // Показываем результаты
  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;
}
```

Результат показан на рисунке ниже.



Результат ретуширования, когда маска заранее неизвестна

В первом ряду показаны исходное изображение и выделенная из него маска. Во втором ряду слева – результат ретуширования методом Телеа, а справа – методом Навье-Стокса.

Из этого примера видно, что получить идеальную маску не всегда возможно. Иногда включаются посторонние части изображения, например фон или шум. Но результаты ретуширования все равно приемлемы, т. к. получающиеся изображения близки к неповрежденным.

Очистка от шумов

Очисткой от шумов называется удаление шума из сигналов, полученных от аналогового или цифрового устройства. В этом разделе мы сосредоточимся на очистке от шумов цифровых изображений и видео.

Хотя сглаживание и медианный фильтр сами по себе неплохо справляются с очисткой изображения от шумов, в OpenCV имеются и другие алгоритмы для этой цели. Это алгоритм нелокального среднего и **TV-L1 (полной вариации в L_1)**. Основная идея алгоритма нелокального среднего заключается в том, чтобы заменить цвет пикселя средним цветом, вычисленным по нескольким областям изображения, похожим на ту, что содержит данный пиксель. Алгоритм же TV-L1 является вариационной моделью очистки от шумов и реализован путем решения двойственной задачи оптимизации.

Дополнительные сведения об алгоритмах нелокального среднего и TV-L1 смотрите на страницах `http://www.ipol.im/pub/art/2011/bcm_nlm` и `http://znah.net/rof-and-tv-l1-denoisingwith-primal-dual-algorithm.html` соответственно.

В OpenCV имеется четыре функции для очистки от шумов цветных и полутоновых изображений методом нелокального среднего и одна, реализующая алгоритм TV-L1, а именно:

- `void fastNlMeansDenoising(InputArray src, OutputArray dst, float h = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: очищает от шумов одного полутоновое изображение `src`. Параметры `templateWindowSize` и `searchWindowSize` задают размеры в пикселях блока-трафарета, используемого для вычисления весов, и окна, используемого для вычисления взвешенного среднего для данного пикселя. Оба числа должны быть нечетными, рекомендуемые значения равны 7 и 21 соответственно. Параметр `h` регулирует работу алгоритма. Чем больше `h`, тем больше удаляется дефектов, вызванных шумов, но ценой про-

падания части деталей. Результирующее изображение сохраняется в `dst`.

- `void fastNlMeansDenoisingColored(InputArray src, OutputArray dst, float h = 3, float hForColorComponents = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: модификация предыдущей функции для цветных изображений. Преобразует изображение `src` в цветовое пространство CIELAB, а затем раздельно очищает от шумов компоненты L и AB, применяя функцию `fastNlMeansDenoising`.
- `void fastNlMeansDenoisingMulti(InputArrayOfArrays srcImgs, OutputArray dst, int imgToDenoiseIndex, int temporalWindowSize, float h = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: применяет очистку от шумов к последовательности изображений. Функции передаются два дополнительных параметра: `imgToDenoiseIndex` и `temporalWindowSize`. Первый задает индекс очищаемого изображения в последовательности `srcImgs`, а второй говорит, сколько соседних изображений использовать для очистки. Это число должно быть нечетным.
- `void fastNlMeansDenoisingColoredMulti(InputArrayOfArrays srcImgs, OutputArray dst, int imgToDenoiseIndex, int temporalWindowSize, float h = 3, float hForColorComponents = 3, int templateWindowSize = 7, int searchWindowSize = 21)`: основана на функциях `fastNlMeansDenoisingColored` и `fastNlMeansDenoisingMulti`. Объяснение параметров приведено выше.
- `void denoise_TVL1(const std::vector<Mat>& observations, Mat& result, double lambda, int niters)`: записывает в `result` изображение, полученное очисткой зашумленных изображений в векторе `observations`. Параметры `lambda` и `niters` управляют поведением алгоритма и количеством итераций.

Пример программы

В программе **denoising** ниже показано, как с помощью описанной выше функций `fastNlMeansDenoisingColored` очистить от шумов цветное изображение. Поскольку исходное изображение не зашумлено, придется добавить в него шум искусственно. Для этого служат строки:

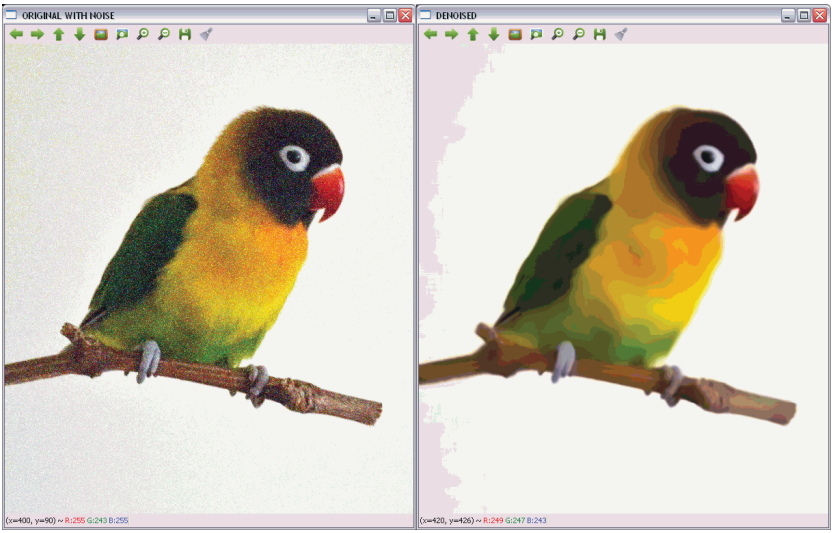
```
Mat noisy = src.clone();
Mat noise(src.size(), src.type());
randn(noise, 0, 50);
noisy += noise;
```

Матрица `Mat` имеет тот же размер и тип, что исходное изображение, а хранится в ней случайный шум, сгенерированный функцией `randn`. В результате сложения клонированного изображения с исходным получается зашумленное изображение.

Ниже приведен полный код программы.

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

using namespace cv;

int main( int argc, char** argv )
{
  // Читаем исходный файл
  Mat src;
  src = imread(argv[1]);

  // Добавляем случайный шум
  Mat noisy = src.clone();
  Mat noise(src.size(), src.type());
  randn(noise, 0, 50);
  noisy += noise;

  // Применяем алгоритм очистки от шумов
  Mat dst;
  fastNlMeansDenoisingColored(noisy, dst,30,30,7,21);

  // Показываем результаты
  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;
}
```

Результат показан на рисунке ниже.



Результат очистки изображения от шумов

Резюме

В этой главе мы рассказали о методах улучшения и коррекции изображений, а именно: очистка от шумов, выделение границ, морфологические операции, геометрические преобразования и восстановление поврежденных изображений. Были представлены различные варианты, чтобы у читателя сложилось более полное представление о том, что предлагает OpenCV.

В следующей главе будут рассмотрены цветовые пространства и способы перехода от одного к другому. Кроме того, мы рассмотрим сегментацию на основе цветовых пространств и методы цветопереноса.

ГЛАВА 4.
Работа с цветом

Цвет – это характеристика, воспринимаемая зрительной системой человека в ответ на свет видимой области спектра, попадающий на сетчатку. Цвет изображения содержит большой объем информации, которую можно использовать для анализа изображений, идентификации и выделения объектов. Все эти действия выполняются путем рассмотрения значений пикселей в некотором цветовом пространстве. В этой главе будут рассмотрены следующие вопросы:

- цветовые пространства в OpenCV и способы преобразования изображения из одного в другое;
- пример сегментации изображения путем рассмотрения цветового пространства, в котором оно определено;
- перенос цветов из одного изображения в другое.

Цветовые пространства

Зрительная система человека способна различать сотни тысяч цветов. Для этого служат три типа колбочковидных клеток сетчатки, реагирующих на падающий свет. Поэтому большую часть аспектов восприятия цвета человеком можно смоделировать с помощью трех числовых компонентов, называемых первичными цветами.

Для задания цвета с помощью трех или более характеристик применяются различные системы, называемые **цветовыми пространствами** или **цветовыми моделями**. Какую выбрать, зависит от выполняемых над изображением операций. Например, в цветовом пространстве RGB изменение яркости затрагивает все три канала, а в некоторых алгоритмах обработки изображений это нежелательно. В следующем разделе описываются цветовые пространства, используемые в OpenCV, и методы преобразования одного в другое.

Преобразования цветовых пространств (cvtColor)

В OpenCV более 150 методов преобразования цветовых пространств. В модуле `imgproc` имеется функция `void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0)`, принимающая следующие аргументы:

- `src`: входное изображение, представленное 8-разрядными целыми без знака, 16-разрядными целыми без знака (CV_16UC) или числами с плавающей точкой одинарной точности;
- `dst`: выходное изображение, имеющее такие же размер и глубину, как входное;
- `code`: код преобразования цветового пространства. Символическое значение параметра имеет вид `COLOR_SPACEsrc2SPACEdst`, например: `COLOR_BGR2GRAY` или `COLOR_YCrCb2BGR`.
- `dstCn`: число каналов выходного изображения. Если этот параметр равен 0 или опущен, то число каналов автоматически определяется по `src` и `code`.

Примеры применения этой функции будут приведены в последующих разделах.

Функция `cvtColor` умеет преобразовывать только из RGB в другое цветовое пространство или из другого пространства в RGB, поэтому для преобразования между двумя пространствами, отличными от RGB, нужно выполнить промежуточное преобразование в RGB.

Далее обсуждаются различные цветовые пространства, поддерживаемые в OpenCV.

RGB

RGB представляет собой аддитивную модель, в которой изображение состоит из трех независимых каналов, или плоскостей: красного, зеленого и синего (и дополнительного четвертого канала, определяющего прозрачность, который обычно называют альфа-каналом). Для задания цвета пикселя нужно указать значения всех трех компонент. Чем больше значение, тем пиксель ярче. Это цветовое пространство

находит широкое применение, потому что соответствует трем фоторецепторам глаза человека.

Цветовую модель, подразумеваемую по умолчанию в OpenCV, часто называют RGB, хотя на самом деле пиксели хранятся в формате BGR (с измененным порядком каналов).

Пример программы

В программе **BGRsplit** ниже показано, как загрузить RGB-изображение, разделить его на каналы и показать каждый канал в полутоновом и цветном виде. Первая часть программы загружает и показывает изображение.

```
#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)
{
  // Загружаем изображение
  Mat image = imread("BGR.png");
  imshow("Picture",image);
```

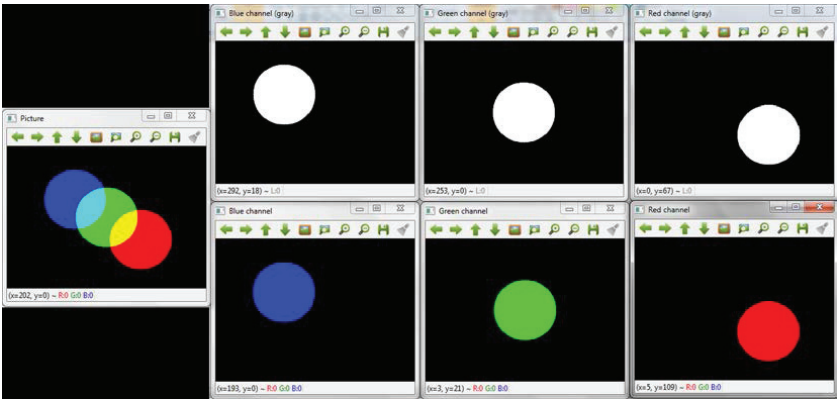
Следующая часть разделяет изображение на каналы и показывает их.

```
  vector<Mat> channels;
  split( image, channels );

  // показываем каналы в полутоновом режиме
  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]);

  // показываем каналы в формате 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;
}
```

Обратите внимание на использование функции `split(InputArray m, OutputArrayOfArrays mv)` для разделения изображения `m` на три канала и сохранения их в векторе `mv` типа `Mat`. Обратная ей функция `void merge(InputArrayOfArrays mv, OutputArray dst)` объединяет каналы `mv` в одно изображение `dst`. Функция, которую мы назвали `showSeparatedChannels`, создает три цветных изображения, представляющих отдельные каналы. Для каждого канала она генерирует вектор `vector<Mat> aux`, включающий сам канал и два дополнительных канала, в которых все значения пикселей равны 0, а затем объединяет все три канала в одно изображение, в котором присутствует только один канал. Она будет еще не раз использоваться в примерах из этой главы, а ее код приведен ниже.

```
vector<Mat> showSeparatedChannels(vector<Mat> channels){
    vector<Mat> separatedChannels;

    // создаем по одному изображению на каждый канал
    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;
}
```



Исходное RGB-изображение и оно же с разделенными каналами

Полутоновая модель

В полутоновой модели каждый пиксель представляется одним значением, несущим информацию о яркости, т. е. в изображении присутствуют только оттенки серого цвета. Для преобразования между полутоновым пространством (`GRAY`) и RGB применяется функция `cvtColor` с кодами `COLOR_BGR2GRAY`, `COLOR_RGB2GRAY`, `COLOR_GRAY2BGR` и `COLOR_GRAY2RGB`. Преобразования вычисляются по следующим формулам:

```
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)
```

Как видим, восстановить цвета, имея только полутоновое изображение, невозможно.

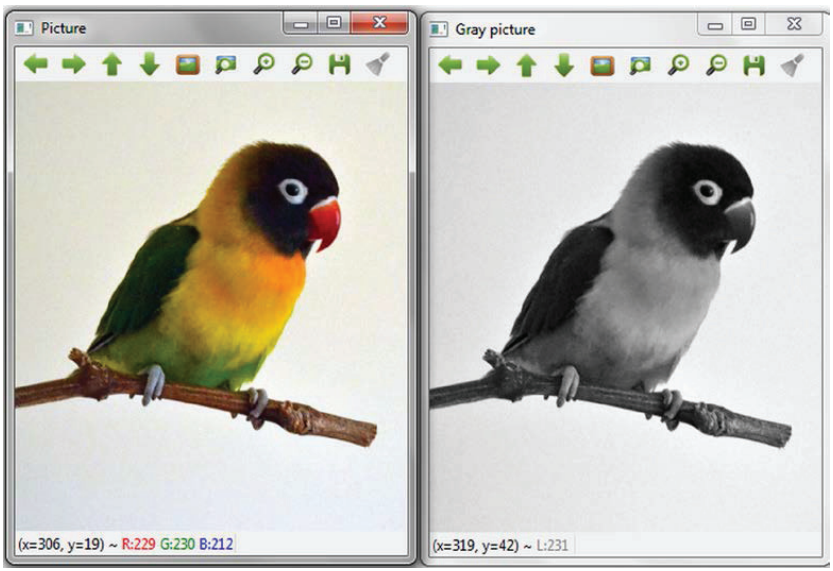
Пример программы

В программе **Gray** ниже показано, как преобразовать RGB-изображение в полутоновое.

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

using namespace cv;

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

```
{
  // Загружаем изображение
  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;
}
```

Результат показан на рисунке ниже.



Исходное RGB-изображение и оно же после преобразования в полутоновое

У этого метода преобразования RGB-изображения в полутоновое есть недостаток: теряется контрастность оригинала. В главе 6 «Вычислительная фотография» описан процесс обесцвечивания, лишенный этого недостатка.

CIE XYZ

В системе CIE XYZ цвет описывается фотометрической яркостью излучения Y, которая связана с чувствительностью глаза к яркости света и двумя дополнительными каналами X и Z, стандартизованными **Международной комиссией по освещению (CIE)** на основе результатов экспериментов с группой испытуемых. Это цветовое пространство используется в измерительных приборах, например колориметрах и спектрофотометрах, и полезно, когда требуется согласованная информация о цветах, не зависящая от устройства. Основная проблема состоит в том, что цвета масштабируются неодинаково, что послужило причиной для разработки цветовых моделей CIE L*a*b* и CIE L*u*v*.

Для преобразования между пространствами RGB и CIE XYZ применяется функция `cvtColor` с кодами `COLOR_BGR2XYZ`, `COLOR_RGB2XYZ`, `COLOR_XYZ2BGR` и `COLOR_XYZ2RGB`. Преобразования вычисляются следующим образом:

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

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

Пример программы

В программе **CIExyz** ниже показано, как преобразовать RGB-изображение в цветовое пространство CIE XYZ. Каналы разделяются и показываются по отдельности в полутоновом режиме и в цвете. Первая часть программы загружает и преобразует изображение.

```
#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)
```

```
{
  // Загружаем изображение
  Mat image = imread("Lovebird.jpg");
  imshow("Picture",image);

  // Преобразуем в CIE XYZ
  cvtColor(image,image,COLOR_BGR2XYZ);
```

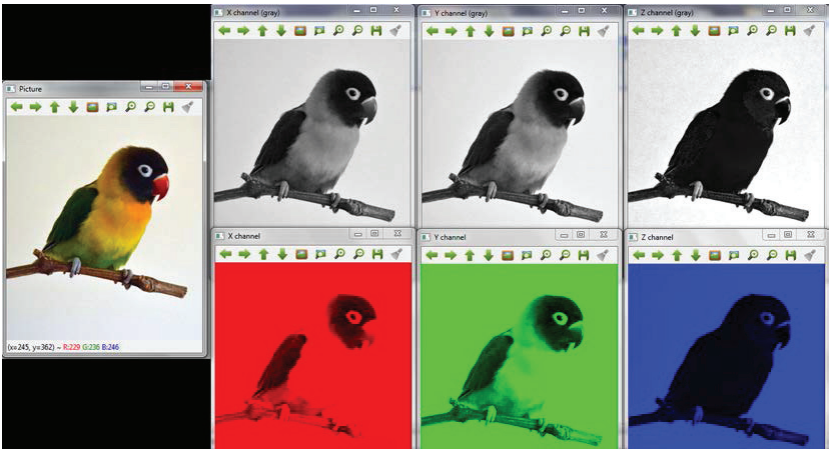
Вторая часть разделяет и показывает каналы CIE XYZ.

```
  vector<Mat> channels;

  split( image, channels );

  // показываем каналы в полутоновом режиме
  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]);

  // показываем каналы в формате BGR
  vector<Mat> separatedChannels=showSeparatedChannels(channels);
  for (int i=0;i<3;i++){
     cvtColor(separatedChannels[i],separatedChannels[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;
}
```

Результаты показаны на рисунке на сдедующей странице.

YCrCb

Это цветовое пространство широко используется в таких алгоритмах сжатия изображений и видео, как JPEG и MPEG. Это не самостоятельное цветовое пространство, а лишь способ кодирования пространства RGB. Канал Y представляет фотометрическую яркость, а Cr и Cb – красная и синяя цветоразностные компоненты (соответственно разность между каналами R и B RGB-изображения и Y).



Исходное RGB-изображение и оно же после разделения каналов CIE XYZ

Преобразование между пространствами RGB и YCrCb производится функцией `cvtColor` с кодами `COLOR_BGR2YCrCb`, `COLOR_RGB2YCrCb`, `COLOR_YCrCb2BGR` и `COLOR_YCrCb2RGB` по следующим формулам:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$
$$Cr = (R - Y) * 0.713 + delta$$
$$Cb = (B - Y) * 0.564 + delta$$
$$R = Y + 1.403 * (Cr - delta)$$
$$G = Y - 0.714 * (Cr - delta) - 0.344 * (Cb - delta)$$
$$B = Y + 1.773 * (Cb - delta)$$

где *delta* вычисляется так:

$$delta = \begin{cases} 128 & \text{для 8-разрядных изображений} \\ 32768 & \text{для 16-разрядных изображений} \\ 0.5 & \text{для изображений с плавающей точкой} \end{cases}$$

Пример программы

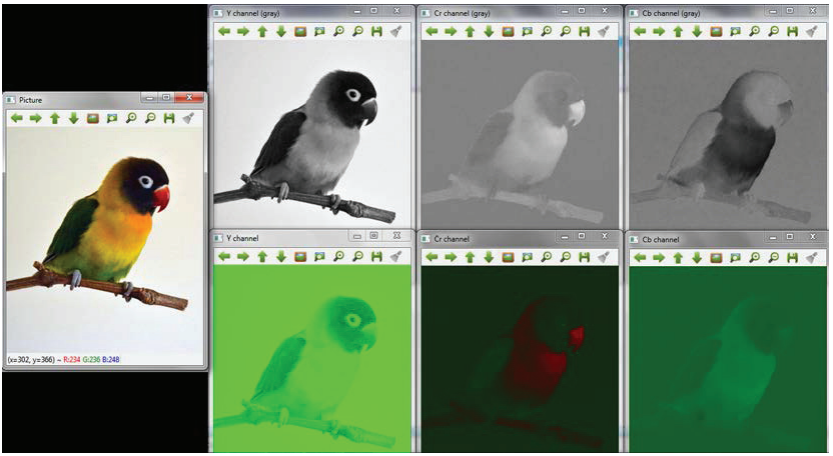
В программе **YCrCbcolor** ниже показано, как преобразовать RGB-изображение в цветовое пространство YCrCb. Каналы разделяются и показываются по отдельности в полутоновом режиме и в цвете. Первая часть программы загружает и преобразует изображение.

```
#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)
{
  // Загружаем изображение
  Mat image = imread("Lovebird.jpg");
  imshow("Picture",image);

  // Преобразуем в YCrCb
  cvtColor(image,image,COLOR_BGR2YCrCb);
```

Вторая часть разделяет и показывает каналы YCrCb.

```
  vector<Mat> channels;

  split( image, channels );

  // показываем каналы в полутоновом режиме
  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]);

  // показываем каналы в формате 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;
}
```

Результаты показаны на рисунке ниже.

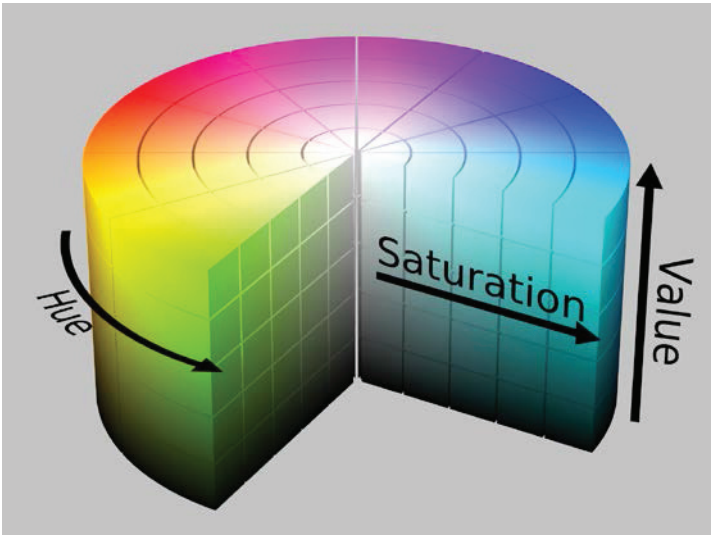


Исходное RGB-изображение и оно же после разделения каналов YCrCb

HSV

Цветовое пространство HSV принадлежит к группе так называемых оттеночных цветовых систем координат, цель которых точно смоделировать восприятие цветов человеком. Если в других цветовых моделях, например RGB, изображение рассматривается как результат сложения трех основных цветов, то три канала пространства **HSV** представляют **оттенок** (**H** (hue) – мера спектрального состава цвета), **насыщенность** (**S** (saturation) определяет долю чистого света доминирующей длины волны, то есть чем больше этот параметр, тем «чище» свет, и, наоборот, чем он меньше, тем ближе цвет к серому такой же яркости) и **значение** (**V** (value) задает яркость относительно белого света такой же силы). Эти каналы соответствуют интуитивно воспринимаемым понятиям тона, насыщенности и светлоты. Модель HSV часто используется для сравнения цветов, потому что канал H представляет практически ни от чего не зависящие вариации цвета. На следующем рисунке показано представление каналов этой модели в виде цилиндра.

Преобразование между пространствами RGB и YCrCb производится функцией `cvtColor` с кодами `COLOR_BGR2HSV`, `COLOR_RGB2HSV`, `COLOR_HSV2BGR` и `COLOR_HSV2RGB`. Стоит отметить, что в данном случае, если формат входного изображения `src` – 8- или 16-разрядное целое, то `cvtColor` сначала преобразует его в формат с плавающей точкой, приводя значения пикселей к диапазону от 0 до 1. Затем преобразование вычисляется по формулам:



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

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

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

Если $H < 0$, то $H = H + 360$. В конце значения преобразуются обратно к требуемому типу данных.

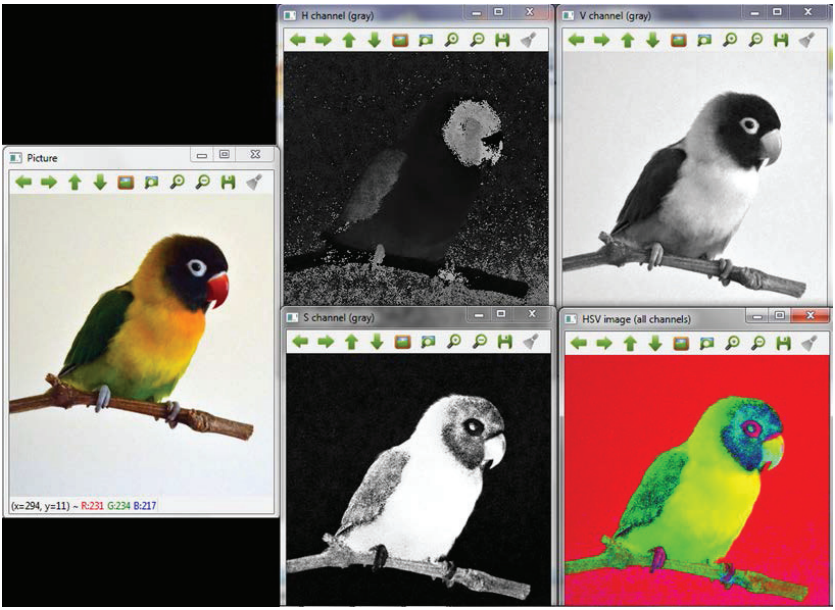
Пример программы

В программе **HSVcolor** ниже показано, как преобразовать RGB-изображение в цветовое пространство HSV. Каналы разделяются и показываются по отдельности в полутоновом режиме и в виде HSV-изображения.

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

using namespace std;
using namespace cv;

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

  // Преобразуем в HSV
  cvtColor(image,image,COLOR_BGR2HSV);

  vector<Mat> channels;

  split( image, channels );

  // показываем каналы в полутоновом режиме
  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;
}
```

Результаты показаны на рисунке ниже.

Функция `imshow` предполагает, что подлежащее показу изображение представлено в модели RGB, поэтому изображение показано неправильно. Если цвета изображения представлены в другом пространстве, то для правильного отображения необходимо предварительно преобразовать его в формат RGB.



Исходное RGB-изображение и оно же после разделения каналов HSV

HLS

Цветовое пространство HLS, как и HSV, принадлежит к группе оттеночных цветовых систем координат. В этой модели цвет представлен оттенком, светлотой и насыщенностью. Отличие от модели HSV заключается в том, что светлота чистого цвета, определенная в HLS, равна светлоте среднего серого, тогда как яркость чистого цвета, определенная в HSV, равна яркости белого.

Преобразование между пространствами RGB и HLS производится функцией `cvtColor` с кодами `COLOR_BGR2HLS`, `COLOR_RGB2HLS`, `COLOR_HLS2BGR` и `COLOR_HLS2RGB`. Как и в случае HSV, если формат входного изображения `src` – 8- или 16-разрядное целое, то `cvtColor` сначала преобразует его в формат с плавающей точкой, приводя значения пикселей к диапазону от 0 до 1. Затем преобразование вычисляется по формулам:

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

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

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

$$H = \begin{cases} \dfrac{60(G - B)}{S} & \text{, если } Vmax = R \\ 120 + \dfrac{60(B - R)}{S} & \text{, если } Vmax = G \\ 240 + \dfrac{60(R - G)}{S} & \text{, если } Vmax = B \end{cases}$$

Если $H < 0$, то $H = H + 360$. В конце значения преобразуются обратно к требуемому типу данных.

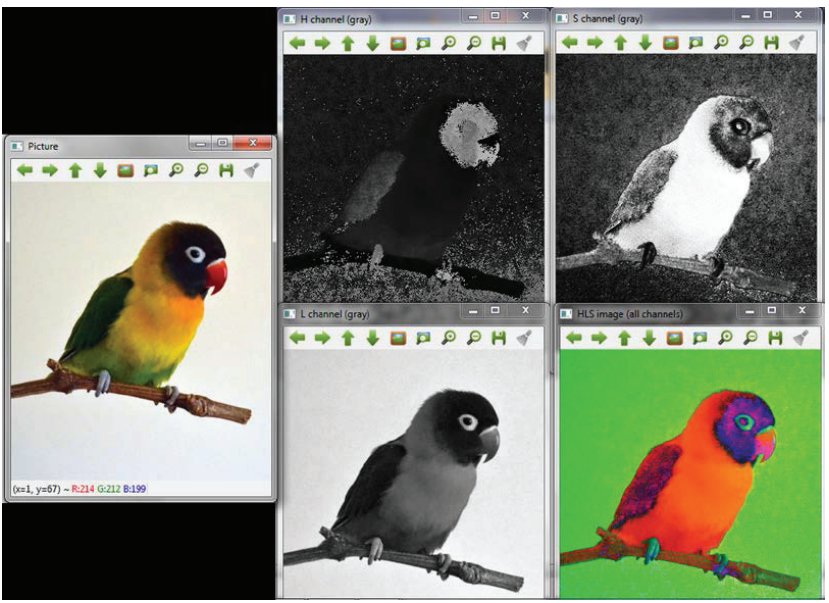
Пример программы

В программе **HLScolor** ниже показано, как преобразовать RGB-изображение в цветовое пространство HSV. Каналы разделяются и показываются по отдельности в полутоновом режиме и в виде HLS-изображения.

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

using namespace std;
using namespace cv;

int main(int argc, const char** argv)
{
  // Загружаем изображение
  Mat image = imread("Lovebird.jpg");
```

```
    imshow("Picture",image);

    // Преобразуем в HSV
    cvtColor(image,image,COLOR_BGR2HLS);

    vector<Mat> channels;

    split( image, channels );

    // показываем каналы в полутоновом режиме
    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;
}
```

Результаты показаны на рисунке ниже.



Исходное RGB-изображение и оно же после разделения каналов HLS

CIE L*a*b*

Цветовое пространство CIE L*a*b – второе однородное пространство, стандартизованное Международной комиссией по освещению (CIE) после CIE L*u*v*, которое основано на пространстве CIE XYZ и эталонной точке белого. Это самое полное цветовое пространство, определенное CIE; оно, как CYE XYZ, призвано быть независимым от устройства и служить эталоном. Три канала представляют соответственно светлоту цвета (L*) и его положение в диапазоне от пурпурного до зеленого (a*) и от желтого до синего (b*).

Преобразование между пространствами RGB и CIE L*a*b* производится функцией `cvtColor` с кодами `COLOR_BGR2Lab`, `COLOR_RGB2Lab`, `COLOR_Lab2BGR` и `COLOR_Lab2RGB`. Порядок вычислений объясняется в документе `http://docs-hoffmann.de/cielab03022003.pdf`.

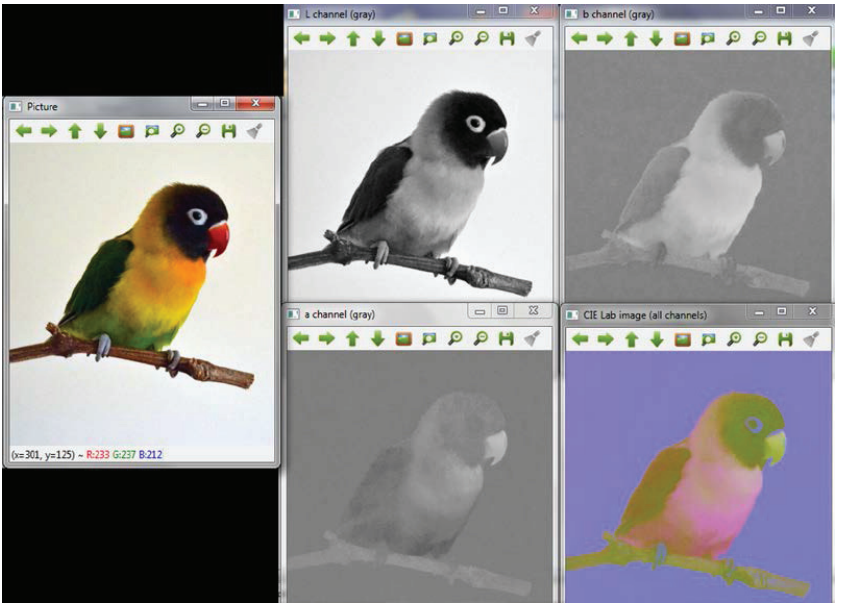
Пример программы

В программе **CIElab** ниже показано, как преобразовать RGB-изображение в цветовое пространство CIE L*a*b*. Каналы разделяются и показываются по отдельности в полутоновом режиме и в виде CIE L*a*b*-изображения.

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

using namespace std;
using namespace cv;

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

  // Преобразуем в CIE Lab
  cvtColor(image,image,COLOR_BGR2Lab);

  vector<Mat> channels;

  split( image, channels );

  // показываем каналы в полутоновом режиме
  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;
}
```

Результаты показаны на рисунке ниже.



Исходное RGB-изображение и оно же после разделения каналов CIE L*a*b*

CIE L*u*v*

CIE L*u*v* – первое однородное цветовое пространство, стандартизованное CIE. Это вычислительно простое преобразование пространства CIE XYZ и эталонной точки белого, ставящее целью добиться равномерности восприятия. Как и CIE L*a*b*, это пространство не зависит от устройства. Три его канала представляют светлоту цвета (L*), положение в диапазоне от зеленого до красного (u*) и от синего до пурпурного (v*). Эта цветовая модель полезна для описания аддитивных смесей цветов в силу своих свойств линейности.

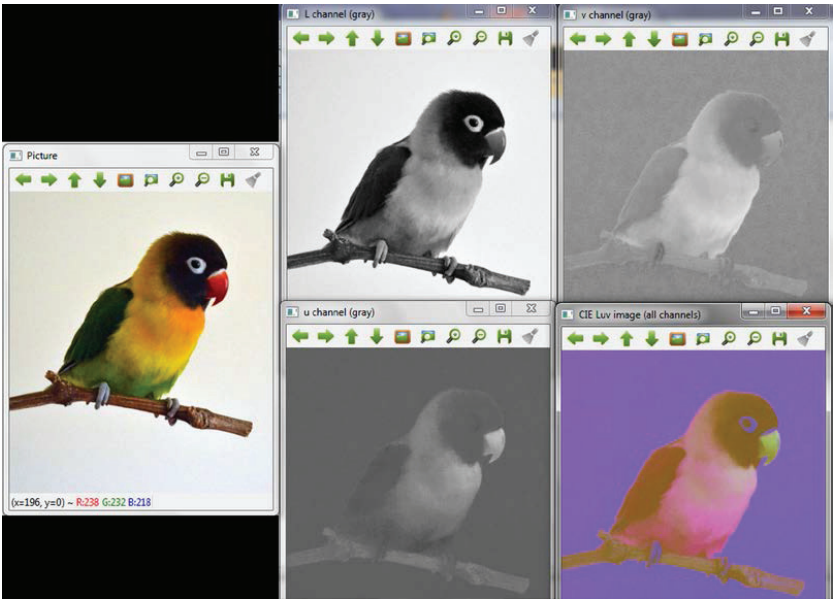
Преобразование между пространствами RGB и CIE L*u*v* производится функцией `cvtColor` с кодами `COLOR_BGR2Luv`, `COLOR_RGB2Luv`, `COLOR_Luv2BGR` и `COLOR_Luv2RGB`. Порядок вычисления объясняется в документе `http://docs.opencv.org/trunk/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor`.

Пример программы

В программе **CIELuvcolor** ниже показано, как преобразовать RGB-изображение в цветовое пространство CIE L*u*v*. Каналы разделяются и показываются по отдельности в полутоновом режиме и в виде CIE L*u*v*-изображения.

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

using namespace std;
using namespace cv;

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

  // Преобразуем в CIE Luv
  cvtColor(image,image,COLOR_BGR2Luv);

  vector<Mat> channels;

  split( image, channels );

  // показываем каналы в полутоновом режиме
  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;
}
```

Результаты показаны на рисунке ниже.

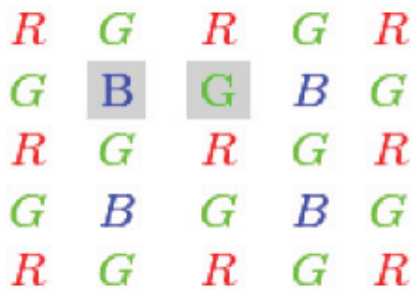


Исходное RGB-изображение и оно же после разделения каналов CIE L*u*v*

Фильтры Байера

Цветной фильтр Байера широко используется в цифровых камерах с одним фотоприемником. В отличие от камер с тремя фотоприемниками (по одному на каждый канал RGB, способному получать всю информацию о данной составляющей цвета), в камере с одним фотоприемником пиксели накрыты фильтрами разных цветов и, следовательно, измеряют только соответствующий цвет. Недостающая цифровая информация экстраполируется по соседним пикселям с помощью метода Байера. Это позволяет получать полноцветные изображения, имея всего одну плоскость, в которой пиксели чередуются, как показано на рисунке ниже.

Отметим, что в массиве фильтров Байера зеленых пикселей больше, чем красных и синих, потому что человеческий глаз более восприимчив к частотам зеленого цвета.



Массив цветных фильтров Байера

Существуют модификации, показанной выше схемы расположения, получаемые путем сдвига одного пикселя в любом направлении. Преобразование между пространствами RGB и и цветовым пространством Байера определяется элементами во втором и третьем столбцах второй строки (X и Y соответственно). Оно обозначается кодом `COLOR_BayerXY2BGR`. Например, на рисунке выше показано преобразование типа «BG», т. е. ему соответствует код `COLOR_BayerBG2BGR`.

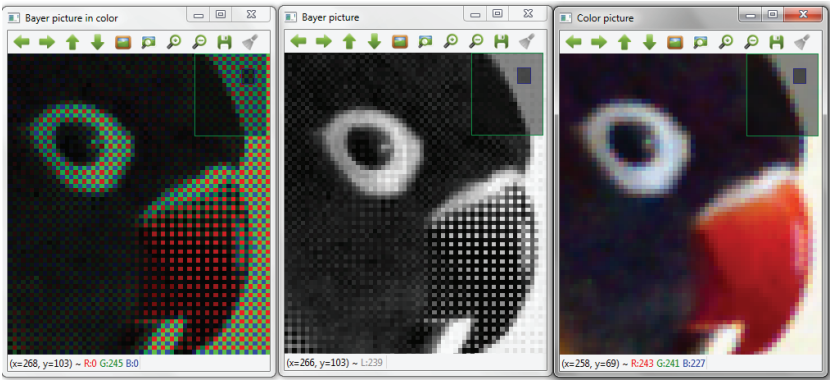
Пример программы

В программе **Bayer** ниже показано, как преобразовать изображение, полученное от камеры с RG-фильтром Байера, в формат RGB.

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

using namespace cv;

int main(int argc, const char** argv)
{
  // Показываем байеровское изображение в цвете
  Mat bayer_color = imread("Lovebird_bayer_color.jpg");
  namedWindow("Bayer picture in color", WINDOW_AUTOSIZE );
  imshow("Bayer picture in color",bayer_color);

  // Загружаем байеровское изображение
  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;
}
```

Результаты показаны на рисунке ниже.



Расположение фильтров Байера и изображение после преобразования в формат RGB

Сегментация на основе цветового пространства

В любом цветовом пространстве изображение представляется числовыми результатами измерения некоторой характеристики по каждому каналу. Анализируя эти характеристики, мы можем разбить цветовое пространство по линейным границам (например, плоскостями в трехмерном пространстве каналов), что позволяет классифицировать пиксели в соответствии с тем, в какую область они попали, а, значит, выбирать множества пикселей с заранее определенными характеристиками. Этой идеей можно воспользоваться для сегментации интересующих нас объектов на изображении.

В OpenCV имеется функция `void inRange(InputArray src, InputArray lowerb, InputArray upperb, OutputArray dst)`, которая проверяет, лежит ли некоторый массив элементов между двумя другими массивами. В применении к сегментации на основе цветовых пространств эта функция позволяет отобрать множество пикселей входного изображения `src`, для которых значения каналов заключены

между нижней (lowerb) и верхней (upperb) границами, и сформировать из них выходное изображение dst.

Границы lowerb и upperb обычно задаются в виде Scalar(x, y, z), где x, y и z – числовые значения каналов, рассматриваемые как нижняя и верхняя граница.

В примерах ниже показано, как найти пиксели, которые можно рассматривать как кожу. Замечено, что цвет кожи отличается больше по интенсивности, чем по цветности, поэтому обычно при распознавании кожи яркость не рассматривается. Поэтому выделить кожу в изображении, представленном в формате RGB трудно, т. к. это цветовое пространство сильно зависит от яркости. В связи с этим используются цветовые модели HSV или YCrCb. Отметим, что для такого типа сегментации необходимо заранее знать или как-то получить граничные значения для каждого канала.

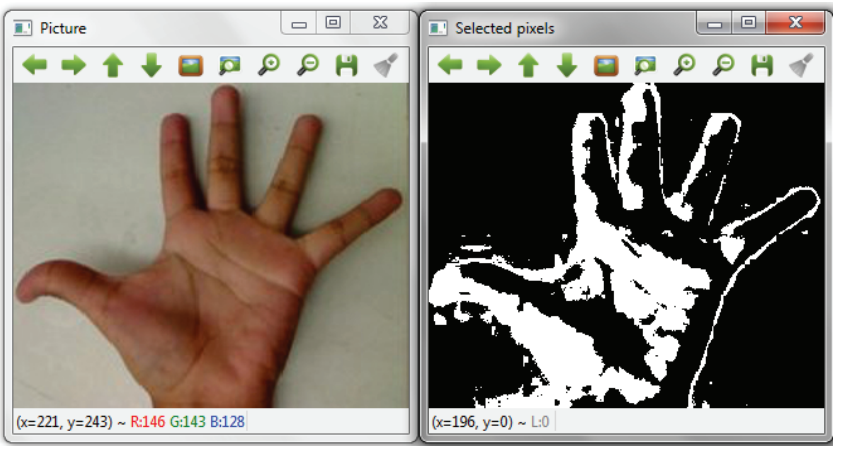
HSV-сегментация

Как сказано выше, модель HSV широко применяется для сравнения цветов, потому что составляющая H почти не зависит от изменения освещенности. Поэтому она полезна для распознавания кожи. В данном примере для решения вопроса о том, считать ли пиксель частью кожи, выбраны нижние границы (0, 10, 60) и верхние (20, 150, 255).

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

using namespace std;
using namespace cv;

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

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

  // Выбираем пиксели
  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;
}
```

Результаты показаны на рисунке ниже.



Распознавание кожи в цветовом пространстве HSV

YCrCb-сегментация

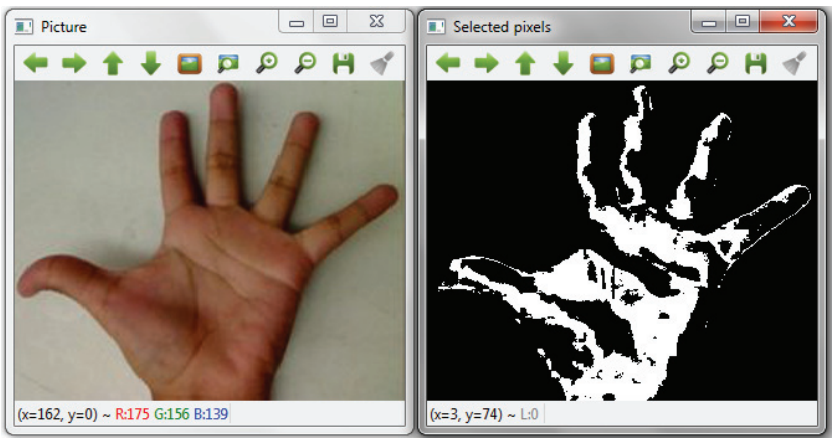
Цветовое пространство YCrCb уменьшает избыточность цветовых каналов RGB и представляет цвет с помощью трех независимых компонент. Поскольку яркость и цветность разделены, то это пространство хорошо подходит для распознавания кожи.

В следующем примере для распознавания кожи применяется пространство YCrCb с нижней границей (0, 133, 77) и верхней (255, 173, 177).

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

using namespace std;
using namespace cv;

int main()
```

```
{
  // Загружаем изображение
  Mat image = imread("hand.jpg");
  namedWindow("Picture", WINDOW_AUTOSIZE );
  imshow("Picture",image);

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

  // Выбираем пиксели
  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;
}
```

Результаты показаны на рисунке ниже.



Распознавание кожи в цветовом пространстве YCrCb

Дополнительные сведения о методах сегментации изображений можно найти в главе 4 книги «OpenCV Essentials», вышедшей в издательстве Packt Publishing.

Цветоперенос

Еще одна типичная задача обработки изображений – изменение цвета изображения, особенно когда необходимо удалить нежелательное преобладание какого-то одного цвета. Один из методов ее решения называется цветопереносом (color transfer), в этом случае цвет корректируется так, чтобы цветовые характеристики исходного изображения перенеслись на конечное.

Пример программы

В программе **colorTransfer** ниже показано, как перенести цвет с исходного изображения на конечное. Сначала изображение преобразуется в пространство CIE L*a*b*. Затем каналы исходного и конечного изображения разделяются. После этого статистическое распределение каналов одного изображения, определяемое средним и стандартным отклонением, применяется к другому. И наконец, каналы объединяются и выполняется обратное преобразование в пространство RGB.

Теоретическое обоснование использованного в примере преобразования можно найти в статье «Color Transfer between Images» по адресу `http://www.cs.tau.ac.il/~turkel/imagepapers/ColorTransfer.pdf`.

В первой части программы изображение преобразуется в цветовое пространство CIE L*a*b*, при этом тип изображения меняется на `CV_32FC1`:

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

using namespace std;
using namespace cv;

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

  // Преобразуем в пространство Lab и тип 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);
```

Далее производится цветоперенос в соответствии с описанным выше планом.

```
    // Находим среднее и стандартное отклонение для каждого канала
    // обоих изображений
    Mat mean_src, mean_tar, stdd_src, stdd_tar;
    meanStdDev(src_lab, mean_src, stdd_src);
    meanStdDev(tar_lab, mean_tar, stdd_tar);

    // Разделяем на отдельные каналы
    split( src_lab, src_chan );
    split( tar_lab, tar_chan );

    // Для каждого канала вычисляем распределение цветов
    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);
    }

    // Объединяем каналы, преобразуем в формат CV_8UC1, а затем в BGR
    Mat output;
    merge(tar_chan, output);
    output.convertTo(output,CV_8UC1);
    cvtColor(output, output, COLOR_Lab2BGR );

    // Показываем результаты
    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;
}
```

Результаты показаны на рисунке ниже.

Перенос цветов с ночного изображения на дневное

Резюме

В этой главе мы рассмотрели цветовые пространства, используемые в OpenCV, и показали, как переходить из одного в другое с помощью функции `cvtColor`. Кроме того, мы обсудили возможности обработки изображений в разных цветовых моделях и подчеркнули важность выбора цветового пространства, подходящего для выполняемой операции. Попутно мы реализовали сегментацию на основе цветового пространства и метод цветопереноса.

В следующей главе мы рассмотрим методы обработки последовательности изображения или видео. Мы покажем, как в OpenCV реализовать стабилизацию видео, сверхвысокое разрешение и сшивку изображений.

ГЛАВА 5. Обработка видео

В этой главе описаны различные методы обработки видео. Большинство классических алгоритмов обработки изображений работает со статическими картинками, но обработка видео становится все более популярной и доступной по цене.

В этой главе рассматриваются следующие вопросы:

- стабилизация видео;
- сверхвысокое разрешение;
- сшивка изображений.

Мы будем работать как с последовательностями видеокадров, так и непосредственно с камерой. Результатом обработки может быть либо набор модифицированных изображений, либо полезная информация высокого уровня. В большинстве методов изображение рассматривается как двумерный цифровой сигнал, к которому можно применять различные операции. Нас будет интересовать улучшение имеющейся или получаемой с камеры последовательности изображений, т. е. мы добавляем третье измерение – время.

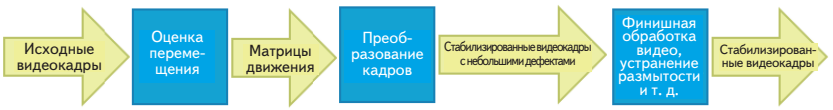
Стабилизация видео

Под стабилизацией видео понимается семейство методов, применяемых для уменьшения дрожания из-за перемещения камеры. Иными словами, стабилизация компенсирует угловые смещения, связанные с поворотом вокруг любой из трех осей и параллельным сдвигом камеры. Первые стабилизаторы изображений появились еще в начале 1960-х годов. Такие системы могли немного компенсировать тряску камеры и непроизвольные движения оператора. Они управлялись гироскопами и акселерометрами, которые могли свести на нет или уменьшить нежелательные перемещения за счет изменения положения объектива. В наши дни такие методы широко применяются в биноклях, видеокамерах и телескопах.

Существуют различные способы стабилизации видео, в этой главе мы остановимся на наиболее распространенных.

- **Механические системы стабилизации**: в них используется механическая система привода объектива, в которой акселерометры и гироскопы распознают движение, после чего генерируется ответное перемещение объектива. Такие системы мы рассматривать не будем.
- **Цифровые системы стабилизации**: обычно используются в видеокамерах и воздействуют непосредственно на полученное от камеры изображение. В таких системах площадь стабилизированного изображения чуть меньше площади исходного. Когда камера перемещается, захваченное изображение тоже перемещается, так чтобы компенсировать движение камеры. И хотя эти методы успешно справляются с компенсацией движения путем уменьшения полезной площади, за это приходится расплачиваться разрешением и четкостью изображения.

Алгоритмы стабилизации видео обычно состоят из следующих шагов:



Общие шаги алгоритма стабилизации видео

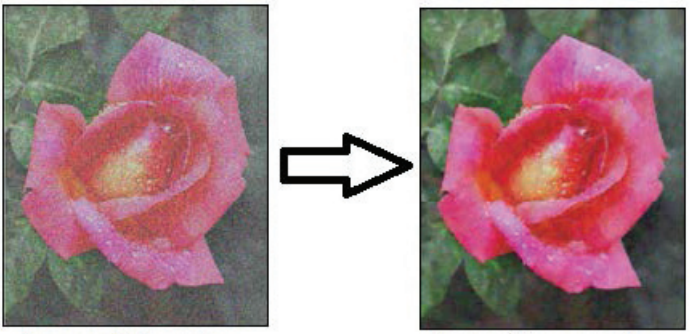
В этой главе мы будем изучать модуль `videostab` из версии OpenCV 3.0 Alpha, который содержит функции и классы для решения задачи стабилизации видео.

Рассмотрим процесс более детально. Сначала необходимо с помощью метода RANSAC вычислить перемещение между соседними кадрами. По завершении этого шага мы получаем массив матриц 3×3, каждая из которых описывает перемещение двух пар соседних кадров. Глобальное вычисление перемещения на этом шаге очень важно, от него зависит точность окончательной стабилизированной последовательности.

Подробное описание метода RANSAC смотрите на странице `http://ru.wikipedia.org/wiki/RANSAC`.

На втором шаге на основе вычисленного перемещения генерируется новая последовательность кадров. Для повышения качества стабилизации выполняется дополнительная обработка: сглаживание, уменьшение размытия, экстраполяция границ и т. д.

На третьем шаге устраняются раздражающие возмущения – см. рисунок ниже. Существуют методы, в основе которых лежат допущения о модели движения камеры, они дают хорошие результаты, если какие-то разумные допущения возможны.



Изображение с нерегулярными возмущениями

Изображение после коррекции

Устранение нерегулярностей

Среди примеров, входящих в дистрибутив OpenCV, есть программа стабилизации видео ([каталог исходного кода OpenCV]/samples/cpp/videostab.cpp). Для рассматриваемого ниже проекта `videoStabilizer.pro` необходимы следующие библиотеки: `lopencv_core300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300` и `lopencv_videostab300`.

В программе **videoStabilizer** используется модуль `videostab` из версии 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 - Подготовить и проверить входное видео
  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 << "число кадров (примерное): " << source->count() << endl;

  // 2 - Подготовить вычислитель перемещения
  // Сначала настроить построитель вычислителей 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);

  // Затем создать детектор признаков
  int nkps = 1000;
  Ptr<GoodFeaturesToTrackDetector> feature_detector =
       makePtr<GoodFeaturesToTrackDetector>(nkps);

  // На третьем шаге создать вычислитель перемещения
  Ptr<KeypointBasedMotionEstimator> motionEstBuilder =
       makePtr<KeypointBasedMotionEstimator>(est);
  motionEstBuilder->setDetector(feature_detector);
  Ptr<IOutlierRejector> outlierRejector =
       makePtr<NullOutlierRejector>();
  motionEstBuilder->setOutlierRejector(outlierRejector);

  // 3 - Подготовить стабилизатор
  StabilizerBase *stabilizer = 0;

  // Сначала подготовить одно- или двухпроходный стабилизатор
  bool isTwoPass = 1;
```

```
    int radius_pass = 15;
    if (isTwoPass)
    {
      // с двухпроходным стабилизатором
      bool est_trim = true;
      TwoPassStabilizer*twoPassStabilizer=newTwoPassStabilizer();
      twoPassStabilizer->setEstimateTrimRatio(est_trim);
      twoPassStabilizer->setMotionStabilizer(
          makePtr<GaussianMotionFilter>(radius_pass));
      stabilizer = twoPassStabilizer;
    }
    else
    {
      // с однопроходным стабилизатором
      OnePassStabilizer*onePassStabilizer=newOnePassStabilizer();
      onePassStabilizer->setMotionFilter(
          makePtr<GaussianMotionFilter>(radius_pass));
      stabilizer = onePassStabilizer;
    }

    // Затем настроить параметры
    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);

    // Привести стабилизатор к простому интерфейсу источника
    // кадров, чтобы можно было читать стабилизированные кадры
    stabilizedFrames.reset(dynamic_cast<IFrameSource*>(stabilizer));

    // 4 - Обработать стабилизированные кадры. Показать и сохранить
    // результаты
    processing(stabilizedFrames, outputPath);
  }
  catch (const exception &e)
  {
    cout << "ошибка: " << 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;

// для каждого стабилизированного кадра
while (!(stabilizedFrame = stabilizedFrames->nextFrame()).empty())
{
 nframes++;

 // инициализировать объект-писатель (один раз) и сохранить
 // стабилизированный кадр
 if (!outputPath.empty())
 {
  if (!writer.isOpened())
     writer.open(outputPath,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 << "обработано кадров: " << nframes << endl;
cout << "завершено " << endl;
}
```

Эта программа принимает имя входного видеофайла, а если оно не задано, по умолчанию берет файл `.\cube4.avi`. Получившееся в результате обработки видео сохраняется в файле `.\cube4_stabilized.avi`. Обратите внимание на включение заголовка `videostab.hpp` и использование пространства имен `cv::videostab`. Программа состоит из четырех шагов. На первом шаге подготавливается путь к видеофайлу; мы используем для задания файла стандартный механизм передачи аргументов в командной строке (`inputPath = argv[1]`). Если файл не задан, по умолчанию используется видеофайл `.\cube4.avi`.

На втором шаге строится вычислитель перемещения. Мы пользуемся устойчивым глобальным двумерным алгоритмом RANSAC, применяя встроенный в OpenCV интеллектуальный указатель `Ptr<object>` (`Ptr<MotionEstimatorRansacL2> est = makePtr<MotionEstimatorRansacL2>(MM_AFFINE)`). Существуют и другие модели перемещения, применяемые для стабилизации видео:

- MM_TRANSLATION = 0
- MM_TRANSLATION_AND_SCALE = 1

- MM_ROTATION = 2
- MM_RIGID = 3
- MM_SIMILARITY = 4
- MM_AFFINE = 5
- MM_HOMOGRAPHY = 6
- MM_UNKNOWN = 7

Приходится выбирать между точностью стабилизации и временем вычислений. Самые простые модели не слишком точны, зато работают быстро; сложные модели дают более высокую точность, но работают дольше.

Итак, объект RANSAC создан (`RansacParams ransac = est->ransacParams()`), и его параметры настроены (`ransac.size`, `ransac.thresh` и `ransac.eps`). Для вычисления перемещения между соседними кадрами стабилизатору необходим еще детектор признаков. В этой программе мы используем метод `GoodFeaturesToTrackDetector` для обнаружения `nkps = 1000` особых точек в каждом кадре. Затем, располагая устойчивым алгоритмом RANSAC и детектором признаков, мы создаем объект-вычислитель перемещения `Ptr<KeypointBasedMotionEstimator> motionEstBuilder = makePtr<KeypointBasedMotionEstimator>(est)` и задаем для него детектор признаков: `motionEstBuilder->setDetector(feature_detector)`.

Параметры RANSAC	
Size	Размер подмножества
Thresh	Максимальная ошибка, при которой точка еще считается регулярной (inlier)
Eps	Максимальный коэффициент для выбросов
Prob	Вероятность успеха

На третьем шаге создается стабилизатор, которому необходим ранее созданный вычислитель перемещения. Можно выбрать одно- (`isTwoPass = 1`) или двухпроходный стабилизатор. В последнем случае (`TwoPassStabilizer *twoPassStabilizer = new TwoPassStabilizer()`) результаты обычно получаются лучше, но время вычисления увеличивается. В случае однопроходного стабилизатора (`OnePassStabilizer *onePassStabilizer = new OnePassStabilizer()`) для получения ответа требуется меньше времени, но результаты хуже. Для работы стабилизатора нужно задать ряд параметров, например источник видеофайла

(`stabilizer->setFrameSource(source)`) и вычислитель перемещения (`stabilizer->setMotionEstimator(motionEstBuilder)`). Кроме того, для чтения стабилизированных кадров стабилизатор необходимо привести к типу простого источника видеокадров (`stabilizedFrames.reset(dynamic_cast<IFrameSource*> (stabilizer))`).

На последнем шаге созданный стабилизатор используется для стабилизации видео. Этим занимается функция `processing(Ptr<IFrameSource> stabilizedFrames)`, которой необходимо передать путь к файлу, где будет сохраняться результат (`string outputPath = ".//stabilizedVideo.avi"`), и скорость воспроизведения (`double outputFps = 25`). Эта функция продолжает работать, пока остаются кадры (`stabilizedFrame = stabilizedFrames-> nextFrame().empty()`). Первым делом она вызывает стабилизатор для вычисления перемещения между кадрами. Функция создает объект записи видео (`writer.open(outputPath,VideoWriter::fourcc('X','V','I','D'), outputFps, stabilizedFrame.size())`), который сохраняет кадры в формате **XVID**. Затем она сохраняет и показывает стабилизированные кадры, но останавливается, если пользователь нажмет клавишу **Esc**.

Продемонстрируем с помощью этой программы стабилизацию виде в OpenCV. Для этого запустим ее из командной строки:

```
<bin_dir>\videoStabilizer.exe .\cube4.avi .\cube4_stabilized.avi
```

Файл `cube4.avi` находится в каталоге примеров OpenCV. При его подготовке камера активно перемещалась, так что это прекрасный кандидат для стабилизации.

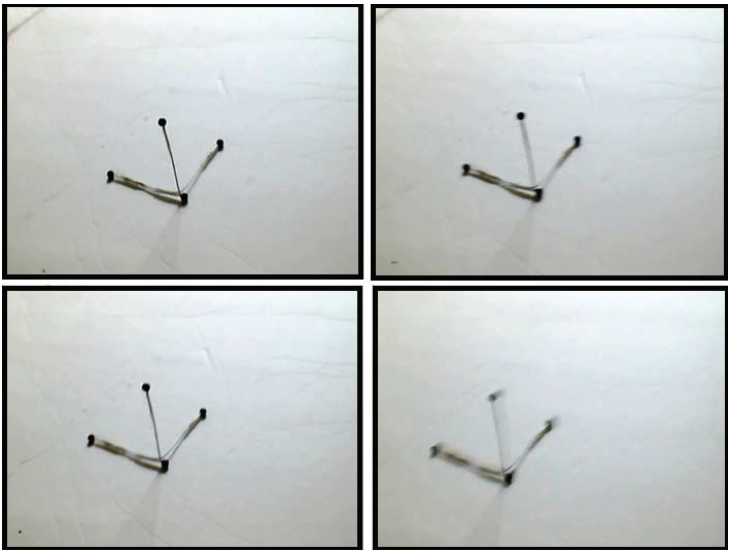
На первом рисунке на следующей странице показаны первые четыре кадра файла `cube4.avi`, а затем результат наложения первых десяти кадров до стабилизации (слева) и после стабилизации (справа).

Из правого рисунка видно, что дрожание вследствие движения камеры после стабилизации уменьшилось.

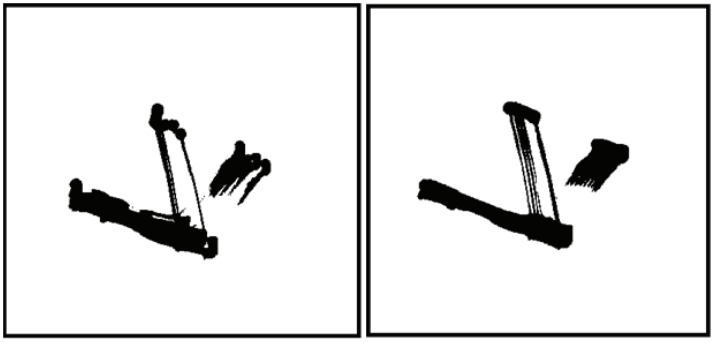
Сверхвысокое разрешение

Под **сверхвысоким разрешением** понимают технические средства и алгоритмы, призванные увеличить пространственное разрешение изображения или видео обычно на основе последовательности изображений более низкого разрешения. Эта техника отличается от

традиционного масштабирования, при котором увеличивается разрешение одного изображения с сохранением резкости границ. Идея сверхвысокого разрешения – объединив информацию из нескольких изображений одной и той же сцены, получить детали, которых не было ни на одном из исходных изображений.



Первые четыре кадра из видеофайла `cube4.avi`, полученные в результате перемещения камеры

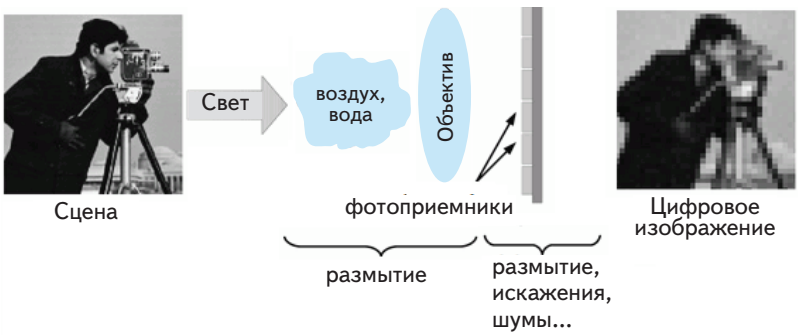


Десять наложенных друг на друга кадров до и после стабилизации

Процесс захвата изображения или видео реальной сцены состоит из нескольких шагов.

- **Дискретизация:** это преобразование непрерывной сцены в идеальную дискретную последовательность без зубцеобразных дефектов.
- **Геометрическое преобразование:** применение ряда преобразований, например параллельного переноса или поворота, компенсирующих перемещение камеры и особенности оптической системы, для выделения деталей сцены, попавших на каждый фотоприемник.
- **Размытие:** обусловлено влиянием оптической системы или реального движения внутри сцены в течение обрабатываемого промежутка времени.
- **Субдискретизация:** на этом этапе фотоприемник интегрирует те пиксели, которые оказались в его распоряжении.

Описанный процесс захвата показан на следующем рисунке.



Процесс захвата изображения реальной сцены

В ходе этого процесса детали сцены интегрируются различными фотоприемниками, так что каждый пиксель в разных кадрах несет различную информацию. Для получения сверхвысокого разрешения необходимо найти соотношения между разными кадрами, на которых запечатлены разные детали одной сцены, и создать новое изображение, содержащее больше информации. Таким образом, сверхвысокое разрешение применяется для восстановления дискретизированной сцены в более высоком разрешении.

Получить сверхвысокое разрешение можно разными способами – от интуитивно очевидных, применяемых к пространственной области, до анализа спектра частот. Все методы делятся на оптические (использование объективов, оптическое увеличение масштаба и т. д.)

и алгоритмические, основанные на обработке изображений. В этой главе нас будут интересовать последние. В них используются другие части одного изображения низкого разрешения или несколько изображений, и на их основе делается вывод о том, как должно выглядеть изображение высокого разрешения. Все такие алгоритмы можно разделить на пространственные и частотные. Поначалу методы сверхвысокого разрешения хорошо работали только для полутоновых изображений, но недавно разработанные алгоритмы адаптируются и к цветным тоже.

Вообще говоря, для вычисления сверхвысокого разрешения требуется много времени и памяти, поскольку размер изображений как низкого, так и высокого разрешения велик. Поэтому для генерации результирующего изображения может понадобиться несколько сотен секунд. В попытках уменьшить время вычислений в оптимизаторах обычно используются предобуславливатели, повышающие скорость сходимости алгоритма. Другой вариант – использовать **графические процессоры**, потому что алгоритмы сверхвысокого разрешения естественно распараллеливаются.

В этом разделе мы рассмотрим модуль `superres` из версии OpenCV 3.0 Alpha, который содержит функции и классы для решения задачи увеличения разрешения. В нем реализовано несколько методов получения изображений со сверхвысоким разрешением. Нас будет интересовать метод **двустороннего сверхвысокого разрешения TV-L1 (BTVL1)**. Основная сложность – вычислить деформирующую функцию построения изображения сверхвысокого разрешения. В методе BTVL1 для этого применяется оптический поток.

Подробные сведения о двустороннем методе TV-L1 можно найти в статье `http://www.ipol.im/pub/art/2013/26/`, а об оптическом потоке – в статье википедии по адресу `https://ru.wikipedia.org/wiki/Оптический_поток`.

В дистрибутиве OpenCV имеется простой пример вычисления сверхвысокого разрешения (`[исходный код opencv]/samples/gpu/super_resolution.cpp`).

Скачать его можно также из GitHub-репозитория OpenCV по адресу `https://github.com/Itseez/opencv/blob/master/samples/gpu/super_resolution.cpp`.

В файл проекта `superresolution.pro` включены следующие библиотеки: `lopencv_core300`, `lopencv_imgproc300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300` и `lopencv_superres300`.

```
#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 - Задаем начальные параметры

 // Входной и выходной видеофайл
 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;  // масштабный коэффициент

 const int iterations = 180;  // количество итераций

 const int temporalAreaRadius = 4; // радиус временной области поиска

 string optFlow = "farneback"; // алгоритм вычисления оптического потока
    // optFlow = "farneback";
    // optFlow = "tvl1";
    // optFlow = "brox";
```

```
    // optFlow = "pyrlk";

double outputFps = 25.0; // скорость воспроизведения выходного видео

// 2 - Создаем вычислитель оптического потока
Ptr<DenseOpticalFlowExt> optical_flow = createOptFlow(optFlow);

if (optical_flow.empty()) return -1;

// 3 - Создаем вычислитель сверхвысокого разрешения и задаем
//     его параметры
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);

// Первый кадр пропускаем
Mat frame;
frameSource->nextFrame(frame);

// 4 - Обрабатываем входное видео алгоритмом сверхвысокого разрешения
// Показываем начальные параметры
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;

  // Вычисляем время обработки
  start_time = getTickCount();
  superRes->nextFrame(result);
  finish_time = getTickCount();
  cout << (finish_time - start_time)/getTickFrequency() << "
          secs, Size: " << result.size() << endl;

  if (result.empty()) break;
```

```
    // Показываем результат
    imshow("Super Resolution", result);

    if (waitKey(1000) > 0) break;

    // Сохраняем результат в выходном файле
    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 << "Недопустимый алгоритм оптического потока - " << name
          << endl;

  return Ptr<DenseOpticalFlowExt>();
}
```

Эта программа (**superresolution**) получает видеофайл со сверхвысоким разрешением. Она принимает путь к входному видеофайлу или открывает файл по умолчанию (`.\tree.avi`). Обработанное видео сохраняется в файле `.\tree_superresolution.avi`. В самом начале включаются заголовки `superres.hpp` и `superres/optical_flow.hpp` и объявляется об использовании пространства имен `cv::superres`. Программа состоит из четырех шагов.

На первом шаге задаются начальные параметры. Это путь к видеофайлу, задаваемый в первом аргументе командной строки (`inputVideoName = argv[1]`); если путь не задан, то берется файл по умолчанию. Путь к выходному файлу задается во втором аргументе (`outputVideoName = argv[2]`), а, если он не задан, то результат сохраняется в файле `.\tree_superresolution`. Задается также скорость вос-

произведения результата (`double outputFps = 25.0`). Другие важные параметры алгоритма сверхвысокого разрешения: коэффициент масштабирования (`const int scale = 4`), количество итераций (`const int iterations = 100`), радиус временно́й области поиска (`const int temporalAreaRadius = 4`) и алгоритм вычисления оптического потока (`string optFlow = "farneback"`).

На втором шаге создается вычислитель оптического потока, который находит отличительные признаки и прослеживает их во всех кадрах. Функция `static Ptr<DenseOpticalFlowExt> createOptFlow(string name)` выбирает один из имеющихся алгоритмов: **farneback**, **tvl1**, **brox** и **pyrlk**. Наиболее важны методы Farneback (`createOptFlow_Farneback()`) и TV-L1 (`createOptFlow_DualTVL1()`). Первый основан на алгоритме Гуннера Фарнебэка (Gunner Farneback), который вычисляет оптический поток для всех точек кадра. Второй вычисляет оптический поток между двумя изображениями, основываясь на двойственной формулировке задачи о полной вариации и поточечной бинаризации. С вычислительной точки зрения, второй алгоритм эффективнее.

Сравние методов вычисления оптического потока

Метод	Сложность	Допускает распараллеливание
Farneback	Квадратичная	Нет
TV-L1	Линейная	Да
Brox	Линейная	Да
PyrLK	Линейная	Нет

Дополнительные сведения о методе Фарнебэка смотрите в статье по адресу `http://www.diva-portal.org/smash/get/diva2:273847/FULLTEXT01.pdf`.

На третьем шаге создается и инициализируется вычислитель сверхвысокого разрешения `Ptr<SuperResolution> superRes`, в котором используется двусторонний алгоритм **TV-L1** (`superRes = createSuperResolution_BTVL1()`) со следующими параметрами:

- `scale`: масштабный коэффициент;
- `iterations`: количество итераций;
- `tau`: асимптотическое значение в методе перевала (наискорейшего спуска);

- `lambda`: весовой параметр для выбора компромисса между членом данных и членом гладкости;
- `alpha`: параметр пространственного распределения в алгоритме Bilateral-TV;
- `btvKernelSize`: размер ядра фильтра Bilateral-TV;
- `blurKernelSize`: размер ядра фильтра размытия Гаусса;
- `blurSigma`: параметра «сигма» фильтра размытия Гаусса;
- `temporalAreaRadius`: радиус временно́й области поиска;
- `opticalFlow`: плотный алгоритм оптического потока.

Эти параметры задаются следующим образом:

```
superRes->set ("параметр", значение);
```

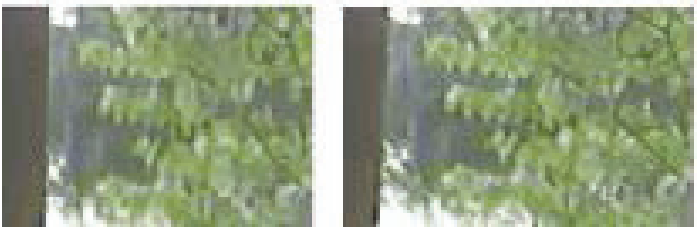
Задаются только показанные ниже параметры, остальные принимают значения по умолчанию:

```
superRes->set("opticalFlow", optical_flow);
superRes->set("scale", scale);
superRes->set("iterations", iterations);
superRes->set("temporalAreaRadius", temporalAreaRadius);
```

Затем выбирается кадр входного изображения (`superRes->setInput (frameSource)`).

На последнем шаге вычисляются все кадры выходного изображения со сверхвысоким разрешением (`superRes->nextFrame(result)`); поскольку вычисление медленное, мы показываем время обработки, чтобы пользователь видел: что-то происходит. И наконец, все обработанные кадры показываются (`imshow("Super Resolution", result)`) и сохраняются (`writer << result`).

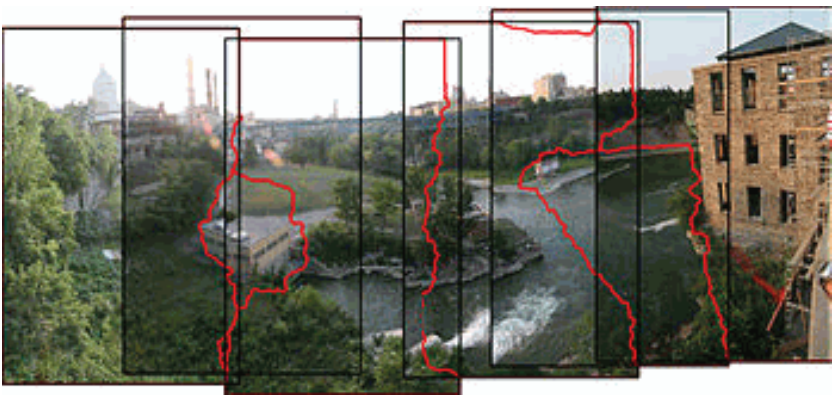
Для оценки результатов работы сравните небольшой участок первого кадра в файлах `tree.avi` (без сверхвысокого разрешения) и `tree_superresolution.avi` (со сверхвысоким разрешением) на рисунке ниже:



Благодаря сверхвысокому разрешению справа видно больше деталей листьев и ветвей дерева.

Сшивка изображений

В процессе сшивки изображений делается попытка установить соответствие между частично перекрывающимися изображениями. В результате объединения из набора изображений пересекающихся частей сцены получается панорама или изображение с более высоким разрешением. Большинству алгоритмов сшивки для получения бесшовного результата необходимо почти точное пересечение изображений. Некоторые цифровые камеры умеют самостоятельно сшивать изображения для получения панорамы. Пример показан на рисунке ниже.



Панорама, полученная в результате сшивки изображений

Это изображение и дополнительные сведения о сшивке изображений смотрите в статье по адресу `http://en.wikipedia.org/wiki/Image_stitching`.

Процесс сшивки состоит из трех шагов.

- **Регистрацией** называется сопоставление признаков в наборе изображений и нахождение такого сдвига, при котором достигает минимума сумма абсолютных величин разностей между соответственными пикселями. Для получения более точных результатов можно применить методы прямого совмещения (direct alignment). Пользователь может также добавить грубую модель панорамы, чтобы упростить работу на этапе сопо-

ставления; в этом случае результаты обычно получаются точнее, а вычисления занимают меньше времени.

- Задача **калибровки** изображения заключается в минимизации искажений, вносимых оптической системой камеры: влияния положения камеры и оптических дефектов – дисторсии, неправильной выдержки, хроматических аберраций и т. д.
- На этапе **композиции** результаты калибровки объединяются с преобразованием изображений в конечную проекцию. Кроме того, корректируются цвета, чтобы компенсировать различия в выдержке. Изображения соединяются, а линии соединения подправляется, так чтобы не были заметны швы.

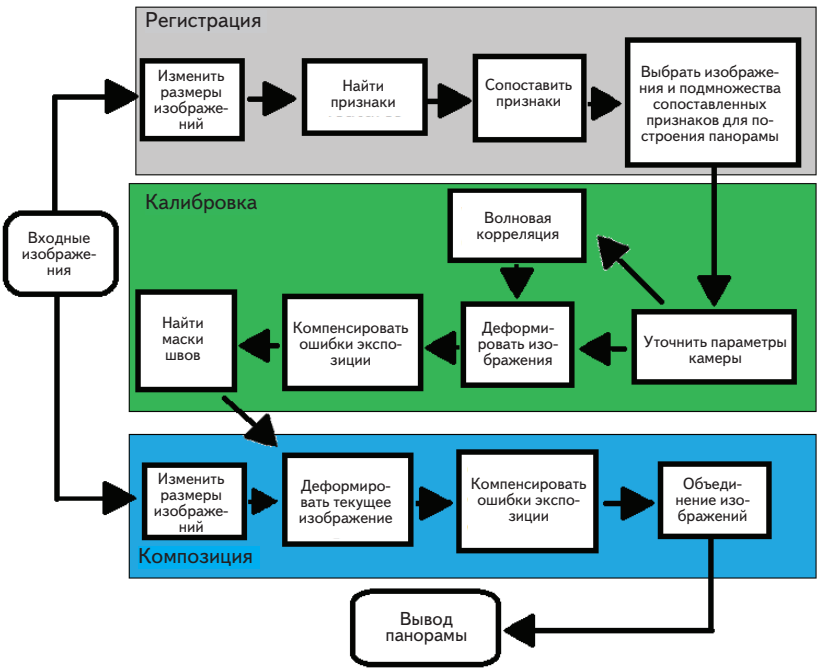
Когда имеются участки изображения, снятые из одной и той же точки, сшивку можно выполнять в одной из нескольких картографических проекций. Ниже перечислены наиболее употребительные проекции.

- **Прямолинейная проекция:** в этом случае сшитое изображение проецируется на плоскость, пересекающую сферу панорамы в единственной точке. Прямые линии остаются прямыми на изображении независимо от их направления. При больших углах обзора (порядка 120 градусов) заметны искажения на краях.
- **Цилиндрическая проекция:** на сшитом изображении горизонтальное поле зрения показано полностью (360 градусов), а вертикальное ограничено. Изображение как бы навернуто на цилиндр и рассматривается из его внутренней точки. При проецировании на плоскость горизонтальные прямые кажутся искривленными, а вертикальные остаются прямыми.
- **Сферическая проекция:** на сшитом изображении горизонтальное поле зрения показано полностью (360 градусов), а вертикальное – наполовину (180 градусов). Изображение как бы навернуто на сферу и рассматривается из ее внутренней точки. При проецировании на плоскость горизонтальные прямые кажутся искривленными, как в цилиндрической проекции, а вертикальные остаются вертикальными.
- **Стереографическая проекция**, или **рыбий глаз**: можно использовать для формирования панорамы малой планеты, когда виртуальная камера направлена прямо вниз, а поле зрения достаточно велико, чтобы показать всю поверхность и часть

области над ней. Если виртуальная камера направлена вверх, то создается эффект туннеля.

- **Проекция Панини:** специализированная проекция, которая имеет эстетические преимущества по сравнению со стандартными картографическими проекциями. В этом случае различные проекции комбинируются в одном изображении для тонкой настройки окончательного вида панорамного изображения.

В этом разделе мы рассмотрим модуль `stitching` и его подмодуль `detail` в версии OpenCV 3.0 Alpha. В них содержатся функции и классы для реализации **сшивателя**. Модули позволяют конфигурировать или пропускать шаги. Реализованный пример сшивки описывается следующей диаграммой:



В дистрибутиве OpenCV есть два простых примера: `[исходный код opencv]/samples/cpp/stitching.cpp` и `[исходный код opencv /samples/cpp/stitching_detailed.cpp]`.

Ниже приведен более сложный пример **stitchingAdvanced**; файл проекта `stitchingAdvanced.pro` должен включать следующие библиотеки: `lopencv_core300`, `lopencv_imgproc300`, `lopencv_highgui300`, `lopencv_features2d300`, `lopencv_videoio300`, `lopencv_imgcodecs300` и `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[])
{
 // Параметры по умолчанию
 vector<String> img_names;
 double scale = 1;
 string features_type = "orb"; // тип признаков: "surf" или "orb"
 float match_conf = 0.3f;
 float conf_thresh = 1.f;
 string adjuster_method = "ray"; // метод уравнивания "reproj" или "ray"
 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 - Входные изображения
 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");
}

// Проверяем, достаточно ли изображений
int num_images = static_cast<int>(img_names.size());
if (num_images < 2) {cout << "Мало изображений" << endl; return -1; }

// 2 - Изменение размеров изображений и поиск признаков
cout << "Поиск признаков..." << endl;
double t = getTickCount();

Ptr<FeaturesFinder> finder;
if (features_type == "surf")
   finder = makePtr<SurfFeaturesFinder>();
else if (features_type == "orb")
   finder = makePtr<OrbFeaturesFinder>();
else {
    cout << "Неизвестный тип 2D-признаков: '" << 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 << "Не могу открыть изображение " << 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 << "Признаки в изображении #" << i+1 << ": " <<
          features[i].keypoints.size() << endl;
}

finder->collectGarbage();
```

```
full_img.release();
img.release();
cout << "Поиск признаков завершен, время: "
     << ((getTickCount() - t) / getTickFrequency()) << " c" << endl;

// 3 - Сопоставление признаков
cout << "Попарное сравнение" << endl;
t = getTickCount();
vector<MatchesInfo> pairwise_matches;
BestOf2NearestMatcher matcher(false, match_conf);
matcher(features, pairwise_matches);
matcher.collectGarbage();
cout << "Попарное сравнение завершено, время: "
     << ((getTickCount() - t) / getTickFrequency()) << " c" << endl;

// 4 - Выбор изображений и подмножества сопоставленных признаков
// для построения панорамы
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;

// Грубая оценка параметров камеры
HomographyBasedEstimator estimator;
vector<CameraParams> cameras;
if (!estimator(features, pairwise_matches, cameras))
{
 cout << "Ошибка при вычислении гомографии." << 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 << "Начальный внутренний параметр #" << indices[i]+1 << ":\n"
      << cameras[i].K() << endl;
}

// 5 - Глобальное уточнение параметров камеры
```

```
Ptr<BundleAdjusterBase> adjuster;
if (adjuster_method == "reproj") // метод "reproj"
   adjuster = makePtr<BundleAdjusterReproj>();
else // метод "ray"
   adjuster = makePtr<BundleAdjusterRay>();

adjuster->setConfThresh(conf_thresh);
if (!(*adjuster)(features, pairwise_matches, cameras)) {
   cout << "Ошибка при уточнении параметров камеры." << endl;
   return -1;
}

// Ищем медианную фокусную точку
vector<double> focals;
for (size_t i = 0; i < cameras.size(); ++i)
{
  cout << "Камера #" << 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 - Волновая коррекция (факультативно)
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 - Деформирование изображений
cout << "Деформирование изображений (вспомогательное)... " << 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);

// Подготовка масок
for (int i = 0; i < num_images; ++i)
```

```
{
 masks[i].create(images[i].size(), CV_8U);
 masks[i].setTo(Scalar::all(255));
}

// Задание картографической проекции
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 << "Ошибка при создании проектора типа " << 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 << "Деформирование изображений завершено, время: "
     << ((getTickCount() - t) / getTickFrequency()) << " c" << endl;

// 8 - Компенсация ошибок экспозиции
Ptr<ExposureCompensator> compensator =
```

```
    ExposureCompensator::createDefault(expos_comp_type);
compensator->feed(corners, images_warped, masks_warped);

// 9 - Поиск масок швов
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>(
        GraphCutSeamFinderBase::COST_COLOR);
else if (seam_find_type == "gc_colorgrad")
  seam_finder = makePtr<GraphCutSeamFinder>(
        GraphCutSeamFinderBase::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 << "Ошибка при создании обнаружителя швов типа " << seam_find_type
       << endl;
  return 1;
}

seam_finder->find(images_warped_f, corners, masks_warped);

// Освободить память
images.clear();
images_warped.clear();
images_warped_f.clear();
masks.clear();

// 10 - Создать смеситель
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 << "Многополосный смеситель, число полос: " << mb->numBands()
        << endl;
}
else if (blend_type == Blender::FEATHER)
```

```
{
   FeatherBlender* fb = dynamic_cast<FeatherBlender*>(blender.get());
   fb->setSharpness(1.f/blend_width);
   cout << "Перьевой смеситель, резкость: " << fb->sharpness() << endl;
}

blender->prepare(corners, sizes);
// 11 - Композиция
cout << "Композиция..." << 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 << "Композиция изображени #" << indices[img_idx]+1 << endl;

  // 11.1 - Прочитать изображение и при необходимости изменить размер
  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 - Деформировать текущее изображение
  warper->warp(img, K, cameras[img_idx].R, INTER_LINEAR,
               BORDER_REFLECT, img_warped);

  // Деформировать текущую маску
  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 - Компенсация ошибки экспозиции
  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 - Смешивание изображений
    blender->feed(img_warped_s, mask_warped, corners[img_idx]);
  }
  Mat result, result_mask;
  blender->blend(result, result_mask);
  cout << "Композиция завершена, время: "
       << ((getTickCount() - t) / getTickFrequency()) << " c" << endl;

  imwrite(result_name, result);

  cout << "Все сделано, общее время: "
       << ((getTickCount() - start_time) / getTickFrequency()) << " c"
       << endl;
  return 0;
}
```

Эта программа сшивает изображения, выполняя описанные выше шаги. Она принимает имена входных файлов изображений или берет файлы по умолчанию (`.\panorama_image1.jpg` и `panorama_image2.jpg`). Результирующее изображение сохраняется в файле `.\panorama_result.jpg`. В начале включаются заголовки модулей `stitching` и `detail`, а затем объявляется об использовании пространства имен `cv::detail`. Задаются параметры, настраивающие процесс сшивки. Если нужна необычная конфигурация, то очень полезно обратиться к общей диаграмме сшивки (см. рисунок выше). В этом примере 11 шагов. На первом считываются и проверяются входные изображения. Для работы программе необходимо как минимум два изображения.

На втором шаге программа изменяет размеры изображений, используя параметр `double scale = 1`, и находит в каждом изображении признаки. Можно использовать один из двух алгоритмов поиска признаков: **Surf** (`finder = makePtr<SurfFeaturesFinder>()`) или **Orb** (`finder = makePtr<OrbFeaturesFinder>()`), он задается параметром `string features_type = "orb" parameter`. Для изменения размеров изображений вызывается функция `resize(full_img, img, Size(), scale, scale)`, а для поиска признаков — `(*finder)(img, features[i])`.

Дополнительные сведения о SURF- и ORB-дескрипторах смотрите в главе 5 книги «OpenCV Essentials», выпущенной издательством Packt Publishing.

На третьем шаге найденные признаки сопоставляются. При создании сопоставителя `BestOf2NearestMatcher matcher(false, match_conf)` задается параметр `match_conf = 0.3f`.

На четвертом шаге выбираются изображения и сопоставляются их участки для создания панорамы. Затем отбираются лучшие признаки и производится их сопоставление с помощью функции `vector<int> indices = leaveBiggestComponent(features, pairwise_matches, conf_thresh)`. Имея эти признаки, создается новое подмножество.

На пятом шаге производится глобальное уточнение параметров камеры с помощью **блочного уравнивания** (`Ptr<BundleAdjusterBase> adjuster`). Задача блочного уравнивания определяется следующим образом: имея набор изображений, на которых изображены точки на плоскости или в пространстве, снятые разными камерами, требует одновременно уточнить координаты точек, описать геометрию сцену, а также параметры относительного движения и оптические характеристики камер. При этом критерий оптимальности учитывает соответствие проекций всех точек на изображении. Существует два метода блочного уравнивания: `reproj` (`adjuster = makePtr<BundleAdjusterReproj>()`) и `ray` (`adjuster = makePtr<BundleAdjusterRay>()`), для задания этого параметра служит переменная `string adjuster_method = "ray"`. Выбранный уравниватель вызывается следующим образом: `(*adjuster)(features, pairwise_matches, cameras)`.

Шестой шаг факультативный (`bool do_wave_correct = true`), на нем с помощью волновой корреляции уточняются параметры камеры. Вид волновой корреляции выбирается с помощью параметра `WaveCorrectKind wave_correct_type = WAVE_CORRECT_HORIZ`, а вычисляется она с помощью функции `waveCorrect(rmats, wave_correct_type)`.

На седьмом шаге создается деформатор изображения, которому необходима картографическая проекция. Проекции были описаны выше, они обозначаются строками *rectilinear*, *cylindrical*, *spherical*, *stereographic*, *panini*. На самом деле, в OpenCV проекций больше. Проекция задается параметром `string warp_type = "spherical"`. Деформатор создается в результате обращения `Ptr<RotationWarper> warper = warper_creator-> create(static_cast<float>(warped_image_scale * scale))`, а изображение деформируется так: `warper->warp(masks[i], K, cameras[i].R, INTER_NEAREST, BORDER_CONSTANT, masks_warped[i])`.

На восьмом шаге мы создаем компенсатор ошибок экспозиции (`Ptr<ExposureCompensator> compensator = ExposureCompensator::createDefault(expos_comp_type)`) и применяем его к каждому деформированному изображению (`compensator->feed(corners, images_warped, masks_warped)`).

На девятом шаге производится поиск масок швов, т. е. участков, по которым лучше всего соединять изображения в панораму. В OpenCV есть несколько способов решения этой задачи: `NoSeamFinder` (этот метод бесполезен), `VoronoiSeamFinder`, `GraphCutSeamFinderBase::COST_COLOR`, `GraphCutSeamFinderBase::COST_COLOR_GRAD`, `DpSeamFinder::COLOR` и `DpSeamFinder::COLOR_GRAD`. Мы выбрали метод, задав параметр `string seam_find_type = "gc_color"`.

На десятом шаге создается смеситель, который объединяет изображения в панораму. В OpenCV реализовано два смесителя: `MultiBandBlender* mb = dynamic_cast<MultiBandBlender*>(blender.get())` и `FeatherBlender* fb = dynamic_cast<FeatherBlender*>(blender.get())`, мы выбрали первый, задав параметр `int blend_type = Blender::MULTI_BAND`. Затем мы подготавливаем смеситель, вызывая его метод `blender->prepare(corners, sizes)`.

На последнем шаге производится композиция изображений, в результате чего мы получаем окончательную панораму. Здесь понадобится все сделанное на предыдущих шагах для конфигурирования сшивки. Для вычисления панорамы нужно четыре подшага. На первом читаются все изображения (`full_img = imread(img_names[img_idx])`) и при необходимости изменяются их размеры (`resize(full_img, img, Size(), scale, scale)`). На втором к изображениям применяется ранее созданный деформатор (`warper->warp(img, K, cameras[img_idx].R, INTER_LINEAR, BORDER_REFLECT, img_warped)`). На третьем применяется компенсатор ошибок экспозиции (`compensator->apply(img_idx, corners[img_idx], img_warped, mask_warped)`). И наконец, изображения объединяются с помощью смесителя. Получившаяся панорама сохраняется в файле с именем `string result_name = "panorama_result.jpg"`.

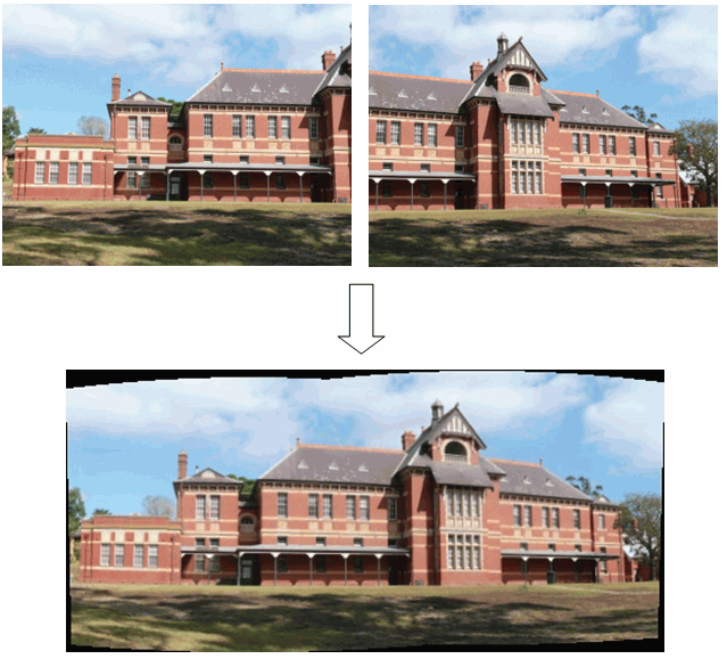
На рисунке на следующей странице показан результат сшивки панорамы из двух изображений с помощью программы **stitchingAdvanced**.

Резюме

В этой главе мы познакомились с тремя модулями OpenCV, предназначенными для обработки изображений в составе видео. Были также изложены теоретические основы каждого модуля.

В каждом разделе был приведен полный код примера на C++ с пояснениями. Показаны результаты применения написанных программ.

В следующей главе дается введение в изображения с широким динамическим диапазоном и методы их обработки в OpenCV. Такие изображения обычно рассматриваются в рамках так называемой «вычислительной фотографии». Не вдаваясь в подробности, можно сказать, что вычислительная фотография изучает способы дополнения стандартных средств цифровой фотографии. Это может быть специальное оборудование или модификация существующего, но по большей части речь идет о программных методах. Так или иначе, на выходе мы имеем изображения, которые невозможно получить с помощью «традиционной» цифровой камеры.



ГЛАВА 6. Вычислительная фотография

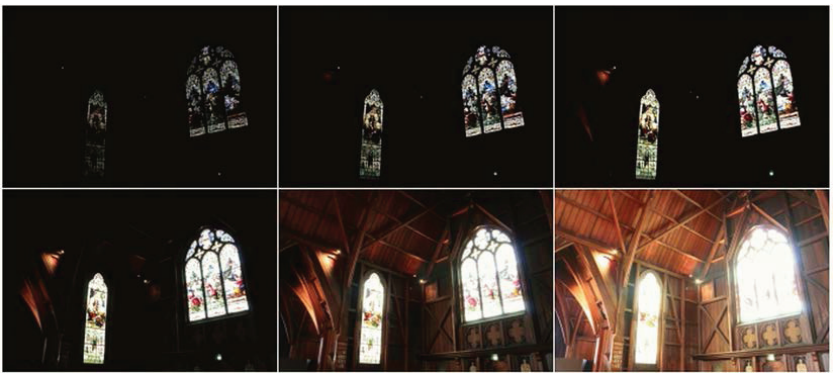
Вычислительная фотография изучает методы, дополняющие типичные возможности цифровой фотографии. Это может быть специальное оборудование или модификация существующего, но по большей части речь идет о программных методах. Так или иначе, на выходе мы имеем изображения, которые невозможно получить с помощью «традиционной» цифровой камеры. В этой главы мы рассмотрим некоторые малоизвестные методы вычислительной фотографии, реализованные в OpenCV: обработка изображений с широким динамическим диапазоном, бесшовное клонирование, обесцвечивание и нефотореалистичный рендеринг. Все они находятся в модуле `photo`. Другие части этого модуля (ретуширование и очистка от шумов) были рассмотрены в предыдущих главах.

Изображения с широким динамическим диапазоном

В типичном изображении пиксели 8-разрядные (8 бит на пиксель – bpp). В цветных изображениях каждый канал также представлен 8-разрядными пикселями, т. е. допускается не более 256 значений интенсивности цвета. Такое ограничение существовало на протяжении всей истории цифровой обработки изображений. Однако очевидно, что в природе у света куда больше уровней яркости. Поэтому возникает вопрос, желательна ли такая дискретизация и достаточна ли она. Известно, что человеческий глаз способен воспринять намного больший динамический диапазон (число уровней между самым тусклым и самым ярким) – от одного до 100 миллионов градаций. Если уровней яркости всего 256, то иногда яркие цвета кажутся пересвеченны-

ми или чрезмерно насыщенными, тогда как темные сцены выглядят просто черными.

Существуют камеры, способные формировать пиксели с большим числом битов. Но самый распространенный способ создания изображений с широким динамическим диапазоном (HDR-изображений) – воспользоваться обычной камерой 8 bpp и сделать фотографии с разными значениями выдержки. При этом возникают очевидные проблемы из-за ограниченности динамического диапазона. Взгляните, к примеру, на следующий рисунок:

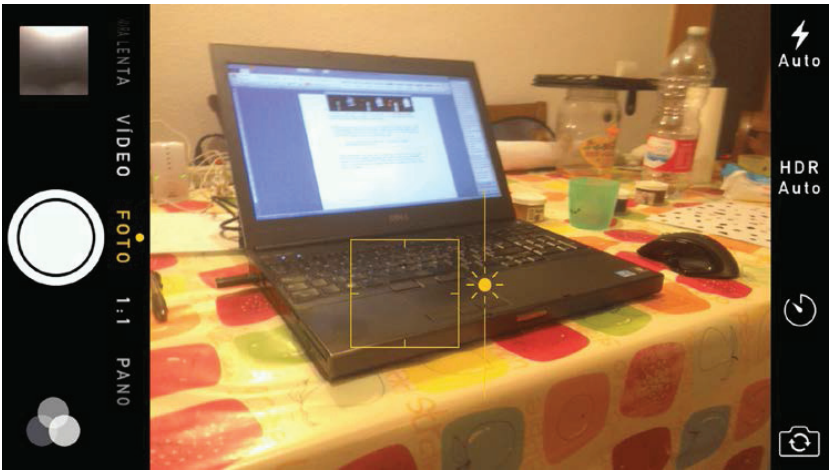


Сцена, снятая с шестью разными значениями выдержки

Верхнее левое изображение почти черное, но детали окна все же видны. Наоборот, на правом нижнем изображении видны детали помещения, зато детали окна едва различимы.

Сделать фотографии с разной выдержкой можно и с помощью современных смартфонов. К примеру, на iPhone и iPads с системой iOS 8 изменить выдержку при съемке встроенной камерой очень просто. Если коснуться экрана, то появится желтый квадрат со значком солнца сбоку. Сдвинув это значок пальцем вниз или вверх, вы сможете изменить выдержку (см. рисунок ниже).

Диапазон уровней выдержки очень велик, поэтому менять ее можно многократно. Если на вашем устройстве установлена более ранняя версия iOS, то можете загрузить стороннее приложение, например *Camera+*, которое позволяет фокусироваться на указанной точке и менять выдержку.



Задание выдержки для камеры, встроенной в iPhone 5S

Для Android в магазине Google Play полно приложений, которые позволяют выставлять выдержку. Например, *Camera FV-5* в платной и бесплатной версии.

Если вы снимаете портативной камерой, то важно, чтобы она была неподвижна. Лучше пользоваться треножником. Иначе изображения, снятые с разной выдержкой, не совместятся. Кроме того, движущаяся камера неизбежно порождает артефакты в виде паразитных изображений. В большинстве случаев достаточно трех снимков, с малой, средней и большой выдержкой.

Смартфоны и планшеты удобны для получения нескольких изображений с разной выдержкой. Чтобы создать HDR-изображение, нужно знать время выдержки каждого изображения. Не все приложения позволяют задавать время вручную (или хотя бы видеть его), например, встроенная в iOS 8 камера не дает такой возможности. На момент написания этой книги для iOS существовало по меньшей мере два бесплатных приложения с возможностью задания времени выдержки: *Manually* и *ManualShot!*. Для Android бесплатное приложение *Camera FV-5* позволяет выставлять и видеть время выдержки. Отметим, что для управления выдержкой служат еще два параметра: F/Stop и ISO.

Снятые изображения можно скопировать на компьютер и использовать для создания HDR-изображения.

В системе iOS 7 во встроенном приложении камеры имеется режим HDR, в котором автоматически делаются три фотографии с разной выдержкой. Затем они также автоматически объединяются в одно изображение (иногда оно действительно получается лучше).

Создание HDR-изображений

Как объединить несколько (например, три) изображения с разной выдержкой в одно HDR-изображение? Если взять только один канал и один пиксель, то необходимо преобразовать три его значения (по одному на каждое время экспозиции) в одно значение, принадлежащее более широкому выходному диапазону (скажем, 16 bpp). Сделать это не просто. Прежде всего, нужно понимать, что интенсивности пикселей – это грубая мера плотности излучения (количества света, падающего на фотоприемник камеры). Цифровые камеры измеряют плотность излучения, и для любой камеры определена нелинейная функция отклика, которая преобразует плотность излучения в уровень интенсивности пикселя от 0 до 255. Чтобы отобразить это значение в более широкий дискретный диапазон, необходимо оценить функцию отклика камеры.

А как оценить функцию отклика? По самим пикселям! Функция отклика – это S-образная кривая для каждого цветового канала, ее можно вычислить, зная все пиксели (при трех уровнях выдержки на пиксель мы имеем три точки на кривой). Но поскольку это очень долго, то обычно выбирают случайное подмножество пикселей.

Осталась только одна вещь. Ранее мы упомянули о вычислении соотношения между плотностью излучения и интенсивностью пикселя. А откуда мы знаем плотность излучения? Для каждого фотоприемника она прямо пропорциональна времени экспозиции (или, что то же самое, скорости срабатывания затвора). Вот потому-то нам и нужно знать выдержку!

И наконец, HDR-изображение вычисляется в виде взвешенной суммы значений плотности излучения, восстановленных по пикселям, полученным при нескольких уровнях выдержки. Отметим, что такое изображение нельзя показать на обычном экране, диапазон которого ограничен.

По обработке изображений с широким динамическим диапазоном можно порекомендовать хорошую книгу: Reinhard и др. «High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting», вышедшую в издательстве Morgan Kaufmann. К книге прилагается DVD, содержащий изображения в различных HDR-форматах.

Пример

В OpenCV (только в версии 3.0) имеются функции для создания HDR-изображений по набору фотографий, снятых с разной выдержкой. Есть даже учебная программа *hdr_imaging*, которая читает список файлов изображений и время экспозиции каждого (из текстового файла) и создает HDR-изображение.

Для запуска программы hdr_imaging необходимо скачать файлы изображений и текстовые файлы, содержащие данные о них. Все это есть на странице `https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/hdr`.

Классы `CalibrateDebevec` и `MergeDebevec` реализуют два алгоритма Дебевека: вычисление функции отклика камеры и объединение изображений с разной выдержкой в одно HDR-изображение. В программе **createHDR** ниже показано, как пользоваться этими классами.

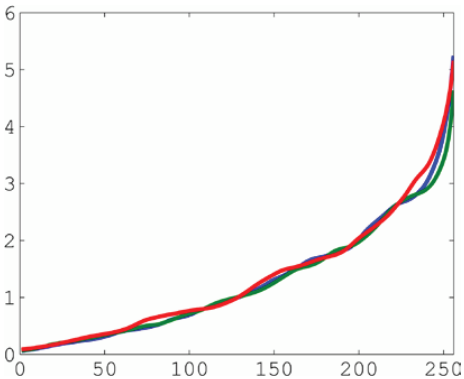
```
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
 vector<Mat> images;
 vector<float> times;

 // Загружаем изображения и данные о выдержке
 Mat img1 = imread("1div66.jpg");
 if (img1.empty())
 {
  cout << "Ошибка при загрузке входного изображения...\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);

// Вычисляем функцию отклика камеры
Mat response;
Ptr<CalibrateDebevec> calibrate = createCalibrateDebevec();
calibrate->process(images, response, times);

// Показываем вычисленную функцию отклика
cout << response;

// Создаем и записываем HDR-изображение
Mat hdr;
Ptr<MergeDebevec> merge_debevec = createMergeDebevec();
merge_debevec->process(images, hdr, times, response);
imwrite("hdr.hdr", hdr);

cout << "\nГотово. Для выхода нажмите любую клавишу.\n";
waitKey(); // Ждем нажатия клавиши
return 0;
}
```

В этом примере использованы три изображения чашки (они включены в состав сопроводительных материалов к книге). Фотографии были сделаны с помощью вышеупомянутого приложения *ManualShot!* с выдержкой 1/66, 1/32 и 1/12 секунды.

Три изображения, использованные в примере программы

Отметим, что метод `createCalibrateDebevec` ожидает, что изображения и время экспозиции передаются в STL-векторе (STL – это стандартная библиотека C++, содержащая полезные общеупотребительные функции и структуры данных). Функция отклика камеры представлена вектором, содержащим 256 вещественных значений. Она описывает отображение между плотностью излучения и значением пикселя. На самом деле, это не вектор, а матрица 256×3 (по одному столбцу на каждый из трех цветовых каналов). На следующем рисунке показана функция отклика, вычисленная в данном примере.



Вычисленная функция отклика RGB-камеры

На стандартный вывод `cout` выводится матрица в формате, используемом в двух популярных пакетах программ для численных расчетов: MATLAB и Octave. Чтобы увидеть графическое представление функции, скопируйте выведенные данные и вставьте их в MATLAB или Octave.

Результирующее HDR-изображение сохраняется в формате RGBE без потери информации. В этом формате каждый цветовой канал кодируется одним байтом, а дополнительный байт содержит общий показатель степени. Принцип такой же, как для представления чисел с плавающей точкой: общий показатель степени позволяет представить гораздо более широкий диапазон значений. Формату RGBE соответствует расширение `.hdr`. Поскольку это формат без потери информации, то `hdr`-файлы довольно велики. В нашем примере входные RGB-изображения с разрешением 1224×1632 занимали от 100 до 200 КБ, а выходной `hdr`-файлы – 5.9 МБ.

В этом примере использовался метод Дебевека и Малика, но OpenCV предлагает еще одну функцию калибровки, основанную на методе Робертсона. Соответствующие функции калибровки и объединения называются `createCalibrateRobertson` и `MergeRobertson`.

Дополнительные сведения об этих функциях и стоящей за ними теории можно найти на странице `http://docs.opencv.org/trunk/modules/photo/doc/hdr_imaging.html`.

Наконец, отметим, что наша программа не выводит получившееся изображение. HDR-изображение невозможно показать на стандартном экране, поэтому необходим дополнительный шаг – тональная компрессия (tone mapping).

Тональная компрессия

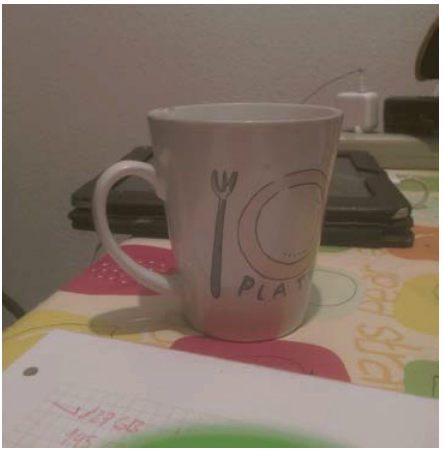
При отображении HDR-изображения на экране или на бумаге информация теряется. Связано это с тем, что коэффициент контрастности компьютерного экрана ограничен, а количество оттенков в печатных материалах обычно не превышает 256. Поэтому значения интенсивности пикселей HDR-изображения необходимо привести к более узкому диапазону. Это и называется тональной компрессией.

Но простого масштабирования значений пикселей в уменьшенный диапазон устройства отображения недостаточно для создания реалистичного результата, т. к. при этом обычно теряются детали (контрастность) и искажается истинный вид сцены. Алгоритмы тональной компрессии стремятся породить изображения, визуально схожие с оригинальной сценой (т. е. с тем, что увидел бы рассматривающий ее человек). Было предложено несколько таких алгоритмов, и до сих пор в этой области активно ведутся исследования. В следующем фрагменте тональная компрессия применяется к созданному выше HDR-изображению:

```
Mat ldr;
Ptr<TonemapDurand> tonemap = createTonemapDurand(2.2f);
tonemap->process(hdr, ldr); // ldr – изображение, представленное числами с
ldr=ldr*255;            // плавающей точкой в диапазоне [0..1]
imshow("LDR", ldr);
```

Здесь используется алгоритм, предложенный Дюраном (Durand) и Дорси (Dorsey) в 2002 году. Конструктор принимает ряд параметров,

от которых зависит результат. На рисунке ниже показано выходное изображение. Отметим, что оно необязательно *лучше* любого из трех исходных изображений.



Результат применения тональной компрессии

В OpenCV есть еще три алгоритма тональной компрессии: `createTonemapDrago`, `createTonemapReinhard` и `createTonemapMantiuk`.

HDR-изображение (в формате RGBE, в файле с расширением `.hdr`) можно отобразить в MATLAB. Для этого достаточно таких трех строчек кода:

```
hdr=hdrread('hdr.hdr');
rgb=tonemap(hdr);
imshow(rgb);
```

pfstools – набор командных программ с открытым исходным кодом для чтения, записи и отображения HDR-изображений. В него включены средства для чтения изображений в формате `hdr` и других, а также алгоритмы калибровки камеры и тональной компрессии. На основе pfstools создана программа с графическим интерфейсом Luminance HDR.

Совмещение

Сцена, снимаемая с разной выдержкой, должна быть неподвижной, как и камера. Но даже если оба условия выполнены, все равно рекомендуется произвести совмещение.

В OpenCV имеется алгоритм совмещения изображений, предложенный Дж. Уордом (G. Ward) в 2003 году. Основная функция, `createAlignMTB`, принимает параметр, определяющий максимальный сдвиг (точнее, двоичный логарифм максимального сдвига по каждому измерению). Перед вычислением функции отклика камеры в предыдущем примере нужно вставить такие строчки:

```
vector<Mat> images_(images);
Ptr<AlignMTB> align=createAlignMTB(4);// 4=макс сдвиг - 16-пикселей
align->process(images_, images);
```

Экспозиционное объединение

Изображения с разной выдержкой можно объединить даже без калибровки камеры и создания промежуточного HDR-изображения. Эта процедура, называемая экспозиционным объединением (exposure fusion), была предложена Мертенсом с соавторами в 2007 году. Ниже показано, как выполнить экспозиционное объединение (входные изображения хранятся в STL-векторе `images`; см. предыдущий пример):

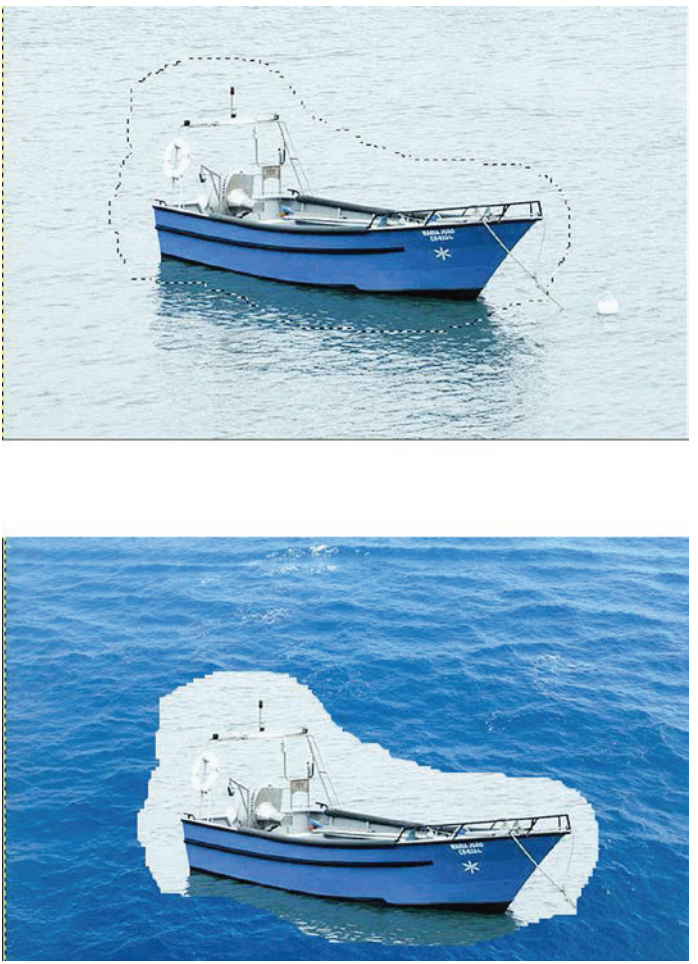
```
Mat fusion;
Ptr<MergeMertens> merge_mertens = createMergeMertens();
// fusion - изображение, представленное числами с плавающей
// точкой в диапазоне [0..1]
merge_mertens->process(images, fusion);
fusion=fusion*255;
imwrite("fusion.png", fusion);
```

На рисунке ниже показан результат.

Результат экспозиционного объединения

Бесшовное клонирование

Целью фотомонтажа обычно является вырезание объекта или человека из одного изображения и вставка в другое. Конечно, это можно сделать, бесхитростно скопировав объект. Но реалистичного эффекта так не получится. Так, на рисунке ниже мы попытались вставить судно из верхнего изображения в море в нижнем изображении.



Клонирование

В версии OpenCV 3 имеется функция бесшовного клонирования `seamlessClone`, позволяющая получить более реалистичный результат. Она основана на алгоритме, который предложили Перес (Perez) и Гангнет (Gangnet) в 2003 году. В программе **seamlessCloning** показано, как ей пользоваться.

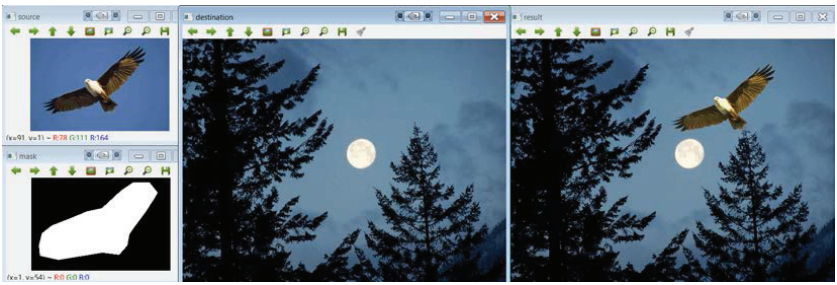
```
#include <opencv2/photo.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main(int, char** argv)
{
 // Загружаем и показываем изображения
 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 находится рядом с правым верхним углом
 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 << "\nГотово. Для выхода нажмите любую клавишу.\n";
 waitKey(); // Ждем нажатия клавиши
 return 0;
}
```

Здесь нет ничего сложного. Функция `seamlessClone` принимает начальное и конечное изображения, масочное изображение и точку в конечном изображении, в которой должен быть вставлен вырезанный объект (все три изображения можно скачать со страницы `https://github.com/Itseez/opencv_extra/tree/master/ testdata/cv/cloning/Normal_Cloning`). Результат показан на рисунке на следующей странице.

Последний параметр функции `seamlessClone` задает метод клонирования (есть три метода, дающие разные результаты). В библиотеке имеются также дополнительные функции:

Функция	Эффект
`colorChange`	Умножает каждый из трех цветовых каналов исходного изображения на коэффициент, ограничиваясь для клонирования области, заданной маской, только умножением
`illuminationChange`	Изменяет освещенность исходного изображения в области, заданной маской
`textureFlattening`	Размывает текстуры в области исходного изображения, заданной маской

В отличие от `seamlessClone`, эти функции принимают только исходное и масочное изображения.



Бесшовное клонирование

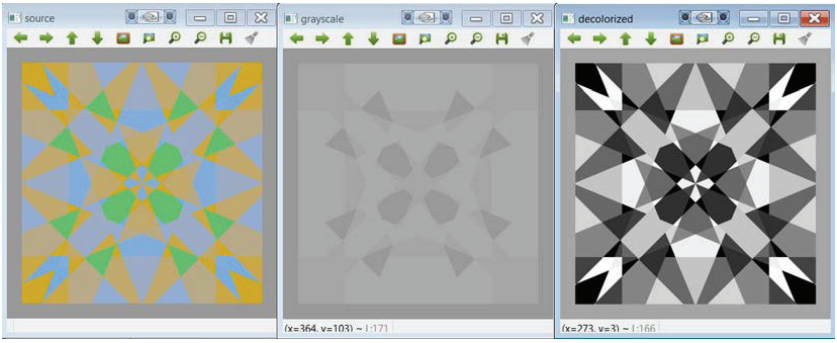
Обесцвечивание

Обесцвечиванием называет процесс преобразования цветного изображения в полутоновое. У читателя, прочитавшего такое определение, возникает законное недоумение: «Ведь у нас уже есть преобразование в полутоновое изображение, разве нет?». Да, такое преобразование относится к числу базовых, оно имеется и в OpenCV и в любой другой библиотеке обработки изображений. Стандартное преобразование основано на линейном комбинировании каналов R, G и B. Проблема в том, что при таком преобразовании может теряться контрастность из-за того, что два разных цвета (воспринимаемые в исходном изображении как контрастные) отображаются на одно и то же значение яркости. Пусть есть два цвета, A и B, и мы хотим преобразовать их в оттенки серого. Предположим, что B получается из A разнонаправленным изменением каналов R и G:

$A = (R,G,B) => G = (R+G+B)/3$
$B = (R-x,G+x,B) => G = (R-x+G+x+B)/3 = (R+G+B)/3$

Цвета A и B, воспринимаемые как различные, отображаются в одно и то же значение яркости! Это видно на изображениях из следующей программы **decolorization**:

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

using namespace cv;
using namespace std;

int main(int, char** argv)
{
 // Загружаем и показываем изображения
 Mat source = imread("color_image_3.png", IMREAD_COLOR);
 imshow("source", source);

 // сначала вычисляем и показываем стандартное полутоновое преобразование
 Mat grayscale = Mat(source.size(),CV_8UC1);
 cvtColor(source, grayscale, COLOR_BGR2GRAY);
 imshow("grayscale",grayscale);

 // теперь вычисляем и показываем обесцвечивание
 Mat decolorized = Mat(source.size(),CV_8UC1);
 Mat dummy = Mat(source.size(),CV_8UC3);
 decolor(source,decolorized,dummy);
 imshow("decolorized",decolorized);

 cout << "\nГотово. Для выхода нажмите любую клавишу.\n";
 waitKey(); // Ждем нажатия клавиши
 return 0;
}
```



Результат обесцвечивания

Пример тривиален. Сначала мы читаем изображение и показываем результат стандартного полутонового преобразования. А затем выполняем обесцвечивание с помощью функции `decolor`. Использованное изображение (`color_image_3.png`) включено в репозиторий opencv_extra по адресу `https://github.com/Itseez/opencv_extra/tree/master/testdata/cv/decolor`.

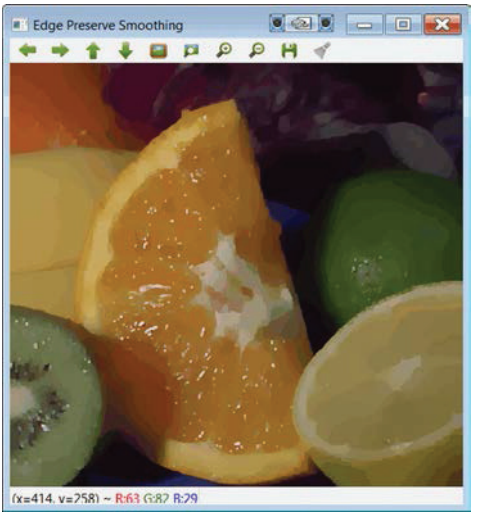
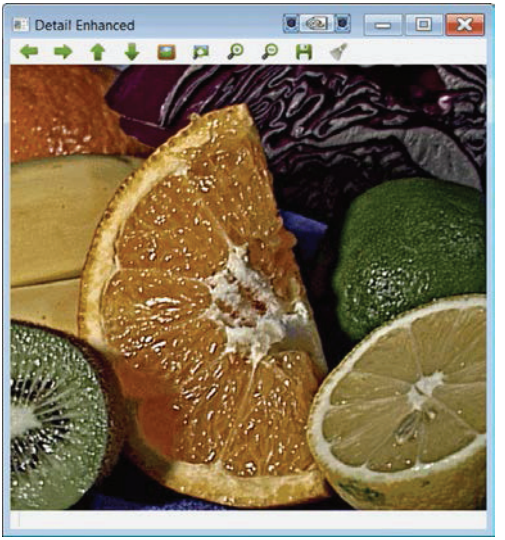
Это изображение – конечно, крайний случай. Цвета в нем специально подобраны так, чтоб результат стандартного полутонового преобразования, выглядел однородно.

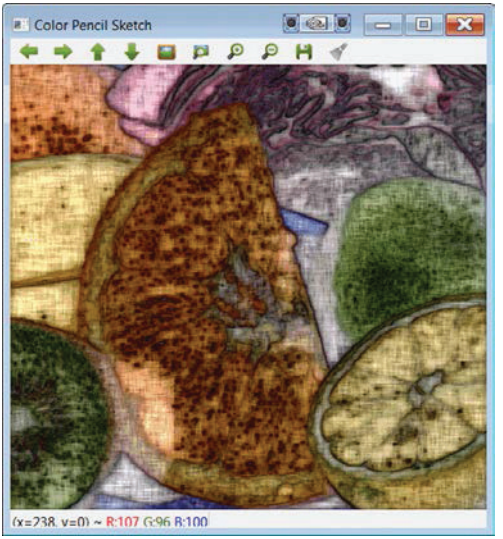
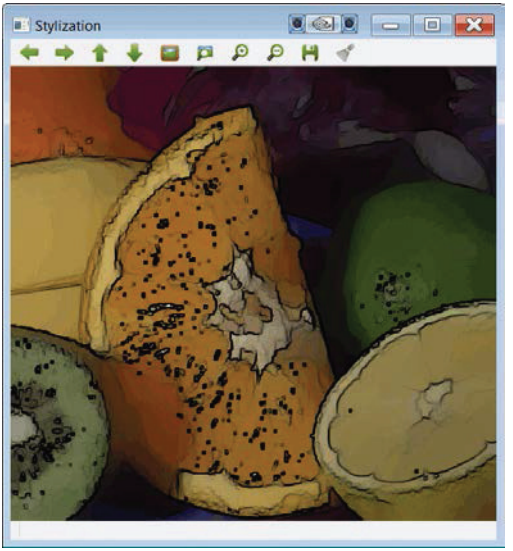
Нефотореалистичный рендеринг

В модуле `photo` есть четыре функции, которые преобразуют изображение так, что получается нереалистичный, но художественный результат. Ими легко пользоваться, и в OpenCV есть симпатичный пример (`npr_demo`). Взгляните на изображение `fruits.jpg`, входящее в состав OpenCV:

Исходное изображение для сравнения

Для иллюстрации приведем таблицу, в которой описаны эффекты всех функций.

Функция	Эффект
`edgePreservingFilter`	Сглаживание – часто применяемый фильтр. Эта функция осуществляет сглаживание, сохраняя детали на границах объектов 
`detailEnhance`	Усиливает детали 

Функция	Эффект
`pencilSketch`	Вариант входного изображения с эффектом рисования карандашом 
`stylization`	Эффект рисования акварелью 

Резюме

В этой главе мы познакомились с вычислительной фотографией и относящимися к ней функциями в OpenCV 3. Мы рассказали о самых важных функциях в модуле `photo`, отметив, что другие находящиеся в нем функции (ретуширование и очистка от шумов) были рассмотрены в предыдущих главах. Вычислительная фотография – быстро развивающаяся область, которая имеет тесные связи с компьютерной графикой. Поэтому можно ожидать, что этот модуль в последующих версиях будет расширяться.

Следующая глава посвящена важному, но еще не рассмотренному аспекту: времени. Многие из вышеупомянутых функций работают довольно долго. Мы расскажем, как справиться с этой проблемой, используя современное оборудование.

ГЛАВА 7. Ускорение обработки изображений

В этой главе рассматривается ускорение обработки изображений с помощью графических процессоров общего назначения (General Purpose Graphics Processing Units, **GPGPU**), или просто **GPU**, со встроенными средствами распараллеливания вычислений. GPU – это, по существу, сопроцессор, предназначенный для обработки графики или выполнения операций с плавающей точкой. Его задача – повысить производительность таких приложений, как видеоигры и интерактивная трехмерная графика. Пока GPU занимается обработкой графики, ЦП может выполнять другие вычисления (например, относящиеся к встроенным в игру алгоритмам искусственного интеллекта). Каждый GPU содержит сотни простых процессорных ядер, что позволяет организовать массивно параллельное выполнение сотен «простых» математических операций над числами с плавающей точкой (как правило).

Центральные процессоры, похоже, достигли предела быстродействия и температурного режима. Конструирование компьютеров с несколькими ЦП стало трудной задачей. Тут-то и вступают в игру GPU. Вычисления на GPU, призванные повысить общую производительность компьютера, стали новой парадигмой. В первых графических процессорах были реализованы лишь некоторые параллельные операции, так называемые графические примитивы, оптимизированные для обработки графики. Один из самых распространенных примитивов в трехмерной графике – антиалиасинг, т. е. придание границам объектов более реалистичного вида. Другие примитивы – рисование прямоугольников, треугольников, окружностей и дуг. Современные GPU поддерживают сотни функций общего назначения, выходящих далеко за пределы отрисовки графики. Особенно полезны они для решения задач, допускающих распараллеливание, которые часто встречаются в алгоритмах компьютерного зрения.

OpenCV поддерживает архитектуры GPU OpenCL и CUDA. В CUDA реализовано много алгоритмов, но она работает только с графическими картами компании NVIDIA. CUDA – это платформа параллельных вычислений и модель программирования, созданная NVIDIA и реализованная в производимых ей GPU. В этой главе мы будем рассматривать архитектуру OpenCL, которая поддерживается большим числом устройств и даже некоторыми графическими картами NVIDIA.

Open Computing Language (OpenCL) – это каркас для написания программ, которые можно исполнять на ЦП или GPU, подключенных к хост-процессору (ЦП). В нем определен похожий на C язык для программирования функций, называемых ядрами, которые исполняются вычислительными устройствами. OpenCL позволяет параллельно выполнять ядра на всех или некоторых процессорных элементах (PE), будь то ЦП или GPU.

Кроме того, в OpenCL определен **интерфейс прикладного программирования (API)**, который позволяет программам, работающим на хосте (ЦП), запускать ядра на вычислительных устройствах и управлять их памятью, которая (по крайней мере, концептуально) отделена от памяти хоста. Предполагается, что программы на языке OpenCL компилируются на этапе выполнения, поэтому написанные на OpenCL приложения переносимы на различные хост-системы. OpenCL – открытый стандарт, поддерживаемый некоммерческим консорциумом Khronos Group (`https://www.khronos.org/`).

OpenCV содержит классы и функции для ускоренной реализации функциональности OpenCV с помощью OpenCL. В настоящее время OpenCV предоставляет прозрачный API, который допускает унификацию оригинального API с ускорением средствами OpenCL. Поэтому достаточно написать только один вариант кода. Введена новая унифицированная структура данных **UMat**, которая отвечает за копирование данных в память GPU, когда это возможно и необходимо.

Поддержка OpenCL в OpenCV спроектирована так, чтобы облегчить использование, знать OpenCL при этом необязательно. На верхнем уровне ее можно рассматривать как набор способов ускорения, которые задействуют средства современных ЦП и GPU, если они доступны.

Для запуска OpenCL-программ поставщик оборудования должен обеспечить среду выполнения OpenCL, обычно в виде драйвера устройства. Кроме того, библиотеке OpenCV необходим совместимый комплект средств разработки (SDK). В настоящее время существует пять SDK для OpenCL.

- **AMD APP SDK**: поддерживает OpenCL на ЦП и GPU, в частности на ЦП с набором команд X86+SSE2 (или выше), AMD Fusion, AMD Radeon, AMD Mobility и GPU ATI FirePro.
- **Intel SDK**: поддерживает OpenCL на процессорах семейства Intel Core и GPU Intel HD GPU, например: Intel+SSE4.1, SSE4.2 или AVX, Intel Core i7, i5 и i3 (поколения 1, 2 и 3), Intel HD Graphics, Intel Core 2 Solo (Duo Quad и Extreme), и ЦП Intel Xeon.
- **IBM OpenCL Development Kit**: поддерживает OpenCL на серверах IBM таких, как IBM Power, IBM PERCS и IBM BladeCenter.
- **IBM OpenCL Common Runtime**: поддерживает OpenCV ЦП и GPU, например: ЦП с набором команд X86+SSE2 (или выше), GPU AMD Fusion и Radeon, NVIDIA Ion, NVIDIA GeForce and NVIDIA Quadro.
- **Nvidia OpenCL Driver and Tools**: поддерживает OpenCL на некоторых графических картах Nvidia, например: NVIDIA Tesla, NVIDIA GeForce, NVIDIA Ion и NVIDIA Quadro.

Установка OpenCV с поддержкой OpenCL

Процедура установки, описанная в главе 1, для включения OpenCL нуждается в дополнительных шагах. Ниже объясняется, какое программное обеспечение необходимо.

Для компиляции и установки OpenCV с поддержкой OpenCL в Windows предъявляются дополнительные требования.

- **GPU или CPU с поддержкой OpenCL**: это самое важное требование. Отметим, что OpenCL поддерживают многие, но не все вычислительные устройства. Можно проверить, относится ли ваш процессор или графическая карта к их числу. В этой главе предполагается наличие AMD APP SDK для графического процессора AMD FirePro W5000.

Список вычислительных устройств, поддерживающих этот SDK, опубликован на странице `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/systemrequirements-`

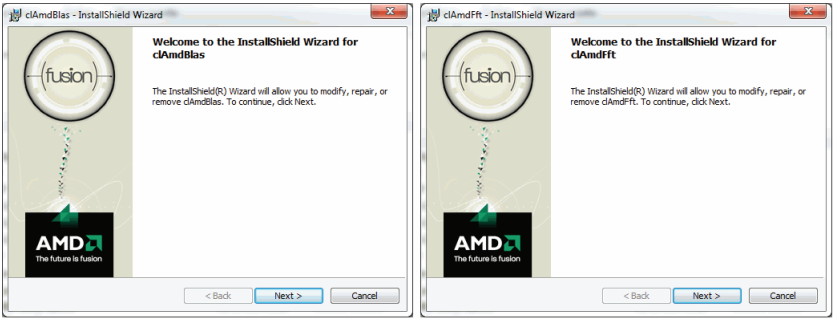
driver-compatibility/. Там же можно узнать, какая минимальная версия SDK необходима.

- **Компиляторы**: OpenCV с поддержкой OpenCL совместима с компиляторами Microsoft и MinGW. Можно установить бесплатную версию Visual Studio Express. Но если вы собираетесь компилировать OpenCV с помощью продуктов Microsoft, то рекомендуется версия не ниже Visual Studio 2012. Впрочем, в этой главе используется компилятор MinGW.
- **AMD APP SDK**: в этом SDK реализованы передовые технологии, позволяющие использовать совместимые вычислительные устройства для выполнения и ускорения самых разных приложений, а не только графических. SDK можно скачать по адресу http://developer.amd.com/tools-and-sdks/opencl-zone/amd-acceleratedparallel-processing-app-sdk/. В этой главе мы используем версию SDK 2.9 (для 64-разрядных версий Windows); процедура установки показана на снимке экрана ниже.

Установка AMD APP SDK

Если на этом шаге происходит ошибка, то, возможно, следует обновить драйвер графической карты. Драйверы AMD можно скачать по адресу `http://www.amd.com/en-us/innovations/software-technologies`.

- **OpenCL BLAS**: базовый набор подпрограмм линейной алгебры (Basic Linear Algebra Subroutines, BLAS) – набор математических библиотек с открытым исходным кодом для параллельных вычислений на устройствах AMD. Его можно скачать по адресу `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-math-libraries/`. В этой главе используется версия BLAS 1.1 для 32- или 64-разрядных версий Windows 32, на рисунке ниже (слева) показано, как выглядит процедура установки.
- **OpenCL FFT**: быстрое преобразование Фурье (FFT) – крайне полезная функция, которая используется во многих алгоритмах обработки изображений, и потому предоставляется ее параллельная реализация для устройств AMD. Ее можно скачать с той же страницы, что и выше. В этой главе используется версия FFT 1.1 для 32- или 64-разрядных версий Windows 32, на рисунке ниже (справа) показано, как выглядит процедура установки.



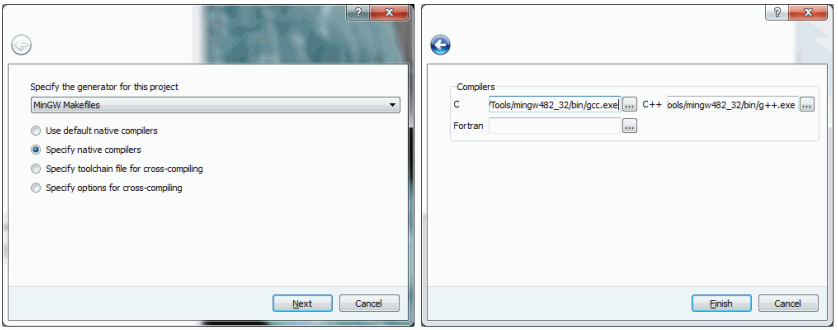
Установка BLAS и FFT для OpenCL

- **Библиотеки Qt для компилятора C++**: в этой главе для компиляции OpenCV с поддержкой OpenCL используются двоичные библиотеки Qt, собранные компилятором MinGW. Альтернатива – установить последнюю версию Qt и восполь-

зоваться компилятором Visual C++. Версию Qt и компилятор выбираете вы сами. Для загрузки различных версий Qt можно воспользоваться менеджером пакетов MaintenanceTool.exe, который находится в каталоге C:\Qt\Qt5.3.1. В этой главе для компиляции OpenCV с поддержкой OpenCL используется версия Qt 5.3.1 и 32-разрядный компилятор MinGW 4.8.2.

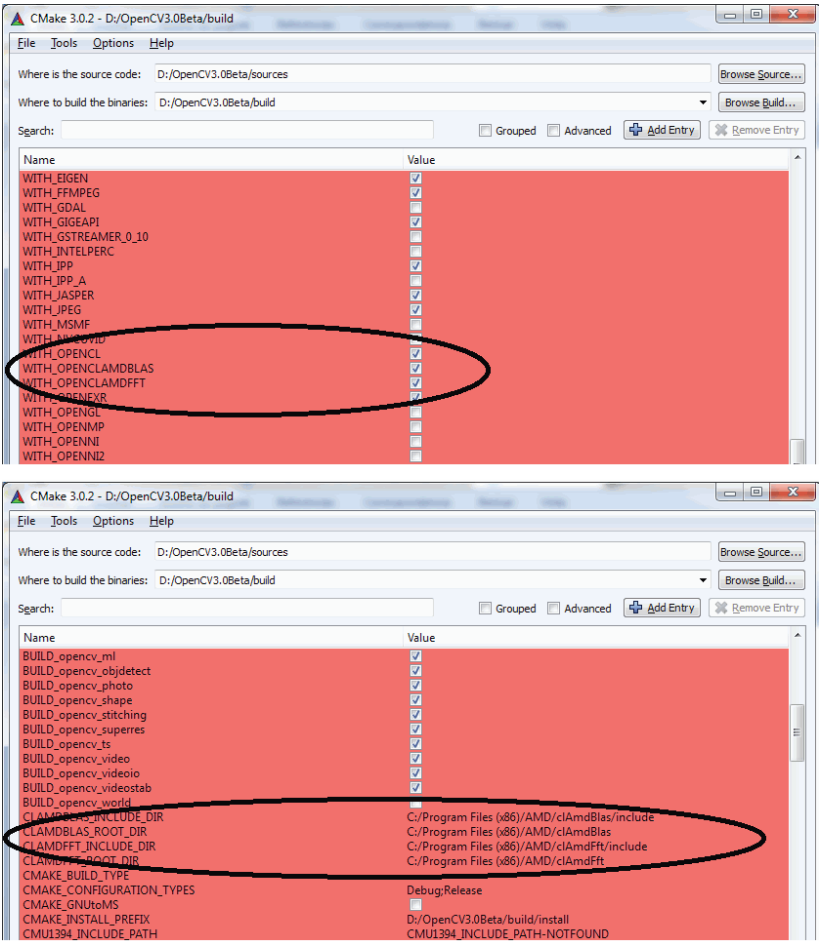
Если все эти требования выполнены, то можно сгенерировать новую конфигурацию сборки с помощью CMake. В нескольких местах этот процесс отличается от описанного в первой главе. Различия перечислены ниже.

- При выборе генератора проекта можно указать версию компилятора, соответствующую установленному окружению. В этой главе для компиляции OpenCV с поддержкой OpenCL используется MinGW, поэтому для указания системы компиляции установлен переключатель «Specify native compilers» (Платформенные компиляторы) и выбран вариант **MinGW Makefiles** (см. рисунок ниже).



Выбор генератора проекта в CMake

- На снимке экрана ниже показаны параметры сборки OpenCV с поддержкой OpenCL. Необходимо включить параметры **WITH_OPENCL**, **WITH_OPENCLAMDBLAS** и **WITH_OPENCLAMDFFT**. В параметрах **CLAMDBLAS_INCLUDE_DIR**, **CLAMDBLAS_ROOT_DIR**, **CLAMDFFT_INCLUDE_DIR** и **CLAMDFFT_ROOT_DIR** следует указать пути к BLAS и FFT. Кроме того, нужно включить параметр **WITH_QT** и выключить **WITH_IPP** (см. главу 1). Рекомендуется также включить **BUILD_EXAMPLES**.



Задание основных параметров в CMake

Наконец, необходимо откомпилировать проект, сгенерированный CMake. Поскольку он генерировался с расчетом на MinGW, для его сборки понадобится этот компилятор. Сначала перейдите в папку `[opencv_build]/` в консоли Windows, а затем выполните команду:

```
./mingw32-make.exe -j 4 install
```

Параметр `-j 4` задает количество процессорных ядер в системе, которая будет использоваться для параллельного выполнения.

Теперь OpenCV с поддержкой OpenCL готова к использованию. В системный список путей нужно добавить путь к двоичным файлам библиотеки, в данном случае `[opencv_build]/install/x64/mingw/bin`.

Не забудьте удалить путь к старым двоичным файлам OpenCV из переменной окружения PATH.

Краткое описание установки OpenCV с поддержкой OpenCL

Перечислим кратко основные шаги процедуры установки.

1. Скачать и установить AMD APP SDK со страницы `http://developer.amd.com/ tools-and-sdks/opencl-zone/amd-acceleratedparallel-processing-app-sdk`.
2. Скачать и установить BLAS и FFT для AMD со страницы `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-acceleratedparallel-processing-math-libraries`.
3. Настроить процедуру сборки OpenCV с помощью CMake. Включить параметры **WITH_OPENCL**, **WITH_OPENCLAMDBLAS, WITH_OPENCLAMDFFT**, **WITH_QT** и **BUILD_EXAMPLES**. Выключить **WITH_IPP**. Указать пути к BLAS и FFT в параметрах **CLAMDBLAS_INCLUDE_DIR**, **CLAMDBLAS_ROOT_DIR**, **CLAMDFFT_INCLUDE_DIR** и **CLAMDFFT_ROOT_DIR**.
4. Откомпилировать проект OpenCV с помощью `mingw32-make.exe`.
5. Включить в переменную окружения PATH путь к каталогу bin OpenCV (например, `[opencv_build]/install/x64/mingw/bin`).

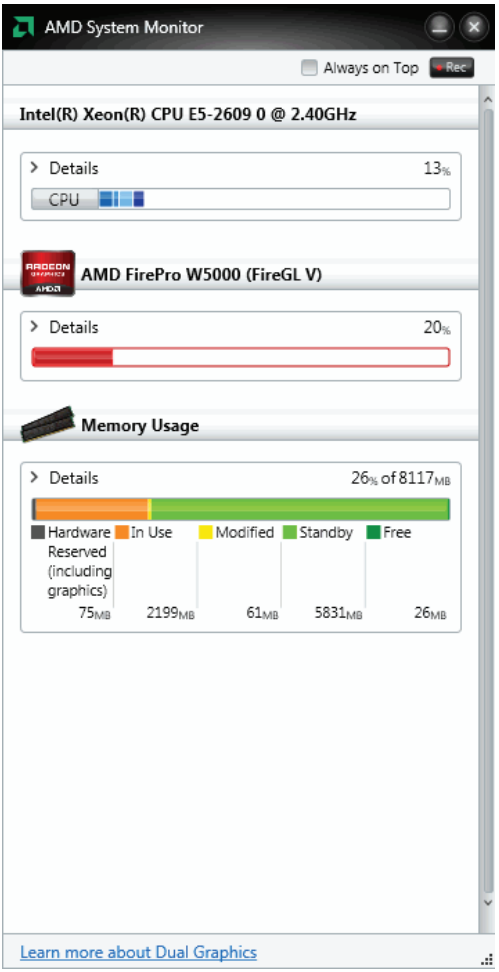
Проверка использования GPU

В штатной поставке Windows нет средств для измерения использования GPU. А знать это полезно по двум причинам:

- убедиться, что GPU действительно используется;
- отслеживать процент потребления ресурсов GPU.

Но на рынке есть несколько приложений для этой цели. В этой главе мы будем использовать **AMD System Monitor**. Это приложение

следит за использованием ЦП, памяти и GPU. Как оно выглядит, показано на снимке экрана ниже.



AMD System Monitor следит за использованием ЦП, памяти и GPU

Программу AMD System Monitor для Microsoft Windows (32 и 64-разрядных версий) можно скачать со страницы `http://support.amd.com/es-xl/kb-articles/Pages/AMDSystemMonitor.aspx`.

Ускорение собственных функций

В этом разделе мы приведем три примера использования OpenCV с поддержкой OpenCL. В первом примере проверяется, доступен ли SDK, и печатается полезная информация о найденных вычислительных устройствах, поддерживающих OpenCL. Во втором примере показаны две версии одной программы, написанные с использованием ЦП и GPU. И последний пример представляет собой законченную программу для обнаружения и пометки лиц. Попутно в нем производится сравнение производительности.

Проверка поддержки OpenCL

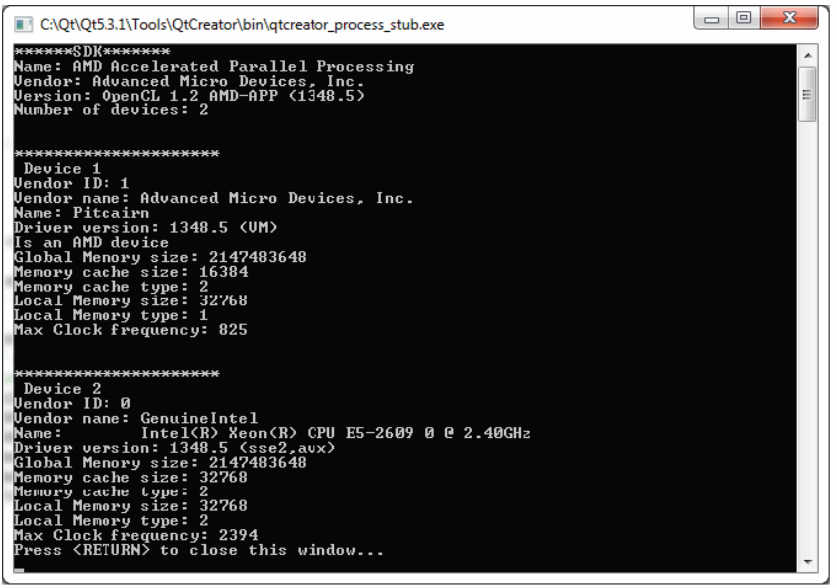
Следующая простая программа, **checkOpenCL**, проверяет, доступен ли SDK и какие есть вычислительные устройства. Пользуясь модулем `ocl` библиотеки OpenCV, она выводит список вычислительных устройств.

```
#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 << "Название: " << sdk.name() << endl;
 cout << "Поставщик: " << sdk.vendor() << endl;
 cout << "Версия: " << sdk.version() << endl;
 cout << "Число устройств: " << sdk.deviceNumber() << endl;

 for (int i=0; i<sdk.deviceNumber(); i++){
  Device device;
  sdk.getDevice(device, i);
  cout << "\n\n*********************\n Устройство " << i+1 << endl;
  cout << "Ид поставщика: " << device.vendorID() << endl;
  cout << "Название поставщика: " << device.vendorName() << endl;
  cout << "Название: " << device.name() << endl;
```

```
    cout << "Версия драйвера: " << device.driverVersion() << endl;
    if (device.isAMD()) cout << "Это устройство AMD " << endl;
    if (device.isIntel()) cout << " Это устройство Intel " << endl;
    cout << "Размер глобальной памяти: " << device.globalMemSize() <<endl;
    cout << "Размер кэша памяти: " << device.globalMemCacheSize() <<endl;
    cout << "Тип кэша памяти: " << device.globalMemCacheType() <<endl;
    cout << "Размер локальной памяти: " << device.localMemSize() << endl;
    cout << "Тип локальной памяти: " << device.localMemType() << endl;
    cout << "Макс тактовая частота: " << device.maxClockFrequency() <<endl;
  }

  return 0;
}
```

Пояснения к коду

На снимке экрана ниже показаны результаты выполнения этой программы.



Информация об установленном SDK и вычислительных устройствах

Эта программа выводит версию установленного SDK и сведения о вычислительных устройствах, совместимых с OpenCL. В начале включается заголовок `core/ocl.hpp` и объявляется об использовании пространства имен `cv::ocl`.

Для получения информации об установленном SDK вызывается метод `getPlatfomsInfo(info)`. Эта информация сохраняется в векторе `vector<ocl::PlatformInfo> info`, а для получения доступа к ней мы пишем `PlatformInfo sdk = info.at(0)`. Затем выводятся основные характеристики SDK: название, поставщик, номер версии и количество совместимых с OpenCL вычислительных устройств.

Наконец, обращаясь к методу `sdk.getDevice(device, i)`, мы получаем информацию о каждом вычислительном устройстве: ид и название поставщика, версию драйвера, размер глобальной памяти, размер кэша памяти и т. д.

Ваша первая программа для GPU

Ниже показаны два варианта одной программы: в одном все вычисления производятся только на ЦП, а во втором задействован также GPU (при посредстве OpenCL). Варианты называются соответственно **calculateEdgesCPU** и **calculateEdgesGPU**.

Сначала – программа **calculateEdgesCPU**.

```
#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;
}
```

Затем – **calculateEdgesGPU**.

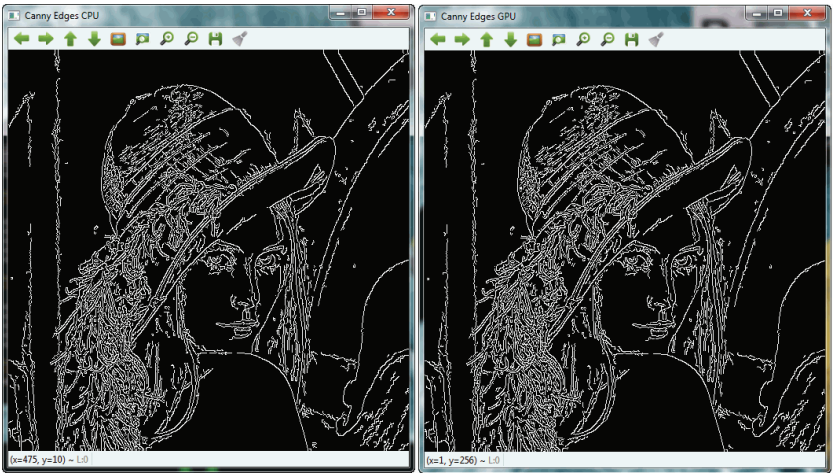
```
#include "opencv2/opencv.hpp"
#include "opencv2/core/ocl.hpp"

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;
}
```

Пояснения к коду

Как видно из рисунка ниже, результаты обеих программ одинаковы. Обе читают изображение, заданное в командной строке, преобразуют его в полутоновое и применяют гауссово размытие и оператор Кэнни.

Во втором примере выделены места, в которых программа для GPU отличается. Во-первых, необходимо активировать OpenCL, вызвав функцию `setUseOpenCL(true)`. Во-вторых, для выделения памяти на GPU используется тип данных **UMat (Unified Mat)**; так выделена память для переменных `gpuFrame`, `gpuBW`, `gpuBlur`, `gpuEdges`. В-третьих, входное изображение копируется из оперативной памяти

в память GPU с помощью функции `cpuFrame.copyTo(gpuFrame)`. Теперь, если используется функция, для которой имеется реализация на OpenCL, то она будет выполнена на GPU. Те же функции, которые не реализованы на OpenCL, будут выполняться на ЦП, как обычно. Как показывают замеры, этот пример на GPU выполняется в 10 раз быстрее.



Результате выполнения обеих программ

А теперь в реальном времени

Одно из основных преимуществ GPU – значительное ускорение вычислений. Это позволяет исполнять вычислительно сложные алгоритмы в приложениях реального времени: стереозрение, обнаружение пешеходов, вычисление оптического потока, распознавание лиц. Приведенная ниже программа **detectFaces** обнаруживает лица в потоке данных, поступающем от видеокамеры. Заодно программа позволяет выбрать режим работы – на ЦП или на GPU – и сравнить время вычислений.

Среди примеров в дистрибутиве OpenCV есть детектор лиц (`[opencv_source_code]/samples/cpp/facedetect.cpp`). В проекте `detectFaces.pro` используются следующие библиотеки: `-lopencv_core300`, `-lopencv_imgproc300`, `-lopencv_highgui300`, `-lopencv_videoio300`, `-lopencv_objdetct300`.

В программе **detectFaces** используется модуль `ocl`:

```
#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 - Задаем начальные параметры
// Вектор для хранения лиц
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 - Загружаем xml-файл для использования классификатора
if (!face_cascade.load(face_cascade_name))
{
  cout << "Ошибка при загрузке xml-файла!" << endl;
  return -1;
}

namedWindow("Video Capture");

// 3 - Устанавливаем режим работы: CPU или GPU
if (argc < 2)
{
  cout << "./detectFaces [CPU/GPU | C/G]" << endl;
  cout << "Попытаюсь использовать GPU..." << endl;
  setUseOpenCL(true);
}
else
{
  cout << "./detectFaces пытается использовать " << argv[1] << endl;
  if(argv[1][0] == 'C')
    // Попытка использовать ЦП для обработки
    setUseOpenCL(false);
```

```
  else
    // Попытка использовать GPU для обработки
    setUseOpenCL(true);
}

Rect r;
double start_time, finish_time, start_total_time, finish_total_time;
int counter = 0;

// 4 - Распознавание лиц на каждом захваченном кадре
start_total_time = getTickCount();
while (!finish)
{
  start_time = getTickCount();
  cap >> frame;
  if (frame.empty())
  {
     cout << "Больше нет кадров --> конец" << endl;
     break;
  }

  cvtColor(frame, frameGray, COLOR_BGR2GRAY);
  equalizeHist(frameGray,frameGray);

  // Распознать лица
  face_cascade.detectMultiScale(frameGray, faces, scale_factor,
     min_neighbours, 0|CASCADE_SCALE_IMAGE, Size(face_size,face_size));

  // Для каждого распознанного лица
  for (int f = 0; f < faces.size(); f++)
  {
     r = faces[f];
     // Обводим лицо прямоугольником
     rectangle(frame, Point(r.x, r.y), Point(r.x + r.width,
        r.y + r.height), Scalar(0,255,0), 3);
  }

  // Показываем результаты
  imshow("Video Capture",frame);

  // Вычисляем время работы
  finish_time = getTickCount();
  cout << "Время обработки кадра: "
       << (finish_time - start_time)/getTickFrequency() << " с" << endl;
  counter++;

  // Нажать Esc для завершения
  if(waitKey(1) == 27) finish = true;
}

finish_total_time = getTickCount();
```

```
    cout << "Среднее время на один кадр: "
         << ((finish_total_time - start_total_time)
             /getTickFrequency())/counter << " c" << endl;

    return 0;
}
```

Пояснения к коду

На первом шаге задаются параметры, в частности xml-файл (`String face_cascade_name argv[2]`), содержащий данные классификатора для распознавания лиц, минимальный размер распознанного лица (`face_size=30`), масштабный коэффициент (`scale_factor = 1.1`) и минимальное число соседей (`min_neighbours = 2`), определяющее компромисс между ложноположительными и ложноотрицательными результатами распознавания. Для возможности использования GPU нужно только изменить тип данных (`UMat frame, frameGray`).

В папке `[opencv_source_code]/data/haarcascades/` есть также xml-файлы для распознавания других частей тела: глаз, нижних конечностей, улыбок и т. д.

На втором шаге создается детектор с использованием заданного ранее xml-файла. Он основан на классификаторе Хаара по признакам – эффективном алгоритме распознавания объектов, предложенном Полом Виолой и Майклом Джонсом. Этот классификатор очень точно распознает лица. На этом шаге xml-файл загружается методом `face_cascade.load( face_cascade_name)`.

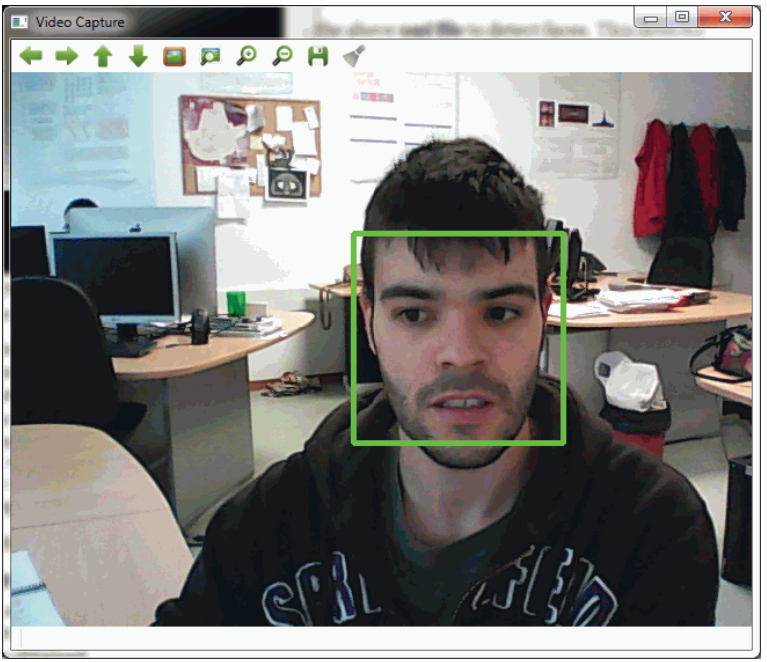
Подробные сведения об алгоритме Виолы и Джонса можно найти в статье `https://ru.wikipedia.org/wiki/Метод_Виолы_—_Джонса`.

На третьем шаге выбирается режим работы: на ЦП или на GPU (`setUseOpenCL(false)` или `setUseOpenCL(true)`). Режим задается с помощью аргумента командной строки `argv[1]`, например:

```
<bin_dir>/detectFaces CPU pathClassifier
<bin_dir>/detectFaces GPU pathClassifier
```

Если этот аргумент не задан, то по умолчанию подразумевается GPU.

Четвертый шаг – распознавание лиц на каждом кадре, захваченном видеокамерой. Но предварительно производится преобразование изображения в полутоновое (`cvtColor(frame, frameGray, COLOR_BGR2GRAY)`) и выравнивание гистограммы (`equalizeHist(frameGray, frameGray)`). Затем вызывается метод `face_cascade.detectMultiScale(frameGray, faces, scale_factor, min_neighbours, 0|CASCADE_SCALE_IMAGE, Size(face_size,face_size))` созданного детектора лиц, который ищет лица в текущем кадре. И наконец, каждое найденное лицо обводится зеленым прямоугольником. На рисунке ниже показан результат работы программы.



Результат работы программы распознавания лиц

Производительность

В этой программы мы вычисляли время работы обоих вариантов и печатали среднее время обработки одного кадра.

Производительность – важнейшее преимущество использования GPU. Поэтому мы и замеряли время, чтобы оценить ускорение по сравнению с вычислениями на ЦП. Текущее время запоминается в

начале программы путем вызова функции `getTickCount()`. А в конце та же функция вызывается еще раз. Кроме того, мы запоминаем счетчик кадров, что позволяет вычислить среднее время обработки одного кадра. В этой программе среднее время обработки кадра на GPU было равно 0.057 с (17.5 кадров/с), а на ЦП 0.335 с (2.9 кадра/с). Таким образом, получено *ускорение в 6 раз*. Это отличный результат, особенно если учесть, что нам пришлось изменить всего несколько строк кода. Однако есть возможность добиться значительно большего ускорения, но для этого нужно точнее представлять себе задачу и опуститься на уровень проектирования ядер.

Резюме

В этой главе мы узнали, как установить OpenCV с поддержкой OpenCL и разрабатывать приложения, исполняемые на вычислительных устройствах, совместимых с OpenCL. Для этого необходима последняя версия OpenCV.

Мы объяснили, что такое OpenCL и какие существуют SDK. Напомним, что для правильной работы с OpenCL необходим SDK, предназначенный для конкретного типа устройств. Во втором разделе была описана процедура установки OpenCV с поддержкой OpenCL, и далее мы использовали AMD APP SDK. В последнем разделе было приведено три примера программирования GPU (второй пример был для сравнения написан в двух вариантах: для ЦП и для GPU). И кроме того, мы провели сравнение производительности программы, выполняемой на ЦП и на GPU. Оказалось, что версия для GPU работает в шесть раз быстрее.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: **тел. +7 (499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Луис Педро Коэльо, Вилли Ричарт

Обработка изображений с помощью OpenCV

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 18,50.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф