

Перевод книги Learning OpenCV



Содержание

Введение	0
Обзор	1
Что такое OpenCV?	1.1
Кто использует OpenCV?	1.2
Что такое компьютерное зрение?	1.3
История появления OpenCV	1.4
Загрузка и установка OpenCV	1.5
Получение последней версии OpenCV через CVS	1.6
Документация по OpenCV	1.7
Структура и содержание OpenCV	1.8
Переносимость	1.9
Упражнения	1.10
Введение в OpenCV	2
Приступая к работе	2.1
Первая программа - Отображение картинки	2.2
Вторая программа - AVI видео	2.3
Создание ползунка	2.4
Простые преобразования	2.5
Не столь простые преобразования	2.6
Захват видео с камеры	2.7
Запись видео в формате AVI	2.8
Резюме	2.9
Упражнения	2.10
Знакомство с OpenCV	3
Примитивные типы данных в OpenCV	3.1
Структура CvMat	3.2
Структура IplImage	3.3
Операции над матрицами и изображениями	3.4
Рисование	3.5
Сохранность данных	3.6

IPP - Integrated Performance Primitives	3.7
Резюме	3.8
Упражнения	3.9
HighGUI	4
Набор графических инструментов	4.1
Создание окна	4.2
Загрузка изображения	4.3
Отображение изображения	4.4
Работа с видео	4.5
Функция cvConvertImage()	4.6
Упражнения	4.7
Обработка изображений	5
Краткий обзор	5.1
Сглаживание	5.2
Морфологические преобразования	5.3
Заливка	5.4
Изменение размера	5.5
Пирамиды изображения	5.6
Пороговое преобразование	5.7
Упражнения	5.8
Преобразования изображений	6
Краткий обзор	6.1
Свертка	6.2
Градиенты и оператор Собеля	6.3
Лапласиан	6.4
Canny	6.5
Преобразования Хафа	6.6
Remap	6.7
Растягивание, сжатие, деформация и поворот	6.8
CartToPolar и PolarToCart	6.9
LogPolar	6.10
Дискретное преобразование Фурье	6.11
Дискретно косинусное преобразование	6.12
Интегральное изображение	6.13

Дистанционные преобразования	6.14
Коррекция гистограмм	6.15
Упражнения	6.16
Гистограммы и сопоставления	7
Базовый тип Histogram	7.1
Доступ к Histogram	7.2
Базовые манипуляции над гистограммами	7.3
Более сложные манипуляции	7.4
Упражнения	7.5
Контуры	8
Хранилище памяти	8.1
Последовательности	8.2
Поиск контура	8.3
Примеры поиска контура	8.4
Другие манипуляции с контурами	8.5
Сопоставление контуров	8.6
Упражнения	8.7
Составные части и сегментация изображения	9
Составные части и сегментация	9.1
Вычитание фона	9.2
Алгоритм водораздела	9.3
Восстановление изображения	9.4
Mean-Shift сегментация	9.5
Триангуляция Delaunay, тесселяция Voronoi	9.6
Упражнения	9.7
Слежение и движение	10
Основы слежения	10.1
Поиск углов	10.2
Углы субпикселя	10.3
Инвариантность особенностей	10.4
Оптические потоки	10.5
Mean-Shift и Camshift слежение	10.6
Шаблоны движения	10.7

Оценочная функция	10.8
Алгоритм условно плотного распространения	10.9
Упражнения	10.10
Модели камер и их калибровка	11
Модель камеры	11.1
Калибровка	11.2
Удаление искажений	11.3
Пример реализации калибровки	11.4
Преобразования Rodrigues	11.5
Упражнения	11.6
Проекция и 3D Vision	12
Проекции	12.1
Аффинные и перспективные преобразования	12.2
POSIT: оценивание трехмерного представления	12.3
Стерео зрение	12.4
Структура движения	12.5
Установка линий в 2-ух и 3-х мерных измерениях	12.6
Упражнения	12.7
Машинное обучение	13
Что такое машинное обучение	13.1
Универсальные методы библиотеки ML	13.2
Расстояние Mahalanobis	13.3
K-means	13.4
Naïve/Normal Bayes Classifier	13.5
Binary Decision Trees	13.6
Boosting	13.7
RandomTrees	13.8
Распознавание лиц или классификатор Хаара	13.9
Другие алгоритмы машинного обучения	13.10
Упражнения	13.11
Будущее OpenCV	14
Прошлое и будущее	14.1
Направления	14.2
OpenCV для разработчиков	14.3

Copyright © 2008 Gary Bradski and Adrian Kaehler

Исключительное право на произведение "Learning OpenCV" принадлежит авторам книги **Gary Bradski и Adrian Kaehler**, 2008.

Оригинал книги: <http://shop.oreilly.com/product/9780596516130.do>

От авторов перевода

Обращаем ваше внимание на то, что данный материал не подлежит распространению с целью получения любой финансовой или материальной выгоды и носит исключительно ознакомительный характер

Добро пожаловать на проект, посвященный переводу книги Learning OpenCV (1-е издание). На данный момент проект имеет статус **любительский**, т.к. перевод выполнялся группой людей, чья деятельность не связана с выполнением качественного профессионального перевода. Поэтому не стоит считать данный материал качественным в плане трактовки некоторых языковых конструкций и специфичных терминов.

Обзор

[П]||[РС]||(РП) Что такое OpenCV

OpenCV - библиотека компьютерного зрения с открытым исходным кодом (<http://opensource.org>), доступная по адресу <http://SourceForge.net/projects/opencvlibrary>. Библиотека написана на С и С++ и работает на компьютерах под управлением Linux, Windows, Mac OS X. Так же активно развиваются интерфейсы библиотеки для Python, Ruby, Matlab и других языков программирования.

Библиотека OpenCV была разработана с целью повышения вычислительной эффективности и с уклоном на приложения реального времени. OpenCV написана с использованием оптимизированного С и может использовать многоядерные процессоры. В дальнейшем, если потребуется автоматическая оптимизация на аппаратных платформах Intel, существует возможность покупки библиотеки IPP (Integrated Perfomance Primitives), которая состоит из процедур с низкоуровневой оптимизацией для различных алгоритмических областей. OpenCV будет автоматически использовать библиотеку IPP во время выполнения программы.

Одной из основных целей OpenCV является предоставление простого в использовании интерфейса, который позволит людям довольно таки быстро строить сложные приложения, использующие компьютерное зрение. Библиотека OpenCV содержит более 500 функций, которые охватывают многие области компьютерного зрения, такие как: инспекция фабричной продукции, медицина, безопасность, пользовательский интерфейс, калибровка камеры, стереозрение и робототехника. И все это благодаря тому, что компьютерное зрение и машинное обучение часто идут "рука об руку", к тому же OpenCV полностью включает в себя библиотеку общего назначения MLL (Machine Learning Library). MLL библиотека ориентирована на распознавание статических образов и кластеризацию. MLL очень полезна для задач компьютерного зрения, которые составляют основу OpenCV, но она довольно-таки обобщенная, чтобы решать конкретные проблемы машинного обучения.

[П]||[PC]||(РП) Кто использует OpenCV

Большинство ученых и практиков из IT осознают роль, которую играет компьютерное зрение, но мало кто знает обо всех областях, в которых оно используется. К примеру, большинство людей знает, что компьютерное зрение используется в системах наблюдения, а также при работе с изображениями и видео в интернете. Вместе с тем, мало кому доводилось видеть, как компьютерное зрение используется в компьютерных играх. Мало кто знает, что аэросъемка и уличные карты (например, сервис Google's Street View) широко используют калибровку камеры и технологию сшивания изображений. Немногие знают какое место занимают подобного рода приложения в таких областях, как безопасность, конструирование беспилотников и в биомедицинских анализах. Немногие знают, как широко стало распространение компьютерного зрения и в производстве: практически все массовое производство в какой-то определенной момент технологического процесса задействует компьютерное зрение.

Лицензия OpenCV была структурирована таким образом, что позволяет создавать коммерческий продукт используя функционал OpenCV частично или полностью. Вы не обязаны делать ваш продукт open-source или же делиться вашими улучшениями для OpenCV, однако мы будем рады вашим улучшениям. Отчасти из-за этих либеральных условий лицензирования, появилось большое сообщество, состоящее из людей из различных крупных компаний (IBM, Microsoft, Intel, SONY, Siemens, Google) и научно-исследовательских центров (Stanford, MIT, CMU, Cambridge, INRIA). На [Yahoo](#) есть группа, где пользователи могут задавать и обсуждать различные вопросы, связанные с OpenCV; в ней уже 20.000 участников. OpenCV популярна по всему миру, с наибольшими сообществами в Китае, Японии, России, Европе и Израиле.

С момента выхода альфа-версии в январе 1999 года, OpenCV была использована во многих приложениях и научно-исследовательских работах: сшивка изображений спутниковых карт, выравнивание отсканированных изображений, уменьшение шума на медицинских снимках, анализ объектов, системы обнаружения вторжения, автоматический мониторинг, системы контроля, калибровка камеры, приложения для военных, беспилотники, наземные и подводные аппараты. Библиотека так же была задействована для распознавания звуков и музыки, при помощи анализа спектрограмм. OpenCV была ключевой частью робота "Stanley" из Стенфорда, который выиграл 2 миллиона долларов в пустынной гонке роботов.

[П]||[РС]||(РП) Что такое компьютерное зрение

Компьютерное зрение - процесс преобразования данных, полученных с фотоаппаратов и видеокамер, в новое представление. Все эти преобразования выполняются для достижения какой-то конкретной цели. Входные данные могут содержать некоторую контекстную информацию, такую как "камера установлена в машине" или "датчик глубины определил объект в радиусе 1 метра". Решением может быть "есть человек в этой сцене" или "есть 14 опухолевых клеток на снимке". Новым представлением может быть процесс превращения цветного изображения в черно-белое или устранение эффекта движения камеры из последовательности изображений.

Не стоит себя обманывать и думать, что задачи компьютерного зрения такие уж и легкие. Как труден может быть поиск, скажем, автомобиля, когда имеется только его образ? Интуиция может быть весьма обманчивой. Человеческий мозг разделяет сигнал поступивший от зрения на множество каналов, которые впоследствии передают различного вида информацию в ваш мозг. Мозг устроен таким образом, что внимание концентрируется только на важных участках изображения, исключая при этом остальные. Существует масса ответных сигналов, перемещающиеся в зрительном канале, которые пока плохо изучены. Сигналы, поступающие на ассоциативные входы, пришедшие от датчиков контроля мышц и всех других чувств, позволяют мозгу опираться на перекрестные ассоциации, накопленные за годы прожитых лет. За счет обратной связи в мозге, процесс повторяется вновь и вновь и включает в себя датчик (глаз), который механически управляет освещением с помощью радужной оболочки и настаивает прием на поверхности сетчатки.

Однако, в системе компьютерного зрения все что получает компьютер это сетку с числами от камеры или с диска. По большей части, нет не встроенного распознавания образов, не автоматического управления фокусом и диафрагмой, не перекрестных ассоциаций с многолетним опытом работы. По большей части, системы компьютерного зрения все еще довольно примитивны. На рисунке 1-1 изображен автомобиль.

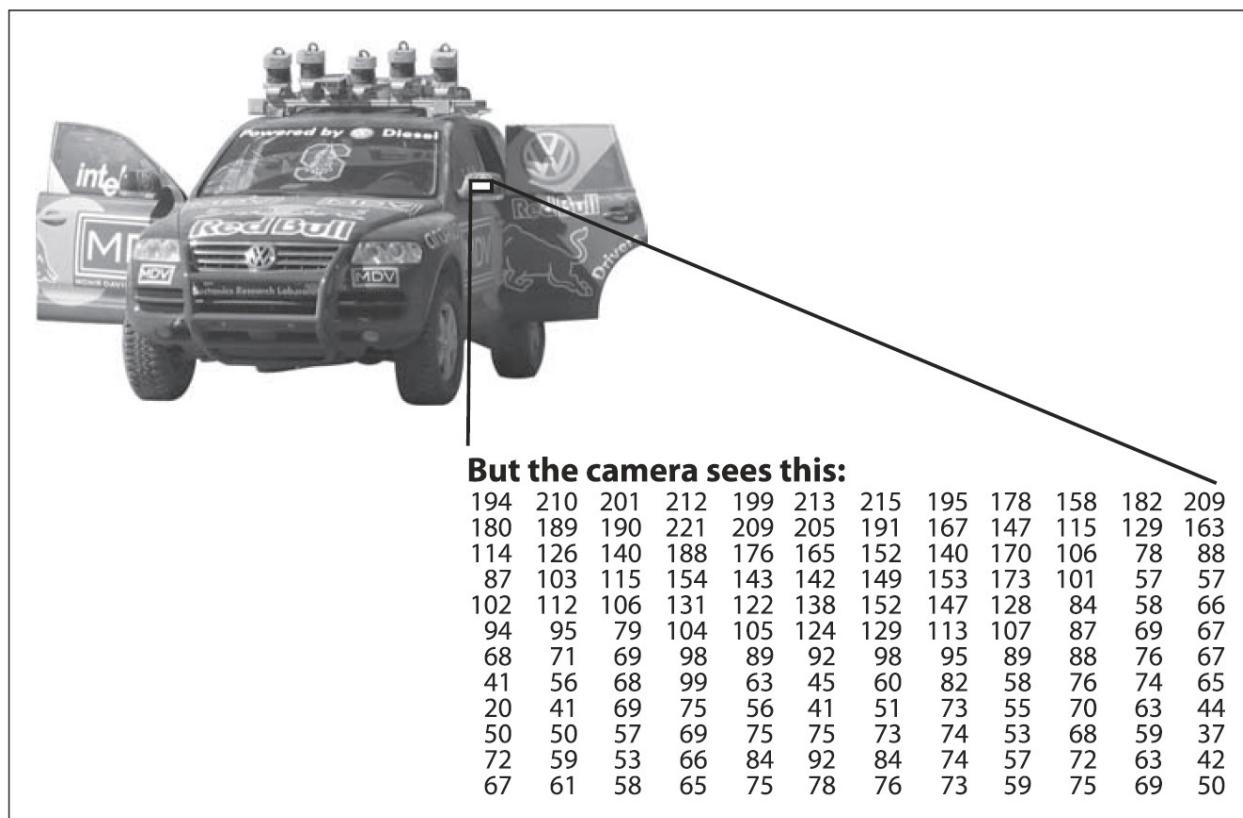


Рисунок 1-1. Компьютер видит боковое зеркало авто просто как сетку чисел

На этом рисунке человек видит боковое зеркало со стороны водителя. Компьютер же "видит" только сетку с числами. Любой номер из этой сетки имеет довольно таки большую шумовую составляющую и сам по себе дает нам мало информации, однако, эта сетка чисел это все что "видит" компьютер. Наша задача сводиться к преобразованию сетки чисел к виду: "боковое зеркало". Рисунок 1-2 дает нам чуть больше представления о том, почему компьютерное зрение это так трудно.

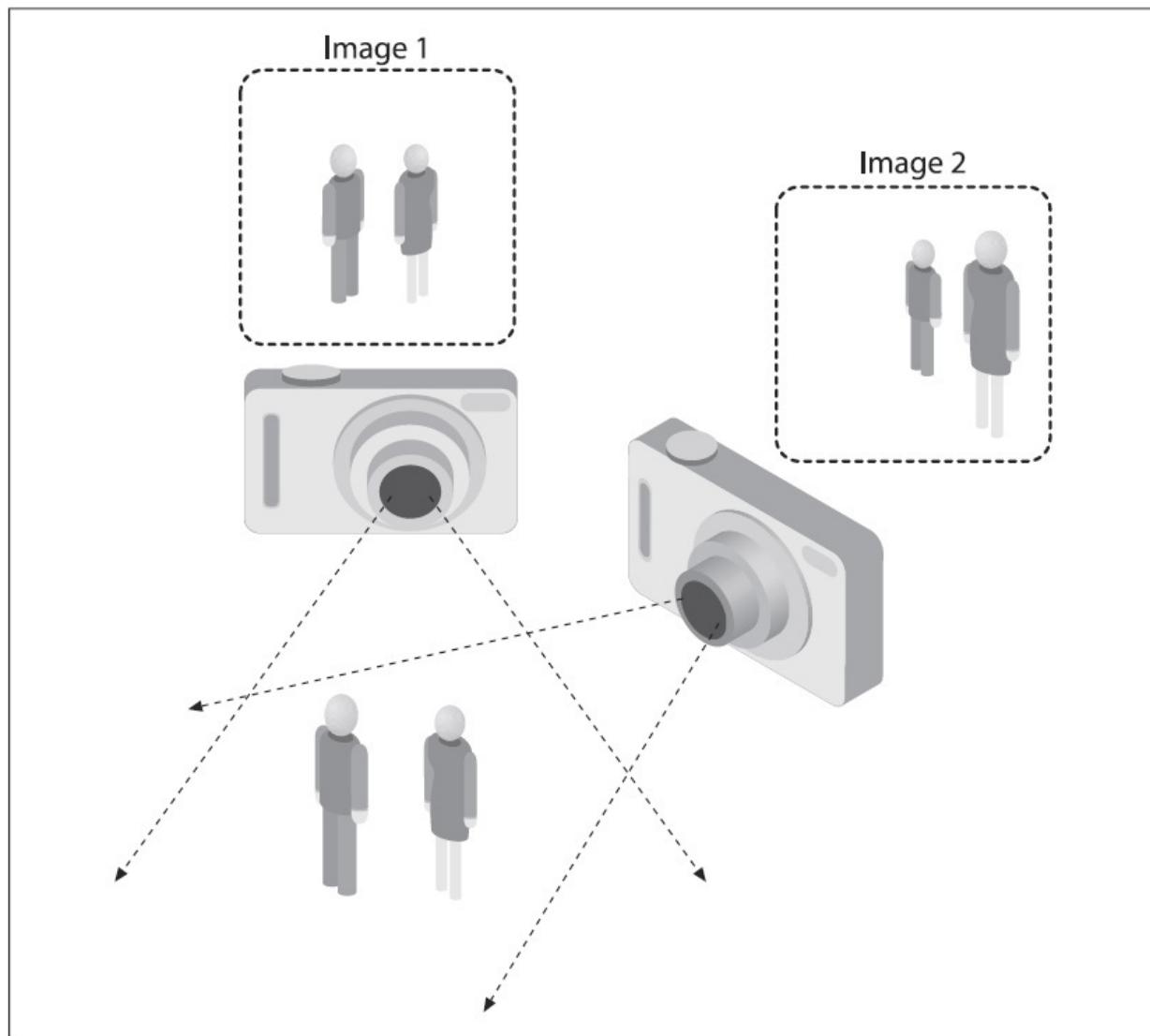


Рисунок 1-2. Некорректная природа зрения: двумерное представление объекта может радикально измениться в зависимости от позиции, с которой на него посмотреть

На самом деле, проблема, о которой шла речь ранее, формально невозможно решить. Имея только двумерное представление трехмерного мира, невозможно однозначно восстановить 3D сигнал. Формально, такого рода некорректная задача не имеет однозначного или окончательного решения. Двумерное изображение можно представить любым из бесконечных сочетаний 3D-сцен, даже если данные были совершены. К тому же, как упоминалось ранее, данные подвержены шуму и искажениям. Такого рода повреждения связаны с изменениями в мире (погода, освещение, отражения, перемещения), недостатками в объективе и механических установках, конечным временем интеграции на датчике (размытие при движении), электрическими шумами в датчике или другой электронике, артефактами сжатия после захвата изображения. Учитывая все эти проблемы, как мы можем добиться хоть какого-то прогресса?

При проектировании практической системы, дополнительные знания контекста зачастую могут быть использованы, чтобы обойти ограничения, накладываемые визуальными датчиками. Рассмотрим следующий пример: мобильный робот должен найти и подобрать степлер. Робот может использовать следующий факт: стол — это объект, который находится внутри офиса и чаще всего степлер можно найти на столах. Это дает неявную ссылку о размере; степлер должен поместиться на столе. Это так же помогает устраниить ложные "распознавания" расположения степлера в невозможных местах (например, на потолке или окне). Робот может проигнорировать 200-футовый рекламный дирижабль с рекламой о степлере, потому что дирижаблю не хватает на фоне древесины стола. В иных случаях, когда приходиться извлекать изображения степлера из базы данных, может оказаться так, что на этих изображениях степлер будет огромных размеров или же иметь необычную форму, что повлечет за собой исключение этих изображений из рассмотрения. То есть фотограф, вероятно, взял только изображения степлеров реальных размеров. Люди также имеют тенденцию снимать объекты так, чтобы они были в центре изображения, и были повёрнуты «по-обычному» - скажем, вряд ли кто-то станет фотографировать человека вверх ногами. Таким образом, фотографии, сделанные людьми, содержат немного неявной информации.

Контекстная информация также может быть смоделирована явно методами машинного обучения. Скрытые переменные, такие как размер, ориентация притяжения и т.д. можно соотнести с их значениями в маркированном обучающем множестве. В альтернативе, можно попытаться измерить скрытые переменные смещения с помощью дополнительных датчиков. Используя датчик глубины, можно точно измерить размер объекта.

Следующая проблема компьютерного зрения — это шум. Как правило, мы имеем дело с шумом, когда используем статистические методы. Например, может быть невозможно обнаружить контур в изображении простым сравнением соседних точек. Но если мы соберем статистику по локальной области, то задача обнаружения контура становится легче. Контур должен появиться в виде строки непосредственных ответов по локальной области, каждый из которых ориентирован в соответствии с его соседями. Так же возможно компенсировать шум, принимая статистические данные в течение долгого времени. Тем не менее иные методы учета шума или искажения путем создания четких моделей позволяют распознавать непосредственно из имеющихся данных. Например, природа искажений от объектива хорошо известна, и чтобы почти полностью исправить такие искажения, необходимо знать только параметры простой полиномиальной модели.

Те или иные действия или решения, принимаемые в компьютерном зрении, сделаны на основе данных полученных с камеры в контексте конкретной цели или задачи. Можно удалить шум или повреждения в изображении так, что система безопасности

будет выдавать предупреждения, если кто-то попытается залезть на забор или же система мониторинга будет подсчитывать, сколько людей переступило через определенную область в парке аттракционов. ПО с использованием компьютерного зрения для роботов, которые передвигаются по офису будут использовать иные стратегии, нежели стационарные камеры видеонаблюдения, поскольку эти две системы имеют существенно различные контексты и задачи. Как правило, контекст задач компьютерного зрения ограничен и потому, рассчитывая на эти ограничения, задачу можно упростить и, как следствие, конечное решение будет более надежным.

OpenCV направлена на обеспечение основных инструментов, необходимых для решения проблем компьютерного зрения. В некоторых случаях, высоко функциональной библиотеки будет достаточно, чтобы решить более сложные проблемы в области компьютерного зрения. Даже если это не так, то основные компоненты библиотеки являются достаточно полными, чтобы позволить создать новое решение самостоятельно, тем самым решить почти любую проблему компьютерного зрения. Есть несколько проверенных и надежных методов, которые используют многие компоненты библиотеки. Как правило, после создания проекта обнаруживаются слабые места и исправляются с помощью собственного кода и умений (более известно, как "решить проблему которая фактически имеется, а не ту, которую вы вообразили"). Тогда можно использовать данное решение как эталон для оценки улучшений, которые будут сделаны в дальнейшем. С этого момента любые слабые места могут быть решены за счет встроенного решения.

[П]|[РС]|(РП) История появления OpenCV

OpenCV появился по инициативе Intel Research для продвижения ресурсоемких приложений. С этой целью, Intel запустил ряд проектов, в числе которых были: трассировка лучей в реальном времени и 3D отображение. Один из авторов, работник Intel, будучи гостем университета, заметил, что некоторые ведущие университеты, такие как MIT Media Lab, имели на тот момент хорошо развитую и внутренне открытую инфраструктуру по развитию компьютерного зрения и код, который "кочевал" от студента к студенту, тем самым давая каждому последующему студенту бесценный опыт для развития их собственных проектов компьютерного зрения. Так, новому студенту не нужно было изобретать основные функции самому, а использовать те, что были написаны до него.

Таким образом, задумка OpenCV — это способ сделать компьютерное зрение общедоступным. При содействии Intel Performance Library Team, задача реализации кода ядра и алгоритмических спецификаций OpenCV была направлена команде Intel из **РОССИИ**. Все начиналось в исследовательской лаборатории корпорации Intel при использовании ПО Performance Libraries наряду с опытом из **РОССИИ**.

Главным в российской команде был **Вадим Писаревский**, которому удалось написать и оптимизировать большую часть OpenCV и который до сих пор прикладывает немало усилий для развития OpenCV. Вместе с ним в разработке ранней инфраструктуры участвовал **Виктор Ерухимов**, а также управляющий российской лабораторией Валерий Kuriakin. В начале перед OpenCV ставились несколько задач:

- Способствовать исследованиям в области компьютерного зрения путем разработки не только открытого, но и тщательно оптимизированного кода. Теперь никто больше не должен изобретать велосипед.
- Распространять знания через общую инфраструктуру, что бы разработчики могли строить приложения с более читаемым и легко передаваемым кодом.
- Продвигать коммерческие приложения, базирующихся на компьютерном зрении и собираемые с общедоступным производительно-оптимизированным кодом - лицензия не требует от коммерческих приложений, что бы они были открытыми и общедоступными.

Эти цели отвечают на вопрос "почему" OpenCV. Включение компьютерного зрения в приложения приводило к увеличению востребованности в быстрых процессорах. Обновление драйверов более быстрых процессоров стало приносить Intel больше доходов, нежели продажа дополнительного ПО. Возможно, поэтому открытый и

свободный код возник у поставщика оборудования, а не у компаний производителей дополнительного ПО. В некотором смысле, аппаратные компании менее ограничены в создании инновационного ПО.

В любом open source проекте важно достичь критической массы, при которой проект станет само поддерживаемым. На данный момент библиотека OpenCV скачена около двух миллионов раз, это число растет и в среднем составляет 26 000 загрузок в месяц. Число активно использующих пользователей приближается к 20 000. Пользователи все больше и больше вносят свой вклад в развитие OpenCV и центр развития в значительной степени вышел за пределы Intel. Путь развития OpenCV показан на рисунке 1-3.

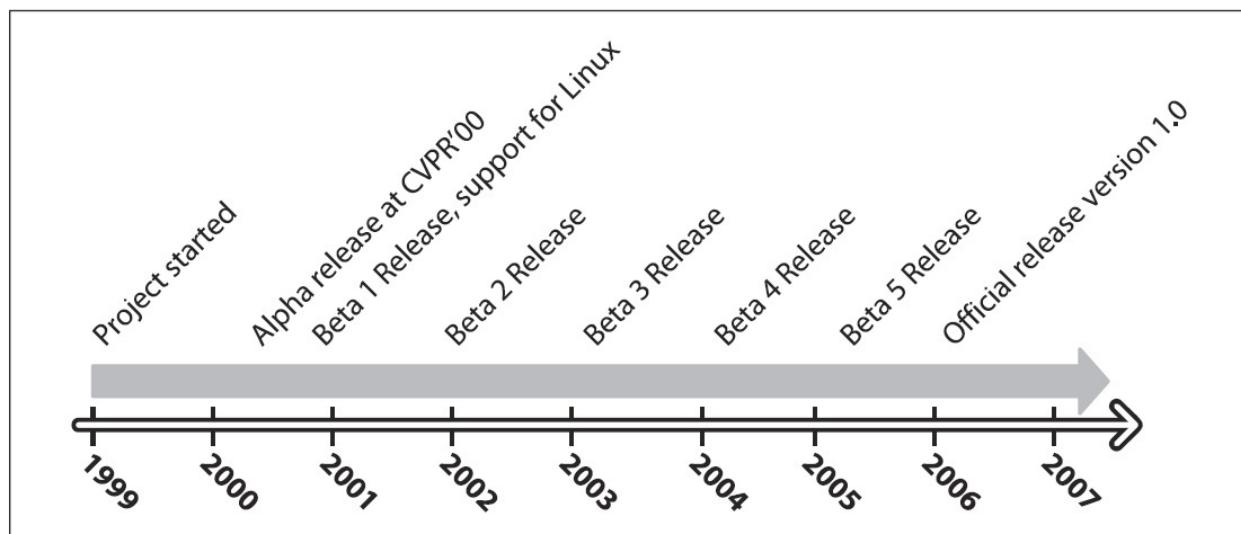


Рисунок 1-3. Путь развития OpenCV

За это время, OpenCV пострадала от бума и спада доткомов (компании, чья бизнес-модель целиком основывается на работе в рамках сети Интернет), а также от многочисленных изменений в руководстве. Во время этих колебаний, были времена, когда работники Intel не принимали участия в развитии библиотеки вообще. Тем не менее, с появлением многоядерных процессоров и множества новых приложений компьютерного зрения, значение OpenCV стало расти. Сегодня OpenCV активно развивается в ряде институтов, так что в скором времени ожидается множество новых обновлений, связанных с: калибровкой сразу нескольких камер, восприятием глубины сцены, методами смешивания зрения с датчиками глубины, лучшим распознаванием образов и еще большей поддержкой для роботов. Для получения дополнительной информации о будущем OpenCV, обратитесь к главе 14.

Ускорение OpenCV с помощью IPP

Поскольку "пристанищем" OpenCV была команда Intel Performance Primitives и основные разработчики были друзьями этой команды, OpenCV имеет функцию ручной настройки на использование высоко оптимизированного кода IPP для само ускорения. Ускорение от использования IPP может быть существенным. На рисунке 1-4 представлено сравнение OpenCV и OpenCV с IPP с двумя другими библиотеками компьютерного зрения LTI и VXL. Обратите внимание, что производительность была ключевой задачей OpenCV; библиотека требовалась для запуска кода с использованием зрения в режиме реального времени.

OpenCV написана при помощи высокопроизводительного кода на С и С++. Но это никоем образом не связано с IPP. OpenCV автоматически подключает IPP для повышения производительности.

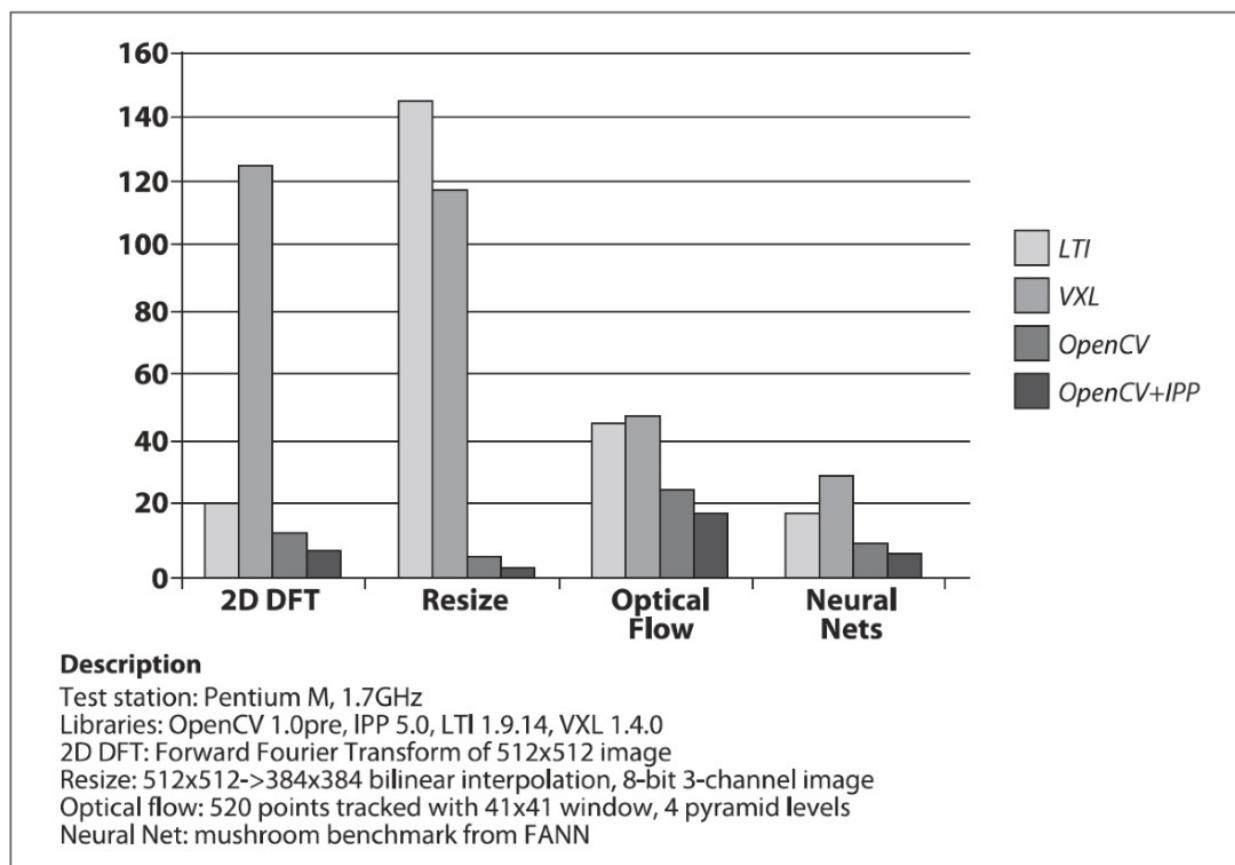


Рисунок 1-4. Сравнение двух других библиотек компьютерного зрения (LTI и VXL) с OpenCV (с и без IPP) по четырем критериям эффективности: четыре полосы для каждого теста отражают результаты, пропорциональные времени выполнения для каждой из указанных библиотек; во всех случаях, OpenCV превосходит другие библиотеки, а OpenCV с IPP превосходит OpenCV без IPP

Кто владелец OpenCV?

Хотя Intel и начал разработку OpenCV, библиотеку всегда можно было использовать в коммерческих и научно-исследовательских работах. Поэтому она является открытой и бесплатной, а сам код может быть использован или встроен (частично или полностью) в другие приложения, хоть коммерческие, хоть научно-исследовательские. При этом не обязательно делать код вашего приложения открытым и общедоступным. Не требуется вносить ваши улучшения в библиотеку, хотя мы надеемся, что вы будете это делать.

[П]|[РС]|(РП) Загрузка и установка OpenCV

Основным сайтом OpenCV является [SourceForge](#) и [Wiki страница](#). Installer для Linux - opencv-1.0.0.tar.gz; для Windows - OpenCV_1.0.exe. Самую актуальную версию всегда можно скачать с CVS сервера SourceForge.

Установка

После загрузки библиотеки, потребуется её установить. Подробная инструкция по установке на Linux или Mac OS, всегда можно найти в файле INSTALL в каталоге .../opencv/; этот файл также описывает процесс сбора и запуск процедур тестирования OpenCV. В файле INSTALL также перечислены дополнительные программы, которые потребуются для сбора версии OpenCV для разработчиков, например, autoconf, automake, libtool, и swig.

Windows

Для начала необходимо скачать и запустить инсталлятор с SourceForge. Он установит OpenCV, зарегистрирует фильтры DirectShow и выполнит некоторые постинсталляционные процедуры. После этого все готово для использования OpenCV. Помимо этого, всегда можно посетить директорию .../opencv/_make и открыть opencv.sln с MSVC++ или MSVC.NET 2005, или открыть opencv.dsw для более старой версии MSVC++ и собрать или каким-то образом изменить библиотеку.

Для оптимизации производительности при помощи коммерческой библиотеки IPP для Windows, необходимо приобрести установщик с сайта [Intel](#). После установки убедитесь, что папка bin (например, c:/program files/intel/ipp/5.1/ia32/bin) находится по указанному пути. Теперь IPP будет автоматически распознаваться OpenCV и подгружаться во время выполнения (более подробно об этом пойдет речь в главе 3).

Linux

Бинарники для Linux не включены в версию OpenCV из-за большого разнообразия версий GCC и GLIBC для различных дистрибутивов (SuSE, Debian, Ubuntu и т.д.). Если в вашем дистрибутиве нет OpenCV, необходимо собрать её из исходников согласно файлу .../opencv/INSTALL.

Для сбора библиотеки и демок потребуется GTK+ 2.x или выше (включая заголовки). Так же потребуются pkgconfig, libpng, zlib, libjpeg, libtiff, и libjasper. Потребуется Python (включая заголовки) версии 2.3, 2.4 или 2.5 (пакет разработчика). Так же нужен

libavcodec и другие libav* библиотеки (включая заголовки) из ffmpeg 0.4.9-pre1 или более поздней версии (svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg).

Скачать ffmpeg можно по адресу <http://ffmpeg.mplayerhq.hu/download.html>. ffmpeg имеет лицензию GPL. Это позволяет использовать ffmpeg с ПО, у которого не GPL лицензия (например, OpenCV):

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

В итоге: /usr/local/lib/libavcodec.so., /usr/local/lib/libavformat.so., /usr/local/lib/libavutil.so., и заголовочные файлы /usr/local/include/libav.

Для построения OpenCV:

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

Путь по умолчанию для установленной библиотеки: /usr/local/lib/ и /usr/local/include/opencv/. После необходимо добавить /usr/local/lib/ в /etc/ld.so.conf (и запустить ldconfig) или в переменную окружения LD_LIBRARY_PATH.

Установка библиотеки IPP (оптимизация производительности) в Linux происходит тем же способом, что был описан ранее. Предположим, что библиотека установлена в /opt/intel/ipp/5.1/ia32/. В скрипте инициализации (.bashrc или подобный) необходимо добавить /bin/ и /bin/linux32 в LD_LIBRARY_PATH:

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

Или, построчно добавить /bin и /bin/linux32 в файл /etc/ld.so.conf и запустить команду ldconfig из под root-а (или использовать sudo).

Все готово к использованию. Теперь OpenCV сможет найти и использовать IPP. Для получения более подробной информации, обратитесь к файлу .../opencv/INSTALL.

MacOS X

На момент написания книги, присутствуют некоторые ограничения для MacOS X (например, запись AVI); данные ограничения описаны в файле .../opencv/INSTALL.

Сбор библиотеки под MacOS X аналогичен случаю для Linux, со следующими исключениями:

- По умолчанию, Carbon используется вместо GTK+
- По умолчанию, QuickTime используется вместо ffmpeg
- pkg-config не обязателен (используется только в скрипте samples/c/build_all.sh)
- RPM и Igconfig не поддерживаются по умолчанию. Используйте configure+make+sudo make install для сборки и установки OpenCV, обновите LD_LIBRARY_PATH (если ./configure --prefix=/usr не используется)

Для нормальной функциональности, необходимо доставить libpng, libtiff, libjpeg и libjasper от darwinports и/или fink и сделать их доступными для ./configure (см. ./configure --help). Для получения актуальной информации, посетите OpenCV Wiki по адресу <http://opencvlibrary.SourceForge.net>.

[П]||[PC]||(РП) Получение последней версии OpenCV через CVS

OpenCV активно развивается и ошибки довольно таки быстро правятся, благодаря предоставлению подробного отчета и кода, демонстрирующий ошибку. Однако, официальные релизы OpenCV выходят один или два раза в год. Если есть серьезный проект, то могут потребоваться исправления и обновления, как только они станут доступны. Для их получения, необходимо получить доступ к OpenCV's Concurrent Versions System (CVS) на SourceForge.

Данная книга не учит работе с CVS. Если вы уже работали над другими проектами с открытым исходным кодом, то вероятнее всего уже знакомы с CVS. Если же нет, то изучите труд *Essential CVS* от Jennifer Vesperman (O'Reilly). Командная строка CVS поставляется с Linux, OS X и в большинстве UNIX-подобных системах. Для пользователей Windows, рекомендуем использовать [TortoiseCVS](#), который прекрасно встраивается в Windows Explorer.

Для получения последней версии OpenCV из репозитория CVS, необходимо получить доступ к каталогу CVSROOT:

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

Для пользователей Linux необходимо ввести две команды:

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

Когда программа запросит пароль, нажмите возврат. Затем используйте следующее:

```
cvs -z3 -d:pserver:anonymous@opencvlibrary.cvs.sourceforge.net:/cvsroot/opencvlibraryco -
```

[П]|[РС]|(РП) Документация по OpenCV

Основной документацией по OpenCV является HTML документация, поставляемая вместе с исходным кодом. В дополнение к этому, в интернете существует OpenCV Wiki и устаревшая версия HTML документации.

Документация. Формат HTML

HTML документация поставляется вместе с OpenCV .../opencv/docs. Загрузите файл *index.html* и увидите следующие ссылки:

CXCORE

Структура данных, матричная алгебра, преобразования данных, хранение объектов, управление памятью, обработка ошибок, динамическая загрузка кода, а также рисование, текст и основы математики

CV

Обработка изображений, анализ структуры изображений, движение и слежение, распознавание образов и калибровка камеры

Machine Learning (ML)

Функции кластеризации, классификации и анализа данных

HighGUI

Графический интерфейс, хранилища изображений/видео и recall.

CVCAM

Работа с камерой

Haartraining

Обучение каскадного детектора (.../opencv/apps/HaarTraining/doc/haartraining.htm)

Директория .../opencv/docs также содержит файл оригинального руководства пользователя для OpenCV IPLMAN.pdf. Он существует и по сей день, но его следует использовать с осторожностью, хоть он и включает подробное описание алгоритмов и то, какие типы изображений могут быть использованы с тем или иным алгоритмом. Данную книгу же можно считать отправной точкой.

Документация. Формат Wiki

Документация в данном формате является более новой, чем HTML версия; к тому же в Wiki есть контент, которого нет в HTML. Адрес Wiki - <http://opencvlibrary.SourceForge.net>. Содержимое Wiki:

- Инструкция сборки OpenCV, используя Eclipse IDE.
- Распознавание лиц с OpenCV
- Библиотека видеонаблюдения
- Список литературы
- Совместимость камер
- Ссылки на китайские и корейские сообщества

Другой wiki-ресурс, расположенный по адресу

<http://opencvlibrary.SourceForge.net/CvAux> содержит уникальную информацию о вспомогательных функциях, о которых пойдет речь в разделе "Структура и содержимое OpenCV". CvAux включает в себя следующие функциональные зоны:

- Стерео соответствие
- Морфинг
- 3D слежение в режиме стерео
- Собственные значения объекта (PCA) для функций распознавания объектов
- Внедрение скрытых марковских моделей (СММ)

Данный ресурс переведен на китайский

<http://www.opencv.org.cn/index.php/%E9%A6%96%E9%A1%B5>.

Зачастую, независимо от источника документации, бывает трудно узнать:

- С каким типом изображения работает функция (floating, integer, byte; 1–3 channels)
- Какие и где функции уместно применять
- Детали более сложных функций (например, contours)
- Детали работы примеров .../opencv/samples/c/
- Что делается, а не как
- Как устанавливать параметры некоторых функций

Одна из целей этой книги состоит в решении этих проблем.

[П]|[РС]|(РП) Структура и содержимое OpenCV

OpenCV структурирован по пяти основным компонентам, четыре из которых показаны на рисунке 1-5. Компонент **CV** содержит основные алгоритмы обработки изображений и высокоуровневые алгоритмы компьютерного зрения; **ML** - библиотеку машинного обучения, которая включает в себя средства статистической классификации и кластеризации. **HighGUI** содержит процедуры и функции ввода/вывода для хранения и загрузки видео и изображений. **CXCore** содержит основные структуры данных.

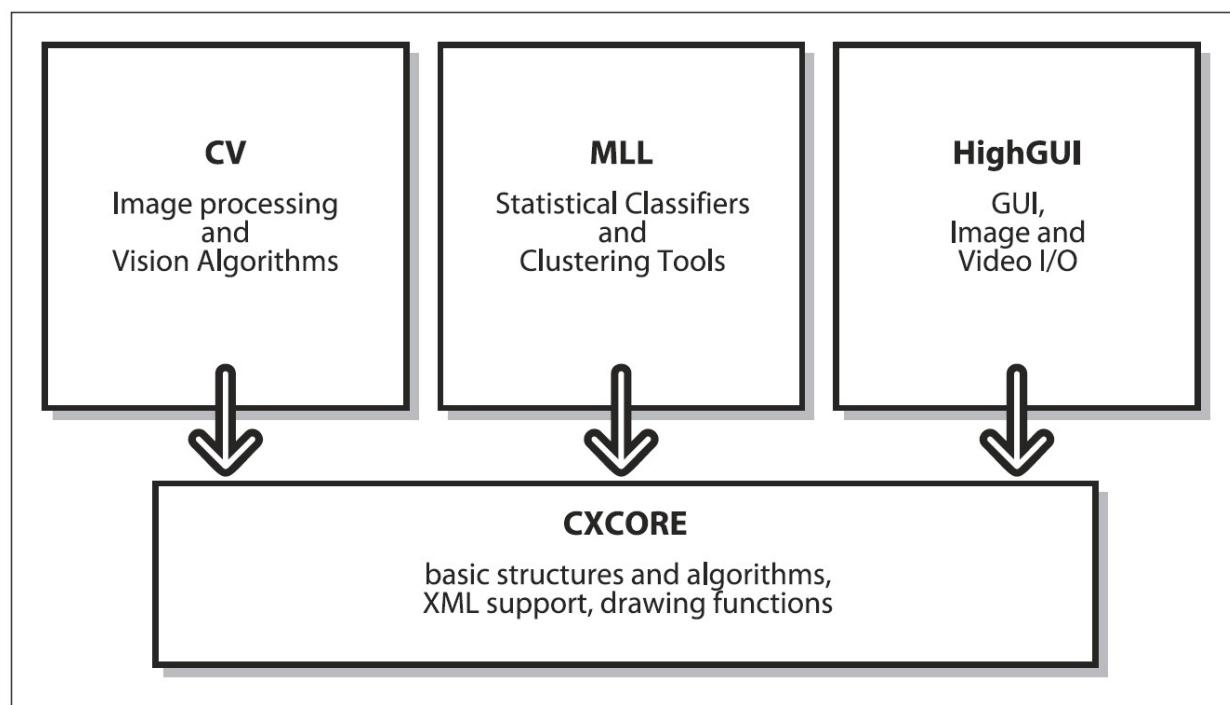


Рисунок 1-5. Базовая структура OpenCV

Рисунок 1-5 не включает **CvAux**, который содержит устаревшие области (встроенные СММ для распознавания лиц) и экспериментальные алгоритмы (background/foreground сегментация). **CvAux** не в должной мере задокументирован на Wiki и вовсе отсутствует в .../opencv/docs. **CvAux** содержит:

- Собственные значения объектов, вычислительно эффективный метод распознавания, который, по сути, оперирует шаблонным сопоставлением
- 1D и 2D скрытую марковскую модели, технику статистического распознавания с использованием метода динамического программирования
- Встроенную СММ
- Распознавание жестов при помощи стерео зрения
- Расширенную триангуляцию Delaunay, последовательности и т.д.

- Стерео зрение
- Сопоставление образцов в области контура
- Дескриптор текстур
- Слежение за глазами и ртом
- 3D tracking
- Поиск скелета (центральных линий) объектов сцены
- Искривление промежуточного обзора между двумя камерами
- Background-foreground сегментацию
- Видеонаблюдение (см. Wiki FAQ для получения дополнительной информации)
- Калибровку камеры классами C++ (функции C и "движок" **CV**)

Некоторые из этих функций могут в дальнейшем мигрировать в **CV**; иные, возможно, никогда.

[П]||[PC]||(РП) Переносимость

OpenCV разрабатывалась как портативная. Изначально была написана при помощи Borland C++, MSVC++, и Intel compilers. Это означало, что C и C++ код должен был быть довольно таки стандартным для того, чтобы было легче осуществлять кроссплатформенную поддержку. На Рисунке 1-6 изображены платформы, на которых OpenCV работает. Начиналось все с поддержки 32-битной архитектуры Intel (IA32) в Windows, затем на Linux с такой же архитектурой. Поддержка Mac OS X началась после того, как Apple начал использовать процессоры Intel. (Поддержка OS X не столь качественная, как у Windows или Linux, но данная ситуация резко меняется). Затем появилась поддержка 64-битной архитектуры Intel (IA64). Для Sun и других операционных систем поддержка на низком уровне.

Если архитектура или ОС отсутствуют на рисунке 1-6, это не означает, что они не поддерживают OpenCV. OpenCV былаパーティонирована почти на все коммерческие системы, включая PowerPC Macs собаки-робота. OpenCV также работает на процессорах AMD, где так же доступна IPP для оптимизации производительности. Для этого в процессорах AMD существует мультимедийное расширение (MMX).

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

Рисунок 1-6. Руководство по OpenCV для версии 1.0: слева ОС, сверху тип архитектуры

[П]|[РС]|(РП) Упражнения

1. Загрузите и установите последнюю версию OpenCV. Соберите библиотеку в режиме debug и release.
2. Скачайте и соберите последнее обновление OpenCV из CVS.
3. Опишите как минимум три неоднозначных аспекта преобразования 3D в 2D представление. Как бы вы преодолели эти неоднозначности?

Введение в OpenCV

[П]||[РС]||(РП) Приступая к работе

После установки OpenCV, первым делом, естественно, хочется сделать что-то интересное. Для этого прежде всего необходимо настроить среду для программирования.

В Visual Studio необходимо в свойствах проекта указать библиотеки (для debug, необходимо использовать библиотеки с *d* на конце) *highgui.lib*, *cxcore.lib*, *ml.lib*, *cv.lib* и путь к заголовочным файлам OpenCV *.../opencv/* /include*. Как правило, "include" - это что-то типа *C:/program files/opencv/cv/include*, *.../opencv/cxcore/include*, *.../opencv/ml/include* и *.../opencv/otherlibs/highgui*. После выполнения этих настроек, все готово для написания первой программы.

Некоторые заголовочные файлы могут существенно облегчить вашу жизнь. Множество полезных макросов можно найти в *.../opencv/cxcore/include/cxtypes.h* и *cxmisc.h*. При помощи них возможно делать такие вещи, как: инициализацию структур и массивов, сортировку списка и т.д. Наиболее важные заголовочные файлы: *.../cv/include/cv.h* и *.../cxcore/include/cxcore.h* для компьютерного зрения, *.../otherlibs/highgui/highgui.h* для ввода/вывода, *.../ml/include/ml.h* для машинного обучения

[П]||[РС]||(РП) Первая программа - отображение картинки

OpenCV содержит средства для чтения широкого спектра типов изображений, а также видеофайлов с диска и видеопотока с камеры. Эти средства являются частью компонента *HighGUI*, который включен в OpenCV. В дальнейшем, часть этих средств будет использована для создания простой программы - открытие изображения и отображения его на экране (Пример 2-1).

Пример 2-1. Простая OpenCV программа - загрузка изображения с диска и отображение его на экране

```
#include "highgui.h"

int main( int argc, char** argv ) {
    IplImage* img = cvLoadImage( argv[1] ); // Получение имени изображения
    cvNamedWindow( "DisplayPicture", CV_WINDOW_AUTOSIZE ); // Создание окна
    cvShowImage( "DisplayPicture", img ); // Показ изображения
    cvWaitKey(0); // Ожидание
    cvReleaseImage( &img ); // Освобождение памяти из под
    cvDestroyWindow( "DisplayPicture" ); // Удаление окна
}
```

После компиляции и запуска из командной строки с одним аргументом, программа загрузит изображение в память и отобразит его на экране. Программа будет ожидать нажатие любой клавиши для закрытия окна и завершения работы. Давайте разберем программу построчно и разберемся, что делает каждая из приведенных строк.

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

Эта строка загружает изображение. Функция *cvLoadImage()* является высокоуровневой: определяет формат файла по его имени; автоматически выделяет память, необходимую для изображения. Обратите внимание, функция *cvLoadImage()* поддерживает широкий спектр типов изображений: BMP, DIB, JPEG, JPE, PNG, PBM, PGM, PPM, SR, RAS и TIFF. Функция возвращает указатель типа *IplImage*. Это наиболее часто используемый тип данных в OpenCV. Используется для обработки всех видов изображений: одноканальные, многоканальные, целочисленные, вещественные и т.д. В дальнейшем манипуляция изображением производиться через созданный указатель *IplImage*.

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

Еще одна высокоуровневая функция `cvNamedWindow()`, открывает окно для размещения и отображения изображения. Функция из библиотеки HighGUI, присваивает имя окну (в нашем случае "DispalyPicture"). Последующие вызовы `HighGUI` будут взаимодействовать с созданным окном по его имени.

Второй аргумент функции `cvNameWindow()` определяет свойства окна. Может быть установлен как 0 (значение по умолчанию) или как `CV_WINDOW_AUTOSIZE`. В первом случае, размер окна не будет зависеть от размера загружаемого изображения, изображение будет подстраиваться под размеры окна. Во втором случае, окно будет подстраиваться под размеры загружаемого изображения.

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

Всякий раз, когда есть изображение в виде указателя `IplImage`, его можно отобразить в существующем окне с помощью функции `cvShowImage()`. Функция требует, чтобы имя окна уже существовало (создано с помощью `cvNamedWindow()`). Вызов `cvShowImage()` перерисовывает окно, с соответствующим изображением в нем, и изменяет его размеры, если при создании был указан флаг `CV_WINDOW_AUTOSIZE`.

```
cvWaitKey(0);
```

Функция `cvWaitKey()` заставляет программу ожидать нажатие любой клавиши. Если аргумент - положительное число, программа будет ожидать заданное количество миллисекунд, а затем продолжит выполнение программы, даже если ничего не будет нажато. Если параметр - 0 или отрицательное число, программа будет бесконечно ожидать нажатие клавиши.

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

Когда изображение больше не требуется, нужно освободить выделенную память. Для выполнения этой операции необходимо передать указатель типа `IplImage*`. После выполнения операции, указатель `img` будет установлен в `NULL`.

```
cvDestroyWindow( "DisplayPicture" );
```

В завершении, нужно уничтожить окно. Функция `cvDestroyWindow()` закрывает окно и высвобождает любую, связанную с этим окном, память (внутренний буфер для изображения, который содержит копию информации о пикселях из `img`). Для столь простой программы (пример 2-1), можно было и не использовать `cvDestroyWindow()` или `cvReleaseImage()`, т.к. все ресурсы и окна автоматически уничтожаются операционной системой при выходе, однако использование этих функций хорошая привычка.

В итоге имеем простую программу, с которой можно ещё "поиграться" различными способами, но пока не будем забегать вперед. Следующей задачей будет построение почти столь же простой программы - программы для чтения и отображения AVI файла. После этого перейдем к более серьезным вещам.

[П]|[РС]|(РП) Вторая программа - AVI видео

Воспроизвести видео в OpenCV почти также легко, как и отображать одно изображение. Разница лишь в том, что нужно организовать некий цикл для обработки последовательности кадров; также нужен способ для выхода из цикла, если фильм скучный.

Пример 2-2. Простая программа для воспроизведения видеофайла с диска при помощи OpenCV

```
#include "highgui.h"

int main( int argc, char** argv ) {
    cvNamedWindow( "PlayVideo", CV_WINDOW_AUTOSIZE );           // Создание окна
    CvCapture* capture = cvCreateFileCapture( argv[1] );        // Открытие видеофайла для фо
    IplImage* frame;                                            // Кадр

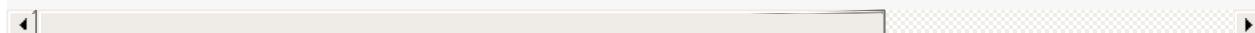
    while(1) {
        frame = cvQueryFrame( capture );                         // Последовательное чтение кадров

        if( !frame ) {                                         // Конец файла, кадров больше нет
            break;
        }

        cvShowImage( "PlayVideo", frame );                      // Отображение кадра
        char c = cvWaitKey(33);                                // Ожидание 33 мс, получение ASCII кода клави

        if( c == 27 ) {                                       // Если Esc - выход из цикла
            break;
        }
    }

    cvReleaseCapture( &capture );                            // Закрытие файла
    cvDestroyWindow( "PlayVideo" );                          // Уничтожение окна
}
```



Функция *main()* начинается с создания окна, в примере имя окна "PlayVideo". После начинается самое интересное.

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

Функция *cvCreateFileCapture()* принимает в качестве аргумента имя видеофайла, а возвращает указатель на структуру типа *CvCapture*. Эта структура содержит всю информацию из видеофайла, включая информацию о состоянии.

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

После, в цикле `while(1)`, происходит чтение видеофайла кадр за кадром. Функция `cvQueryFrame()` принимает в качестве аргумента указатель на структуру **CvCapture** и помещает последующий кадр в память (которая на самом деле часть структуры **CvCapture**). Возвращает указатель полученного кадра. В отличии от `cvLoadImage()`, которая выделяет память под изображение, `cvQueryFrame()` использует выделенную память в структуре **CvCapture**. Поэтому не требуется использовать `cvReleaseImage()` для указателя на "кадр". Вместо этого память из-под кадра будет освобождена, когда произойдет уничтожение структуры **CvCapture**.

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

После отображения кадра, программа будет ожидать нажатие клавиши в течении 33 мс. Если пользователь нажмет клавишу, то будет получен ASCII код этой клавиши; иначе будет получена -1. Если пользователь нажмет Esc (ASCII 27), то чтение кадров прекратиться. Если в течении 33 мс пользователь не нажмет клавишу, то произойдет чтение последующего кадра.

Стоит отметить, что в этом простом примере, не происходит явного контроля скорости воспроизведения видео. Все зависит лишь от таймера `cvWaitKey()`. В более сложном проекте было бы целесообразно полагаться на скорость из структуры **CvCapture**.

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

После окончания воспроизведения видеофайла - обработаны все кадры или пользователь нажал Esc - необходимо освободить память, связанную со структурой **CvCapture**.

[П]||[РС]||(РП) Создание ползунка

Хорошо, уже не плохо. Теперь пришло время расширить функциональность нашей программы. Первое, что бросается в глаза в примере 2-2 - это отсутствие возможности передвигаться по видео. Следующей задачей будет добавление ползунка, который позволит устраниить эту проблему.

Инструментарий *HighGUI* предоставляет ряд простых средств для работы с изображениями и видео. Одним из таких механизмов является слайдер, который с лёгкостью перемещает с одного конца видео в другой. Для создания ползунка необходимо вызвать функцию *cvCreateTrackBar()*, указать окно, в котором этот ползунок появится и установить функцию-обработчик. Детали реализации в примере 2-3.

Пример 2-3. Программа добавления слайдера; при перемещении ползунка, происходит вызов *onTrackbarSlider()* и установка нового значения слайдера

```

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

int      g_slider_position = 0;      // Позиция ползунка
CvCapture* g_capture          = NULL; // Структура для видеофайла

// Вызывается каждый раз, когда происходит изменение положения ползунка
//
void onTrackbarSlide( int pos ) {
    // Установка свойства: положение
    //
    cvSetCaptureProperty(
        g_capture
        ,CV_CAP_PROP_POS_FRAMES
        ,pos // Номер кадра
    );
}

int main( int argc, char** argv ) {
    cvNamedWindow( "TrackBar", CV_WINDOW_AUTOSIZE );      // Создание окна
    g_capture = cvCreateFileCapture( argv[1] );             // Открытие видеофайла для формир

    // Получение кол-ва кадров
    //
    int frames = (int) cvGetCaptureProperty(
        g_capture
        ,CV_CAP_PROP_FRAME_COUNT
    );

    if( 0 != frames ) {
        // Создание ползунка
        //
        cvCreateTrackbar(
            "Position"           // Имя ползунка
            ,"TrackBar"          // Окно для вывода ползунка
            ,&g_slider_position // Начальная позиция
            ,frames              // Максимальная позиция
            ,onTrackbarSlide     // Функция обработчик
        );
    }

    IplImage* frame; // Кадр

    // Цикл последовательного чтения кадров (пример 2-2)
    ...
    // Освобождение памяти и уничтожение окна
    ...

    return(0);
}

```

В сущности, стратегия заключается в добавлении глобальной переменной для хранения позиции слайдера и функции-обработчик для обновления этой переменной. Давайте взглянем на детали.

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

Во-первых, нужно определить глобальную переменную для позиции ползунка. Функции-обработчик требуется объект захвата, поэтому соответствующая переменная также объявлена глобально. Поступим как хорошие люди и сделаем наш код легко читаемым и понятным. Для этого добавим ведущую приставку `g_` к нашим переменным.

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

Функция-обработчик вызывается каждый раз, когда пользователь изменяет положение ползунка. Эта функция принимает 32-разрядное целое число, которое устанавливает положение ползунка.

Вызов функции `cvSetCaptureProperty()`, наряду с функцией `cvGetProperty()`, является наиболее используемой функцией. Эти функции позволяют настроить различные свойства объекта **CvCapture**. В нашем случае, флаг `CV_CAP_PROP_POS_FRAMES` указывает на то, что устанавливается позиция для чтения кадров в единицах (также можно использовать `AVI_RATIO` вместо `FRAMES`, если потребуется установить позицию в процентах). Как результат - новое положение позиции ползунка. Библиотека **HighGUI** довольно-таки умна и потому такие ситуации, как запрос к не ключевому кадру будут обрабатываться автоматически; произойдет откат назад к ближайшему ключевому кадру и уже от него произойдет перемотка вперед до нужного кадра.

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

Как было отмечено ранее, для получения данных из структуры *CvCapture* используется функция *cvGetCaptureProperty()*. В нашем случае, требуется узнать количество кадров в видео для откалибровки ползунка.

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

Для создания ползунка используется функция *cvCreateTrackbar()*, которая принимает в качестве аргументов: имя объекта ползунка, имя окна, переменную положения ползунка, максимальное значение ползунка и функцию-обработчик (или NULL). Ползунок не будет создан, если *cvGetCaptureProperty()* возвратит ноль кадров. Такое возможно, так как общее количество кадров может быть не доступно из-за используемого метода кодирования видеофайла. В таком случае прокрутить видео будет не возможно.

Ползунок из *HighGUI* не является полнофункциональным. Несомненно, существует возможность использовать иные варианты создания ползунка сторонними средствами для устранения этой проблемы, однако, использование ранее описанного подхода позволяет легко и быстро создать простой в использовании ползунок.

И в заключении, не был рассмотрен кусок кода, необходимый для отображения авто перемещения ползунка поверх видео. Эта задача остается в качестве упражнения.

[П]|[РС]|(РП) Простые преобразования

Отлично, теперь с помощью OpenCV можно создать собственный видеоплеер, который не будет сильно отличаться от бесчисленного множества уже существующих плееров. Однако, предметом нашего обсуждения является компьютерное зрение и потому хочется сделать что-то большее, чем просто видеоплеер. Многие базовые задачи компьютерного зрения включают применение различного вида фильтров для обработки видео потока. Давайте попробуем модифицировать уже существующую программу путем добавления простой операции преобразования каждого кадра из видео потока.

Одна из самых простых операций это сглаживание изображения, которая эффективно снижает информативность изображения, сворачивая его гауссовой или другой аналогичной функцией ядра. Сделать нечто подобное довольно таки легко. Начнем с создания нового окна "Transform-out", где будем отображать результат преобразований. После отображения захваченного исходного кадра (функция `cvShowImage()`), произведем сглаживание изображения из кадра и отобразим в окне результата.

Пример 2-4. Загрузка и сглаживание изображения с последующим отображением результата на экране

```

#include <cv.h>
#include <highgui.h>

void TransformImg( IplImage* image ) {
    cvNamedWindow( "Transform-in" );      // Окно для отображения исходного изображения
    cvNamedWindow( "Transform-out" );     // Окно для преобразованного изображения

    cvShowImage( "Transform-in", image );

    // Создание контейнера для преобразованного изображения
    //

    IplImage* out = cvCreateImage(
        cvGetSize( image )
        , IPL_DEPTH_8U
        , 3
    );

    // Сглаживание
    //

    cvSmooth( image, out, CV_GAUSSIAN, 3, 3 );

    // Отображение результата преобразования
    //

    cvShowImage( "Transform-out", out );

    // Высвобождение выделяемой памяти
    //

    cvReleaseImage( &out );

    // Ожидание нажатия клавиши для завершения программы
    //

    cvWaitKey( 0 );
    cvDestroyWindow( "Transform-out" );
    cvDestroyWindow( "Transform-in" );
}

int main( int argc, char** argv ) {
    IplImage* img = cvLoadImage( argv[1] ); // Получение имени изображения
    TransformImg( img );                  // Преобразование исходного изображения
    cvReleaseImage( &img );              // Освобождение памяти из под изображения
}

```

Ранее, новый кадр создавался при содействии функции *cvCreateFileCapture()*. Происходил покадровый перебор всех кадров из структуры **CvCapture**, один за другим, с размещением их в одном и том же указателе. В нашем случае использование данного указателя не приемлемо и потому создается собственная структура под результат сглаживания. Для этого используется новая функция *cvCreateImage()*. Аргументы функции *cvCreateImage(CvSize size, int depth, int channels)*:

- `size` - размер существующей структуры изображения (удобно получать при помощи функции `cvGetSize()`)
- `depth` - тип данных для каждого канала
- `channels` - количество каналов

В примере 2-3 создается 8-ми битное 3-х канальное изображение того же размера, что и исходное изображение.

Операция сглаживания это всего лишь вызов функции из OpenCV, которой передается указатель на исходное изображение, указатель на результирующее изображение, тип и параметры сглаживания. В примере 2-3 используется размытие по Гауссу с ядром 3x3.

Не забывайте освобождать ресурсы, выделенные под результирующее изображение (используйте `cvReleaseImage()`).

[П]|[РС]|(РП) Не столь простые преобразования

Уже не плохо, переходим к еще более интересным вещам. В примере 2-4 создавалась новая структура под единичное преобразование изображения. По идеи, существует возможность работать только с исходным изображением, без создания дополнительного контейнера под преобразованное изображение, но зачастую это плохая идея. Например, некоторые операторы не работают с изображениями такого же размера, типом каналов и количеством каналов, как исходное изображение. Как правило, производиться цепочка преобразований и потому использование стороннего контейнера неизбежно.

В таких случаях, зачастую полезно использовать функции-обертки, чтобы произвести все необходимые преобразования. Рассмотрим пример сжатия изображения в 2 раза. В OpenCV это достигается при помощи функции `cvPyrDown()`, которая выполняет гауссово сглаживание, а затем удаляет каждую вторую строку из изображения. Это полезно в самых разнообразных и важных алгоритмах компьютерного зрения.

Реализация функции-обертки представлена в примере 2-5.

Пример 2-5. Использование `cvPyrDown()` для сжатия исходного изображения в 2 раза

```
IplImage* doPyrDown( IplImage* in, int filter = IPL_GAUSSIAN_5x5 ) {
    // Проверка деления исходного изображения на 2
    //
    assert( 0 == in->width%2 && 0 == in->height%2 );

    // Создание контейнера с размером вдвое меньшим, чем исходное изображение.
    // В остальном параметры те же
    //
    IplImage* out = cvCreateImage(
        cvSize( in->width/2, in->height/2 )
        ,in->depth
        ,in->nChannels
    );

    // Сжатие исходного изображения
    //
    cvPyrDown( in, out );

    return( out );
};
```

Обратите внимание, что под преобразованное изображение создается новая структура на основе параметров исходного изображения. В OpenCV все важные типы данных реализованы как структуры, в которых нет private данных, и все крутится вокруг указателей на эти структуры.

Переходим к чуть более сложному примеру с использованием функции определения контура **Canny edge detector**. В этом примере конечное изображение сохранит исходные размеры, но воспользуется только одним каналом.

Пример 2-6. Определение контура при помощи функции Canny. Результат - одноканальное изображение

```
IplImage* doCanny(
    IplImage* in          // Одноканальное изображение
    ,double    lowThresh   // Нижний порог
    ,double    highThresh  // Верхний порог
    ,double    aperture    // Размер оператора Собеля
) {
    // Canny работает только с одноканальными изображениями
    //
    if( 1 != in->nChannels ) {
        return(0);
    }

    // Создание контейнера под конечное изображение
    //
    IplImage* out = cvCreateImage(
        cvSize( cvGetSize( in ) )
        ,IPL_DEPTH_8U
        ,1
    );

    // Преобразования
    //
    cvCanny( in, out, lowThresh, highThresh, aperture );

    return( out );
}
```

В итоге, можно довольно-таки легко соединять несколько различных операторов. Например, требуется уменьшить изображение в два раза, а затем найти контур в уменьшенном еще раз вдвое изображении.

Пример 2-7. Сочетание сжатия изображения (дважды) и оператора *Canny*

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

// делаем что-нибудь с img3
...
//

cvReleaseImage( &img1 );
cvReleaseImage( &img2 );
cvReleaseImage( &img3 );
```

Важно отметить, что создание дополнительных переменных под каждое преобразование не является хорошей идеей. Зачастую, из-за своей лени, происходит утечка памяти в связи с отсутствием соответствующей функции освобождения памяти `cvReleaseImage()` в функции-обертки.

Механизм "само очистки" будет более аккуратным, если использовать эту функцию, однако существует еще одна проблема: что, если нам потребуется что-то сделать с промежуточным изображением? Ничего может не получиться из-за возможного отсутствия доступа к этому изображению. Решить проблему поможет подход из примера 2-8.

Пример 2-8. Упрощение примера 2-7

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

// делаем что-нибудь с 'out'
...
//

cvReleaseImage ( &out );
```

В заключении, стоит отметить, что освобождать память необходимо из-под выделяемой под преобразования структуры. Рассмотрим небольшой пример: указатель `IplImage`, возвращаемый `cvCreateFileCapture()`, указывает на структуру типа `CvCapture` и инициализируется только в момент загрузки видеофайла. Освобождение памяти, вызовом функции `cvReleaseImage()`, приведет к некоторым неожиданным проблемам. Мораль сей басни такова: очистка мусора важна, но только мусора, которой создавали сами!

[П]|[РС]|(РП) Захват видео с камеры

В мире компьютеров, компьютерное зрение может означать множество связных с ним вещей. В некоторых случаях анализируются кадры, полученные из заранее неизвестного источника. В других, анализируется видео с диска. И наконец, возникает потребность в анализе видео потока в реальном времени, получаемого с камеры.

OpenCV - а точнее ее часть, *HighGUI* - помогает с легкостью справиться с подобного рода ситуациями. Подход схож с тем, как происходило чтение AVI файла. Вместо вызова *cvCreateFileCapture()*, используется *cvCreateCameraCapture()*. В качестве аргумента используется не имя файла, как ранее, а идентификационный номер камеры, и то имеет смысл его использовать, когда доступно сразу несколько камер. Значение по умолчанию равно -1, что означает "просто выбрать одну"; естественно, это работает, когда доступна только одна камера (см. Главу 4 для получения более подробной информации).

Функция *cvCreateCameraCapture()* возвращает указатель на структуру **CvCapture**, которая обрабатывается так же, как и в примере с обработкой видео файла с диска. Конечно, большая часть работы происходит "за кулисами" для того, чтобы изображение с камеры просматривалось как видео, все эти проблемы скрыты от пользователя. Нужно лишь захватить изображение с камеры, когда это потребуется. Для большинства разрабатываемых приложений требуется обеспечить видео поток в реальном времени, что при наличии уникальной структуры **CvCapture** с легкостью позволяет это сделать.

Пример 2-9. После инициализации переменной захвата, уже не важно, изображение с камеры или из файла

```
...
CvCapture* capture; // Переменная захвата

if( 1 == argc ) {
    capture = cvCreateCameraCapture(0);      // Захват изображения с камеры
}
else {
    capture = cvCreateFileCapture( argv[1] ); // Захват изображения из файла
}

assert( capture != NULL ); // Проверка успешного захвата
...
```

Пример 2-10. Простой пример вывода изображения, получаемого с камеры

```
#include <highgui.h>
#include <cv.h>

int main() {
    CvCapture *capture = cvCreateCameraCapture(0); // Захват изображения с камеры

    // Проверка успешного захвата
    //

    if( capture != NULL ) {
        IplImage *frame = NULL; // Кадр
        cvNamedWindow( "Camera", CV_WINDOW_AUTOSIZE ); // Создание окна

        while( 1 ) {
            frame = cvQueryFrame( capture ); // Получение кадра, так же как и из видео

            if( !frame ) { // Конец файла, кадров больше нет
                break;
            }

            cvShowImage( "Camera", frame ); // Отображение кадра
            char c = cvWaitKey(33); // Ожидание 33 мс, получение ASCII кода

            if( 27 == c ) { // Если Esc - выход
                break;
            }
        }

        cvDestroyWindow( "Camera" ); // Уничтожение окна
    }

    // Очистка памяти
    //

    cvReleaseCapture( &capture );

    return 0;
}
```

[П]|[РС]|(РП) Запись видео в формате AVI

Во многих приложениях требуется записывать потоковое видео в файл и OpenCV предоставляет простые в использовании средства для этого. Просто нужно создать устройство захвата, прочитать из него кадры один за одним и при помощи созданного устройства записи поместить кадры один за другим в видео файл. Устройство записи создается при помощи функции `cvCreateVideoWriter()`.

Покадровое чтение производится при помощи функции `cvWriteFrame()`. Освобождение памяти, занимаемая устройством записи, производится при помощи функции `cvReleaseVideoWriter()`. В примере 2-11 представлена простая программа, в которой происходит чтение содержимого видео файла с диска, преобразование его в формат **logpolar** (см Главу 6 для более подробной информации) и запись в новый видео файл.

Пример 2-11. Преобразование 3-х канального видео файла в одноканальное (цветное -> оттенки серого)

```
// argv[1]: имя исходного видео файла
// argv[2]: имя нового видео файла
#include <cv.h>
#include <highgui.h>

int main( int argc, char* argv[] ) {
    CvCapture* capture = 0; // Переменная захвата
    capture = cvCreateFileCapture( argv[1] ); // Чтение видео файла с диска

    // Не удалось захватить видео поток с диска
    //
    if( !capture ) {
        return -1;
    }

    // Получение первого кадра из видео потока с диска
    //
    IplImage *bgr_frame=cvQueryFrame( capture );

    // Частота кадров
    //
    double fps = cvGetCaptureProperty( capture, CV_CAP_PROP_FPS );

    // Размер кадра
    //
    CvSize size = cvSize(
        (int)cvGetCaptureProperty( capture, CV_CAP_PROP_FRAME_WIDTH )
        ,(int)cvGetCaptureProperty( capture, CV_CAP_PROP_FRAME_HEIGHT )
    );
}
```

```
// Создание устройства записи
//
CvVideoWriter *writer = cvCreateVideoWriter(
    argv[2]
    ,CV_FOURCC( 'M','J','P','G' )      // Кодек
    ,fps
    ,size
);

// Контеинер для промежуточного преобразования кадра
//
IplImage* logpolar_frame = cvCreateImage(
    size
    ,IPL_DEPTH_8U
    ,3
);

// Чтение кадров, пока не закончится
//
while( (bgr_frame = cvQueryFrame( capture )) != NULL ) {
    // Преобразование
    //
    cvLogPolar(
        bgr_frame           // Исходный кадр
        ,logpolar_frame     // Преобразованный кадр
        ,cvPoint2D32f(       // Центр преобразования
            bgr_frame->width/2
            ,bgr_frame->height/2
        )
        ,40                 // Масштабный коэффициент
        ,CV_INTER_LINEAR     // Сочетание метода интерполяции и необязательног
        + CV_WARP_FILL_OUTLIERS // флага CV_WARP_FILL_OUTLIERS и/или CV_WARP_INVE
    );
}

// Запись преобразованного кадра
//
cvWriteFrame( writer, logpolar_frame );
}

// Очистка занимаемой памяти
//
cvReleaseImage( &logpolar_frame );
cvReleaseVideoWriter( &writer );
cvReleaseCapture( &capture );

return(0);
}
```

Все начинается с открытия видео файла с диска; затем берется первый кадр при помощи функции *cvQueryFrame()* для определения необходимых свойств при помощи функции *cvGetCaptureProperty()*. Потом создается устройство для записи и покадрово записываются преобразованные кадры в формате log-polar в новый видео файл. В заключении происходит освобождение занимаемой памяти.

Вызов функции *cvCreateVideoWriter()* сопровождается передачей в нее нескольких параметров:

```
cvCreateVideoWriter(const char* filename, int fourcc, double fps, CvSize frame_size, int
```

- fileName - имя нового видео файла
- foutcc - видео кодек, которым будет сжиматься видеопоток.
- fps - частота кадров созданного видеопотока
- frame_size - размер кадра
- is_color - 1: кодирование цветных кадров; 0: кодирование кадров в оттенках серого

Есть бесчисленное множество кодеков; прежде чем выбрать тот или иной кодек, убедитесь, что он доступен на вашей машине (кодеки устанавливаются отдельно от OpenCV). В примере выбран довольно популярный кодек **MJPG**; на это указывает макрос *CV_FOURCC()*, который принимает четыре символа в качестве аргументов. Эти символы составляют "четырехзначный код" кодека, каждый кодек имеет такой код. Четырехзначный код движущейся jpg-картинки есть MJPG (motion jpeg), потому и указано *CV_FOURCC('M','J','P','G')*.

Список кодеков, включённых в OpenCV:

```
'X', 'V', 'I', 'D' - кодек XviD
'P', 'I', 'M', '1' - MPEG-1
'M', 'J', 'P', 'G' - motion-jpeg
'M', 'P', '4', '2' - MPEG-4.2
'D', 'I', 'V', '3' - MPEG-4.3
'D', 'I', 'V', 'X' - MPEG-4
'U', '2', '6', '3' - H263
'I', '2', '6', '3' - H263I
'F', 'L', 'V', '1' - FLV1
```

[П]||[РС]||(РП) Резюме

Прежде чем перейти к следующей главе, подведем итоги. Уже известно, как API OpenCV предоставляет множество простых в использовании инструментов для загрузки изображений из файлов, чтения видео файлов с жёсткого диска и захвата видео с камер. Так же известно, что библиотека предоставляет множество простых функций для работы с этими изображениями. Однако, всё ещё не были представлены более мощные инструменты OpenCV, которые позволяют производить более сложные манипуляции с абстрактными типами данных.

В следующих нескольких главах будет более детальное погружение в основы работы OpenCV, будет рассмотрено как устроены некоторые типы данных и функции для работы с ними. Продолжим разбираться с простыми функциями для преобразования изображений, а затем перейдем и к более сложными. После этого перейдем к изучению специализированных функций, которые предоставляют нам API, для таких задач, как калибровка камеры, отслеживание движений и распознавания образов.

Готовы? Поехали!

[П]|[РС]|(РП) Упражнения

Скачайте и установите OpenCV. Изучите структуру каталогов. В каталоге `docs`, загрузив файл `index.html`, можно изучить документацию по OpenCV.

- **Cvcore** содержит основные структуры данных и алгоритмы
- **cv** содержит механизмы обработки изображений и алгоритмы видения
- **ml** содержит алгоритмы машинного обучения и кластеризации
- **otherlibs/highgui** содержит функции ввода/вывода

Проверте наличие директорий `_make` (содержит файлы для сборки OpenCV) и `samples`, где хранятся примеры кода.

1. Перейдите в директорию `../opencv/_make`. Если Windows, откройте `opencv.sln`; если Linux, откройте соответствующий файл сборки. Соберите библиотеку в обоих режимах: `debug` и `release`. Это может занять некоторое время. В результате вы должны получить библиотеки и `dll` файлы.
2. Перейдите в директорию `../opencv/samples/c/`. Создайте проект и импортируйте файл `/kdemo.c` (пример отслеживания движения). Подключите камеру к компьютеру и запустите код. Нажмите '`r`', чтобы инициализировать отслеживание. Добавление точек на видео производится нажатием клавиши мыши. Переключение в режим без фона производится нажатием клавиши '`n`'. Повторное нажатие клавиши '`n`' вернет фон.
3. Совместите код из примера 2-11 и код из примера 2-5, чтобы создать программу, которая будет считывать видеопоток с камеры и сохранять цветное изображение с пониженной дискретизацией на диск.
4. Модифицируйте предыдущий код добавлением в него кода из примера 2-1, чтобы отобразить процесс преобразования кадров на экран.
5. Модифицируйте предыдущий код добавлением в него слайдера из примера 2-3, чтобы пользователь смог динамически изменять пирамидальное размытие между 2 и 8. Можно опустить этап записи результата на диск. (Судя по всему речь идет об изменении третьего аргумента функции `pyrDown()` при помощи функции `Size()`. Прототип `pyrDown(const CvArr src, CvArr dst, int filter=CV_GAUSSIAN_5x5)`. Как результат, чем выше показатель, тем более размытым будет изображение)

Знакомство с OpenCV

[П]|[РС]|(РП) Примитивные типы данных в OpenCV

OpenCV включает в себя множество примитивных типов данных. Эти данные не являются примитивными с точки зрения C, но являются самыми элементарными с точки зрения OpenCV. Посмотреть, как устроены все эти структуры, можно в файле `cxtypes.h` в директории `.../OpenCV/cxcore/include`.

Самый простейший тип - `CvPoint`. Это простая структура состоит только из двух полей `x` и `y` типа `int`. `CvPoint2D32f` содержит два поля `x` и `y` типа `float`. `CvPoint3D32f` содержит три поля `x`, `y` и `z` типа `float`.

`CvSize` содержит два поля `width` и `height` типа `int`. `CvSize2D32f` содержит два поля `width` и `height` типа `float`.

`CvRect` содержит четыре поля `x`, `y`, `width` и `height` типа `int`.

`CvScalar` содержит четыре переменные типа `double`. На самом деле `CvScalar` включает в себя одно поле `val`, которое является указателем на массив, содержащий четыре числа типа `double`.

Все эти типы данных имеют конструкторы с именами похожими на `cvSize` (обычно именуются так же, как и структуры, только первая буква – строчная). Помните, что это C, а не C++, и все эти "конструкторы" всего лишь `inline` функции, принимающие список аргументов и возвращающие желаемую структуру со значениями установленными соответствующим образом.

Конструкторы для типов данных из таблицы 3-1 - `cvPointXXX()`, `cvSize()`, `cvRect()`, `cvScalar()` - чрезвычайно полезны, потому что упрощают ваш код. Например, чтобы нарисовать белый прямоугольник с координатами углов (5, 10) и (20, 30), достаточно написать следующий код:

```
cvRectangle(
    myImg           // Изображение
    ,cvPoint(5,10)   // Верхний левый угол
    ,cvPoint(20,30)   // Нижний правый угол
    ,cvScalar(255,255,255) // Цвет
);
```

Таблица 3-1. Структура точка, размер, прямоугольник и скаляр

Структура	Поля	Описание
CvPoint	int x, y	Пиксель
CvPoint2D32f	float x,y	2D точка
CvPoint3D32f	float x, y, z	3D точка
CvSize	int width, height	Размер изображения
CvRect	int x, y, width, height	Часть изображения
CvScalar	double val[4]	Значение RGBA

cvScalar в отличии от всех остальных структур содержит три конструктора. Первый может принимать один, два, три или четыре аргумента и присваивать их соответствующим элементам *val*[*l*]. Второй конструктор *cvRealScalar()* принимает один аргумент и устанавливает соответствующее значение *val*[0], остальные элементы устанавливаются в 0. Третий конструктор *cvScalarAll()* так же принимает один аргумент и инициализирует все элементы *val*[*l*] этим значением.

Типы изображений, представленные матрицами

На рисунке 3-1 представлена иерархия классов или структур трёх типов изображений. При использовании OpenCV, неоднократно будет использоваться тип *IplImage*. *IplImage* это базовая структура, используемая для кодирования того, что мы называем "изображение". Эти изображения могут быть чёрно-белыми, цветными, 4-х канальными (RGB+Alpha), и каждый канал может содержать либо целые, либо вещественные значения. Следовательно, этот тип является более общим, чем вездесущие 3-х канальные 8-битные изображения, которые сразу приходят на ум.

OpenCV располагает обширным арсеналом операторов для работы с этими изображениями, которые позволяют изменять размеры изображений, извлекать отдельные каналы, складывать два изображения, и т.д. В этой главе такие операторы будут рассмотрены более тщательно.

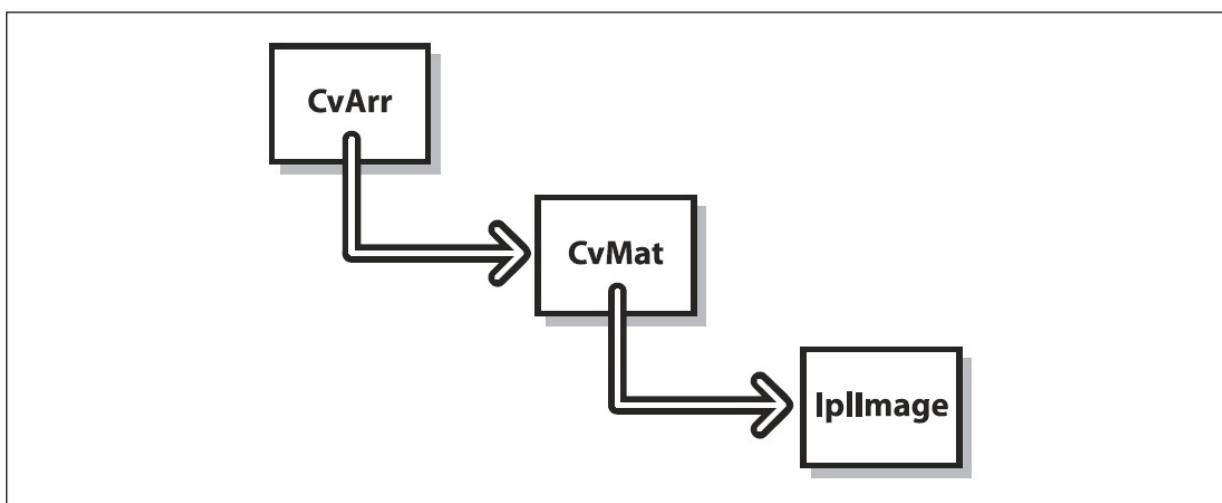


Рисунок 3-1. Несмотря на то, что OpenCV написана на С, структуры, используемые в OpenCV, объектно-ориентированные; в действительности, *IplImage* происходит от *CvMat*, которая является производной от *CvArr*.

Прежде, чем перейти к деталям, необходимо взглянуть на другой тип данных: *CvMat*, структура для матриц. Хотя OpenCV полностью написана на С, относительная взаимосвязь между *CvMat* и *IplImage* похожа на наследование в C++. *IplImage* можно рассматривать как производную от *CvMat*. Класс *CvArr* можно рассматривать как абстрактный базовый класс. В прототипах функций зачастую будет указано *CvArr* (точнее указатель *CvArr*). В таких случаях вместо *CvArr* можно использовать и указатель *CvMat* и указатель *IplImage*.

[П]|[РС]|(РП) Структура CvMat

Прежде, чем приступить к обсуждению данной темы, уточним несколько вещей относительно матриц. Во-первых, в OpenCV нет конструкции "vector". Когда требуется вектор, просто воспользуйтесь матрицей с одной колонкой (или одной строкой, если требуется транспонированный или сопряженный вектор). Во-вторых, понятие матрицы в OpenCV несколько абстрактно, нежели в линейной алгебре. В частности, элементы матрицы не обязательно должны быть просто числами. К примеру, создание новой двумерной матрицы имеет следующий прототип:

```
cvMat* cvCreateMat ( int rows, int cols, int type );
```

Здесь *type* может быть любым из длинного списка предопределенных типов; тип задаётся так: *CV_<глубина в битах>(S|U|F)C<число каналов>*. Например, матрица может состоять из 32-битных чисел с плавающей точкой(*CV_32FC1*), 8-битных без знаковых триплетов(*CV_8UC3*) и из множества других. Элементами *CvMat* могут быть не только числа. Возможность представить один элемент составным значением позволяет делать такие вещи, как представление нескольких цветовых каналов в изображении RGB. С простыми изображениями, содержащие красную, зелёную и синюю составляющую, большинство операторов будут работать с каждым каналом по отдельности (если не указано обратное).

Внутренняя структура *CvMat* является довольно таки простой, что доказывает пример 3-1 (так же в этом можно убедиться, открыв файл *../opencv/cxcore/include/cxtypes.h*). Матрицы имеют ширину, высоту, тип, шаг (длина строки в байтах, а не в *int* или *float*) и указатель на массив данных (и еще несколько вещей, о которых пока не будет рассказано). Получить доступ к элементам матрицы можно непосредственно через указатель на *CvMat* или через специальные функции. Например, для получения размера матрицы можно либо вызвать функцию *cvGetSize(CvMat*)*, либо непосредственно обратиться к соответствующим полям через указатель как *matrix->height* и *matrix->width*.

Пример 3-1. Структура CvMat

```

typedef struct CvMat {
    int type;          // Тип элементов
    int step;          // Шаг
    int* refcount;    // Только для внутреннего использования
    union {
        uchar* ptr;
        short* s;
        int* i;
        float* f1;
        double* db;
    } data; // Данные матрицы
    union {
        int rows;      // Кол-во строк
        int height;   // Высота
    };
    union {
        int cols;      // Кол-во столбцов
        int width;    // Ширина
    };
} CvMat;

```

Эта информация обычно именуется заголовком матрицы. Многие подпрограммы разделяют заголовок и данные, причем данные представлены указателем.

Матрицы могут быть созданы несколькими способами. Наиболее распространенным является подход с использованием `cvCreateMat()`, которая по существу использует более элементарные функции `cvCreateMatHeader()` и `cvCreateData()`.

`cvCreateMatHeader()` создает структуру `CvMat`, без выделения памяти под данные, в то время, как `cvCreateData()` выделяет память под данные. По тем или иным причинам, порой, требуется создание лишь заголовка матрицы. Еще один метод заключается в вызове функции `cvCloneMat(CvMat*)`, которая создает новую матрицу на основе существующей. Когда матрица больше не нужна, память из под неё может быть освобождена вызовом функции `cvReleaseMat(CvMat**)`.

В примере 3-2 представлен список функций, которые только что были описаны, а также некоторые другие, связанные с ними.

Пример 3-2. Создание и уничтожение матриц

```

// Создание rows by cols матрицы типа 'type'.
CvMat* cvCreateMat( int rows, int cols, int type );

// Создание только заголовка матрицы
CvMat* cvCreateMatHeader( int rows, int cols, int type );

// Инициализация заголовка уже существующей матрицы
CvMat* cvInitMatHeader(
    CvMat* mat
    ,int      rows
    ,int      cols
    ,int      type
    ,void*   data = NULL
    ,int      step = CV_AUTOSTEP
);

// Схожа с cvInitMatHeader()
CvMat cvMat(
    int      rows
    ,int      cols
    ,int      type
    ,void*   data = NULL
);

// Создание новой матрицы на основе существующей
CvMat* cvCloneMat( const CvMat* mat );

// Освобождение памяти из-под матрицы
void cvReleaseMat( CvMat** mat );

```

По аналогии с другими структурами OpenCV для *CvMat* также есть конструктор *cvMat()*. Конструктор не выделяет память под данные, а лишь создаёт заголовок (по аналогии с *cvInitMatHeader()*). Хороший способ создать матрицу из уже имеющихся данных показан в примере 3-3.

Пример 3-3. Создание матрицы из уже существующих данных

```

float vals[] = { 0.866025, -0.500000, 0.500000, 0.866025 }; // Данные
CvMat rotmat;
cvInitMatHeader(
    &rotmat      // Матрица
    ,2           // Кол-во строк
    ,2           // Кол-во столбцов
    ,CV_32FC1    // Тип
    ,vals        // Данные
);

```

После того, как матрица создана, можно производить над ней множество интересных действий.

```
cvGetElemType( const CvArr* arr )
cvGetDims( const CvArr* arr, int* sizes = NULL )
cvGetDimSize( const CvArr* arr, int index )
```

Первая функция возвращает тип элемента матрицы (например, `CV_8UC1`, `CV_64_FC4` и т.д.). Вторая принимает указатель на массив и дополнительный указатель на целое число, а возвращает количество измерений (в примере матрица 2×2). Если указатель на число не `NULL`, то будет возвращена размерность принимаемого массива. Последняя функция принимает целое число, указывающее размер в процентах и просто возвращает степень матрицы в указанном измерении.

Доступ к данным матрицы

Существует три варианта получения данных матрицы: простой, сложный и правильный.

Простой способ

Самый простой способ получить данные матриц это воспользоваться макросом `CV_MAT_ELEM()`. Этот макрос (пример 3-4) принимает в качестве аргументов указатель на матрицу, тип элементов, сроку и столбец, а возвращает запрашиваемый элемент.

Пример 3-4. Доступ к данным матрицы через макрос `CV_MAT_ELEM`

```
CvMat* mat = cvCreateMat( 5, 5, CV_32FC1 ); // Создание матрицы
float element_3_2 = CV_MAT_ELEM( *mat, float, 3, 2 ); // Получение элемента матрицы
```

"Под капотом" этот макрос просто вызывает другой макрос `CV_MAT_ELEM_PTR()`. Этот макрос (пример 3-5) принимает в качестве аргументов указатель на матрицу, номер строки и столбца запрашиваемого элемента и возвращает указатель на нужный элемент. Одно важное отличие `CV_MAT_ELEM()` от `CV_MAT_ELEM_PTR()` в том, что `CV_MAT_ELEM()` преобразует указатель в соответствии с типом. Если требуется задать значение элементу матрицы, то нужно непосредственно вызвать `CV_MAT_ELEM_PTR()`; при этом, однако, необходимо сделать приведение типов в явном виде.

Пример 3-5. Установка значения элемента матрицы, используя макрос `CV_MAT_ELEM_PTR()`

```
CvMat* mat = cvCreateMat( 5, 5, CV_32FC1 ); // Создание матрицы
float element_3_2 = 7.7; // Значение элемента в строке
*( (float*)CV_MAT_ELEM_PTR( *mat, 3, 2 ) ) = element_3_2; // Установка элемента
```

К сожалению эти макросы пересчитывают смещение указателя каждый раз при их вызове. Это означает, что указатель каждый раз указывает на первый элемент матрицы; происходит вычисление смещения и добавление полученного значения смещения к указателю на первый элемент матрицы. Таким образом, хоть эти макросы и просты в использовании, это не лучший способ получения доступа к данным матрицы. Лучшим примером "против" использования данного подхода является вариант последовательного перебора элементов матрицы.

Сложный способ

Два макроса, которые были рассмотрены в простом способе, могут работать только с одно- и двумерными матрицами (одномерные массивы или вектора, на самом деле, просто $1 \times n$ матрица). OpenCV предоставляет механизмы для обработки многомерных массивов. Фактически OpenCV позволяет обрабатывать n -мерные матрицы "условно любого" размера.

Для получения доступа к элементам матрицы используется семейство функций `cvPtr*D` и `cvGet*D` описанные в примере 3-6 и 3-7. Семейство `cvPtr*D` содержит функции `cvPtr1D()`, `cvPtr2D()`, `cvPtr3D()` и `cvPtrND()`. Каждая из первых трех принимает указатель на матрицу `CvArr`, соответствующий количеству целых индексов, и необязательный параметр, указывающий на тип выходного параметра. Все эти процедуры возвращают указатель на необходимый элемент. `cvPtrND()` вторым аргументом принимает указатель на массив, содержащий соответствующее количество индексов.

Пример 3-6. Доступ к элементам матрицы через указатель

```

uchar* cvPtr1D(
    const CvArr* arr
    , int idx0
    , int* type = NULL
);

uchar* cvPtr2D(
    const CvArr* arr
    , int idx0
    , int idx1
    , int* type = NULL
);

uchar* cvPtr3D(
    const CvArr* arr
    , int idx0
    , int idx1
    , int idx2
    , int* type = NULL
);

uchar* cvPtrND(
    const CvArr* arr
    , int* idx
    , int* type = NULL
    , int create_node = 1
    , unsigned* precalc_hashval = NULL
);

```

Для простого чтения данных есть и другое семейство функций *cvGet*D*, описанные в примере 3-7 и возвращающие фактическое значение элемента матрицы.

Пример 3-7. Функции для работы с *CvMat* и *IplImage*

```

double cvGetReal1D( const CvArr* arr, int idx0 );                                // Для одно-
double cvGetReal2D( const CvArr* arr, int idx0, int idx1 );                      // двух-
double cvGetReal3D( const CvArr* arr, int idx0, int idx1, int idx2 );            // трёх-
double cvGetRealND( const CvArr* arr, int* idx );                                // N-канальных

CvScalar cvGet1D( const CvArr* arr, int idx0 );
CvScalar cvGet2D( const CvArr* arr, int idx0, int idx1 );
CvScalar cvGet3D( const CvArr* arr, int idx0, int idx1, int idx2 );
CvScalar cvGetND( const CvArr* arr, int* idx );

```

Тип возвращаемого значения для первых четырех функций - число типа *double*, для других четырех *CvScalar*. Это означает, что имеет место значительное увеличение ненужных расходов при использовании этих функций. Они должны быть использованы только там, где это уместно и эффективно; иначе лучше использовать *cvPtr*D*.

Причин, по которой лучше использовать `cvPtr*D()` заключается в том, что эти функции возвращают указатель на необходимый элемент, что в свою очередь позволяет использовать арифметику указателей для перемещения по матрице. Важно помнить, что каналы в многоканальной матрице являются смежными. Например, трехканальная двумерная матрица, представляющая байты для красного, зеленого, синего цветов (RGB), хранит данные как: rgbrgbrgb Поэтому для перемещения указателя на следующий канал, необходимо увеличить указатель на единицу. Если потребуется перейти к следующему "пикселью" или набору элементов, увеличьте указатель на число равное числу каналов (в приведенном примере на 3).

Иной способ узнать шаг элемента матрицы (примеры 3-1 и 3-3) заключается в получении длины строки матрицы в байтах. В структуре `CvMat`, колонок или ширины не достаточно для перемещения между строками матрицы, т.к. эффективное выделение памяти под матрицы или изображения выполняется до ближайшей границы в четыре байта. Таким образом под матрицы шириной в три байта будет выделено четыре байта и последний байт не будет использован. По этой причине, после получения указателя на элемент, необходимо увеличить его на соответствующий шаг. Если имеется матрица типа `int` или `float` и соответствующий указатель на элемент, то шаг для получения следующей строки: для `int` - `step/4`, для `double` - `step/8` (при этом С будет автоматически умножать смещение в соответствии с типом данных).

Семейства функций `cvSetReal*D()` и `cvSet*D()`, описанные в примере 3-8, устанавливают значения элементов матрицы или изображения.

Пример 3-8. Функции для установки значений элементам `CvMat` или `IplImage`

```

void cvSetReal1D( CvArr* arr, int idx0, double value );
void cvSetReal2D( CvArr* arr, int idx0, int idx1, double value );
void cvSetReal3D(
    CvArr* arr
    ,int idx0
    ,int idx1
    ,int idx2
    ,double value
);
void cvSetRealND( CvArr* arr, int* idx, double value );
void cvSet1D( CvArr* arr, int idx0, CvScalar value );
void cvSet2D( CvArr* arr, int idx0, int idx1, CvScalar value );
void cvSet3D(
    CvArr* arr
    ,int idx0
    ,int idx1
    ,int idx2
    ,CvScalar value
);
void cvSetND( CvArr* arr, int* idx, CvScalar value );

```

Для удобства также имеются функции *cvmSet()* и *cvmGet()*, которые имеют дело с числами с плавающей точкой одно-канальных матриц.

```
double cvmGet( const CvMat* mat, int row, int col )
void cvmSet( CvMat* mat, int row, int col, double value )
```

Так, вызов функции *cvmSet()*

```
cvmSet( mat, 2, 2, 0.5000 );
```

эквивалентен вызову функции *cvSetReal2D*

```
cvSetReal2D( mat, 2, 2, 0.5000 );
```

Правильный способ

Со всеми этими функциями можно подумать, что больше не о чем говорить. На самом деле, на практике данный перечень функций крайне редко используется. На протяжение всей работы программы компьютерного зрения интенсивно используют процессор. Использование ранее перечисленных функций оказывается на эффективности. Вместо использования ранее описанных способов нужно разрабатывать собственную арифметику для работы с указателями. Управлять указателями особенно важно, когда требуется произвести изменения каждого элемента матрицы.

Для прямого доступа к элементам матрицы все, что нужно знать, это то, что данные хранятся последовательно в порядке растрового сканирования, где переменные колонок наиболее быстро обрабатываются. Каналы чередуются, что в случае с многоканальной матрицей означает еще более быструю обработку. Пример 3-9 показывает то, как это может быть сделано.

Пример 3-9. Суммирование всех элементов трехканальной матрицы

```

float sum( const CvMat* mat ) {
    float s = 0.0f;

    // Перебор строк матрицы
    //
    for(int row=0; row<mat->rows; row++ ) {
        // Указатель на начало соответствующей строки
        //
        const float* ptr = (const float*)(mat->data.ptr + row * mat->step);

        // Перебор элементов строки
        //
        for( col=0; col<mat->cols; col++ ) {
            s += *ptr++;
        }
    }

    return( s );
}

```

На первом шаге происходит получение указателя данных и изменение его в соответствии со смещением. Как было сказано ранее, смещение в байтах. Для обеспечения безопасности, лучше всего сначала произвести арифметику над указателем, а затем произвести приведение типов (в указанном примере к типу *float*). Хотя структура *CvMat* и содержит поля *width* и *height*, для совместимости со старой структурой *Image*, предпочтительней использовать поля *rows* и *cols*. В заключении, обратите внимание на то, что *ptr* пересчитывается для каждой строки, а не просто берется с самого начала и затем смещается. Это может показаться лишним, но, указывая на *ROI* внутри большого массива, нет никакой гарантии, что данные будут непрерывны по строкам.

Массив точек

Один вопрос который может часто возникать - и который важно понимать – сводится к разнице между многомерным массивом (или матрицей) из многомерных элементов, и многомерным массивом с одномерными элементами. Предположим есть *n* точек в трехмерном измерении, которые необходимо передать некоторой функции OpenCV в виде *CvMat** (скорее *CvArr**). Есть четыре очевидных способа сделать это, при этом не все эквиваленты друг другу. Первый способ: использовать двумерный массив типа *CV32FC1* из *n* строк и трех столбцов (*nx3*). Второй способ: использовать двумерный массив типа *CV32FC1* с 3 строками и *n* столбцов (*3xn*). Третий и четвертый способы: использовать массив типа *CV32FC3* с *n* строк и одного столбца (*nx1*) или массив с одной строкой и *n* столбцов (*1xn*). В некоторых случаях эти способы взаимозаменяемы, чтобы понять почему, посмотрим на рисунок 3-2.

n-by-1 (32FC3)							1-by-n (32FC3)						
points	x_0	y_0	z_0	x_1	y_1	z_1	x_0	y_0	z_0	x_1	y_1	z_1	
$(x_0 \ y_0 \ z_0)$	x_2	y_2	z_2	x_3	y_3	z_3	x_2	y_2	z_2	x_3	y_3	z_3	
$(x_1 \ y_1 \ z_1)$	x_4	y_4	z_4	x_5	y_5	z_5	x_4	y_4	z_4	x_5	y_5	z_5	
$(x_2 \ y_2 \ z_2)$	x_6	y_6	z_6	x_7	y_7	z_7	x_6	y_6	z_6	x_7	y_7	z_7	
$(x_3 \ y_3 \ z_3)$	x_8	y_8	z_8	x_9	y_9	z_9	x_8	y_8	z_8	x_9	y_9	z_9	
$(x_4 \ y_4 \ z_4)$													
$(x_5 \ y_5 \ z_5)$													
n-by-3 (32FC1)							3-by-n (32FC1)						
	x_0	y_0	z_0	x_1	y_1	z_1	x_0	x_1	x_2	x_3	x_4	x_5	
$(x_6 \ y_6 \ z_6)$	x_2	y_2	z_2	x_3	y_3	z_3	x_6	x_7	x_8	x_9	y_0	y_1	
$(x_7 \ y_7 \ z_7)$	x_4	y_4	z_4	x_5	y_5	z_5	y_2	y_4	y_4	y_5	y_6	y_7	
$(x_8 \ y_8 \ z_8)$	x_6	y_6	z_6	x_7	y_7	z_7	y_8	y_9	z_0	z_1	z_2	z_3	
$(x_9 \ y_9 \ z_9)$	x_8	y_8	z_8	x_9	y_9	z_9	z_4	z_5	z_6	z_7	z_8	z_9	

Рисунок 3-2. Набор из 10 точек, каждая из которых представлена тремя числами типа *float*; в трех случаях из четырех распределение памяти одинаково

Как можно видеть на рисунке, точки отображаются в памяти одинаковым образом в трех из четырех случаев, описанных ранее. Ситуация становится еще более запутанной для N-мерного массива с-мерных точек. Главное помнить, что расположение точек определяется по формуле:

$$\delta = (\text{row}) \cdot \text{Ncols} \cdot \text{Nchannels} + (\text{col}) \cdot \text{Nchannels} + (\text{channel})$$

где *Ncols* и *Nchannels* количество колонок и каналов, соответственно. По этой формуле можно сказать, что N-мерный массив с с-мерными элементами не то же самое, как (N+c)-мерный массив одномерных объектов. В частном случае при N=1 (т.е. вектор вида $1 \times n$ или $n \times 1$) матрица является специально вырожденной (данний случай представлен на рисунке 3-2), что может быть иногда использовано для повышения производительности.

В заключении пару слов о типах *CvPoint2D* и *CvPoint2D32f*. Эти типы данных определены как структуры С и, следовательно, имеют строго определенную компоновку памяти. В частности, числа типа *int* или *float*, которые содержат эти структуры образуют последовательный "канал". Как результат одномерный массив в стиле С этих объектов имеет такое же распределение памяти, как и n-by-1 или 1-by-n массивы типа *CV32FC2*. Тоже самое справедливо и для массивов структур типа *CvPoint3D32f*.

[П]|[РС]|(РП) Структура IplImage

Теперь, со всеми ранее полученными знаниями, можно перейти к изучению структуры *IplImage*. В сущности, это объект *CvMat*, но с некоторыми дополнениями для интерпретации матрицы как изображение. Изначально структура была частью библиотеки IPL (Intel's Image Processing). Подробное определение структуры *IplImage* отображает пример 3-10.

Пример 3-10. Заголовок структуры *IplImage*

```
typedef struct _IplImage {
    int      nSize;          // sizeof(IplImage)
    int      ID;             // Версия (=0)
    int      nChannels;       // Число каналов
    int      alphaChannel;   // Не используется в OpenCV
    int      depth;           // Глубина в битах: IPL_DEPTH_8U, IPL_DEPTH_8S,
                           // IPL_DEPTH_16S, IPL_DEPTH_32S, IPL_DEPTH_32F и
                           // IPL_DEPTH_64F
    char    colorModel[4];    // Не используется в OpenCV
    char    channelSeq[4];    // Не используется в OpenCV
    int      dataOrder;        // Расположение каналов.
                           // 0 - чередование цветных каналов
                           // 1 - раздельное расположение цветных каналов
    int      origin;          // Начало координат
                           // 0 - верхний левый угол
                           // 1 - нижний левый угол
    int      align;            // Выравнивание строк изображения (4 или 8)
                           // В OpenCV используется widthStep
    int      width;            // Ширина в пикселях
    int      height;           // Высота в пикселях
    struct _IplROI*          roi;              // ROI. Если NULL, то выделяется все изображение
    struct _IplImage*         maskROI;          // может быть NULL
    void*               imageId;          // может быть NULL
    struct _IplTileInfo*      tileInfo;         // может быть NULL
    int                  imageSize;        // Память выделенная под изображение в байтах
                           // == image->height*image->widthStep в случае
                           // чередования данных
    char*               imageData;        // Указатель на данные изображения
    int                  widthStep;         // Число байт в одной строке изображения
    int                  BorderMode[4];     // Не используется в OpenCV
    int                  BorderConst[4];    // Не используется в OpenCV
    char*               imageDataOrigin; // Используется для правильного освобождения памяти
} IplImage;
```

Как бы безумно это не прозвучало, но стоит уделить некоторое время обсуждению некоторых переменных из этой структуры. Некоторые из них просты, однако, многие очень важны для понимания принципов работы OpenCV с изображениями.

После вездесущих переменных ширины и высоты, наиболее важными являются переменные глубина и кол-во каналов. Переменная глубина принимает одно из множества значений, определенных в *ipl.h*, которые (к сожалению) не в полной мере были описаны в главе про матрицы. Это все из-за того, что в случае с изображением приходиться иметь дело с глубиной и количеством каналов по отдельности (тогда, как в случае с матрицами со всем сразу). Возможные значения глубины приведены в таблице 3-2.

Таблица 3-2. Типы изображений в OpenCV

Макрос	Тип изображения
IPL_DEPTH_8U	Без знаковое 8-битное целое (8u)
IPL_DEPTH_8S	Знаковое 8-битное целое (8s)
IPL_DEPTH_16S	Знаковое 16-битное целое (16s)
IPL_DEPTH_32S	Знаковое 32-битное целое (32s)
IPL_DEPTH_32F	32-битное типа float (32f)
IPL_DEPTH_64F	64-битное типа double (64f)

Возможное число каналов - 1, 2, 3, 4.

Следующие немало важные поля структуры *origin* и *dataOrder*. Переменная *origin* может принимать одно из двух значений: *IPL_ORIGIN_TL* или *IPL_ORIGIN_BL*, что соответствует верхнему левому или нижнему левому углу изображения, соответственно. Отсутствие стандартного подхода (верхний против нижнего) является важным источником ошибок в процедурах компьютерного зрения. В частности, в зависимости от того, откуда пришло изображение, какие операционная система, кодек, формат хранения и т.д. – все это влияет на расположение начала координат конкретного изображения. Например, можно подумать, что выборка из пикселей находится на лицевой стороне в верхнем квадранте изображения, в действительности же на изнаночной стороне в нижней четверти. Лучше всего проверять этот момент перед выводом изображения на экран.

Параметр *dataOrder* может принимать значения *IPL_DATA_ORDER_PIXEL* или *IPL_DATA_ORDER_PLANE*. Эти значения указывают каким образом должны быть упакованы данные по каналам, пиксель за пикселим (чредуются или стандартно).

Параметр *widthStep* содержит количество байт между пикселями одного столбца (количество байт в одной строке). Переменной ширины не достаточно для того, чтобы рассчитать это расстояние, потому что каждая строка может быть выравнена с определенным числом байт для достижения более быстрой обработки изображения; соответственно, могут существовать зазоры между окончанием i-ой строки и началом (i+1) строки.

Параметр *imageData* содержит указатель на первую строку данных изображения. Если изображение содержит несколько плоскостей (когда *dataOrder* = *IPL_DATA_ORDER_PLANE*), то они размещаются последовательно, в виде отдельных изображений с *height*nChannels* строками, но, как правило, они чередуются таким образом, что количество строк равно высоте, а каждая строка содержит чередующие каналы. OpenCV поддерживает только *IPL_DATA_ORDER_PIXEL*, в то время как IPL/IPP поддерживает *IPL_DATA_ORDER_PIXEL* и *IPL_DATA_ORDER_PLANE*.

В заключении, регион интереса (ROI) также очень важен и является на самом деле экземпляром другой структуры, *IplROI*. *IplROI* содержит *xOffset*, *yOffset*, *width*, *height* и *coi* (канал интересов). Идея использования ROI состоит в том, что функции работают с частью, указанной в ROI, изображения, а не со всем изображением. Все функции OpenCV работают с ROI, если он установлен. Если COI не NULL, то некоторые операции будут выполняться только на указанном канале, однако, многие функции OpenCV игнорируют этот параметр.

Доступ к данным изображения

Обычно требуется обрабатывать данные изображения быстро и эффективно. Это означает, что не стоит использовать функции вида *cvSet*D* или аналогичные. На самом деле, лучший путь получения данных изображения, получать данные на прямую. Теперь, зная внутреннее устройство структуры *IplImage*, можно получать данные изображения быстрее и эффективнее.

Однако, даже при наличии в OpenCV хорошо оптимизированных процедур, выполняющих большой спектр задач, всегда найдутся такие задачи, с которыми эти процедуры не справятся. Рассмотрим случай трехканального HSV изображения, в котором будем изменять насыщенность и значение цвета от 0 до 255 (максимальное значение для 8-битного изображения), оставляя оттенок неизменным. Лучше всего сделать это с использованием указателя, также как это было сделано с матрицами в примере 3-9. Тем не менее есть незначительные отличия, которые связаны с различиями в устройстве структур *IplImage* и *CvMat*. Пример 3-11 отражает наиболее быстрый способ обработки трехканального HSV изображения.

Пример 3-11. Установка "S" и "V" составляющих HSV изображения в 255

```

void saturate_sv( IplImage* img ) {
    for( int y=0; y<img->height; y++ ) {
        uchar* ptr = (uchar*) (
            img->imageData + y * img->widthStep
        );
        for( int x=0; x<img->width; x++ ) {
            ptr[3*x+1] = 255;
            ptr[3*x+2] = 255;
        }
    }
}

```

В начале происходит вычисление указателя *ptr*, который указывает на начало соответствующей строки *y*. Затем значение насыщенности устанавливается в 255. Так, как это трехканальное изображение, расположение канала *c* в колонке *x* вычисляется как $3x+c^*$.

Одно важное отличие *IplImage* от *CvMat* заключается в поведении *imageData*. Данные *CvMat* являются объединением, поэтому существует возможность задать тип указателя. Указатель *imageData* задается жестко как *uchar**. Уже известно, что указатель данных не обязательно может быть типа *uchar*, в случае же с изображением арифметика над указателем сводится к добавлению к указателю значения *widthStep*, который так же в байтах, и потому уже не нужно беспокоиться о приведении типов, после получения результата. При работе с матрицами, необходимо уменьшать смещение, так как указатель данных не всегда может быть типа *byte*, в то время, как указатель данных изображения всегда типа *byte*, смещение можно использовать "как есть".

Подробнее о *ROI* и *widthStep*

ROI и *widthStep* имеют большое практическое значение, так как во многих ситуациях ускоряют операции компьютерного зрения, за счет обработки части изображения. Все функции OpenCV поддерживают *ROI* и *widthStep*. Для включения или выключения поддержки *ROI*, используются функции *cvSetImageROI()* и *cvResetImageROI()*. Т.к. выделение части изображения имеет вид прямоугольника, нужно использовать структуру *CvRect*, которую можно передать в функцию *cvSetImageROI()* для "включения *ROI*"; для "выключения *ROI*" необходимо передать указатель на изображение в функцию *cvResetImageROI()*.

```

void cvSetImageROI( IplImage* image, CvRect rect );
void cvResetImageROI( IplImage* image );

```

Чтобы увидеть, как используется ROI, рассмотрим пример загрузки и выделения некоторой области изображения. Код из примера 3-12 загружает изображение, устанавливает *x*, *y*, *width*, *height* предполагаемого ROI и объявляет целое число, для изменения области ROI. Затем задается область ROI, при помощи конструктора *cvRect()*. Важно выключить область ROI с помощью функции *cvResetImageROI* перед выводом изображения на экран, иначе в вывод попадет часть (область ROI) изображения.

Пример 3-12. Использование ROI для увеличения значения пикселей в области изображения

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv ) {
    IplImage* src;

    // argv[1] - путь до изображения
    //
    if( argc == 7 && ((src=cvLoadImage(argv[1],1)) != 0) ) {
        int x = atoi(argv[2]);           // Отступ от левого верхнего угла изображения по оси
        int y = atoi(argv[3]);           // Отступ от левого верхнего угла изображения по оси
        int width = atoi(argv[4]);       // Ширина ROI
        int height = atoi(argv[5]);      // Высота ROI
        int add = atoi(argv[6]);         // Значение увеличения

        // Захват области изображения. Установка ROI
        //
        cvSetImageROI( src, cvRect(x,y,width,height) );

        // Сложение скаляра с изображением
        //
        cvAddS( src, cvScalar(add),src );

        // Сброс ROI
        //
        cvResetImageROI( src );
    }

    return 0;
}
```

Рисунок 3-3 отображает результат выполнения примера 3-12 с добавлением *add=150* к синему каналу изображения кошки с центром ROI на лице.



Рисунок 3-3. Результат добавления 150 к синему каналу изображения

Такого же эффекта можно добиться и с использованием *widthStep*. Чтобы сделать это, необходимо создать дополнительный заголовок изображения и установить ширину и высоту в точности как у *interest_rect*. Кроме этого, необходимо установить начало координат изображения (левый верхний или левый нижний угол) в такое же положение, как у *interest_rect*. Так же установить *widthStep* в значение, равное значению *widthStep* у *interest_rect*. В заключении, установить указатель данных на начало запрашиваемой области изображения.

Пример 3-13. Использование альтернативного метода с использованием *widthStep* для изменения пикселей области изображения *interest_img*

```

// interest_img - исходное изображение
// interest_rect - запрашиваемая область
// sub_img - контейнер под запрашиваемую область

// Создание заголовка изображения (контейнер)
//
IplImage *sub_img = cvCreateImageHeader(
    cvSize(
        interest_rect.width
        ,interest_rect.height
    )
    ,interest_img->depth
    ,interest_img->nChannels
);

// Начало координат
//
sub_img->origin = interest_img->origin;

// Шаг
//
sub_img->widthStep = interest_img->widthStep;

// Установка указателя данных на начало (в соответствии с началом координат)
//
sub_img->imageData = interest_img->imageData +
    interest_rect.y * interest_img->widthStep +
    interest_rect.x * interest_img->nChannels;

// Изменение пикселей запрашиваемой области
//
cvAddS( sub_img, cvScalar(1), sub_img );

// Освобождение памяти, занимаемой контейнером
//
cvReleaseImageHeader( &sub_img );

```

Теперь возникает вопрос, когда использовать метод с *widthStep*, а когда с ROI? Если требуется обрабатывать несколько регионов за раз, то используйте первый метод, т.к. использование ROI подразумевает захват и освобождение запрашиваемой области для каждого региона, что сказывается на производительности программы.

В заключение, пару слов о масках. Функция *cvAddS()* позволяет использовать четыре аргумента, где маска по умолчанию NULL: *const CvArr* mask = NULL*. Это 8-битный одноканальный массив, который позволяет ограничить область обработки произвольной формы, ограниченная ненулевыми пикселями маски. Если ROI устанавливается вместе с маской, область обработки будет ограничена пересечением маски и ROI. Не все функции поддерживают маски.

[П]||[РС]||(РП) Операции над матрицами и изображениями

В таблице 3-3 перечислены различные процедуры манипуляции с матрицами, многие из которых также хорошо работают и с изображениями. Они выполняют такие "обычные" вещи, как приведение к диагональной форме и транспонирование матрицы, а также некоторые более сложные операции, такие как вычисление статистических данных изображения.

Таблица 3-3. Базовые операции

Функция	Описание
cvAbs	Абсолютное значение всех элементов матрицы
cvAbsDiff	Абсолютное значение разницы между двумя элементами матрицы
cvAbsDiffS	Абсолютное значение разницы между элементами матрицы и скаляром
cvAdd	Поэлементное складывание двух матриц
cvAddS	Поэлементное складывание элементов матрицы и скаляра
cvAddWeighted	Поэлементное взвешенное складывание двух матриц (альфа-смешивание)
cvAvg	Среднее значение всех элементов матрицы
cvAvgSdv	Абсолютное значение стандартного отклонения всех элементов матрицы
cvCalcCovarMatrix	Вычисление ковариации набора n-мерных векторов
cvCmp	Применение выбранной операции сравнения для всех элементов в двух матрицах
cvCmpS	Применение выбранной операции сравнения отношения матрицы и скаляра
cvConvertScale	Изменение типа матрицы с изменением масштаба
cvConvertScaleAbs	Абсолютное изменение типа матрицы с изменением масштаба
cvCopy	Копирование элементов из одной матрицы в другую
cvCountNonZero	Количество ненулевых элементов матрицы
cvCrossProduct	Вычисление векторного произведения двух трехмерных векторов

cvCvtColor	Преобразование каналов матрицы из одного цветового пространства в другое
cvDet	Вычисление определителя квадратной матрицы
cvDiv	Поэлементное деление одной матрицы на другую
cvDotProduct	Вычисление скалярного произведения двух векторов
cvEigenVV	Вычисление собственных чисел и собственных векторов квадратной матрицы
cvFlip	Обратный порядок матрицы по выбранной оси
cvGEMM	Обобщенная матрица умножения
cvGetCol	Копирование элементов колонки частичной матрицы
cvGetCols	Копирование элементов из нескольких соседних столбцов матрицы
cvGetDiag	Копирование элементов из диагональной матрицы
cvGetDims	Возвращает количество измерений матрицы
cvGetDimSize	Возвращает размеры всех измерений матрицы
cvGetRow	Копирование элементов строки частичной матрицы
cvGetRows	Копирование элементов из нескольких соседних строк матрицы
cvGetSize	Получение размера двумерной матрицы и возвращение в виде CvSize
cvGetSubRect	Копирование элементов из выделенного региона изображения
cvInRange	Тестирование попадания элементов матрицы в пределы значений двух других матриц
cvInRangeS	Тестирование попадания элементов матрицы между двумя скалярами
cvInvert	Невырожденная матрица
cvMahalanobis	Вычисление расстояния Махalanобиса между двумя векторами
cvMax	Поиск максимального значения, среди двух матриц
cvMaxS	Поиск максимального значения, среди матрицы и скаляра
cvMerge	Слияние нескольких одноканальных изображений в одно многоканальное
cvMin	Поиск минимального значения, среди двух матриц
cvMinS	Поиск минимального значения, среди матрицы и скаляра
cvMinMaxLoc	Поиск максимального и минимального значения матрицы

cvMul	Поэлементное перемножение двух матриц
cvNot	Побитовое инвертирование всех элементов матрицы
cvNorm	Вычисление нормализованной взаимокорреляционной функции между двумя матрицами
cvNormalize	Нормализация элементов массива до некоторого значения
cvOr	Поэлементная побитовая операция OR между двумя матрицами
cvOrS	Поэлементная побитовая операция OR между матрицей и скаляром
cvReduce	Преобразование двумерной матрицы в вектор
cvRepeat	Заполнение выходной матрицы копиями входной матрицы
cvSet	Установка всех элементов матрицы в заданное значение
cvSetZero	Установка всех элементов матрицы в 0
cvSetIdentity	Установка диагональных элементов матрицы в 1, остальных в 0
cvSolve	Решение системы линейных уравнений
cvSplit	Разделение многоканального изображения на несколько одноканальных
cvSub	Поэлементное вычитание одной матрицы из другой
cvSubS	Поэлементное вычитание скаляра из матрицы
cvSubRS	Поэлементное вычитание матрицы из скаляра
cvSum	Суммирование всех элементов матрицы
cvSVD	Вычисление сингулярного разложения двумерной матрицы
cvSVBkSb	Вычисление сингулярного разложения методом обратной замены
cvTrace	Вычисление следа матрицы
cvTranspose	Транспонирование всех элементов матрицы через диагональ
cvXor	Поэлементная битовая операция XOR между двумя матрицами
cvXorS	Поэлементная битовая операция XOR между матрицей и скаляром
cvZero	Установка всех элементов матрицы в 0

cvAbs, cvAbsDiff и cvAbsDiffS

```

void cvAbs(
    const CvArr* src
, const CvArr* dst
);

void cvAbsDiff(
    const CvArr* src1
, const CvArr* src2
, const CvArr* dst
);

void cvAbsDiffS(
    const CvArr* src
, CvScalar value
, const CvArr* dst
);

```

Функция `cvAbs()` вычисляет абсолютные значения элементов `src` и записывает результат в `dst`. Функция `cvAbsDiff()` сначала вычисляет разницу между `src2` и `src1` и потом записывает абсолютное значение в `dst`. Функция `cvAbsDiffS()` делает почти тоже самое, что и `cvAbsDiff()`, только разница находится между элементами `src` и значением скаляра `value`.

`cvAdd`, `cvAddS`, `cvAddWeighted` и альфа-смешивание

```

void cvAdd(
    const CvArr* src1
, const CvArr* src2
, CvArr* dst
, const CvArr* mask = NULL
);

void cvAddS(
    const CvArr* src
, CvScalar value
, CvArr* dst
, const CvArr* mask = NULL
);

void cvAddWeighted(
    const CvArr* src1
, double alpha
, const CvArr* src2
, double beta
, double gamma
, CvArr* dst
);

```

Функция `cvAdd()` складывает все элементы из `src1` с элементами из `src2` и помещает результат в `dst`. Если `mask` не `NULL`, то некоторые элементы `dst` останутся неизменными там, где элемент маски равен 0. Функция `cvAddS()` делает почти тоже самое, что и `cvAdd()`, за исключением того, что значение скаляра добавляется к каждому элементу `src`.

Функция `cvAddWeighted()` аналогична `cvAdd()` за исключением того, что результат записывается в `dst` и вычисляется по формуле:

$$dst(x,y) = \alpha \cdot src1(x,y) + \beta \cdot src2(x,y) + \gamma$$

Эта функция может быть использована для реализации альфа-смешивания. Функция принимает два источника изображения `src1` и `src2`. Эти изображения могут быть различного типа, но при этом оба должны быть одинакового типа. Они также могут быть одноканальными и трехканальными (серыми или цветными), при этом оба изображения с одинаковым кол-вом каналов. Результирующее изображение `dst` также должно быть того же типа, что и исходные изображения. Исходные изображения могут быть разных размеров, поэтому предварительно нужно согласовать этот момент при помощи ROI, иначе OpenCV выдаст ошибку. Параметр `alpha` альфа смешивания относится к `src1`, а `beta` бетта смешивания к `src2`. Уравнение альфа смешивания:

$$dst(x,y) = \alpha \cdot src1(x,y) + \beta \cdot src2(x,y) + \gamma$$

Стандартное уравнение альфа смешивания получается путем выбора α между 0 и 1, $\beta=1-\alpha$, $\gamma=0$:

$$dst(x,y) = \alpha \cdot src1(x,y) + (1-\alpha) \cdot src2(x,y)$$

В общем случае, требуется, чтобы `alpha` и `beta` были больше 0, а их сумма не больше 1; `gamma` может быть установлена в зависимости от среднего или максимального значения изображения для увеличения масштаба пикселей.

Пример 3-14. Альфа смешивание, смещение ROI в `src2` (0,0), смещение ROI в `src1` (x,y)

```

#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv ) {
    IplImage *src1, *src2;

    // Проверка наличия изображений по указанным путям
    // Проверка наличия необходимого кол-ва аргументов
    //
    if ( ( 9 == argc )
        && ( 0 != (src1 = cvLoadImage(argv[1], 1)) )
        && ( 0 != (src2 = cvLoadImage(argv[2], 1)) ) )
    ) {
        int x = atoi(argv[3]);                                // Смещение по оси x
        int y = atoi(argv[4]);                                // Смещение по оси y
        int width = atoi(argv[5]);                            // Ширина области ROI
        int height = atoi(argv[6]);                           // Высота области ROI
        double alpha = (double)atof(argv[7]);                // Значение альфа
        double beta = (double)atof(argv[8]);                 // Значение бета

        // Установка области ROI
        // Примечание: отсутствует логика установки областей
        //             одинакового размера
        cvSetImageROI( src1, cvRect(x, y, width, height) );
        cvSetImageROI( src2, cvRect(0, 0, width, height) );

        // src1 = src1
        // src2 = src2
        // α = alpha
        // β = beta
        // γ = 0.0
        // dst = src1
        cvAddWeighted(src1, alpha, src2, beta, 0.0, src1);

        // Сброс ROI для src1
        //
        cvResetImageROI(src1);
        cvNamedWindow( "Alpha_blend", 1 );
        cvShowImage( "Alpha_blend", src1 );

        cvWaitKey();
    }

    return 0;
}

```

Результат выполнения примера 3-14 представлен на рисунке 3-4.

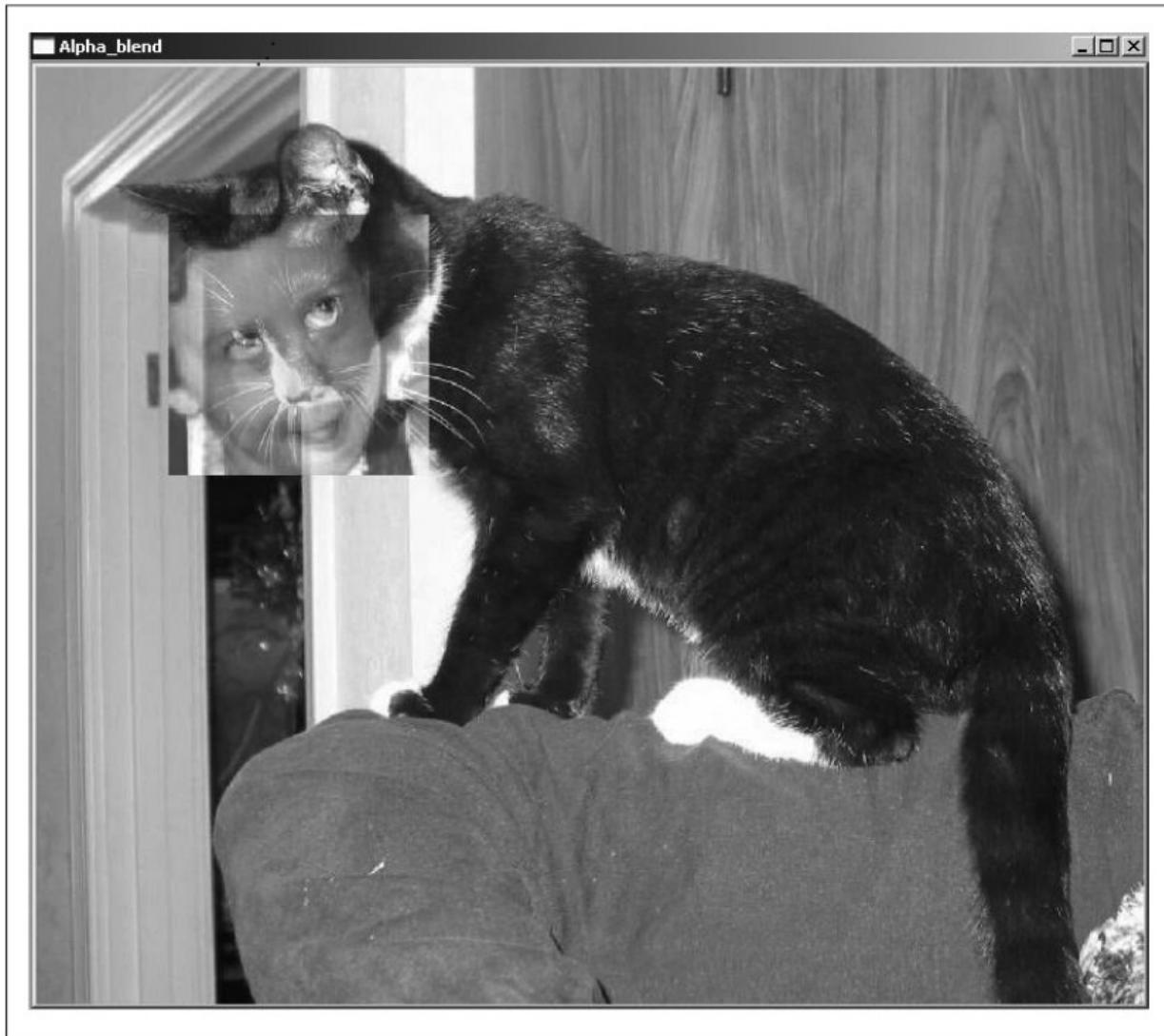


Рисунок 3-4. Лицо ребенка альфа смешано с лицом кошки

cvAnd и cvAndS

```
void cvAnd(
    const CvArr*    src1
    ,const CvArr*    src2
    ,CvArr*         dst
    ,const CvArr*    mask = NULL
);

void cvAndS(
    const CvArr*    src1
    ,CvScalar        value
    ,CvArr*         dst
    ,const CvArr*    mask = NULL
);
```

Эти две функции вычисляют побитовое *AND* на массиве *src1*. В случае *cvAdd()* каждый элемент *dst* вычисляется как побитовое *AND* двух соответствующих элементов *src1* и *src2*. В случае *cvAndS* побитовое *AND* вычисляется со значением скаляра. Если *mask* \neq *NULL*, то под вычисления попадают элементы *dst*, соответствующие ненулевым элементам маски.

src1 и *src2* должны быть любого одинакового типа данных. Если элементы вещественного типа, то результат тоже будет состоять из элементов вещественного типа.

cvAvg

```
CvScalar cvAvg(
    const CvArr* arr
    ,const CvArr* mask = NULL
);
```

Функция *cvAvg()* вычисляет среднее значение пикселей в *arr*. Если *mask* \neq *NULL*, то среднее значение будет рассчитано только на тех пикселях, для которых соответствующее значение маски отлично от нуля.

В настоящее время функция является устаревшим псевдонимом *cvMean()*.

cvAvgSdv

```
void cvAvgSdv(
    const CvArr* arr
    ,CvScalar* mean
    ,CvScalar* std_dev
    ,const CvArr* mask = NULL
);
```

Эта функция похожа на *cvAvg()*, но в дополнение к ней вычисляет еще стандартное отклонение.

В настоящее время функция является устаревшим псевдонимом *cvMean_StdDev()*.

cvCalcCovarMatrix

```
void cvAdd(
    const CvArr** vsects
    ,int count
    ,CvArr* cov_mat
    ,CvArr* avg
    ,int flags
);
```

Учитывая любое количество векторов, `cvCalcCovarMatrix()` вычисляет среднее значение и ковариационную матрицу для аппроксимации гауссового распределения этих точек. Это может пригодится при решении множества задач, на этот случай в OpenCV есть специальные флаги, которые помогут при решении конкретной задачи (таблица 3-4). Возможно объединение нескольких флагов с использованием логической операции *OR*.

Таблица 3-4. Возможные варианты флагов `cvCalcCovarMatrix()`

Наименование флага	Значение
<code>CV_COVAR_NORMAL</code>	Среднее значение и ковариационная матрица
<code>CV_COVAR_SCRAMBLED</code>	Быстрый PCA (principal component analysis - метод главных компонент)
<code>CV_COVAR_USE_AVERAGE</code>	Использование вычисленного среднего значения
<code>CV_COVAR_SCALE</code>	Масштабирование матрицы ковариаций

Во всех случаях вектора представлены как массив массивов `vects` (т.е. указатель на список указателей на массивы). Матрица ковариаций будет размещена в `cov_mat`, а значение среднего `avg` будет зависеть от флага (таблица 3-4).

Флаги `CV_COVAR_NORMAL` и `CV_COVAR_SCRAMBLED` являются взаимоисключающими; совместное использование не допустимо. В случае `CV_COVAR_NORMAL` функция вычисляет среднее значение ковариационной матрицы.

$$\Sigma_{\text{normal}}^2 = z \begin{bmatrix} v_{0,0} - \bar{v}_0 & \cdots & v_{m,0} - \bar{v}_0 \\ \vdots & \ddots & \vdots \\ v_{0,n} - \bar{v}_n & \cdots & v_{m,n} - \bar{v}_n \end{bmatrix} \begin{bmatrix} v_{0,0} - \bar{v}_0 & \cdots & v_{m,0} - \bar{v}_0 \\ \vdots & \ddots & \vdots \\ v_{0,n} - \bar{v}_n & \cdots & v_{m,n} - \bar{v}_n \end{bmatrix}^T$$

Таким образом нормальная ковариация Σ_{normal}^2 вычисляется из m векторов длиной n , где \bar{v}_n - определяет n -ый элемент среднего вектора \bar{v} . Результирующая ковариационная матрица имеет размерность $n \times n$. Коэффициент z является необязательным масштабным коэффициентом; он будет равен 1, если не использовать флаг `CV_COVAR_SCALE`.

В случае `CV_COVAR_SCRAMBLED` функция `cvCalcCovarMatrix()` вычисляет матрицу ковариаций по следующей формуле:

$$\Sigma_{\text{scrambled}}^2 = z \begin{bmatrix} v_{0,0} - \bar{v}_0 & \cdots & v_{m,0} - \bar{v}_0 \\ \vdots & \ddots & \vdots \\ v_{0,n} - \bar{v}_n & \cdots & v_{m,n} - \bar{v}_n \end{bmatrix}^T \begin{bmatrix} v_{0,0} - \bar{v}_0 & \cdots & v_{m,0} - \bar{v}_0 \\ \vdots & \ddots & \vdots \\ v_{0,n} - \bar{v}_n & \cdots & v_{m,n} - \bar{v}_n \end{bmatrix}$$

Эта матрица не является обычной ковариационной матрицей (обратите внимание на расположение оператора транспонирования). Эта матрица вычисляется из тех же m векторов длиной n , но результирующая матрица уже будет размерностью $t \times t$. Эта матрица используется, например, в быстрых PCA для очень больших векторов (техника *eigenfaces* для распознавания лица).

Флаг `CV_COVAR_USE_AVG` используется, когда среднее значение входных векторов уже известно. В этом случае аргумент `avg` используется в качестве входного сигнала, а не выходного, что снижает время выполнения вычислений.

Флаг `CV_COVAR_SCALE` применяется для равномерного масштабирования результирующей ковариационной матрицы. За это отвечает переменная `z`. При использовании в сочетании с флагом `CV_COVAR_NORMAL`, масштабный коэффициент будет равен $1.0/m$ (что эквивалентно $1.0/count$). При использовании в сочетании с флагом `CV_COVAR_SCRAMBLED`, масштабный коэффициент будет равен $1.0/n$.

Входные и выходные массивы функции `cvCalcCovarMatrix()` должны быть одинакового вещественного типа. Размер результирующей матрицы `cov_mat` должен быть $n \times n$ или $t \times t$, в зависимости от типа вычисляемой ковариационной матрицы (стандартная или нет). Стоит отметить, что входные вектора `vects` не должны быть одномерными; они могут быть и двумерными (например, в случае с изображениями).

cvCmp и cvCmpS

```
void cvCmp(
    const CvArr* src1
    ,const CvArr* src2
    ,CvArr* dst
    ,int cmp_op
);

void cvCmpS(
    const CvArr* src
    ,double value
    ,CvArr* dst
    ,int cmp_op
);
```

Обе эти функции выполняют сравнение, либо между соответствующими пикселями двух изображений, либо между пикселями изображения и значением скаляра. Функции `cvCmp()` и `cvCmpS()` принимают в качестве последнего аргумента оператор сравнения, который определен как флаг (таблица 3-5).

Таблица 3-5. Значения `cmp_op`, используемые в `cvCmp()` и `cvCmpS()`, и результат выполняемого сравнения

Значение <code>cmp_op</code>	Сравнение
<code>CV_CMP_EQ</code>	$(src1i == src2i)$
<code>CV_CMP_GT</code>	$(src1i > src2i)$
<code>CV_CMP_GE</code>	$(src1i \geq src2i)$
<code>CV_CMP_LT</code>	$(src1i < src2i)$
<code>CV_CMP_LE</code>	$(src1i \leq src2i)$
<code>CV_CMP_NE</code>	$(src1i \neq src2i)$

Данные функции работают только с одноканальными изображениями. Эти функции полезны в программах, где используется какая-то версия вычитания фона, чтобы замаскировать результат.

cvConvertScale

```
void cvConvertScale(
    const CvArr*   src
    ,CvArr*        dst
    ,double        scale = 1.0
    ,double        shift = 0.0
);
```

Функция `cvConvertScale()` на самом деле состоит из нескольких вложенных функций; при этом может быть использован весь набор вложенных функций, а может и не весь. На первом шаге происходит преобразование типа данных исходного изображения в тип результирующего изображения. Например, имея 8-битное RGB изображение в оттенках серого, можно преобразовать его в 16-битное изображение.

На втором шаге выполняется линейное преобразование данных изображения. После преобразования к новому типу данных, каждое значение пикселя будет умножено на значение `scale`, с добавлением значения смещения `shift`.

Важно понимать, что хоть операция "Конвертирования" и предшествует операции "Масштабирования" в имени функции, фактически эти операции выполняются в противоположном порядке. Это означает, что операция умножения на `scale` и

добавления *shift* происходит до операции преобразования типов.

Использование значений *scale* и *shift* по умолчанию (*scale* = 1.0, *shift* = 0.0) приведет к пропуску соответствующих операций. Для такого случая в OpenCV предусмотрен макрос *CvConvert()*. Функция *cvConvertScale()* может работать со всеми типами данных и любым количеством каналов, однако, количество каналов в исходном изображении и результирующем должно быть одинаково. (Если же требуется преобразовать цветное изображение в изображение с оттенками серого, то необходимо использовать функцию *cvCvtColor()*).

cvConvertScaleAbs

```
void cvConvertScaleAbs(
    const CvArr*   src
    ,CvArr*        dst
    ,double         scale = 1.0
    ,double         shift = 0.0
);
```

Функция *cvConvertScaleAbs()* идентична *cvConvertScale()*, за исключением того, что *dst* содержит абсолютные значения полученных данных. Это означает, что функция *cvConvertScaleAbs()* сначала применяет *scale* и *shift*, затем вычисляет абсолютное значение и в завершении выполняет преобразование типов.

cvCopy

```
void cvCopy(
    const CvArr*   src
    ,CvArr*        dst
    ,const CvArr*  mask = NULL
);
```

Функция выполняет копирование одного изображения в другое. Оба массива должны быть одинакового типа, размера и иметь одинаковое количество измерений. Данную функцию можно использовать для копирования разряженных массивов, но при этом невозможно задействовать маску. Для разряженных массивов и изображений, эффект от маски (если *mask* != *NULL*) будет такой, что будут изменены только те пиксели в *dst*, которым соответствуют ненулевые элементы в маске.

cvCountNonZero

```
int cvCountNonZero( const CvArr* arr );
```

Функция *cvCountNonZero()* возвращает количество ненулевых пикселей в массиве *arr*.

cvCrossProduct

```
void cvCrossProduct(
    const CvArr* src1
    ,const CvArr* src2
    ,CvArr* dst
);
```

Эта функция вычисляет векторное произведение двух трехмерных векторов. Не имеет значение, вектор-столбец или вектор-строка. *src1* и *src2* должны быть одноканальными массивами, массив *dst* должен быть одноканальным и длиною точно 3. Все три массива должны быть одного типа.

cvCvtColor

```
void cvCvtColor(
    const CvArr* src
    ,CvArr* dst
    ,int code
);
```

Ранее была представлена функция, которая преобразует из одного типа данных в другой, при этом количество каналов должно быть одинаково для исходного и результирующего изображений. Функция *cvCvtColor()* преобразует из одного цветового пространства в другое, при этом тип данных исходного и результирующего изображений должны совпадать. Флаг *code*, значения которого перечислены в таблице 3-6, задает процесс преобразования для конкретного случая.

Таблица 3-6. Значения флага для функции *cvCvtColor()*

Флаг	Значение
CV_BGR2RGB	Преобразования между RGB и BGR (с и без альфа канала)
CV_RGB2BGR	
CV_RGBA2BGRA	
CV_BGRA2RGBA	
CV_RGB2RGBA	Добавление альфа канала к RGB или BGR изображению
CV_BGR2BGRA	
CV_RGBA2RGB	Удаление альфа канала из RGB или BGR изображения
CV_BGRA2BGR	
CV_RGB2BGRA	Преобразования RGB в BGR при добавлении или удалении альфа канала

CV_RGBA2BGR	
CV_BGRA2RGB	
CV_BGR2RGBA	
CV_RGB2GRAY	Преобразование RGB или BGR в оттенки серого
CV_BGR2GRAY	
CV_GRAY2RGB	Преобразование из оттенков серого в RGB или BGR (необязательное удаление альфа канала в процессе преобразования)
CV_GRAY2BGR	
CV_RGBA2GRAY	
CV_BGRA2GRAY	
CV_GRAY2RGBA	Преобразование из оттенков серого в RGB или BGR и добавление альфа канала
CV_GRAY2BGRA	
CV_RGB2BGR565	Преобразование из RGB или BGR в BGR565 с добавлением или удалением альфа канала (16-битные изображения)
CV_BGR2BGR565	
CV_BGR5652RGB	
CV_BGR5652BGR	
CV_RGBA2BGR565	
CV_BGRA2BGR565	
CV_BGR5652RGBA	
CV_BGR5652BGRA	
CV_GRAY2BGR565	Преобразование из оттенков серого в BGR565 и наоборот (16-битные изображения)
CV_BGR5652GRAY	
CV_RGB2BGR555	Преобразование из RGB или BGR в BGR555 с необязательным добавлением или удалением альфа канала (16-битные изображения)
CV_BGR2BGR555	
CV_BGR5552RGB	
CV_BGR5552BGR	
CV_RGBA2BGR555	
CV_BGRA2BGR555	

CV_BGR5552RGBA	
CV_BGR5552BGRA	
CV_GRAY2BGR555	Преобразование из оттенков серого в BGR555 и наоборот (16-битные изображения)
CV_BGR5552GRAY	
CV_RGB2XYZ	Преобразование RGB или BGR в CIE XYZ и наоборот (Rec 709 with D65 white point)
CV_BGR2XYZ	
CV_XYZ2RGB	
CV_XYZ2BGR	
CV_RGB2YCrCb	Преобразование RGB или BGR в luma-chroma (также известный как YCC)
CV_BGR2YCrCb	
CV_YCrCb2RGB	
CV_YCrCb2BGR	
CV_RGB2HSV	Преобразование RGB или BGR в HSV (тон, насыщенность, значение) и наоборот
CV_BGR2HSV	
CV_HSV2RGB	
CV_HSV2BGR	
CV_RGB2HLS	Преобразование RGB или BGR в HLS (тон, светлота, насыщенность) и наоборот
CV_BGR2HLS	
CV_HLS2RGB	
CV_HLS2BGR	
CV_RGB2Lab	Преобразование RGB или BGR в CIE Lab и наоборот
CV_BGR2Lab	
CV_Lab2RGB	
CV_Lab2BGR	
CV_RGB2Luv	Преобразование RGB или BGR в CIE Luv
CV_BGR2Luv	
CV_Luv2RGB	
CV_Luv2BGR	
	Преобразование из Bayer pattern (один канал) в RGB или

	BGR
CV_BayerGB2RGB	
CV_BayerRG2RGB	
CV_BayerGR2RGB	
CV_BayerBG2BGR	
CV_BayerGB2BGR	
CV_BayerRG2BGR	
CV_BayerGR2BGR	

Реализация многих из этих преобразований нетривиальны. OpenCV содержит достаточный инструментарий для преобразования "в" и "из" этих различных цветовых пространств.

При преобразованиях цветовое пространство (диапазон значений) изображений должно соответствовать следующим правилам: для 8-битных изображений 0-255; для 16-битных изображений 0-65536; для вещественных 0.0-1.0. Когда изображения в оттенках серого преобразуются в цветные изображения, все компоненты результирующего изображения берутся равными; в противном случае (из RGB или BGR в оттенки серого), значения результирующего изображения вычисляются по перцептивно взвешенной формуле:

$$Y = (0.299)R + (0.587)G + (0.114)B$$

В случае HSV и HLS, тон представляет диапазон [0, 360). Это может привести к проблемам с 8-битными изображениями, поэтому при преобразовании к HSV, тон делится на 2, когда на выходе необходимо получить 8-битное изображение.

cvDet

```
double cvDet( const CvArr* mat );
```

Функция `cvDet()` вычисляет детерминант квадратной матрицы. Матрица может быть любого типа, но только одноканальной. Если матриц мала, то определитель вычисляется по стандартной формуле. Для больших матриц это не особо эффективно, поэтому используется метод Гаусса.

При этом стоит отметить, что если матрица симметрична и имеет положительный определитель, можно воспользоваться решением с помощью сингулярного разложения (SVD). Для получения более подробной информации обратитесь к функции `cvSVD()`.

cvDiv

```
void cvDiv(
    const CvArr*    src1
    const CvArr*    src2
    CvArr*          dst
    double          scale = 1
);
```

cvDiv() это функция деления; она просто поэлементно делит элементы *src1* на элементы *src2* и записывает результат в *dst*. Если *mask* != *NULL*, то любые элементы в *dst*, соответствующие нулевому элементу маски, не подвергаются операции деления. Если потребуется инвертировать все элементы матрицы *src2*, то передайте *src1*, у которого все элементы *NULL*-ы; процедура расценит *src1* как массив с элементами 1s.

cvDotProduct

```
double cvDotProduct(
    const CvArr*    src1
    ,const CvArr*    src2
);
```

Эта функция вычисляет вектор скалярного произведения двух N-мерных векторов. Не имеет значение, если это будут вектор-строка или вектор-столбец. *src1* и *src2* должны быть одноканальными и однотипными.

cvEigenVV

```
double cvEigenVV(
    CvArr* mat
    ,CvArr* evecs
    ,CvArr* evals
    ,double eps = 0
);
```

Имея симметричную матрицу *mat*, функция *cvEigenVV()* вычисляет собственные вектора и соответствующие собственные числа матрицы. Для небольших матриц используется метод *Jacobi*. Этот метод запрашивает параметр прекращения, который является максимальным значением размера недиагональных элементов в результирующей матрице. Необязательный параметр *eps* устанавливает этот параметр прекращения. В процессе вычислений, значения матрицы *mat* будут изменены. При завершении выполнения функции, собственные вектора будут помещены в

переменную `evecs`; собственные числа в переменную `evals`. Порядок собственных векторов всегда будет в порядке убывания величин собственных чисел. На вход функции `cvEigenVV()` необходимо передавать сразу три массива.

Также, как и в случае с функцией `cvDet()`, если известно, что матрица симметрична и имеет положительный определитель, то лучше использовать `SVD` для нахождения собственных векторов и собственных чисел.

cvFlip

```
void cvFlip(
    const CvArr*     src
    CvArr*           dst = NULL
    int              flip_mode = 0
);
```

Эта функция переворачивает изображение вокруг оси x, оси y или обоих. В частности, если аргумент `flip_mode` равен 0, то изображение будет перевернуто по оси x. Если `flip_mode` равен положительному числу (например, +1), изображение будет повернуто по оси y. Если `flip_mode` равен отрицательному числу (например, -1) изображение будет повернуто по обоим осям.

При обработки видео в системах *Win32*, необходимо самостоятельно переключаться между форматами изображения, в зависимости от расположения начала координат (верхний левый или нижний левый угол).

cvGEMM

```
double cvGEMM(
    const CvArr*     src1
    ,const CvArr*     src2
    ,double           alpha
    ,const CvArr*     src3
    ,double           beta
    ,CvArr*           dst
    ,int              tABC = 0
);
```

Обобщенная матрица умножения (*GEMM*) в OpenCV представлена функцией `cvGEMM()`; она выполняет умножение матриц, транспонированное умножение, масштабированное умножение и т.д. В наиболее общей форме, `cvGEMM()` вычисляется по следующей формуле:

$$D = \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot \text{op}(C)$$

A , B и C матрицы $src1$, $src2$ и $src3$ соответственно, α и β числовые коэффициенты, $op()$ необязательное транспонирование матрицы. Аргумент $src3$ может быть установлен в `NULL`. Транспонирование контролируется с помощью дополнительного аргумента $tABC$, который может быть равен 0 или любой комбинации (с помощью OR) из флагов `CV_GEMM_A_T`, `CV_GEMM_B_T` и `CV_GEMM_C_T` (каждый флаг указывает транспонирование соответствующей матрицы).

Ранее OpenCV содержал функции `cvMatMul()` и `cvMatMulAdd()`, но это часто приводило к путанице с функцией `cvMul()`. В настоящее время эти функции существуют в виде макросов, которые неявно вызывают `cvGEMM()` (таблица 3-7).

Таблица 3-7. Макросы для обобщенной функции `cvGEMM()`

Макрос	<code>cvGEMM</code>
<code>cvMatMul(A, B, D)</code>	<code>cvGEMM(A, A, 1, NULL, 0, D, 0)</code>
<code>cvMatMulAdd(A, B, C, D)</code>	<code>cvGEMM(A, A, 1, C, 1, D, 0)</code>

Все матрицы должны быть соответствующего размера и вещественного типа. Функция `cvGEMM()` поддерживает двухканальные матрицы.

`cvGetCol` и `cvGetCols`

```
CvMat* cvGetCol(
    const CvArr* arr
    ,CvMat* submat
    ,int col
);

CvMat* cvGetCols(
    const CvArr* arr
    ,CvMat* submat
    ,int start_col
    ,int end_col
);
```

Функция `cvGetCol()` используется для получения одного столбца из матрицы в виде вектора (т.е. матрицы с одной колонкой). В этом случае заголовок матрицы `submat` будет изменен таким образом, чтобы указывать на конкретный столбец матрицы `mat`. При этом не происходит выделение дополнительной памяти или копирование данных. Содержимое `submat` изменяется таким образом, чтобы правильно указывать на выбранный столбец в `arr`. Поддерживаются все типы данных.

Функция `cvGetCols()` работает похожим образом, только выбирает все столбцы из диапазона `[start_col, end_col]`. Обе функции возвращают указатель на заголовок, соответствующий выбранному столбцу или диапазону столбцов (т.е. `submat`).

cvGetDiag

```
CvMat* cvGetDiag(
    const CvArr* arr
    , CvMat* submat
    , int diag = 0
);
```

Функция `cvGetDiag()` аналогична `cvGetCol()`; она используется, чтобы выбрать одну из диагоналей матрицы и вернуть результат в виде вектора. Аргумент `submat` это заголовок матрицы. Функция `cvGetDiag()` заполняет компоненты заголовка таким образом, чтобы он указывал на нужную информацию в матрице `arr`. Необязательный аргумент `diag` определяет, на какие диагонали указывает `submat`. Если `diag = 0`, то выбирается главная диагональ. Если `diag > 0`, тогда выбирается диагональ выше главной (`diag; 0`); если `diag < 0`, тогда выбирается диагональ ниже главной (`0, -diag`). Матрица не обязательно должна быть квадратной, однако, массив `submat` должен соответствовать входному массиву. Результатом выполнения функции является заголовок `submat`, соответствующий указанной диагонали исходной матрицы.

cvGetDims и cvGetDimSize

```
int cvGetDims(
    const CvArr* arr
    , int* sizes = NULL
);

int cvGetDimSize(
    const CvArr* arr
    , int index
);
```

Функция `cvGetDims()` возвращает количество измерений для массива массивов в общем и (необязательно) размерность вложенных массивов в частности. Если `sizes != NULL`, то это должен быть указатель `n` типа `int`, где `n` размерность измерений. Если заранее не известна размерность измерений, то `sizes` выставляется в `CV_MAX_DIM`.

Функция `cvGetDimSize()` возвращает размерность одного измерения, в соответствии с указанным индексом `index`. Если массив - матрица или изображение, то измерений всегда будет два. Для матриц и изображений, порядок возвращаемых значений в `sizes` из `cvGetDims()` будет следующим: количество строк, количество столбцов.

Пример 3-15. Вычисление общего количества элементов массива

```

int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims(arr, size);

for(i = 0; i < dims; i++ ) {
    total *= sizes[i];
}

```

Функция `cvGetDims()` возвращает размерность массива и массив `sizes` размеров измерений. В случае с матрицами или изображениями, кол-во измерений `dims = 2`, `sizes` будет содержать количество строк и столбцов. Для получения общего количества элементов массива производится перемножение кол-ва строк и кол-ва столбцов.

cvgetRow и cvGetRows

```

CvMat* cvGetRow(
    const CvArr* arr
    ,CvMat* submat
    ,int row
);

CvMat* cvGetRows(
    const CvArr* arr
    ,CvMat* submat
    ,int start_row
    ,int end_row
);

```

Функция `cvGetRow()` выбирает одну строку из матрицы и возвращает её как вектор (т.е матрицу с одной строкой). Как и в `cvGetCol()`, заголовок матрицы `submat` будет координировать выбор конкретной строки в `arr`, при этом модификация этого заголовка не подразумевает выделение памяти или копирование данных. Поддерживаются все типы данных.

Функция `cvGetRows()` работает похожим образом, только выбирает все строки из диапазона `[start_col, end_col]`. Обе функции возвращают указатель на заголовок, соответствующий выбранной строке или диапазону строк (т.е. `submat`).

cvGetSize

```
CvSize cvGetSize( const CvArr* arr );
```

Эта частный случай функций `cvGetDims()` и `cvGetSize()`. Уникальность функции заключается в том, что она может быть использована только на матрицах или изображениях размерностью два. Размер возвращается в виде структуры `CvSize`,

которая подходит, например, для создания новой матрицы или изображения того же размера.

cvGetSubRect

```
CvSize cvGetSubRect(
    const CvArr* arr
    , CvArr* submat
    , CvRect rect
);
```

Функция *cvGetSubRect()* схожа с *cvGetCols()* или *cvGetRows()* за исключением того, что выбирает часть указанного массива, за счет параметра *rect*. Заголовок матрицы *submat* координирует выбор конкретной подматрицы (без выделения памяти и копирования данных).

cvInRange и cvInRangeS

```
void cvInRange(
    const CvArr* src
    , const CvArr* lower
    , const CvArr* upper
    , CvArr* dst
);

void cvInRangeS(
    const CvArr* src
    , CvScalar lower
    , CvScalar upper
    , CvArr* dst
);
```

Эти функции используются для проверки попадания пикселей изображения в указанный диапазон. В случае *cvInRange()* каждый пиксель *src* сравнивается с соответствующими значениями изображений *lower* и *upper*. Если значение *src* больше или равно значения *lower* и меньше, чем *upper*, тогда соответствующее значение в *dst* устанавливается в *0xff*, иначе в 0.

В случае *cvInRangeS()* каждый пиксель *src* сравнивается с соответствующими значениями постоянных (*CvScalar*) *lower* и *upper*. Для обеих функций изображение *src* может быть любого типа; если изображение многоканальное, то каждый канал будет обрабатываться отдельно. Изображение *dst* должно быть 8-битным изображением, того же типа и иметь тоже кол-во каналов, что и *src*.

cvInvert

```
double cvInvert(
    const CvArr*   src
    ,CvArr*        dst
    ,int           method = CV_LU
);
```

Функция *cvInvert()* находит обратную матрицу *src* и помещает результат в *dst*. Эта функция поддерживает несколько методов вычисления обратной матрицы (таблица 3-8); по умолчанию метод Гаусса. Возвращаемое значение зависит от выбранного метода.

Метод	Описание
CV_LU	Метод Гаусса (LU разложение)
CV_SVD	Сингулярное разложение (SVD)
CV_SVD_SYM	SVD для симметричных матриц

В случае *CV_LU*, функция возвращает определитель матрицы *src*. Если определитель равен 0, то обратная матрица не вычисляется и все элементы *dst* устанавливаются в 0s.

В случае *CV_SVD* или *CV_SVD_SYM*, функция возвращает обратное состояние числа *src* (соотношение наименьшего к большему). Если матрица *src* является особенной, то *cvInvert()* вычисляет псевдо-обратную матрицу.

cvMahalonobis

```
CvSize cvMahalonobis(
    const CvArr*   vec1
    ,const CvArr*   vec2
    ,CvArr*         mat
);
```

Расстояние *Mahalonobis* (*Mahal*) определяется как вектор расстояния, измеренного между точкой и центром распределения Гаусса; вычисляется с помощью обратной ковариационной матрицы этого распределения (рисунок 3-5). Интуитивно, аналогично z-счету в базовой статистике, где расстояние от центра распределения измеряется в единицах дисперсии этого распределения. По идеи, расстояние *Mahalonobis* это просто многомерное обобщение.

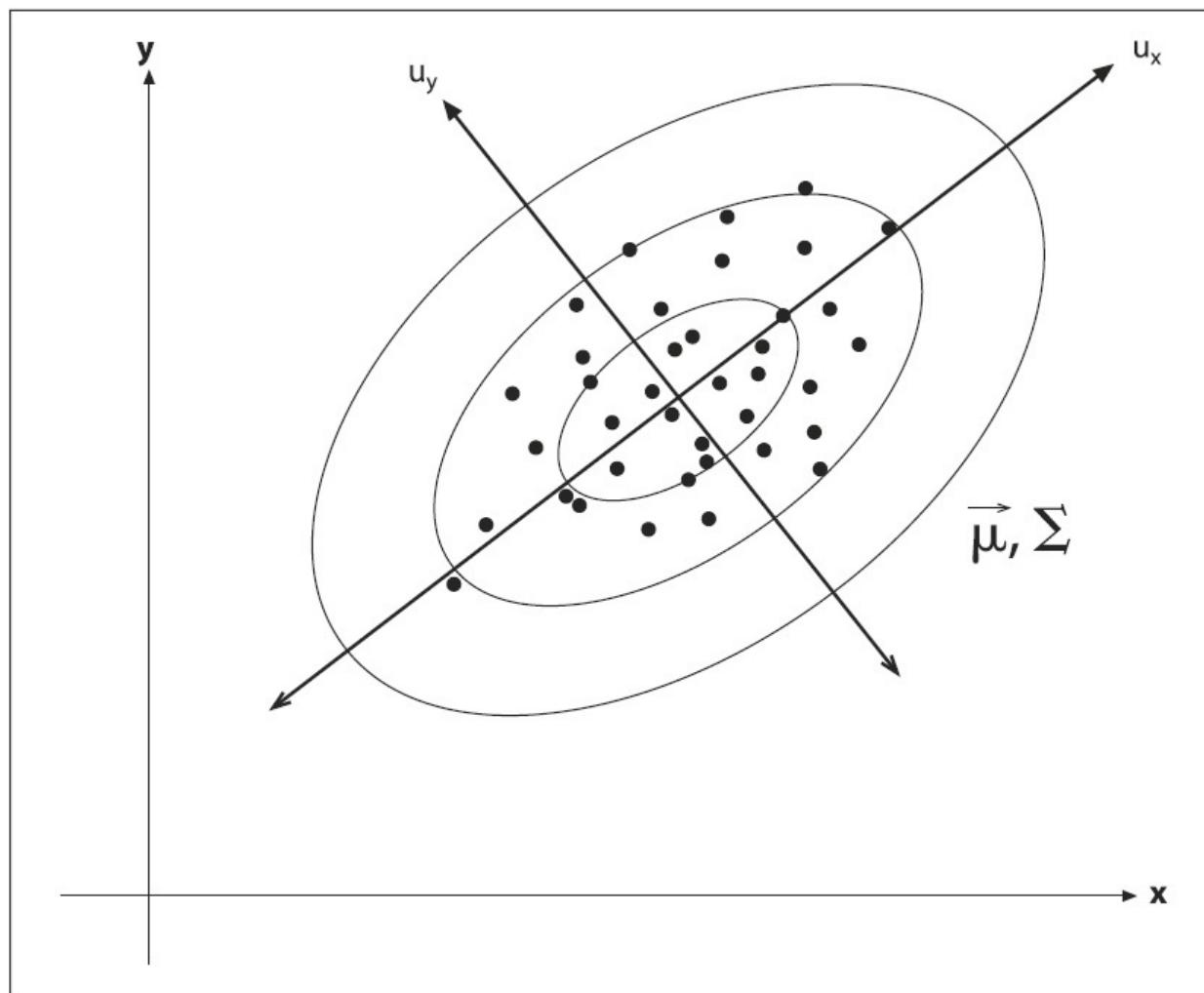


Рисунок 3-5. Распределение точек в двумерном пространстве с наложенными на него эллипсоидами, представляющие 1.0, 2.0 и 3.0 расстояния *Mahalonobis* от среднего распределения

`cvMahalanobis()` вычисляется по следующей формуле:

$$r_{\text{Mahalanobis}} = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}$$

Вектор `vec1` соответствует \mathbf{x} , а вектор `vec2` берется как среднее распределение.

Матрица `mat` это обратная ковариационная матрица.

На практике, эта ковариационная матрица вычисляется при помощи функций `cvCalcCovarMatrix()` и `cvInvert()`. Хорошим тоном программирования считается использования метода `CV_SVD` для `cvInvert()`, потому что встречаются распределения, для которых одно из собственных чисел может быть равно нулю!

cvMax и cvMaxS

```

void cvMax(
    const CvArr* src1
    ,const CvArr* src2
    ,CvArr* dst
);

void cvMaxS(
    const CvArr* src
    ,double value
    ,CvArr* dst
);

```

Функция *cvMax()* вычисляет максимальное значение каждой соответствующей пары пикселей из *src1* и *src2*. В функции *cvMaxS()* элементы *src* сравниваются с постоянным значением *value*. Если *mask != NULL*, то под вычисления попадают все элементы *dst*, соответствующие ненулевым элементам маски.

cvMerge

```

void cvMerge(
    const CvArr* src0
    ,const CvArr* src1
    ,const CvArr* src2
    ,const CvArr* src3
    ,CvArr* dst
);

```

Функция *cvMerge()* выполняет обратную операцию по сравнению с функцией *cvSplit()*. Массивы *src0*, *src1*, *src2* и *src3* объединяются в массив *dst*. Конечно, *dst* должен быть того же типа и иметь размер как все исходные массивы, но при этом может иметь два, три или четыре канала. Исходные массивы могут быть объявлены как *NULL*.

cvMin и cvMinS

```

void cvMin(
    const CvArr* src1
    ,const CvArr* src2
    ,CvArr* dst
);

void cvMinS(
    const CvArr* src
    ,double value
    ,CvArr* dst
);

```

Функция *cvMin()* вычисляет минимальное значение каждой соответствующей пары пикселей из *src1* и *src2*. В функции *cvMinS()* элементы *src* сравниваются с постоянным значением *value*. Если *mask != NULL*, то под вычисления попадают все элементы *dst*, соответствующие ненулевым элементам маски.

cvMinMaxLoc

```
void cvMinMaxLoc(
    const CvArr* arr
    , double* min_val
    , double* max_val
    , CvPoint* min_loc = NULL
    , CvPoint* max_loc = NULL
    , const CvArr* mask = NULL
);
```

Эта функция находит минимальное и максимальное значения массива *arr* и (необязательно) возвращает их расположение. Вычисленные значения расположения максимального и минимального значений помещаются в *max_loc* и *min_loc* соответственно. Необязательно, но возможно задать расположение этих экстремумов, если *min_loc != NULL* и *max_loc != NULL*. Если *mask != NULL*, то под вычисления попадают все элементы *arr*, соответствующие ненулевым элементам маски. Функция *cvMinMaxLoc()* может обрабатывать только одноканальные массивы, поэтому для обработки многоканального массива необходимо использовать функцию *cvSetCOI()*, что бы выделить конкретно интересующий канал.

cvMul

```
void cvMul(
    const CvArr* src1
    , const CvArr* src2
    , CvArr* dst
    , double scale = 1
);
```

Функция *cvMul()* поэлементно перемножает все элементы *src1* и *src2* и помещает результат в *dst*. Если *mask != NULL*, то под вычисления попадают все элементы *dst*, соответствующие ненулевым элементам маски. Отсутствие функции *cvMulS()* объясняется наличием функций *cvScale()* и *cvCvtScale()*.

Необходимо понимать, что данная функция выполняет простое поэлементное перемножение матриц. Для получения обобщенной матрицы перемножения необходимо использовать функцию *cvGEMM()*.

cvNot

```
void cvNot(
    const CvArr*   src
    ,CvArr*        dst
);
```

Функция *cvNot()* инвертирует каждый бит в каждом элементе *src* и помещает результат в *dst*. Таким образом, для 8-битного изображения значение *0x00* будет соответствовать *0xff*, а *0x83* - *0x7c*.

cvNorm

```
double cvNorm(
    const CvArr*   arr1
    const CvArr*   arr2      = NULL
    int            norm_type = CV_L2
    const CvArr*   mask      = NULL
);
```

Эта функция может быть использована для вычисления обобщенной нормы массива *src1*, а также расстояния нормы относительной разности между двумя массивами *src1* и *src2*. Для первого случая, норма вычисляется по формулам из таблицы 3-9.

Таблица 3-9. Вычисление нормы при различных *norm_type* и *src2 = NULL*

norm_type	Результат
CV_C	$\ arr1\ _C = \max_{x,y} \text{abs}(arr1_{x,y})$
CV_L1	$\ arr1\ _{L1} = \sum_{x,y} \text{abs}(arr1_{x,y})$
CV_L2	$\ arr1\ _{L2} = \sqrt{\sum_{x,y} \text{arr1}_{x,y}^2}$

Если второй аргумент *arr2* не *NULL*, то норма вычисляется как разностная норма, т.е. что-то вроде расстояния между двумя массивами. В первых трех случаях, как показано в таблице 3-10, норма является абсолютной, в последних трех случаях она масштабируется по величине второго массива *arr2*.

Таблица 3-10. Вычисление нормы при различных *norm_type* и *src2 != NULL*

norm_type	Результат
CV_C	$\ arr1 - arr2\ _C = \max_{x,y} \text{abs}(arr1_{x,y} - arr2_{x,y})$
CV_L1	$\ arr1 - arr2\ _{L1} = \sum_{x,y} \text{abs}(arr1_{x,y} - arr2_{x,y})$
CV_L2	$\ arr1 - arr2\ _{L2} = \sqrt{\sum_{x,y} (arr1_{x,y} - arr2_{x,y})^2}$
CV_RELATIVE_C	$\frac{\ arr1 - arr2\ _C}{\ arr2\ _C}$
CV_RELATIVE_L1	$\frac{\ arr1 - arr2\ _{L1}}{\ arr2\ _{L1}}$
CV_RELATIVE_L2	$\frac{\ arr1 - arr2\ _{L2}}{\ arr2\ _{L2}}$

Во всех случаях, *arr1* и *arr2* должны быть одного размера и иметь одинаковое количество каналов. При наличии более одного канала, норма вычисляется одновременно по всем каналам (т.е. суммирование в таблицах 3-9 и 3-10 происходит не только по x и y, но и по каналам тоже).

cvNormalize

```
void cvNormalize(
    const CvArr* src
    , CvArr* dst
    , double a = 1.0
    , double b = 0.0
    , int norm_type = CV_L2
    , const CvArr* mask = NULL
);
```

Как и многие другие функции OpenCV, *cvNormalize()* делает больше, чем кажется на первый взгляд. В зависимости от величины *norm_type*, нормализуется изображение *src*, иначе отображается в определенном диапазоне в *dst*. Все возможные значения *norm_type* приведены в таблице 3-11.

Таблица 3-11. Возможные значения *norm_type*

norm_type	Result
CV_C	$\ arr1\ _C = \max_{dst} \text{abs}(I_{x,y}) = a$
CV_L1	$\ arr1\ _{L1} = \sum_{dst} \text{abs}(I_{x,y}) = a$
CV_L2	$\ arr1\ _{L2} = \sqrt{\sum_{dst} I_{x,y}^2} = a$
CV_MINMAX	Map into range [a, b]

В случае *CV_C*, массив *src* будет масштабироваться таким образом, что наибольшее значение абсолютной величины равно *a*. В случае *CV_L1* и *CV_L2*, массив *src* будет масштабироваться таким образом, что норма будет равна *a*. Если *norm_type* установлен в *CV_MINMAX*, то значения массива масштабируются и транслируются таким образом, что они линейно отображаются в промежутке *[a;b]*.

Если *mask* \neq *NULL*, то под вычисления попадают все элементы, соответствующие ненулевым элементам маски.

cvOr и cvOrS

```
void cvOr(
    const CvArr*    src1
    ,const CvArr*    src2
    ,CvArr*          dst
    ,const CvArr*    mask = NULL
);

void cvOrS(
    const CvArr*    src
    ,CvScalar        value
    ,CvArr*          dst
    ,const CvArr*    mask = NULL
);
```

Эти две функции применяют побитовою операцию *OR* к массиву *src1*. В случае *cvOr()*, каждый элемент *dst* вычисляется как побитовая операция *OR* между двумя элементами *src1* и *src2*. В случае *cvOrS()*, операция производится между *src1* и *value*. Если *mask* \neq *NULL*, то под вычисления попадают все элементы, соответствующие ненулевым элементам маски.

Поддерживаются все типы данных, но *src1* и *src2* должны быть одинакового типа.

cvReduce

```
CvSize cvReduce(
    const CvArr*   src
    ,CvArr*        dst
    ,int           dim
    ,int           op   = CV_REDUCE_SUM
);
```

Сокращение является систематическим преобразованием входной матрицы *src* в вектор *dst*, применяя некоторую комбинацию правил *op* к каждой строке (или столбцу) и их соседу, пока не останется только одна строка (или один столбец) (таблица 3-12). Аргумент *op* управляет сокращением (таблица 3-13).

Таблица 3-12. Выбор операции сокращения

Значение <i>op</i>	Результат
CV_REDUCE_SUM	Вычисление суммы по векторам
CV_REDUCE_AVG	Вычисление среднего по векторам
CV_REDUCE_MAX	Вычисление максимума по векторам
CV_REDUCE_MIN	Вычисление минимума по векторам

Таблица 3-13. Аргумент *dim* для контроля направления сокращения

Значение <i>dim</i>	Результат
+1	Свернуть в одну строку
0	Свернуть в один столбец
-1	Свернуть в соответствии с <i>dst</i>

cvReduce() поддерживает многоканальные массивы вещественного типа. Также возможно использовать типы с более высокой точностью для *dst*, чем у *src*. Это прежде всего относится к *CV_REDUCE_SUM* и *CV_REDUCE_AVG*, где возможны проблемы с переполнением.

cvRepeat

```
void cvRepeat(
    const CvArr*   src
    ,CvArr*        dst
);
```

Эта функция копирует содержимое *src* в *dst*, повторяя операцию столько раз, сколько потребуется для заполнения *dst*. В частности, *dst* может быть любого размера относительно *src*.

cvScale

```
void cvScale(
    const CvArr* src
    ,CvArr* dst
    ,double scale
);
```

Функция `cvScale()` на самом деле макрос функции `cvConvertScale()`, который устанавливает аргумент `shift = 0.0`. Таким образом, она может быть использована, чтобы изменять масштаб содержимого массива и для преобразования из одного типа данных в другой.

cvSet и cvSetZero

```
void cvSet(
    CvArr* arr
    ,CvScalar value
    ,const CvArr* mask = NULL
);
```

Эта функция устанавливает все значения во всех каналах массива в значение `value`. Функция `cvSet()` принимает необязательный аргумент `mask`: если маска задана, то данной операции подвергаются лишь те элементы `arr`, которые соответствуют ненулевым значениям маски. Функция `cvSetZero()` просто синоним функции `cvSet(arr, 0.0)`.

cvSetIdentity

```
void cvSetIdentity( CvArr* arr );
```

Функция `cvSetIdentity()` устанавливает все элементы массива в 0 за исключением элементов, у которых строки и столбцы равны; эти элементы равны 1. Функция поддерживает все типы данных, а также не требует, чтобы массив был квадратным.

cvSolve

```
int cvSolve(
    const CvArr* src1
    ,const CvArr* src2
    ,CvArr* dst
    ,int method = CV_LU
);
```

Функция `cvSolve()` обеспечивает быстрый способ для решения систем линейных уравнений, основанный на использовании функции `cvInvert()`. Вычисления производятся по следующей формуле:

$$C = \arg \min_X \|A \cdot X - B\|$$

где A квадратная матрица $src1$, B вектор $src2$ и C результат решения `cvSolve()` для найденного лучшего вектора X . Лучший вектор X помещается в dst . Поддерживаются те же методы *method*, что и в `cvInvert()` (описанные ранее); поддерживаются только вещественные типы данных. Функция возвращает целое число; 0 означает, что решение может быть найдено.

Стоит отметить, что `cvSolve()` может быть использована для нахождения решения переопределенных систем линейных уравнений. Данная система будет решена псевдо обратно, используя метод *SVD* для поиска решения наименьших квадратов для системы линейных уравнений.

`cvSplit`

```
void cvSplit(
    const CvArr*   src
    ,CvArr*        dst0
    ,CvArr*        dst1
    ,CvArr*        dst2
    ,CvArr*        dst3
);
```

Иногда возникают моменты, в которые неудобно работать с многоканальными изображениями. В таких случаях следует использовать функцию `cvSplit()`, что бы скопировать каждый канал в отдельное одноканальное изображение. Исходное изображение src разбивается на $dst0, dst1, dst2, dst3$ по мере необходимости. Конечные изображения должны соответствовать исходному изображению по размеру и типу данных, но при этом, конечно, должны быть одноканальными.

Если исходное изображение имеет менее четырех каналов (зачастую это так), то ненужным аргументам будет присвоено NULL.

`cvSub`, `cvSubS` и `cvSubRS`

```

void cvSub(
    const CvArr* src1
    , const CvArr* src2
    , CvArr* dst
    , const CvArr* mask = NULL
);

void cvSubS(
    const CvArr* src
    , CvScalar value
    , CvArr* dst
    , const CvArr* mask = NULL
);

void cvSubRS(
    const CvArr* src
    CvScalar value
    CvArr* dst
    const CvArr* mask = NULL
);

```

Функция *cvSub* выполняет поэлементное вычитание одного массива *src1* из другого *src2* и помещает результат в *dst*. Если *mask* != *NULL*, то под вычисления попадают все элементы, соответствующие ненулевым элементам маски. *src1*, *src2* и *dst* должны быть одного типа, размера и иметь одинаковое количество каналов; если задействован параметр *mask*, то это должен быть 8-битный массив того же размера и с тем же количеством каналов, что и *dst*.

Функция *cvSubS* выполняет поэлементное вычитание из массива *src* значение *value* и помещает результат в *dst*.

Функция *cvSubRS* выполняет вычитание из значения *value* элементов массива *src* и помещает результат в *dst*.

cvSum

```

CvScalar cvSum(
    CvArr* arr
);

```

Функция *cvSum()* суммирует все элементы во всех каналах массива *arr*. Функция возвращает значение типа *CvScalar* - это означает, что существует возможность обрабатывать многоканальные массивы. В этом случае сумма для каждого канала помещается в соответствующий компонент *CvScalar*.

cvSVD

```
void cvSVD(
    CvArr* A
    ,CvArr* W
    ,CvArr* U      = NULL
    ,CvArr* V      = NULL
    ,int     flags  = 0
);
```

Сингулярное разложение (SVD) является разложением матрицы **A** размера $m \times n$ вида:

$$\mathbf{A} = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T$$

где **W** диагональная матрица, а **U** размера $m \times m$ и **V** размера $n \times n$ унитарные матрицы. Конечно матрица **W** имеет размер $m \times n$, при этом "диагональность" не означает, что элементы строк или столбцов равны 0. Поскольку **W** обязательно диагональная, OpenCV позволяет быть ей либо $m \times n$, либо $n \times 1$ (в этом случае вектор будет содержать только диагональные "особые" значения).

Матрицы **U** и **V** являются необязательными для `cvSVD()` и если они остаются установленными в `NULL`, то значение не будет возвращено. Завершающий аргумент `flags` может быть либо одним из трех возможных вариантов (таблица 3-14), либо сразу всеми.

Таблица 3-14. Возможные варианты флага

Флаг	Значение
<code>CV_SVD MODIFY_A</code>	Позволяет модифицировать матрицу A
<code>CV_SVD_U_T</code>	Возвращает \mathbf{U}^T вместо U
<code>CV_SVD_V_T</code>	Возвращает \mathbf{V}^T вместо V

cvSVBkSb

```
void cvSVBkSb(
    const CvArr*   W
    ,const CvArr*   U
    ,const CvArr*   V
    ,const CvArr*   B
    ,CvArr*         X
    ,int             flags = 0
);
```

Как правило, не так часто потребуется вызывать эту функцию напрямую. В сочетании с `cvSVD()`, эта функция лежит в основе SVD-методов для `cvInvert()` и `cvSolve()`. Но никто не запрещает работать без "посредников" и писать свои собственные функции

инверсии (в ряде случаев это спасет от использования памяти для временных матриц внутри `cvInvert()` и `cvSolve()`).

Функция `cvSVBkSb()` вычисляет обратную матрицу для матрицы **A**, представленную в виде разложения матриц **U**, **W**, и **V**. Результирующая матрица **X** получается по формуле:

$$\mathbf{X} = \mathbf{V} \cdot \mathbf{W}^* \cdot \mathbf{U}^T \cdot \mathbf{B}$$

Матрица **B** является необязательной и если будет задана как NULL, то будет проигнорирована. Диагональные элементы матрицы **W*** вычисляются по формуле $\lambda_i^* = \lambda_i^{-1}$ для $\lambda_i \geq \varepsilon$. Значение ε является порогом, это довольно таки маленькое число и как правило пропорционально сумме диагональных элементов **W** (т.е. $\varepsilon \propto \sum_i \lambda_i$).

cvTrace

```
CvScalar cvTrace( const CvArr* mat );
```

След матрицы — это сумма всех диагональных элементов. Поддерживаются многоканальные массивы, но тогда массив *mat* должен быть вещественного типа.

cvTranspose и cvT

```
void cvTranspose(
    const CvArr*   src
    ,CvArr*        dst
);
```

Функция `cvTranspose()` копирует каждый элемент *src* в *dst* в место, соответствующее перевернутым значениям строки и столбца. Данная функция не поддерживает многоканальные массивы; однако, если используется несколько каналов, представленные комплексными числами, помните, что `cvTranspose()` не делает комплексное сопряжение (быстрое выполнение этой задачи выполняется с помощью `cvXorS`). Макрос `cvT()` просто сокращение для `cvTranspose()`.

cvXor и cvXorS

```
void cvXor(
    const CvArr*    src1
    ,const CvArr*    src2
    ,CvArr*          dst
    ,const CvArr*    mask = NULL
);

void cvXorS(
    const CvArr*    src
    ,CvScalar        value
    ,CvArr*          dst
    ,const CvArr*    mask = NULL
);
```

Эти две функции вычисляют операцию побитового **XOR** для массива *src1*. В случае *cvXor()* каждый элемент *dst* вычисляется как побитовое исключающее двух соответствующих элементов *src1* и *src2*. В случае *cvXorS()*, побитовое исключающее **XOR** вычисляется с постоянным значением *value*. Если *mask != NULL*, то под вычисления попадают все элементы, соответствующие ненулевым элементам маски.

cvZero

```
void cvZero( CvArr* arr );
```

Эта функция устанавливает все значения во всех каналах массива в 0.

[П]|[РС]|(РП) Рисование

Очень часто будет возникать необходимость рисовать или писать что-либо на изображениях. OpenCV предоставляет нам такую возможность, за счёт наличия функций, которые могут рисовать линии, квадраты, круги и т.д.

Линии

Простейшая из всех функций просто рисует линию по алгоритму Bresenham:

```
void cvLine(
    CvArr*      array
    ,CvPoint pt1
    ,CvPoint pt2
    ,CvScalar color
    ,int thickness     = 1
    ,int connectivity   = 8
);
```

Первый аргумент *cvLine()* - *CvArr** указатель на изображение *Image**. Следующие два аргумента это координаты точек начала и окончания линии типа *CvPoint*.

Напомним, *CvPoint* это структура, содержащая два целочисленных значения *x* и *y*. Можно создать *CvPoint* "на лету" с помощью функции *cvPoint(int x, int y)*.

Следующий аргумент *color* типа *CvScalar* задает цвет линии. *CvScalar* также является структурой, которая определяется следующим образом:

```
typedef struct {
    double val[4];
} CvScalar;
```

Эта структура содержит только массив из четырех чисел типа *double*. Первые три числа массива представляют цвета: красный, зеленый и синий; четвертый не используется (при необходимости используется для альфа канала). Для этого аргумента удобно использовать макрос *CV_RGB(r,g,b)*. Этот макрос принимает три числа и упаковывает их в структуру *CvScalar*.

Следующие два аргумента не обязательны. Параметр *thickness* это толщина линии (в пикселях), а *connectivity* устанавливает тип сглаживания. Тип сглаживания по умолчанию 8 - хорошо сглаженная линия. Можно также установить 4, тогда линия

будет менее аккуратной, но зато времени на рисование будет потрачено намного меньше.

cvRectangle() - ещё одна удобная функция. Из названия понятно, что она рисует прямоугольник. Эта функция имеет те же аргументы, что и *cvLine()*, за исключением аргумента *connectivity*, т.к. стороны прямоугольника в OpenCV всегда параллельны осям координат и сглаживать их нет необходимости. В *cvRectangle()* просто передаются две точки для противоположных углов.

```
void cvRectangle(
    CvArr*      array
    ,CvPoint    pt1
    ,CvPoint    pt2
    ,CvScalar   color
    ,int        thickness = 1
);
```

Круги и эллипсы

Так же просто можно нарисовать и круг, функция рисования которого в значительной степени имеет те же аргументы.

```
void cvCircle (
    CvArr*      array
    ,cvPoint    center
    ,int        radius
    ,CvScalar   color
    ,int        thickness     = 1
    ,int        connectivity = 8
);
```

Для круга, прямоугольника и других замкнутых фигур, значение аргумента *thickness* может быть установлено как *CV_FILLED*, что эквивалентно значению -1; как результат - фигура будет залита в тот же цвет, что и края.

Немного более сложная, чем *cvCircle()*, функция *cvEllipse()* рисует эллипс:

```

void cvEllipse(
    CvArr*      img
    ,CvPoint     center
    ,CvSize      axes
    ,double      angle
    ,double      start_angle
    ,double      end_angle
    ,CvScalar    color
    ,int         thickness = 1
    ,int         line_type = 8
);

```

Новый аргумент в этой функции *axes* имеет тип *CvSize*. Структура *CvSize* очень похожа на *CvPoint* и *CvScalar*, только содержит ширину и высоту. Как и для *CvPoint* и *CvScalar* есть вспомогательная функция *cvSize(int width, int height)*, которая возвращает структуру *CvSize*. В данном случае ширина и высота представляют длину большой и малой оси эллипса.

Аргумент *angle* это угол большой оси (в градусах), отсчитываемый от горизонта (т.е. от оси *x*) против часовой стрелки. Аргументы *start_angle* и *end_angle* указывают углы (в градусах) начала и окончания дуги эллипса. Для цельного эллипса необходимо установить эти значения в 0 и 360 соответственно.

Альтернативный способ нарисовать эллипс — это использовать ограничительную рамку:

```

void cvEllipseBox(
    CvArr*      img
    ,CvBox2D    box
    ,CvScalar    color
    ,int         thickness = 1
    ,int         line_type = 8
    ,int         shift      = 0
);

```

В функции используется новая вспомогательная структура *CvBox2D*:

```

typedef struct {
    CvPoint2D32f   center;
    CvSize2D32f    size;
    float          angle;
} CvBox2D;

```

Структура *CvPoint2D32f* аналогична *CvPoint*, а *CvSize2D32f* аналогична *CvSize*, только обе вещественного типа.

Полигоны

И в заключении, в распоряжении есть множество функций для рисования полигонов:

```
void cvFillPoly(
    CvArr*      img
    ,CvPoint**  pts
    ,int*        npts
    ,int         contours
    ,CvScalar   color
    ,int        line_type = 8
);

void cvFillConvexPoly(
    CvArr*      img
    ,CvPoint*   pts
    ,int        npts
    ,CvScalar   color
    ,int        line_type = 8
);

void cvPolyLine(
    CvArr*      img
    ,CvPoint**  pts
    ,int*        npts
    ,int         contours
    ,int        is_closed
    ,CvScalar   color
    ,int        thickness = 1
    ,int        line_type = 8
);
```

Все три функции реализуют одну и ту же идею, с разницей в том, как представлены точки, по которым рисуется полигон.

В *cvFillPoly()* точки представлены в виде массива структур *CvPoint*. Это позволяет нарисовать много полигонов в одном вызове. *npts* является массивом количества точек, по одному для каждого полигона. Если переменная *is_closed* установлена в *true*, то последняя точка полигона будет автоматически соединяться с первой. *cvFillPoly()* вполне надежная функция и может обрабатывать самопересекающиеся полигоны, полигоны с отверстиями и полигоны другой сложности. Однако, это влияет на производительность.

Функция *cvFillConvexPoly()* похожа на *cvFillPoly()*, только рисует один полигон за раз, и работает только с выпуклыми полигонами. Достоинством *cvFillConvexPoly()* является высокое быстродействие.

cvPolyLine() принимает те же аргументы что и *cvFillPoly()*, только рисует не закрашенный полигон и следовательно работает гораздо быстрее.

Шрифты и текст

Ещё одна вещь, которая может пригодится при обработке изображений – это отрисовка текста. Конечно текст создаёт собственный набор сложностей, но как всегда в OpenCV есть простой набор инструментов для решения этой задачи (без ненужных возможностей, которые есть в других библиотеках).

В OpenCV есть одна главная функция *cvPutText()*, которая просто помещает некоторый текст на изображение. Текст указывается в *text* и печатается от нижнего левого угла текстового поля *origin* цвета *color*.

```
void cvPutText(
    CvArr*      img
    ,const char* text
    ,CvPoint     origin
    ,const CvFont* font
    ,CvScalar    color
);
```

Однако, существуют более сложные задачи, чем просто размещение текст на изображении. Для этих случаев существует указатель *CvFont*.

Для получения валидного указателя *CvFont** используется функция *cvInitFont()*. Эта функция принимает группу аргументов, которые настраивают некоторые особенности шрифта для отображения на экране. Те, кто знаком с программированием GUI в других средах, знакомы с *cvInitFont()*.

Для создания *CvFont* и передачи в *cvPutText()*, необходимо сначала объявить переменную *CvFont* и передать её в *cvInitFont()*.

```
void cvInitFont(
    CvFont*   font
    ,int      font_face
    ,double   hscale
    ,double   vscale
    ,double   shear    = 0
    ,int      thickness = 1
    ,int      line_type = 8
);
```

Эта функция немного отличается от аналогичных ей, например от `cvCreateImage()`. Структуру `CvFont` нужно создавать перед вызовом этой функции, в отличии от `cvCreateImage()`, которая возвращает указатель; в `cvInitFont()` нужно передать указатель на уже существующую структуру.

Аргумент `font_face` может принимать одно из перечисленных значений в таблице 3-15 (и изображенных на рисунке 3-6). Также он может быть скомбинирован с `CV_FONT_ITALIC` при помощи логического OR.

Таблица 3-15. Доступные шрифты *HERSHEY*

Переменная	Описание
<code>CV_FONT_HERSHEY_SIMPLEX</code>	Нормальный размер без засечек
<code>CV_FONT_HERSHEY_PLAIN</code>	Маленький размер без засечек
<code>CV_FONT_HERSHEY_DUPLEX</code>	Нормальный размер без засечек; сложнее, чем <code>CV_FONT_HERSHEY_SIMPLEX</code>
<code>CV_FONT_HERSHEY_COMPLEX</code>	Нормальный размер без засечек; сложнее, чем <code>CV_FONT_HERSHEY_DUPLEX</code>
<code>CV_FONT_HERSHEY_TRIPLEX</code>	Нормальный размер без засечек; сложнее, чем <code>CV_FONT_HERSHEY_COMPLEX</code>
<code>CV_FONT_HERSHEY_COMPLEX_SMALL</code>	Уменьшенная версия <code>CV_FONT_HERSHEY_COMPLEX</code>
<code>CV_FONT_HERSHEY_SCRIPT_SIMPLEX</code>	Стиль подчерка
<code>CV_FONT_HERSHEY_SCRIPT_COMPLEX</code>	Более сложный вариант <code>CV_FONT_HERSHEY_SCRIPT_SIMPLEX</code>



Рисунок 3-6. Восемь вариантов шрифта из таблицы 3-15 с $hscale = vscale = 1.0$ и с междусторочным интервалом в 30 пикселей.

$hscale$ и $vscale$ могут быть установлены только равными 1.0 или 0.5. Данные параметры задают масштабный коэффициент относительно основного определения конкретного шрифта.

Параметр $shear$ устанавливает курсивный шрифт; если установлен в 0.0, шрифт не наклонный. Если параметр равен 1.0, то наклон примерно 45 градусов.

Параметры $thickness$ и $line_type$ задают толщину и тип сглаживания линии.

[П]||[РС]||(РП) Сохранность данных

OpenCV предоставляет средства для сериализации и де-сериализации различных типов данных "на" и "с" диска в формате *YAML* или *XML*. В главе **HighGUI** будут рассмотрены функции пользовательского интерфейса, в частности по работе с изображениями (функции *cvSaveImage()* и *cvLoadImage()*). Так же будут рассмотрены функции чтения и записи видеофайлов: *cvGrabFrame()*, которая считывает кадр из файла или с камеры; *cvCreateVideoWriter()* и *cvWriteFrame()*. В данном разделе речь пойдет о хранении объектов в общем: чтение и запись матриц, OpenCV структуры, файлы конфигурации и логирования.

Для начала будут рассмотрены функции сохранения и загрузки матриц. Это функции *cvSave()* и *cvLoad()*. Например, имеется единичная матрица 5x5 (по диагонали элементы равны 1, остальные 0).

Пример 3-16. Сохранение и загрузка матрицы

```
CvMat A = cvMat( 5, 5, CV_32F, the_matrix_data );      // Создание матрицы
cvSave( "my_matrix.xml", &A );                          // Сохранение
...
// Чтение ранее сохраненной матрицы
//
CvMat* A1 = (CvMat*) cvLoad( "my_matrix.xml" );
```

CxCore содержит целый раздел, посвященный сохранению данных. Действительно важно знать, что сохранение данных в OpenCV состоит в создании структуры *CvFileStorage*, которая хранит объекты в памяти в виде дерева. Можно создать и заполнить эту структуру, прочитав данные с диска с помощью *cvOpenFileStorage()* с флагом *CV_STORAGE_READ*, или можно создать и открыть эту структуру с помощью *cvOpenFileStorage()* с флагом *CV_STORAGE_WRITE* для записи, т.е. для заполнения этой структуры данными с помощью соответствующих функций. На диске данные хранятся в формате XML или YAML.

Пример 3-17. Структура *CvFileStorage*; данные доступны через функции сохранности данных **CxCore**

```
typedef struct CvFileStorage {
    // скрытые поля
} CvFileStorage;
```

Данные внутри дерева *CvFileStorage* могут состоять из иерархии скаляров, объектов *CxCore* (матрицы, последовательности и графики) и/или пользовательских объектов.

Например, необходимо записать на диск файл конфигурации, который содержит количество кадров для записи (10), размер кадра (320x240) и цветную матрицу преобразования 3x3.

Пример 3-18. Запись конфигурационного файла "cfg.xml" на диск

```
// Создание файлового хранилища
//
CvFileStorage* fs = cvOpenFileStorage(
    "cfg.xml",
    0,
    CV_STORAGE_WRITE
);

cvWriteInt( fs, "frame_count", 10 );                                // Указание количества кадров
cvStartWriteStruct( fs, "frame_size", CV_NODE_SEQ );                // Создание вложенного узла
cvWriteInt( fs, 0, 320 );                                            // Указание ширина кадра
cvWriteInt( fs, 0, 200 );                                            // Указание высоты кадра
cvEndWriteStruct( fs );                                              // Окончание вложенного узла
cvWrite( fs, "color_cvt_matrix", cmatrix );                          // Указание матрицы преобразования
cvReleaseFileStorage( &fs );                                         // Освобождение занимаемой памяти
```



Функция *cvWriteInt()* задает имя тега для скаляра в структуре дерева. Функция *cvStartWriteStruct()* создает контейнер для тегов. Функция *cvEndWriteStruct()* сигнализирует об окончании созданного контейнера. Глубина вложенности может быть произвольной. Функция *cvWrite()* используется для записи цветной матрицы преобразования. После окончания записи, необходимо вызвать функцию *cvReleaseFileStorage()*. Результат выполнения примера 3-18 представлен в примере 3-19.

Пример 3-19. XML файл cfg.xml

```
<?xml version="1.0"?>
<opencv_storage>
    <frame_count>10</frame_count>
    <frame_size>320 200</frame_size>
    <color_cvt_matrix type_id="opencv-matrix">
        <rows>3</rows>
        <cols>3</cols>
        <dt>f</dt>
        <data>...</data>
    </color_cvt_matrix>
</opencv_storage>
```

Чтение конфигурационного файла показано в примере 3-20

Пример 3-20 Чтение cfg.xml

```
// Открытие файла на чтение
//
CvFileStorage* fs = cvOpenFileStorage(
    "cfg.xml"
, 0
, CV_STORAGE_READ
);

// Получение кол-ва кадров
//
int frame_count = cvReadIntByName(
    fs
, 0
, "frame_count"
, 5 /* значение по умолчанию */
);

// Получение контейнера размера кадра
//
CvSeq* s = cvGetFileNodeByName( fs, 0, "frame_size" )->data.seq;

// Получение ширины кадра
//
int frame_width = cvReadInt(
    (CvFileNode*)cvGetSeqElem( s, 0 )
);

// Получение высоты кадра
//
int frame_height = cvReadInt(
    (CvFileNode*)cvGetSeqElem( s, 1 )
);

// Получение матрицы
//
CvMat* color_cvt_matrix = (CvMat*) cvReadByName(
    fs
, 0
, "color_cvt_matrix"
);

// Освобождение занимаемой памяти
//
cvReleaseFileStorage( &fs );
```

Функции сохранения данных, связанные со структурой *CvFileStorage*, описаны в таблице 3-16.

Таблица 3-16. Функции сохранения данных

Функция	Описание
Открытие и освобождение	
cvOpenFileStorage	Открытие хранилища для чтения и записи
cvReleaseFileStorage	Освобождение занимаемой памяти
Запись	
cvStartWriteStruct	Начало записи новой структуры
cvEndWriteStruct	Окончание записи новой структуры
cvWriteInt	Запись целого числа
cvWriteReal	Запись вещественного числа
cvWriteString	Запись текстовой строки
cvWriteComment	Запись комментария в формате XML или YAML
cvWrite	Запись объекта вида CvMat
cvWriteRawData	Запись нескольких чисел
cvWriteTreeNode	Запись узла в другое хранилище
Чтение	
cvGetRootTreeNode	Получение верхнего узла в хранилище
cvGetTreeNodeByName	Поиск узла в части дерева или хранилище
cvGetHashedKey	Получение уникального указателя по переданному имени
cvGetTreeNode	Поиск узла в части дерева или хранилище
cvGetTreeNodeName	Получение имени узла
cvReadInt	Чтение неименованного целого числа
cvReadIntByName	Чтение именованного целого числа
cvReadReal	Чтение неименованного вещественного числа
cvReadRealByName	Чтение именованного целого числа
cvReadString	Чтение неименованной строки
cvReadStringByName	Чтение именованной строки
cvRead	Декодирование неименованного объекта и получение указателя на него
cvReadByName	Декодирование именованного объекта и получение указателя на него

cvReadRawData	Чтение нескольких чисел
cvStartReadRawData	Чтение узла последовательности
cvReadRawDataSlice	Чтение данных из последовательности

[П]|[PC]|(РП) IPP - Integrated Performance Primitives

У Intel-а есть продукт под названием Integrated Performance Primitives (IPP) library. Эта библиотека позволяет оптимизировать задачи обработки мультимедиа и другие операции процессора, тем самым повышая популярность использования процессоров от Intel.

Как уже говорилось ранее, OpenCV поддерживает тесные отношения с IPP, как на программном уровне, так и на организационном уровне внутри компании. В результате, OpenCV автоматически распознает IPP и использует её для повышения производительности.

Имея такую библиотеку под рукой, можно выполнять широкий спектр задач. В большинстве далее обсуждаемых задачах будут использованы подпрограммы из библиотеки IPP. Не секрет, что обрабатывать большой объем данных выгоднее всего параллельно (MMX, SSE, SSE2 и т.д.).

Проверка установки

Проверить и убедиться, что IPP установлена и работает правильно, позволяет функция `cvGetModuleInfo()` (пример 3-21). Эта функция проверит установленную версию OpenCV и версию дополнений.

Пример 3-21. Использование функции `cvGetModuleInfo()` для проверки

```
char* libraries;
char* modules;
cvGetModuleInfo( 0, &libraries, &modules );
printf("Libraries: %s\nModules: %s\n", libraries, modules );
```

Код в примере 3-21 будет генерировать текстовые строки, описывающие установленные библиотеки и модули. Результат может быть следующим:

```
Libraries cxcore: 1.0.0
Modules: ippcv20.dll, ipp120.dll, ipps20.dll, ippvm20.dll
```

Перечисленные модули являются модулями IPP. Эти модули сами по себе являются фактически прокси для более низкоуровневых определений специфических библиотек. Детали реализации выходят за рамки данной книги.

[П]|[РС]|(РП) Резюме

В этой главе были представлены некоторые основные структуры данных, которые наиболее часто используются. В частности, структура матрица (*CvMat*) и наиболее важная структура изображение (*IplImage*). Также были рассмотрены некоторые детали, касающиеся этих структур, и было показано, что они очень похожи, и что функции работают с этими структурами одинаково хорошо.

[П]|[РС]|(РП) Упражнения

Для получения дополнительной информации при выполнении упражнений обратитесь к документации *CxCore* или *OpenCV Wiki*.

1. Найдите и откройте файл `.../opencv/cxcore/include/cxtypes.h`. Изучите его и найдите функции преобразования.
 - a. Объявите отрицательное вещественное число и примените функции получения абсолютного значения, округлив значение (*round*, *ceiling*, *floor*)
 - b. Сгенерируйте несколько случайных чисел
 - c. Создайте вещественную структуру *CvPoint2D32f* и конвертируйте её в целочисленную структуру *CvPoint*
 - d. Конвертируйте полученную структуру *CvPoint* обратно в *CvPoint2D32f*
2. Это упражнение научит использовать матрицы. Создайте двухмерную трехканальную матрицу типа байт размера 100x100. Установите все значения в 0.
 - a. Нарисуйте круг в матрице, используя `void cvCircle(CvArr img, CvPoint center, int radius, CvScalar color, int thickness = 1, int line_type = 8, int shift = 0)`*
 - b. Выведите полученное изображение на экран, используя методы, описанные в главе #2
3. Создайте двухмерную трехканальную матрицу типа байт размера 100x100 и установите все значения в 0. С помощью функции *CvPtr2D()* получите доступ к "зелёному" каналу изображения, и нарисуйте прямоугольник между точками (20, 5) и (40, 20).
4. Создайте трехканальное RGB изображение размером 100x100. Очистите его. Используя арифметику указателей, нарисуйте зеленый квадрат между точками (20, 5) и (40, 20).
5. Теперь попрактикуемся в использовании области интереса (ROI). Создайте одноканальное изображение типа байт и размером 210x210 и обнулите его. С помощью ROI и *cvSet()* постройте пирамиду значений, т.е.: внешняя граница должна быть равна 0, следующая внутренняя граница должна быть равна 20, следующая внутренняя граница должна быть равна 40, и так далее, пока конечный внутренний квадрат не будет равен 200; все границы должны быть 10 пикселей в ширину. Выведите полученный результат на экран.

6. Загрузите изображение размером не менее 100x100 пикселей. Создайте два дополнительных заголовка изображения и установите их *origin*, *depth*, *nchannels* и *widthstep* как у оригинального изображения. В новых заголовках установите ширину в 20 пикселей, а высоту в 30. Передайте получившиеся заголовки функции в *cvNot()* и выведите получившиеся изображение на экран.
7. Создайте маску, используя функцию *cvCmp()*. Загрузите любое изображение. Используя функцию *cvSplit()* разбейте загруженное изображение на изображения, содержащие только красный канал от исходного изображения, зеленый канал от исходного изображения и синий канал от исходного изображения.
 - a. Выведите на экран изображение, содержащее только зеленый канал
 - b. Дважды клонируйте изображение с зеленым каналом и назовите получившиеся изображения *clone1* и *clone2*
 - c. Найдите максимальное и минимальное значения зеленого канала
 - d. Установите значения *clone1* в $thresh = (unsigned\ char)((maximum - minimum)/2.0)$
 - e. Установите значения *clone2* в 0 и вызовите *cvCmp(green_image, clone1, clone2, CV_CMP_GE)*
 - f. В заключении используйте *cvSubS(green_image, thresh/2, green_image, clone2)* и отобразите результат на экране
8. Создайте целочисленную структуру, состоящая из *CvPoint* и *CvRect*; назовите её *my_struct*
 - a. Напишите две функции *void write_my_struct(CvFileStorage fs, const char name, my_struct ms)* и *void read_my_struct(CvFileStorage fs, CvTreeNode ms_node, my_struct ms)* для записи и чтения структуры *my_struct*
 - b. Запишите и прочтите массив структур *my_struct* размером 10

HighGUI

[П]|[РС]|(РП) Набор графических инструментов

Функции OpenCV, которые позволяют взаимодействовать с операционной системой, файловой системой и аппаратными средствами, такими, как камера, собраны в библиотеке **HighGUI** (что означает "высокоуровневый графический пользовательский интерфейс"). **HighGUI** позволяет открывать окна для отображения изображений, читать и записывать графические файлы (изображения и видео), обрабатывать простые события мыши, указателя и клавиатуры. Данная библиотека также позволяет создавать такие полезные элементы, как ползунок. **HighGUI** имеет достаточный функционал для разработки различного рода приложений. При этом наибольшая польза от использования данной библиотеки в её кроссплатформенности.

библиотека HighGUI состоит из трех частей: аппаратной, файловой и GUI.

Аппаратная часть в первую очередь касается работы с камерой. В большинстве ОС обработка камеры довольно таки утомительна. **HighGUI** предоставляет простые механизмы подключения и последующего получения изображения с камеры.

Все, что касается файловой системы, в первую очередь связано с загрузкой и сохранением изображения. Приятной особенностью библиотеки является наличие методов, которые одинаково обрабатывают видеопоток и из файла, и с камеры. Та же идея (относительно) заложена и в методы обработки изображений. Функции просто полагаются на расширения файлов и автоматически обрабатывают все операции по кодированию и декодированию изображений.

Третья часть HighGUI - GUI. Библиотека предоставляет несколько простых функций, которые позволяют открывать окно и отображать в нем изображения. Тут же (в окне) существует возможность обрабатывать события, поступившие от мыши и клавиатуры.

По мере продвижения по главе, данные части не будут рассматриваться по отдельности. В начале будут рассмотрены самые полезные функции в общем; а уже потом будет происходить постепенное погружение в различные детали.

[П]|[РС]|(РП) Создание окна

cvNamedWindow() - функция для отображения изображения на экране. Эта функция принимает два аргумента: имя окна и флаг. Указанное имя окна отображается в заголовке окна, а также является дескриптором окна. Флаг определяет будет ли окно того же размера, что и изображение, которое в нем отображается, либо размеры окна можно будет менять независимо от передаваемого изображения. Прототип данной функции:

```
int cvNamedWindow(
    const char* name
    , int flags = CV_WINDOW_AUTOSIZE
);
```

Стоит обратить внимание на аргумент *flags*. На момент написания данной книги, доступно два варианта определения данного аргумента: 0 и значение по умолчанию *CV_WINDOW_AUTOSIZE*. Если установлено значение *CV_WINDOW_AUTOSIZE*, то размеры окна подстраиваются под размеры изображения. Установив значение аргумента в 0, пользователю предоставляется возможность в дальнейшем самостоятельно изменять размеры окна.

Функция *cvDestroyWindow()* используется для уничтожения окна. Эта функция принимает один аргумент: имя окна. Работа с окнами в OpenCV построена на использовании имен, а не на использовании какого то непонятного "дескриптора". Преобразования между дескриптором и именем происходят внутри **HighGUI**.

Для тех, кого данный подход не устраивает, в **HighGUI** есть две дополнительные функции:

```
void* cvGetWindowHandle( const char* name );
const char* cvGetWindowName( void* window_handle );
```

Эти функции преобразовывают дескриптор окна в имя и наоборот.

Для изменения размеров окна используется функция *cvResizeWindow()*:

```
void cvResizeWindow(
    const char* name
    , int width
    , int height
);
```

Аргументы *width* и *height* в пикселях.

[П]|[РС]|(РП) Загрузка изображения

Прежде, чем отобразить изображение на экране, его необходимо загрузить. Для этого используется функция `cvLoadImage()`:

```
IplImage* cvLoadImage(
    const char*   filename
    ,int          iscolor = CV_LOAD_IMAGE_COLOR
);
```

При загрузке изображения, `cvLoadImage()` не реагирует на расширение файла. Вместо этого `cvLoadImage()` анализирует первые несколько байт файла (т.н. "магическую последовательность") и определяет какой кодек использовать. Второй аргумент `iscolor` может быть установлено в одно из нескольких значений. По умолчанию изображения загружаются 3-ех канальными и 8-битными; дополнительный флаг `CV_LOAD_IMAGE_ANYDEPTH` может быть использован для загрузки не 8-битных изображений. `CV_LOAD_IMAGE_COLOR` - значение флага по умолчанию. Это означает, что независимо от количества каналов в загружаемом изображение, данное изображение будет при необходимости превращаться в 3-канальное.

Альтернативными флагами являются `CV_LOAD_IMAGE_GRAYSCALE` и `CV_LOAD_IMAGE_ANYCOLOR`. При использовании `CV_LOAD_IMAGE_GRAYSCALE` загружаемое изображение автоматически преобразуется в одноканальное. При использовании `CV_LOAD_IMAGE_ANYCOLOR` загружается цветное, 3-канальное изображение, не обязательно 8-битное. Для загрузки 16-битного цветного изображения необходимо использовать `CV_LOAD_IMAGE_COLOR | CV_LOAD_IMAGE_ANYDEPTH`. При использовании `CV_LOAD_IMAGE_UNCHANGED` изображение загружается "как есть". Если в результате загрузки изображения возникает ошибка, то функция `cvLoadImage()` возвращает `NULL`.

Для сохранения изображения используется функция `cvSaveImage()`, которая принимает два аргумента:

```
int cvSaveImage(
    const char*   filename
    ,const CvArr* image
);
```

Первый аргумент `filename` это имя файла, расширение которого используется для определения формата. Второй аргумент `image` это объект-изображение для сохранения. Стоит напомнить, что `CvArr` это аналог базового класса в объектно-

ориентированных языках; таким образом вместо *CvArr** можно использовать *IplImage**. Функция *cvSaveImage()* сохраняет только 8-битные, одно- и трёх-канальные изображения в большинство используемых форматов. Более новые графические форматы, такие как *PNG*, *TIFF* и *JPEG2000* позволяют сохранять 16-битные изображения, или даже изображения с плавающей точкой; также некоторые форматы позволяют сохранять 4-канальные изображения (*BGR+Alpha*). Функция возвращает 1, если сохранение прошло успешно и 0 - если произошла ошибка.

[П]|[РС]|(РП) Отображение изображения

`cvShowImage()` - функция, при помощи которой можно отобразить изображение в ранее созданном окне:

```
void cvShowImage(
    const char*   name
    ,const CvArr* image
);
```

Первый аргумент - имя окна, в котором отображается изображение, второй - указатель на изображение.

Рассмотрим небольшой пример программы, которая отображает изображение на экране. Вся программа занимает всего 22 строки, включая загрузку изображения, комментарии и очистку памяти.

```
int main( int argc, char** argv ) {
    // Создание окна с таким же именем, как и у загружаемого файла изображения
    //
    cvNamedWindow( argv[1], 1 );

    // Загрузка изображения
    //
    IplImage* img = cvLoadImage( argv[1] );

    // Отображение изображения
    //
    cvShowImage( argv[1], img );

    // Ожидание нажатия клавиши "Esc"
    //
    while( 1 ) {
        if( 27 == cvWaitKey( 100 ) ) break;
    }

    // Очистка памяти
    //
    cvDestroyWindow( argv[1] );
    cvReleaseImage( &img );

    exit(0);
}
```

Для более удобного восприятия результата, получаемого в результате выполнения программы, окну присвоено имя загружаемого файла изображения (рисунок 4-1).

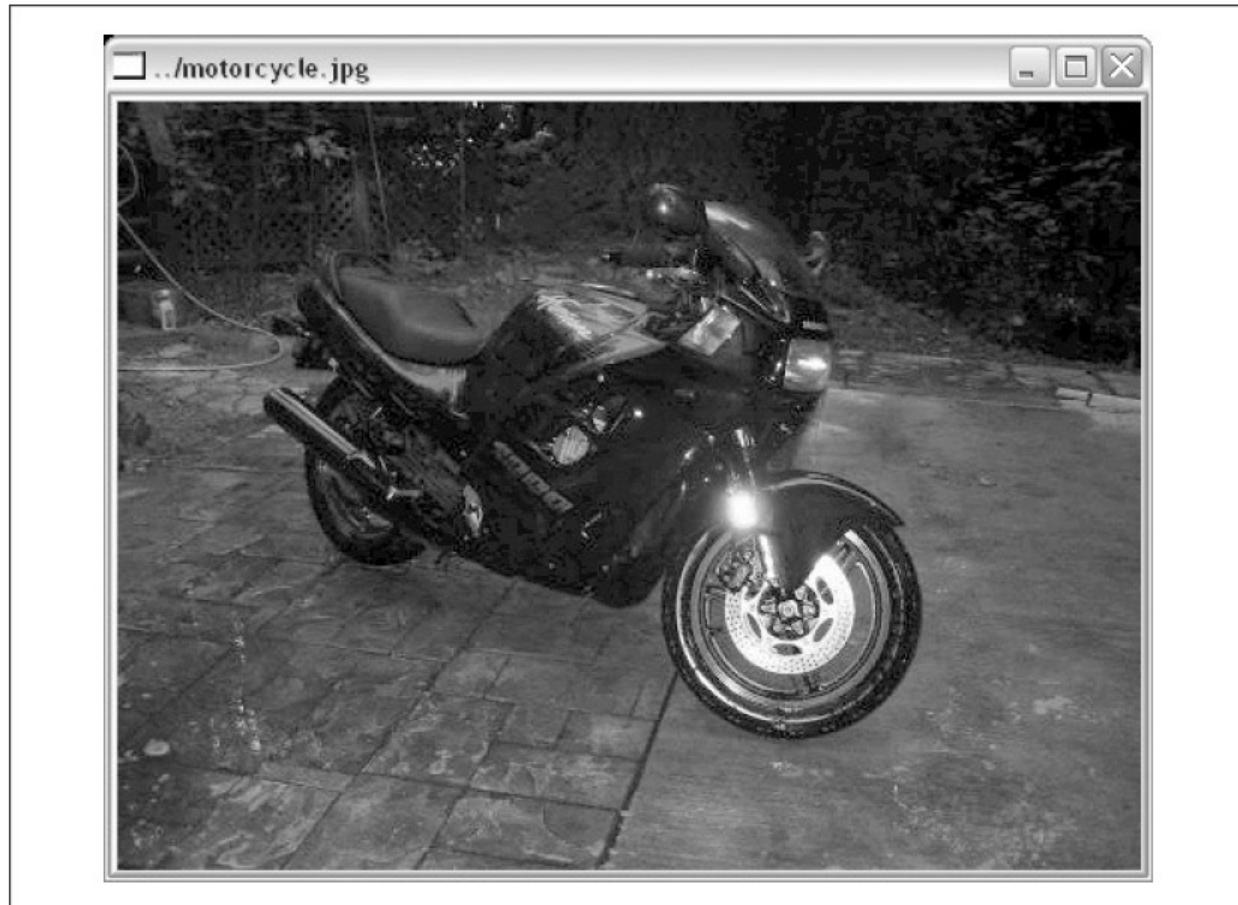


Рисунок 4-1. Отображение изображения при помощи функции *cvShowImage()*

Прежде, чем двигаться дальше, рассмотрим ещё несколько функций для работы с окнами:

```
void cvMoveWindow( const char* name, int x, int y );
void cvDestroyAllWindows( void );
int cvStartWindowThread( void );
```

cvMoveWindow() перемещает окно таким образом, чтобы его верхний левый угол располагался в точке *x*, *y*.

cvDestroyAllWindows() очень полезная функция, которая закрывает все окна и очищает всю занимаемую ими память.

В Linux и MacOS имеется функция *cvStartWindowThread()*, создающая поток, который автоматически обновляет все связанные с окном события. Функция возвращает 0, если не получается запустить ни одного потока; например, при отсутствии поддержки

данной функции в используемой версии OpenCV. Если не создавать окно в отдельном потоке, тогда OpenCV сможет реагировать на действия пользователя только в определенные моменты времени (например, когда вызывается функция `cvWaitKey()`).

WaitKey

Функция `cvWaitKey()` заставляет программу ожидать определенное количество миллисекунд, позволяя тем самым пользователю нажать любую клавишу. Если в течении отведенного времени клавиша будет нажата, то функция вернет код нажатой клавиши, иначе вернет 0.

```
while( 1 ) {  
    if( 27 == cvWaitKey(100) ) break;  
}
```

Функция `cvWaitKey()` ожидает нажатия клавиши 100 мс. Если ASCII код нажатой клавиши равен 27 (клавиша *Esc*), то выполнение цикла прекратиться. Данной подход позволяет задержать показ окна, пока не будет нажата клавиша *Esc*.

Если в качестве аргумента функции `cvWaitKey()` передать 0, то ожидание будет продолжаться неопределенное количество времени, до момента нажатия любой клавиши. В предыдущем примере можно было бы использовать и `cvWaitKey(0)`. Разница между этими двумя вариантами становится более очевидной при обработки видео.

События мыши

Теперь, когда уже есть знания по загрузке и отображению изображения, а также обработке событий от клавиш клавиатуры, можно перейти к обработке событий мыши.

В отличии от событий клавиатуры, события от мыши обрабатываются более типичным образом - при помощи механизма обратного вызова. Это означает, что необходимо писать функции обратного вызова, чтобы обработать щелчки мыши. Функция обратного вызова подлежит регистрации в OpenCV, тем самым информируя OpenCV, что данная функция обрабатывает события мыши над конкретным окном.

Функцией обратного вызова может быть любая функция, которая принимает правильный набор аргументов и возвращает правильный тип. Подобного рода функции должны определять что за мероприятие и где произошло. Так же эти функции должны определять нажатие клавиш Shift и Alt в сочетании с нажатиями клавиш мыши. Точный прототип функции обратного вызова выглядит следующим образом:

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

При вызове данной функции OpenCV будет заполнять аргументы соответствующими значениями. Первый аргумент *event* (событие) будет иметь одно из значений, указанных в таблице 4-1.

Таблица 4-1. Типы событий мыши

Событие	Числовое значение
CV_EVENT_MOUSEMOVE	0
CV_EVENT_LBUTTONDOWN	1
CV_EVENT_RBUTTONDOWN	2
CV_EVENT_MBUTTONDOWN	3
CV_EVENT_LBUTTONUP	4
CV_EVENT_RBUTTONUP	5
CV_EVENT_MBUTTONUP	6
CV_EVENT_LBUTTONDOWNDBLCLK	7
CV_EVENT_RBUTTONDOWNDBLCLK	8
CV_EVENT_MBUTTONDOWNDBLCLK	9

Второй и третий аргументы устанавливают координаты места события мыши. Стоит отметить, что эти координаты представляют собой пиксель на изображение независимо от размера окна.

Четвертый аргумент *flags* - это битовое поле, в котором отдельные биты указывают на особые условия, присутствующие на момент события. Например, флаг *CV_EVENT_FLAG_SHIFTKEY*, имеющий числовое значение 16 (т.е. 5 бит), проверяет нажатие *shift*. Можно использовать сочетание нескольких флагов при помощи операции *AND*. Полный список флагов представлен в таблице 4-2.

Таблица 4-2. Флаги для событий мыши

Флаг	Числовое значение
CV_EVENT_FLAG_LBUTTON	1
CV_EVENT_FLAG_RBUTTON	2
CV_EVENT_FLAG_MBUTTON	4
CV_EVENT_FLAG_CTRLKEY	8
CV_EVENT_FLAG_SHIFTKEY	16
CV_EVENT_FLAG_ALTKEY	32

Последний аргумент - это указатель *void**, который может быть использован для передачи дополнительной информации в виде указателя на необходимую структуру. В общем случае, параметр *param* используется, когда функция обратного вызова является статическим членом класса.

Для регистрации функции обратного вызова используется функция *cvSetMouseCallback()*:

```
void cvSetMouseCallback(
    const char*      window_name
    , CvMouseCallback  on_mouse
    , void*           param = NULL
);
```

Первый аргумент — это имя окна, к которому будет прикреплена функция обратного вызова. Соответственно данная функция обратного вызова будет отслеживать только события, связанные с этим окном. Вторым аргументом является функция обратного вызова. Третий аргумент позволяет указывать дополнительную информацию. Это тот самый *param*, о котором ранее шла речь.

В примере 4-1 описана небольшая программа рисования прямоугольника при помощи мыши. Функция *my_mouse_callback()* реагирует на события мыши и определяет дальнейшие действия.

Пример 4-1. Игрушечная программа, использующая мышь для рисования прямоугольника

```
#include <cv.h>
#include <highgui.h>

// Объявление прототипа функции обратного вызова для реагирования
// на события мыши
//
void my_mouse_callback( int event, int x, int y, int flags, void* param );
```

```

CvRect box;
bool drawing_box = false;

// Функция рисования прямоугольника на изображении
//
void draw_box( IplImage* img, CvRect rect ) {
    cvRectangle (
        img
        ,cvPoint(box.x,box.y)
        ,cvPoint(box.x+box.width,box.y+box.height)
        ,cvScalar(0xff,0x00,0x00) /* red */
    );
}

int main( int argc, char* argv[] ) {
    box = cvRect( -1, -1, 0, 0 );
    IplImage* image = cvCreateImage(
        cvSize( 200, 200 )
        ,IPL_DEPTH_8U
        ,3
    );

    cvZero( image );
    IplImage* temp = cvCloneImage( image );
    cvNamedWindow( "Box Example" );

    // Регистрация функции обратного вызова
    // В качестве значения "param" передается изображение,
    // которое функция обратного вызова будет обрабатывать
    //
    cvSetMouseCallback(
        "Box Example"
        ,my_mouse_callback
        ,(void*) image
    );
}

// Главный цикл. Сначала происходит копирование оригинального изображения
// во временное "temp" изображение, и если пользователь рисует, тогда
// прямоугольник помещается на временное изображение, в заключении результат действий
// отображается на экране. Если после ожидания 15 мс не будет нажата необходимая для
// выхода из цикла клавиша, происходит переход к следующей итерации
//
while( 1 ) {
    cvCopyImage( image, temp );

    if( drawing_box ) {
        draw_box( temp, box );
    }

    cvShowImage( "Box Example", temp );

    if( 27 == cvWaitKey( 15 ) ) {
        break;
}

```

```

        }
    }

    // Освобождение занимаемой памяти
    //
    cvReleaseImage( &image );
    cvReleaseImage( &temp );
    cvDestroyWindow( "Box Example" );
}

// Функция обратного вызова. Если пользователь нажмет левую клавишу
// мыши, активизируется функция рисования. Когда пользователь отпускает
// эту клавишу, происходит добавление прямоугольника на текущее изображение.
// При перемещении мыши происходит изменение размеров прямоугольника
//
void my_mouse_callback( int event, int x, int y, int flags, void* param ) {
    IplImage* image = (IplImage*) param;
    switch( event ) {
        case CV_EVENT_MOUSEMOVE: {
            if( drawing_box ) {
                box.width = x-box.x;
                box.height = y-box.y;
            }
        }
        break;
        case CV_EVENT_LBUTTONDOWN: {
            drawing_box = true;
            box = cvRect(x, y, 0, 0);
        }
        break;
        case CV_EVENT_LBUTTONUP: {
            drawing_box = false;

            if(box.width<0) {
                box.x+=box.width;
                box.width *=-1;
            }

            if(box.height<0) {
                box.y+=box.height;
                box.height*=-1;
            }
        }

        draw_box(image, box);
    }
    break;
}

```

Sliders, Trackbars и Switches

В HighGUI ползунок именуется trackbar-ом. Это все потому, что в оригинальной (исторической) трактовке данный элемент использовался для выбора конкретного кадра из видеопотока. При помощи данного элемента можно делать как привычные вещи, связанные с ползунком, так и более необычные (будут рассмотрены в следующем разделе "No Buttons").

Как и в случае с окном, слайдер имеет уникальное имя (символьная строка), благодаря которому со слайдером можно производить различные манипуляции. Прототип функции создания слайдера:

```
int cvCreateTrackbar(
    const char*        trackbar_name
, const char*        window_name
, int*              value
, int               count
, CvTrackbarCallback on_change
);
```

Первые два аргумента задают имя слайдера и имя окна, к которому будет прикреплен создаваемый слайдер. После создания, слайдер будет добавлен в верхнюю или нижнюю часть окна; при этом он не будет перекрывать ранее добавленное изображение.

Следующие два аргумента: *value* - целочисленный указатель, указывающий позицию ползунка, *count* - максимальное значение ползунка.

Последний аргумент указывает на функцию обратного вызова, которая автоматически вызывается при перемещении ползунка. Прототип функции обратного вызова:

```
void (*callback)( int position )
```

Если функция обратного вызова не требуется, то необходимо передать NULL. В этом случае эффект от использования ползунка заключается в перемещении ползунка в позицию *value*.

И в заключении, вот еще две функции, которые позволяют получать или устанавливать значение ползунка, зная только его имя и имя окна к которому слайдер привязан:

```
int cvGetTrackbarPos(  
    const char*    trackbar_name  
, const char*    window_name  
)  
  
void cvSetTrackbarPos(  
    const char*    trackbar_name  
, const char*    window_name  
, int          pos  
)
```

Вызывать данные функции можно из любого места программы.

No Buttons

К сожалению, **HighGUI** не обеспечивает явную поддержку для кнопок. Таким образом, общая практика преодоления данной проблемы заключается в использовании ползунка только в двух позициях. Другой вариант решения проблемы представлен в [.../opencv/samples/c/](#) и заключается в использовании сочетаний клавиш вместо кнопок (*floodfill* демо в исходниках OpenCV).

Switches (переключатель) это облегченный вариант слайдера, у которого только две позиции "on" (1) и "off" (0) (*count = 1*). Используя слайдер как кнопку, функция обратного вызова автоматически сбрасывает переключатели в 0 (пример 4-2) или же автоматически сбрасывает в 0 другие переключатели (эффект "радиокнопок").

Пример 4-2. Использование слайдера как переключателя

```
// Значение переключателя (глобальная переменная)
//
int g_switch_value = 0;

// Функция обратного вызова
//
void switch_callback( int position ) {
    if( 0 == position ) {
        switch_off_function();
    }
    else {
        switch_on_function();
    }
}

int main( int argc, char* argv[] ) {
    // Создание окна
    //
    cvNamedWindow( "Demo Window", 1 );

    // Создание слайдера с привязкой к окну
    // "Demo Window"
    //
    cvCreateTrackbar(
        "Switch"
        , "Demo Window"
        , &g_switch_value
        , 1
        , switch_callback
    );

    // Ожидание нажатия кнопки Esc
    //
    while( 1 ) {
        if( 27 == cvWaitKey( 15 ) ) {
            break;
        }
    }
}
```

[П]|[РС]|(РП) Работа с видео

Есть несколько функций для работы с видео в OpenCV, наиболее используемыми из которых являются функция чтения и записи видео. Но обо всем по порядку.

CvCapture - структура, содержащая необходимую информацию для чтения кадров с камеры или из видеопотока. В зависимости от источника, необходимо использовать одну из ниже перечисленных функций:

```
CvCapture* cvCreateFileCapture( const char* filename );
CvCapture* cvCreateCameraCapture( int index );
```

cvCreateFileCapture() принимает всего один аргумент - имя AVI или MPG файла. После вызова этой функции OpenCV открывает и подготавливает файл для чтения. Если открытие файла произошло успешно, то функция вернет указатель на структуру *CvCapture*.

Если по какой-то причине, файл не существует или не найден соответствующий кодек, функция вернет NULL. Тонкости работы кодеков сжатия выходят за пределы данной книги, но в целом, достаточно знать в каком случае какой кодек уместно применять и установлен ли он на компьютере. Например, необходимо прочитать файл закодированный с помощью DIVX или сжат с помощью MPG4 на компьютере с ОС Windows. Для корректной обработки должна быть установлена соответствующая библиотека, которая предоставит все необходимые ресурсы для декодирования видео. Крайне необходимо проверять возвращаемое значение на NULL, т.к. все компьютеры имеют разный набор ПО (например, может отсутствовать библиотека для декодирования видео).

Функция *cvCreateCameraCapture()* принимает идентификатор, указывающий к какой камере необходимо получить доступ. Если камера одна, то идентификатор можно установить в 0. *index* - это сумма из порядкового номера и "домена". "Домен" указывает на тип камеры.

Таблица 4-3. Значения домена

Константа	Значение
CV_CAP_ANY	0
CV_CAP_MIL	100
CV_CAP_VFW	200
CV_CAP_V4L	200
CV_CAP_V4L2	200
CV_CAP_FIREWIRE	300
CV_CAP_IEEE1394	300
CV_CAP_DC1394	300
CV_CAP_CMU1394	300

Во время вызова `cvCreateCameraCapture` передается сумма индекса и "домена":

```
CvCapture* capture = cvCreateCameraCapture( CV_CAP_FIREWIRE );
```

В данном примере происходит попытка открыть первую *FireWire* камеру, т.к. индекс равен 0. В большинстве случаев не обязательно указывать "домен", когда используется только одна камера; достаточно использовать `CV_CAP_ANY`. Если в функцию передать значение -1, то OpenCV выведет диалоговое окно для выбора нужной камеры.

Чтение видео

```
int cvGrabFrame( CvCapture* capture );
IplImage* cvRetrieveFrame( CvCapture* capture );
IplImage* cvQueryFrame( CvCapture* capture );
```

Имея верно сформированную структуру `CvCapture`, можно переходить к захвату кадров. Есть два варианта сделать это. Один из них заключается в использовании функции `cvGrabFrame()`, которая принимает указатель на `CvCapture` и возвращает 1, если захват был успешен, и 0 если нет. Функция `cvGrabFrame()` копирует изображение во внутренний, не доступный пользователю, буфер. Возникает вопрос зачем это нужно, захватить кадр и не иметь к нему доступ? Ответ прост - захваченный кадр не обрабатывается, тем самым давая выигрыш в скорости получения кадра.

После вызова `cvGrabFrame()` необходимо вызвать `cvRetrieveFrame()`. Эта функция выполняет все необходимые преобразования кадра (такие, как декомпрессия в кодеке) и возвращает указатель `IplImage*` на другой внутренний буфер (но не стоит на него

рассчитывать, т.к. при следующем вызове `cvGrabFrame()` он будет перезаписан). Если потребуется произвести собственную обработку кадра, то необходимо скопировать данный кадр. При этом указатель на структуру `CvCapture` не должен быть уничтожен до окончания работы с видео, иначе могут возникнуть непредвиденные проблемы с обработкой видео.

`cvQueryFrame()` - иной способ получения кадров из видеопотока. В сущности, это сочетание `cvGrabFrame()` и `cvRetrieveFrame()`; эта функция так же возвращает указатель на `IplImage*`.

Следует отметить, что при чтении видеофайла кадры читаются один за другим при каждом вызове `cvGrabFrame()`, и потому не нужно увеличивать никаких счётчиков.

Освобождение памяти из-под структуры `CvCapture` производится с помощью функции `cvReleaseCapture()`. Как и многие другие функции OpenCV для освобождения памяти, `cvReleaseCapture()` принимает указатель на указатель на `CvCapture`*:

```
void cvReleaseCapture( CvCapture** capture );
```

Есть еще множество других вещей, которые можно делать со структурой `CvCapture`. В частности, можно получить или установить различные свойства источника видео:

```
double cvGetCaptureProperty(
    CvCapture* capture
    ,int      property_id
);
int  cvSetCaptureProperty(
    CvCapture* capture
    ,int      property_id
    ,double   value
);
```

Функция `cvGetCaptureProperty()` принимает любой из идентификаторов, представленных в таблице 4-4:

Таблица 4-4. Свойства `CvCapture`

Свойство	Значение
CV_CAP_PROP_POS_MSEC	0
CV_CAP_PROP_POS_FRAME	1
CV_CAP_PROP_POS_AVI_RATIO	2
CV_CAP_PROP_FRAME_WIDTH	3
CV_CAP_PROP_FRAME_HEIGHT	4
CV_CAP_PROP_FPS	5
CV_CAP_PROP_FOURCC	6
CV_CAP_PROP_FRAME_COUNT	7

POS_MSEC - текущая позиция в видео файле, в миллисекундах. *POS_FRAME* - текущая позиция в файле, в кадрах. *POS_AVI_RATIO* - текущая позиция в файле, заданная числом между 0 и 1 (на самом деле очень полезная вещь при совместном использовании с ползунком для перемотки видео). *FRAME_WIDTH* и *FRAME_HEIGHT* - размер кадра. *FPS* - количество кадров в секунду; влияет на плавность воспроизведимого видео. *FOURCC* - четырехзначный код кодека. *FRAME_COUNT* - количество кадров в видеопотоке (ненадежное значение).

Для всех этих значений будет возвращено значение типа *double*, исключением является значение *FOURCC*. Для верной трактовки случая с *FOURCC*, необходимо произвести еще дополнительное преобразование (пример 4-3).

Пример 4-3. Распаковка четырехзначного кода

```
double f = cvGetCaptureProperty(
    capture
    , CV_CAP_PROP_FOURCC
);

char* fourcc = (char*) (&f);
```

Каждое из этих свойств можно не только получать, но и устанавливать при помощи функции *cvSetCaptureProperty()*. В действительности, не всегда свойства можно изменить. Например, некоторые кодеки могут не поддерживать функцию перемещения по видео.

Запись видео

Операция записи видео в какой-то степени схожа с операцией чтения видеофайла, но с некоторыми дополнительными деталями.

```
CvVideoWriter* cvCreateVideoWriter(
    const char*     filename
    ,int            fourcc
    ,double         fps
    ,CvSize         frame_size
    ,int            is_color = 1
);

int cvWriteFrame(
    CvVideoWriter*   writer
    ,const IplImage* image
);

void cvReleaseVideoWriter(
    CvVideoWriter** writer
);
```

Для начала нужно создать устройство записи, используя структуру *CvVideoWriter*. Для создания этой структуры используется функция *cvCreateVideoWriter()*.

Помимо имени файла, необходимо указать кодек, скорость записи и размер кадра. Необязательно, но можно также указать цвет кадра: цветной или черно-белый (по умолчанию цветной).

Кодек задается при помощи четырехзначного кода. В функцию код передается в упакованном виде. Так как эта операция выполняется относительно часто, OpenCV предоставляет удобный макрос *CV_FOURCC(c0, c1, c2, c3)*.

После создания устройства записи необходимо вызвать функцию *cvWriteFrame()* с передачей в неё *CvVideoWriter** и *IplImage** изображения, которое подлежит записи.

По завершению работы с видео, необходимо вызвать функцию *CvReleaseVideoWriter()* что бы закрыть устройство записи, а затем и файл, в которой производилась запись. Крайне важно не забывать выполнять данную операцию, иначе файл, в который производилась запись, будет поврежден.

[П]|[РС]|(РП) Функция cvConvertImage()

Чисто по историческим причинам, есть в **HighGUI** одна функция "сирота", которая не вписывается ни в одну из ранее представленных категорий. Она чрезвычайно полезна и называется "cvConvertImage()".

```
void cvConvertImage(  
    const CvArr*   src  
    ,CvArr*        dst  
    ,int           flags = 0  
) ;
```

Функция *cvConvertImage()* используется для преобразования между различными типами изображения. Форматы исходного и конечного изображений указываются в *src* и *dst* соответственно (прототип функции принимает аргументы общего типа *CvArr*, который включает в себя *IplImage*).

Изображение источник может быть одно-, трех и четырех канальным изображением, 8-битным или вещественным. Конечное изображение должно быть 8-битным с одним или тремя каналами. Эта функция также позволяет преобразовывать изображение из цветного в серое или из одноканального серого в трех канальное серое (или цветное).

Если задан флаг *flag*, то изображение переворачивается по вертикали. Это полезно в тех случаях, когда формат отображения должен отличаться от формата видео с камеры. В действительности происходит переворачивание пикселей в памяти.

[П]|[РС]|(РП) Упражнения

1. Эта глава завершает введение в основы OpenCV. Следующие упражнения, основанные на всех ранее полученных знаниях, помогут создать набор полезных утилит для дальнейшего использования их в изучении последующих глав.

- a. Напишите программу, которая: (1) читает кадры из видеофайла, (2) конвертирует получаемые кадры в черно-белый формат и (3) выполняет поиск краев, используя функцию `cvCanny()`. Отобразите результат каждого шага в трех разных окнах.
- b. Отобразите все три этапа обработки на одном изображении.

Подсказка: создайте новое изображение с той же высотой и тройной шириной как у кадра. Скопируйте изображения в только что созданное изображение, либо с помощью указателей, либо, что более умно, с помощью создания трёх заголовков изображения, которые указывают на начало, на 1/3 и на 2/3 imageData изображения. Затем используйте cvCopy().

- c. Создайте соответствующие текстовые метки, описывающие обработку каждого изображения.

2. Напишите программу, которая считывает и отображает изображение. При щелчке мыши на изображении, отобразите значения трех каналов (синий, зеленый, красный) в месте щелчка в виде текста на изображении.

- a. Используя программу из 1b, отобразите значения щелков мыши в виде текста на изображении в трех разных окнах.

3. Напишите программу, которая считывает и отображает изображение.

- a. Реализуйте возможность рисования (щелчок левой клавиши, перемещение и отпускание клавиши мыши) прямоугольника на изображении при помощи мыши. Будьте осторожны, сохраните копию исходного изображения в памяти, чтобы не испортить его. При повторном рисовании ранее созданный прямоугольник удаляется, а новый рисуется на исходном изображении.

- b. В отдельном окне нарисуйте график количества используемых значений синего, зеленого и красного каналов. Ось x разбейте на 8 блоков, каждый блок должен иметь диапазон в 32 значения: 0-31, 32-63, ..., 223-255. Ось y должна считать количество найденных пикселей в данном диапазоне. Выполните это для каждого цветного канала.

4. Напишите программу, которая читает, отображает и контролирует, при помощи слайдеров, видео. Первый слайдер контролирует позицию в видео с шагом перемещения 10; второй слайдер контролирует процесс включения и выключения воспроизведения видео. Дайте имена слайдерам соответствующим образом.
5. Напишите упрощённый вариант paint.
 - a. Напишите программу, которая создает изображение (белое полотно), устанавливает все значения в 0 и отображает результат на экране. Обеспечьте возможность рисовать линии, круги, эллипсы и полигоны, используя левую клавишу мыши. Создайте функцию "ластик", которая будет вызываться при нажатии правой клавиши мыши.
 - b. Добавьте возможность "логики рисования" при помощи слайдера. Возможные варианты: *AND*, *OR* и *XOR*. Например, при использовании *AND*, рисование будет происходить только в тех местах полотна, где значение отлично от 0.
6. Напишите программу, которая создает изображение (белое полотно), устанавливает все значения в 0 и отображает результат на экране. В месте нажатия пользователем клавиши мыши Обеспечьте возможность в написании текста. По нажатию *Backspace* обеспечьте удаление введенных символов. По нажатию *Enter* текст должен становиться не редактируемым.
7. Перспективные преобразования
 - a. Напишите программу, которая считывает изображение и использует цифры 1-9 на клавиатуре для управления матрицей перспективного преобразования (функция *cvWarpPerspective()* из 6 главы). Нажатие любой клавиши с цифрой должно увеличивать соответствующую ячейку в матрице перспективного преобразования; в сочетании с *Shift* следует уменьшать соответствующую ячейку в матрице перспективного преобразования (останавливаться при достижении 0). Результат нажатия клавиши клавиатуры выводить в два окна: окно с исходным изображением и окно с преобразованным изображением.
 - b. Добавьте функциональность изменения масштаба.
 - c. Добавьте функциональность изменения угла.
8. Перейдите в директорию */samples/c/* и откройте файл *facedetector.c*. Нарисуйте изображение черепа (или найдите в интернете) и сохраните его на диск. Измените программу *facedetector* для загрузки изображения черепа.
 - a. При обнаружении лица, нарисуйте изображение черепа в прямоугольнике.

Подсказка: изменить размер изображения можно при помощи `cvConvertImage()` или `cvResize()`. Активизировать часть изображения (область ROI) и поместить туда изображение черепа при помощи функции `cvCopy()`.

- b. Добавьте ползунок из 10 значений в интервале от 0.0 до 1.0. Используйте этот ползунок для альфа смещивания черепа и исходного изображения (задействуйте функцию `cvAddWeighted()`).
9. Стабилизация изображения. Перейдите в директорию `/samples/c/` и откройте код `Ikdemo` (код отслеживания движений или оптического потока). Создайте и отобразите видео в окне по размерам превышающие размеры кадра. Немного переместите камеру, используя вектора оптического потока для отображения изображения в окне большего размера. Это элементарная техника стабилизации изображения.

Обработка изображения

[П]|[РС]|(РП) Краткий обзор

На данный момент уже изучены основы OpenCV, необходимые для создания более сложных вещей. Имеется представление о структуре и основных типах данных библиотеки. Изучена библиотека HighGUI, позволяющая отображать получаемые результаты на экране.

Теперь настало время перейти к методам более высокого уровня, которые рассматривают изображения как изображения, а не только как массивы цветных (или черно-белых) значений. Когда говорится об "обработке изображения", имеется ввиду следующее: использование высокоуровневых операторов на изображении в целях выполнения задач, значения которых определены в контексте графических, визуальных образов.

[П]||[РС]||(РП) Сглаживание

Сглаживание или размытие - простая и часто используемая операция обработки изображения. Есть множество причин для сглаживания, но, как правило, данная операция используется для снижения шума или в артефактах камеры. Сглаживание так же применяется при уменьшении размера изображения (детали данной операции будут подробно рассмотрены в последующем разделе текущей главы).

В OpenCV есть пять различных операторов сглаживания. Все они поддерживаются через одну функцию *cvSmooth()*, которая принимает желаемый вариант сглаживания в качестве аргумента.

```
void cvSmooth(
    const CvArr*src
    ,CvArr* dst
    ,int smoothtype = CV_GAUSSIAN
    ,int param1     = 3
    ,int param2     = 0
    ,double param3 = 0
    ,double param4 = 0
);
```

src и *dst* – исходное и конечное изображения. Функция *cvSmooth()* имеет четыре неинформативных аргумента *param1*, *param2*, *param3* и *param4*. Смысл этих аргументов зависит от значения *smoothtype*, который принимает любое из пяти значений, указанных в таблице 5-1 (при некоторых значениях *smoothtype* использование одного изображения в качестве *src* и *dst* не допускается).

Таблица 5-1. Значения *smoothtype*

Тип сглаживания	Наименование	Использование одного изображения	Nc	Тип src	Тип dst
CV_BLUR	Простое размытие	Да	1,3	8u, 32f	8u, 32f
CV_BLUR_NO_SCALE	Простое размытие без масштабирования	Нет	1	8u	16s (для 8u src) или 32f (для 32f src)
CV_MEDIAN	Медианное размытие	Нет	1,3	8u	8u
CV_GAUSSIAN	Размытие по Гауссу	Да	1,3	8u, 32f	8u (для 8u src) или 32f (для 32f src)
CV_BILATERAL	Двусторонняя фильтрация	Нет	1,3		8u

CV_BLUR - простейшая операция размытия (рисунок 5-1). Каждый пиксель на выходе является средним арифметическим области пикселей на входе. Простое размытие поддерживает 1-4 канальные 8 битные или 32 битные вещественные изображения.

Не все операторы сглаживания действуют на одних и тех же видах изображения.

CV_BLUR_NO_SCALE (простое сглаживание без масштабирования) схоже с простым сглаживанием, только без последующего масштабирования. Для этого типа размытия

исходное и конечное изображения должны иметь различные форматы. Исходное изображение должно быть 8-битным изображением, тогда как конечное изображение должно быть *IPL_DEPTH_16S* (*CV_16S*) или *IPL_DEPTH_32S* (*CV_32S*).



Рисунок 5-1. Простое сглаживание; слева исходное изображение, справа конечное изображение

Операцию *CV_BLUR_NO_SCALE* так же возможно выполнять и на 32-битных вещественных изображениях, при условии, что результирующее изображение так же будет 32-битным вещественным. Для данной операции не должно использоваться в качестве *src* и *dst* одно и тоже изображение (это очевидно в случае 8 битного изображения, однако, для 32 битного изображения данное правило так же должно выполняться). Зачастую данный тип сглаживания предпочтительнее, нежели простое сглаживание, т.к. выполняется чуть быстрее.

Медианный фильтр (*CV_MEDIAN*) заменяет каждый пиксель на среднее значение области вокруг центрального пикселя. *CV_MEDIAN* работает с одно-, трёх- и четырёхканальными 8-битными изображениями, но исходное и конечное изображения должны быть разными. Результат работы медианного фильтра показан на рисунке 5-2. Простое сглаживание может быть чувствительно к шуму, в особенности на изображениях с большой изоляцией точек выброса ("дробовый шум"). Большие

различия даже в небольшом количестве точек может вызвать заметные сдвиги среднего значения. Медианный фильтр может игнорировать выбросы путем выбора средней точки.

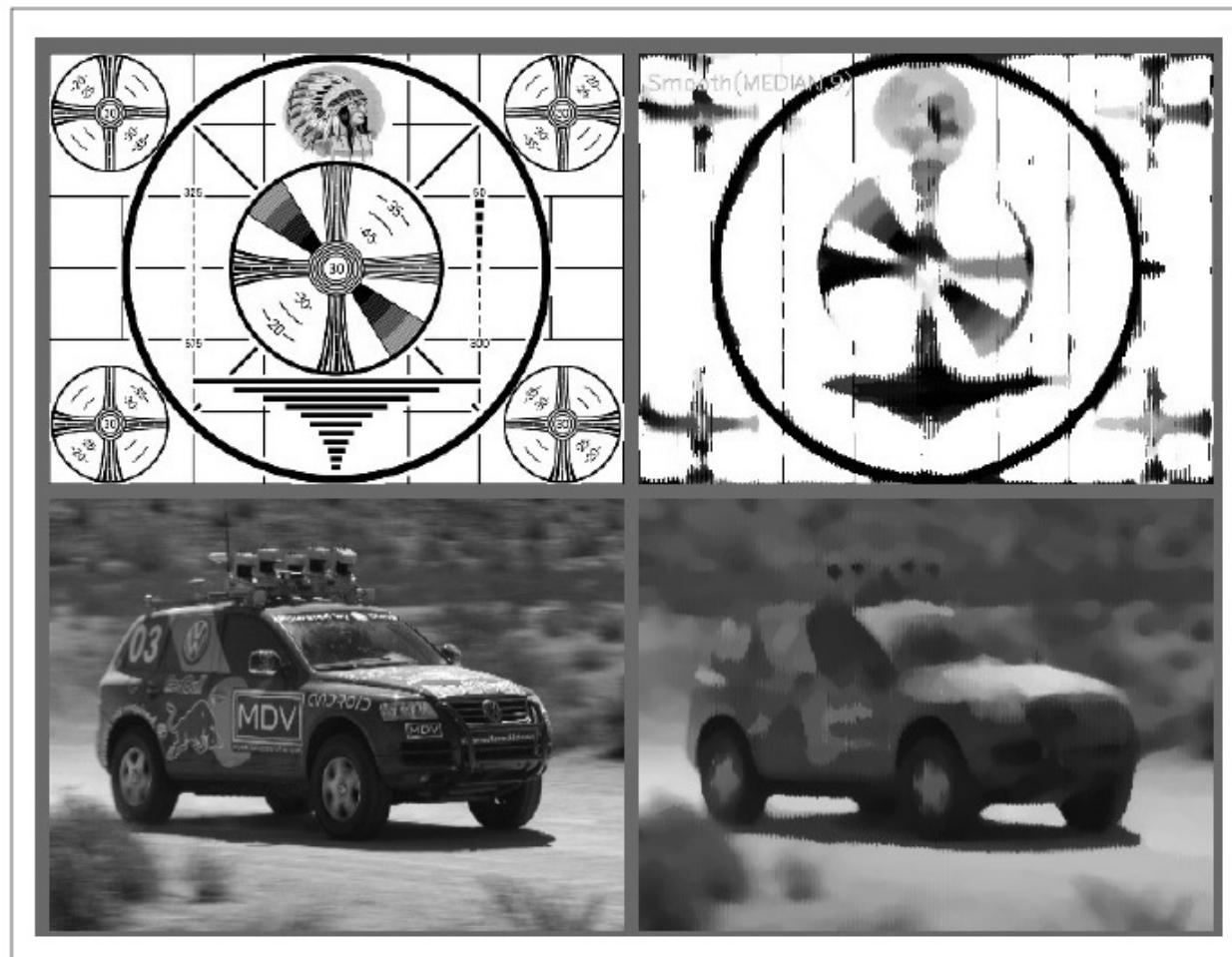


Рисунок 5-2. Медианное сглаживание

Размытие по Гауссу (`CV_GAUSSIAN`) является наиболее полезным, хотя и не самым быстрым. Гауссова фильтрация выполняется путем свертки каждой точки во входном массиве с гауссовым ядром и последующим суммированием для получения выходного массива.

Для сглаживания по Гауссу (рисунок 5-3) первые два аргумента задают ширину и высоту окна фильтрации; (необязательный) третий аргумент задает значение *sigma* гауссова ядра. Если третий аргумент не задан, то он будет рассчитан из ширины и высоты окна по следующим формулам:

$$\sigma_x = \left(\frac{n_x}{2} - 1 \right) \cdot 0.30 + 0.80, \quad n_x = \text{param1}$$

$$\sigma_y = \left(\frac{n_y}{2} - 1 \right) \cdot 0.30 + 0.80, \quad n_y = \text{param2}$$

Если необходимо асимметричное ядро, то необходимо задать (необязательный) четвертый аргумент; в этом случае третий и четвертый аргументы будут соответствовать сигме по горизонтали и вертикали, соответственно.

Если третий и четвёртый аргументы заданы, а первый и второй равны нулю, то они автоматически будут рассчитаны по значению сигмы.

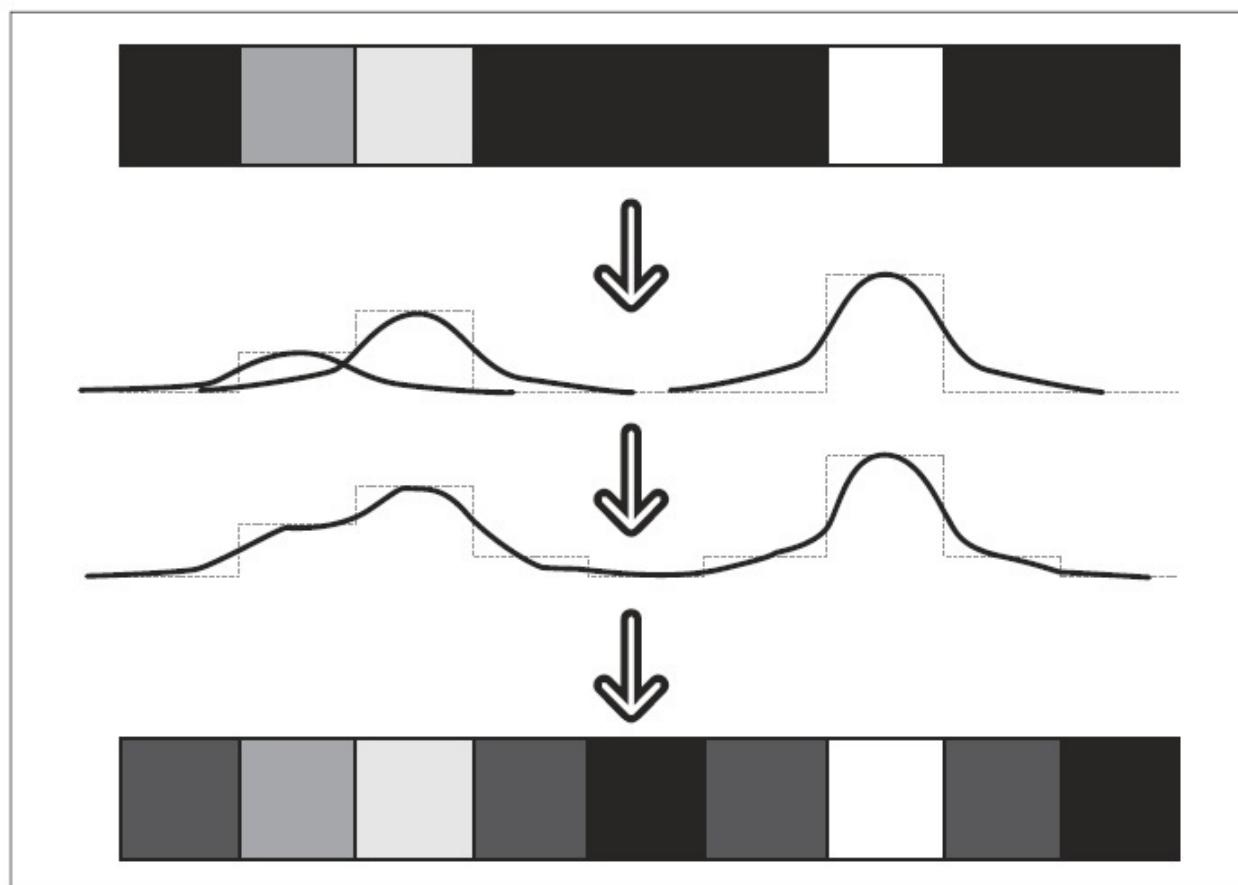


Рисунок 5-3. Размытие по Гауссу на одномерном массиве

Реализация Гауссово сглаживания в OpenCV обеспечивает высокую производительность для ряда распространенных ядер. Ядра размера 3x3, 5x5 и 7x7 имеют "стандартную" сигму (`param3 = 0.0`) и дают наилучшую производительность, чем другие ядра. Размытие по Гауссу поддерживает одно- и трехканальные 8-битные или 32-битные вещественные изображения. В качестве входного и выходного может быть использовано одно и тоже изображение. Результат размытия по Гауссу показан на рисунке 5-4.



Рисунок 5-4. Размытие по Гауссу

Пример двусторонней фильтрации представлен на рисунке 5-5. Данный тип сглаживания более известен как сглаживание с сохранением краев. Проще всего разобраться в принципе работы двусторонней фильтрации можно сравнив его с гауссовым размытием. Размытие по Гауссу снижает уровень шума при сохранении сигнала, но, к сожалению, плохо работает вблизи краёв, которые также размываются. Двусторонняя фильтрация в отличие от Гауссовой по производительности ниже, потому, как предоставляет средства размытия изображения без сглаживания краёв.

Так же, как и в гауссовом сглаживании, в двухстороннем сглаживании создается средневзвешенное каждого пикселя и их соседей. Взвешенное значение состоит из двух компонент, первая из которых использует такой же вес как при Гауссовом сглаживании. Вторая компонента так же основана на весе, используемый в Гауссовом сглаживании, однако, используется не пространственное расстояние от центральной точки, а разница в интенсивности от центральной точки. У более схожих пикселей веса больше, чем у менее схожих. Данное сглаживание может быть полезно как вспомогательное средство при сегментации изображения.

Двусторонняя фильтрация использует два параметра. Первый задает ширину Гауссово ядра в пространственной области, который является аналогом параметра сигма в Гауссовом фильтре. Второй задает ширину Гауссово ядра в области цвета. Чем больше второй параметр, тем шире диапазон интенсивности (или цвета), который будет включен в сглаживание.



Рисунок 5-5. Двустороннее сглаживание

[П] | [РС] | (РП) Морфологические преобразования

OpenCV предоставляет быстрый и простой в использовании интерфейс для выполнения морфологических преобразований над изображениями. Основные морфологические преобразования - расширение и размытие (сужение) - могут быть применены в таких ситуациях, как удаление шума, выделение отдельных элементов и соединение разнородных элементов на изображении. Морфологические преобразования так же могут быть использованы для поиска неровностей интенсивности или отверстий и градиента изображения.

Расширение и размытие

Расширение - свёртка изображения (или области изображения), которое именуется **A**, с некоторым ядром, которое именуется **B**. Ядро, которое может быть любой формы и размера, имеет одну определенную точку привязки (якорь). Чаще всего ядро — это небольшой сплошной квадрат или диск с якорем в центре. Ядро можно рассматривать в качестве шаблона или маски, и его эффект на расширение зависит от оператора локального максимума. При сканировании изображения ядром **B** происходит вычисление локального максимума пикселя, перекрываемого **B**, и затем значение пикселя, лежащего под опорной точкой заменяется этим максимальным значением. Это приводит к появлению ярких областей на изображении; схематично этот рост показан на рисунке 5-6. Этот рост именуется "расширением оператора".

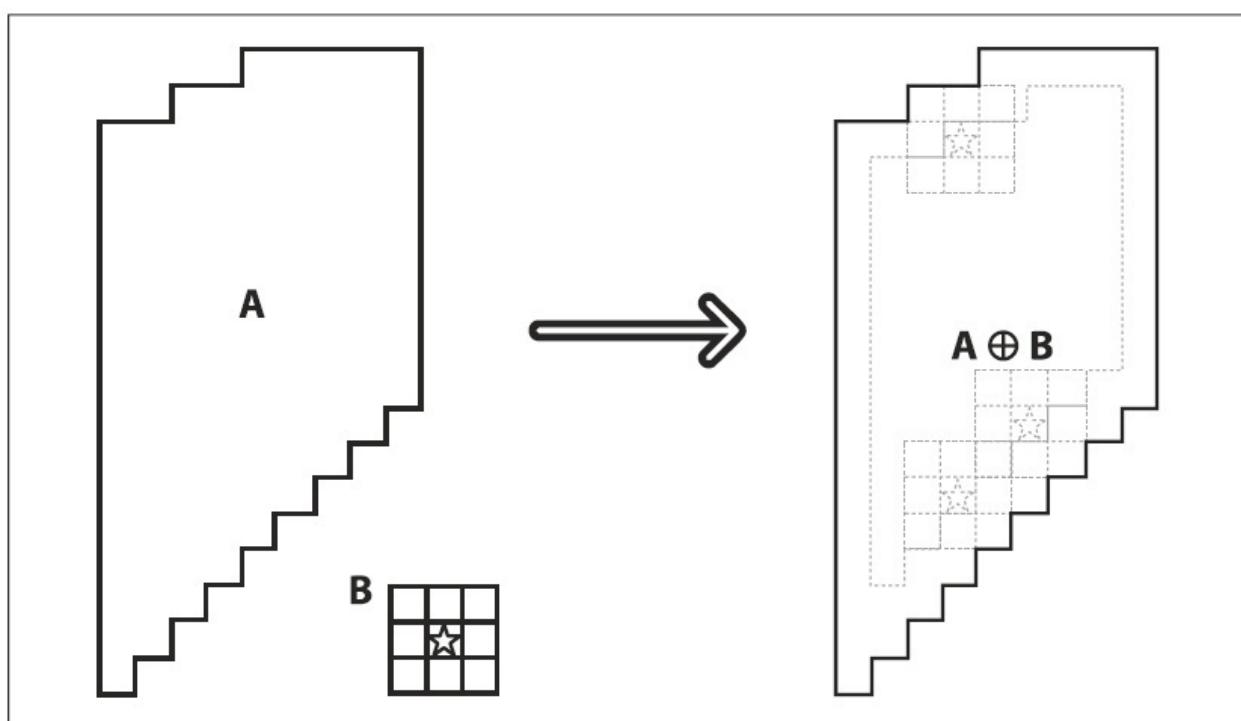


Рисунок 5-6. Морфологическое расширение

Размытие - обратная операция. Действие оператора размытия заключается в вычислении локального минимума под ядром. Данный оператор создаёт новое изображение на основе исходного по следующему алгоритму: при сканировании изображения ядром **B** происходит вычисление локального минимума пикселя, перекрываемого **B**, и затем значение пикселя, лежащего под опорной точкой заменяется этим минимальным значением. Схематично сужение показано на рисунке 5-7.

Морфологические преобразования чаще всего выполняются над бинарными изображениями, получаемые с помощью порогового преобразования.

Расширение расширяет область **A**, а размытие уменьшает область **A**. К тому же, расширение применяется, как правило, чтобы сгладить вкрапления, а размытие, чтобы сгладить выступы. И все же конечный результат будет зависеть от ядра.

В OpenCV для выполнения этих преобразований используются функции *cvErode()* и *cvDilate()*:

```
void cvErode(
    IplImage*      src
    ,IplImage*      dst
    ,IplConvKernel* B = NULL
    ,int           iterations = 1
);

void cvDilate(
    IplImage*      src
    ,IplImage*      dst
    ,IplConvKernel* B = NULL
    ,int           iterations = 1
);
```

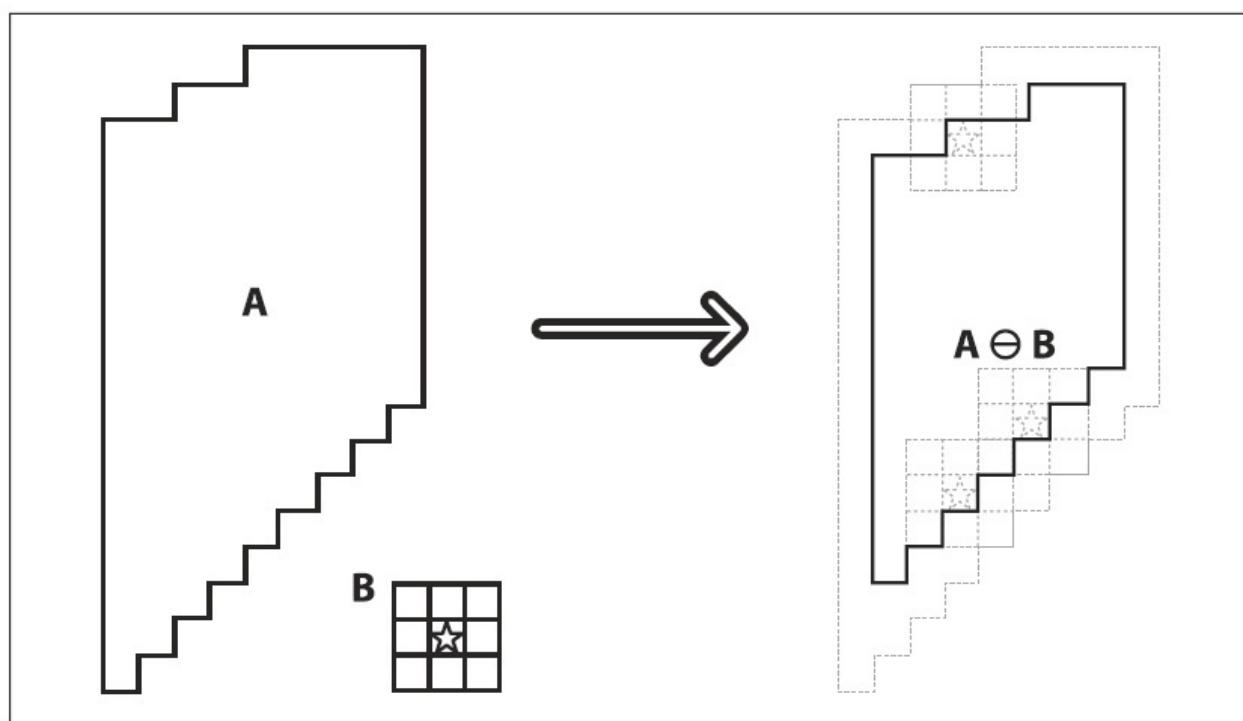


Рисунок 5-7. Морфологическое сужение

Обе функции допускают использование одного и того же изображения в качестве исходного и конечного. Третий аргумент - ядро, которое по умолчанию равно NULL. Если ядро рано NULL, то используется ядро 3x3 с якорем в центре. Четвертый аргумент - количество итераций. Если указанное значение не равно 1, то операция будет применяться несколько раз в течение одного вызова функции. Результат размытия показан на рисунке 5-8, а расширения на рисунке 5-9. Операция размытия зачастую используется для устранения "вкраплений" шума на изображении. Суть операции сужения в том, что вкрапления и шумы размываются, в то время как большие и соответственно более значимые регионы не затрагиваются. Операция расширения зачастую используется при попытке найти связные компоненты (т.е. большие отдельные области аналогичного цвета или интенсивности). Полезность расширения возникает во многом из-за того, что, большая область разбита на несколько более мелких частей шумами, тенями или какими-то аналогичными эффектами. Применение небольшого растягивания приводит к тому, что эти области "плавятся" в одну.

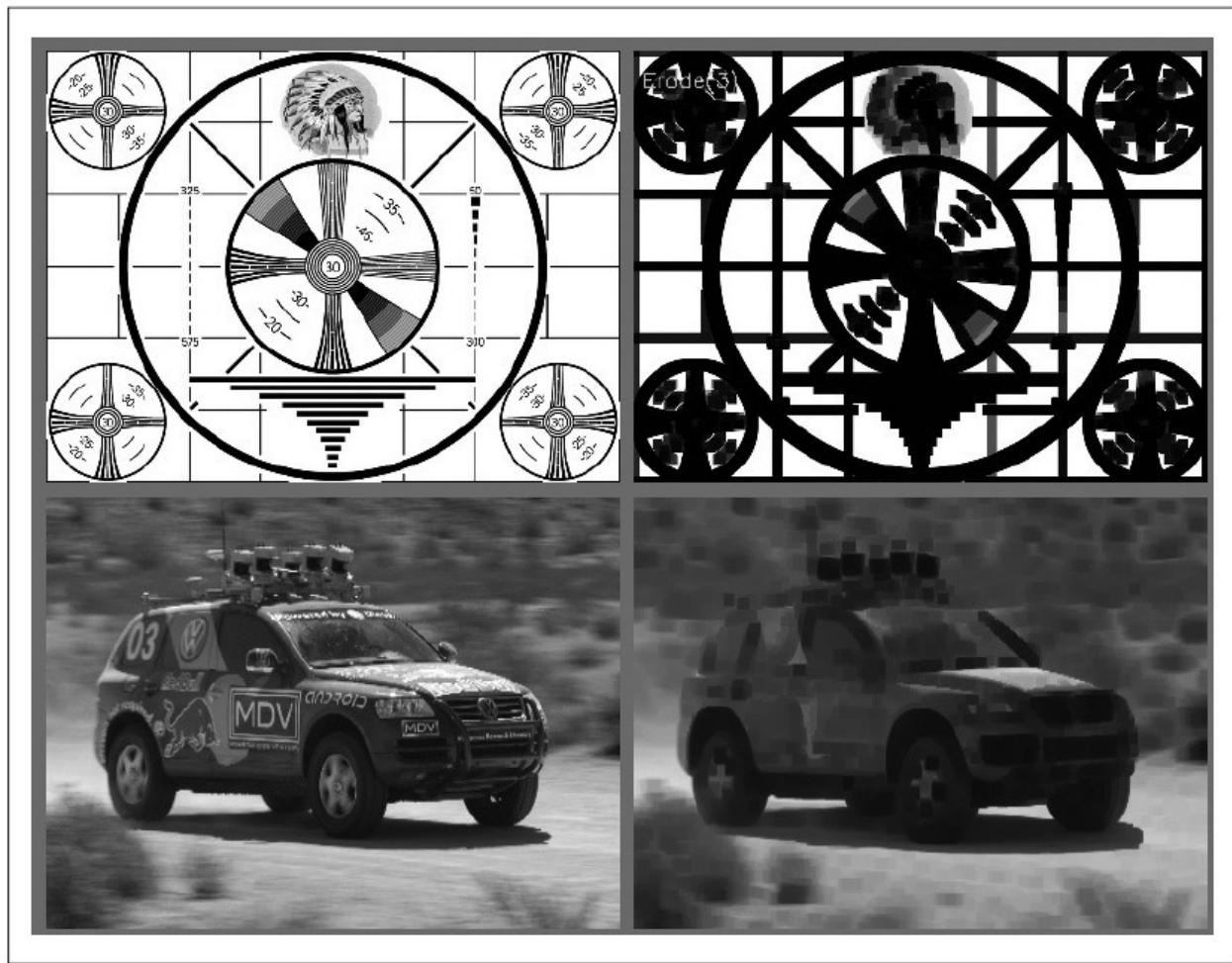


Рисунок 5-8. Результат размытия

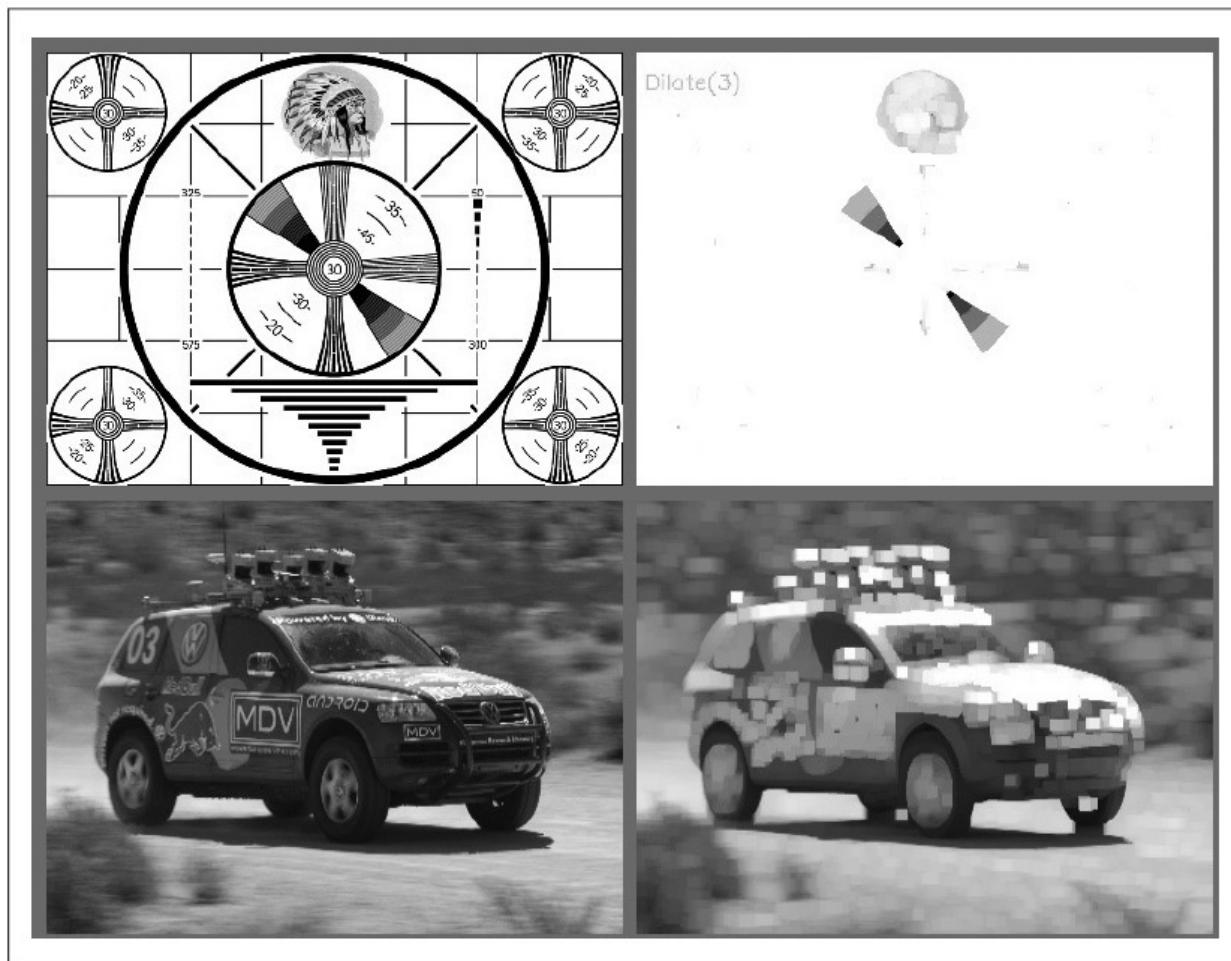


Рисунок 5-9. Результат расширения

Напоминание: во время выполнения функции `cvErode()` значение некоторой точки p устанавливается в минимальное значение всех точек охватываемых ядром, выровненного относительно p , и наоборот в `cvDilate()` значение устанавливается в максимальное:

$$\text{erode}(x, y) = \min_{(x', y') \in \text{kernel}} \text{src}(x + x', y + y')$$

$$\text{dilate}(x, y) = \max_{(x', y') \in \text{kernel}} \text{src}(x + x', y + y')$$

Может показаться странным, зачем знать столь сложные формулы, когда ранее описанного эвристического подхода вполне достаточно. Некоторые читатели на самом деле предпочитают знать эти формулы, но что более важно, формулы охватывают некоторые утверждения общего характера, которые не проявляются в качественном

описании. Если изображение не бинарное, то операторы размытия и расширения работают менее тривиально. В качестве наглядного подтверждения данного утверждения обратитесь еще раз к рисункам 5-8 и 5-9.

Создание собственного ядра

Для создания собственного ядра необходимо воспользоваться структурой *IplConvKernel* и функцией *cvCreateStructuringElementEx()*. Для освобождения занимаемой ядром памяти необходимо использовать функцию *cvReleaseStructuringElement()*.

```
IplConvKernel* cvCreateStructuringElement(
    int     cols
    ,int    rows
    ,int    anchor_x
    ,int    anchor_y
    ,int    shape
    ,int*   values = NULL
);

void cvReleaseStructuringElement( IplConvKernel** element );
```

Морфологическое ядро в отличии от ядра свертки не требует никаких числовых значений. Элементы ядра просто указывают на то место, где будет вычислено максимальное или минимальное значение. Якорь определяет каким образом ядро будет выравниваться на исходном изображении, а так же место размещения вычисляемого значения на результирующем изображении. При создании ядра, *cols* и *rows* определяют размер прямоугольника. Параметры *anchor_x* и *anchor_y* - это координаты якоря в пределах прямоугольника. Параметр *shape* может принимать любое значение из таблицы 5-2. Если использовать *CV_SHAPE_CUSTOM*, то целочисленный вектор *values* используется для определения пользовательской формы ядра в пределах прямоугольника размера *rows* x *cols*. Этот вектор считывается в порядке сканирования раstra. Любое ненулевое значение указывает на то, что пиксель должен быть включен в ядро. Если *values = NULL*, тогда пользовательское ядро интерпретируется как ненулевое и как результат используется ядро прямоугольной формы.

Таблица 5-2. Значения параметра *shape*

Значение	Смысл
CV_SHAPE_RECT	Прямоугольное ядро
CV_SHAPE_CROSS	Крестообразное ядро
CV_SHAPE_ELLIPSE	Эллиптическое ядро
CV_SHAPE_CUSTOM	Ядро пользовательской формы

Сложные морфологические преобразования

Во время работы с двоичными изображениями или масками, операторов расширения и сужения вполне достаточно. Но при работе с цветными или черно-белыми изображениями ряд дополнительных операторов могли бы быть полезны. *cvMorphologyEx()* является наиболее полезной функцией.

```
void cvMorphologyEx(
    const CvArr*    src
    ,CvArr*        dst
    ,CvArr*        temp
    ,IplConvKernel* element
    ,int           operation
    ,int           iterations = 1
);
```

В дополнение к аргументам *src*, *dst*, *element* и *iterations*, которые были рассмотрены в предыдущем разделе, функция *cvMorphologyEx()* имеет ещё два дополнительных параметра. Первый - временный массив *temp*, который используется в ряде операций (таблица 5-3). При необходимости, этот массив должен быть того же размера, что и исходное изображение. Второй новый аргумент *operation* - тип морфологического преобразования.

Таблица 5-3. Морфологические преобразования поддерживаемые *cvMorphologyEx()*

Значение	Морфологический оператор	Потребность во временном изображении
CV_MOP_OPEN	Сначала сужение, а затем расширение	Нет
CV_MOP_CLOSE	Сначала расширение, а затем сужение	Нет
CV_MOP_GRADIENT	Морфологический градиент	Всегда
CV_MOP_TOPHAT	Изоляция ярких регионов	Только если src == dst
CV_MOP_BLACKHAT	Изоляция темных регионов	Только если src == dst

Открытие и закрытие

Первые две операции из таблицы 5-3 - открытие и закрытие — это комбинация операций сужения и расширения. В случае открытия сначала выполняется сужение, а затем расширение (рисунок 5-10). Открытие зачастую используется для подсчета количества регионов на двоичном изображении. Например, имея изображение клеток на стекле под микроскопом после порогового преобразования, можно воспользоваться операцией открытия, чтобы выделить клетки, которые находятся друг около друга, прежде чем подсчитывать количество регионов. В случае с закрытием, сначала выполняется расширение, а затем сужение (рисунок 5-12). Закрытие зачастую используется для устранения нежелательных шумов. Для связных компонентов, как правило, сначала выполняется операция сужения или закрытия, чтобы устраниить элементы, появляющиеся из-за шума, а затем операция открытия для объединения близлежащих крупных элементов (хотя результат использования операций открытия и закрытия схож с результатом действия расширения и сужения, эти новые операции, как правило, сохраняют площадь регионов более точно).

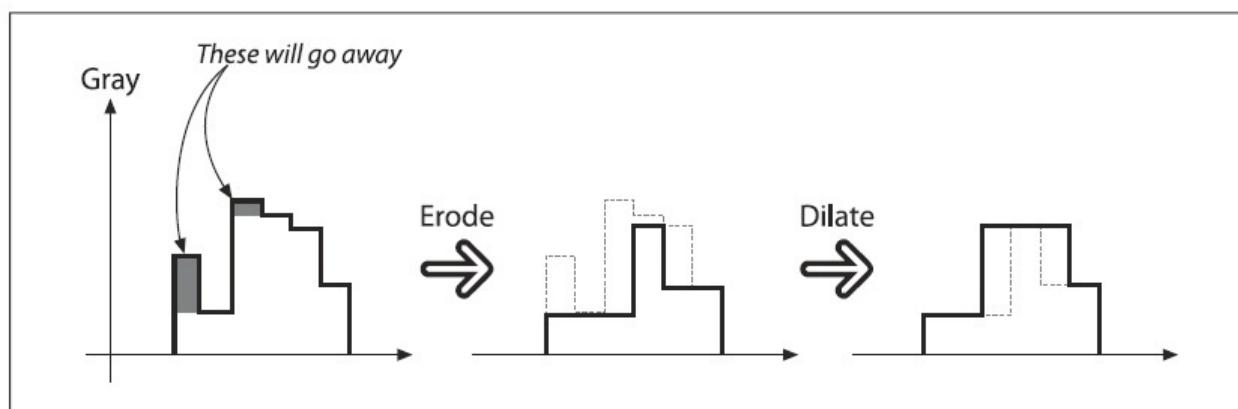


Рисунок 5-10. Морфологическая операция открытие

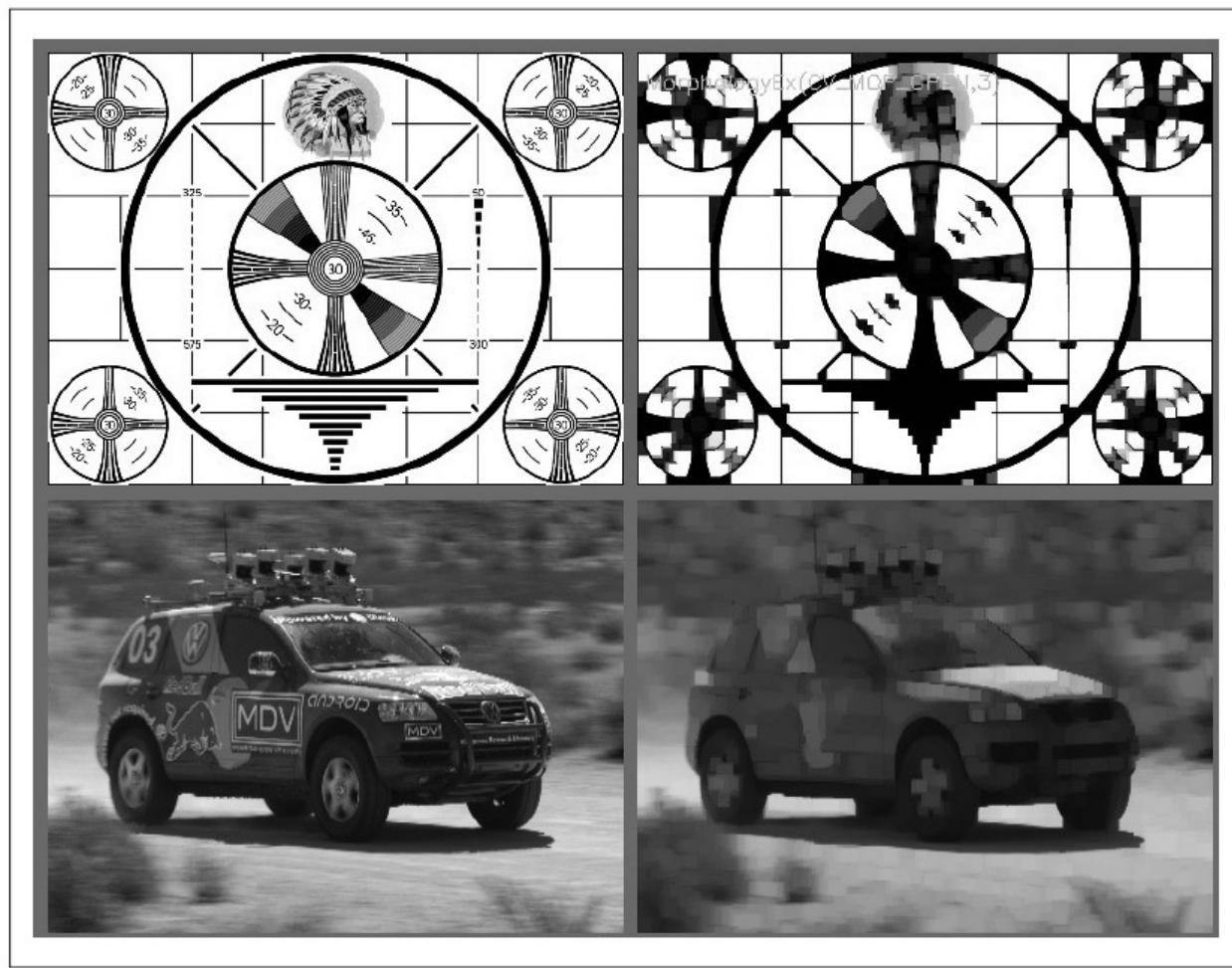


Рисунок 5-11. Результат операции морфологического открытия: мелкие яркие регионы удалены, а остальные яркие регионы изолированы, сохранив свои размеры

Эффект от операции закрытия - устранение локальных выбросов, которые ниже чем у "соседей", а эффект от открытия - устранение выбросов, которые выше чем у "соседей". Результат операции открытия представлен на рисунке 5-11, закрытия на рисунке 5-13.

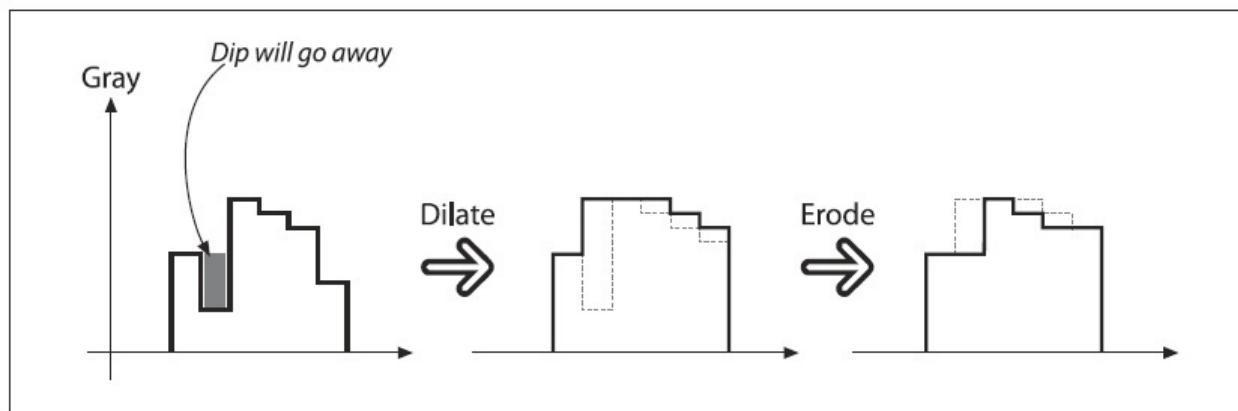


Рисунок 5-12. Морфологическая операция закрытия

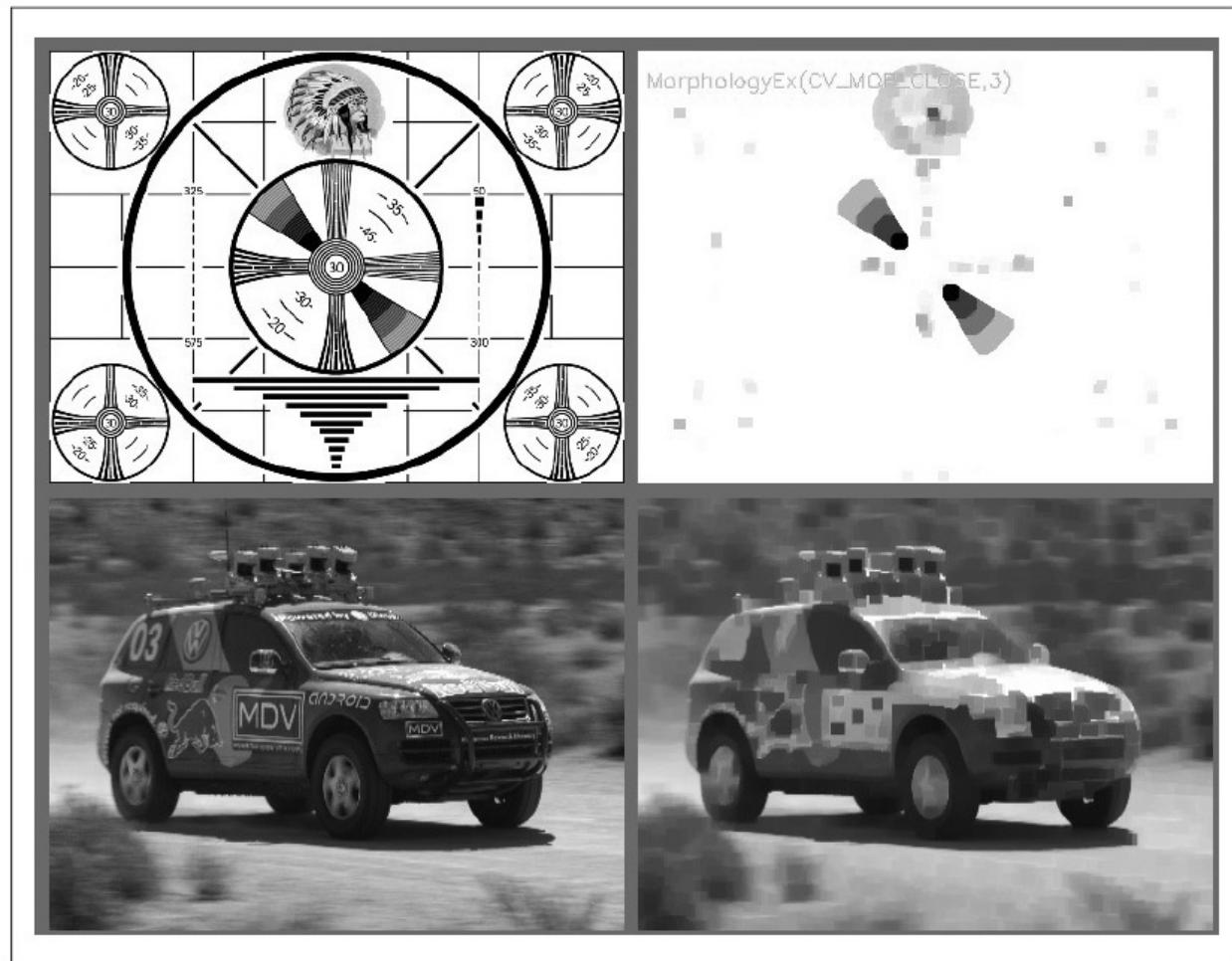


Рисунок 5-13. Результат операции морфологического закрытия: объединение ярких областей, при сохранении своих базовых размеров

И последнее замечание по операциям открытия и закрытия, касающееся количества итераций. Можно было бы ожидать, что две операции закрытия дадут что-то вроде dilate-erode-dilate-erode. Но, как оказалось, это не очень полезно, поэтому преобразование для этого случая происходит следующим образом: dilate-dilate-erode-erode. Таким образом исчезнут не только одиночные, но и соседние пары выбросов.

Морфологический градиент

Наверное, будет проще начать с формулы, а затем разбираться что она означает:

$$\text{gradient(src)} = \text{dilate(src)} - \text{erode(src)}$$

Результат данной операции над двоичным изображением - выделение периметров существующих пятен. Этот процесс схематически изображен на рисунке 5-14, а на рисунке 5-15 показан результат операции.

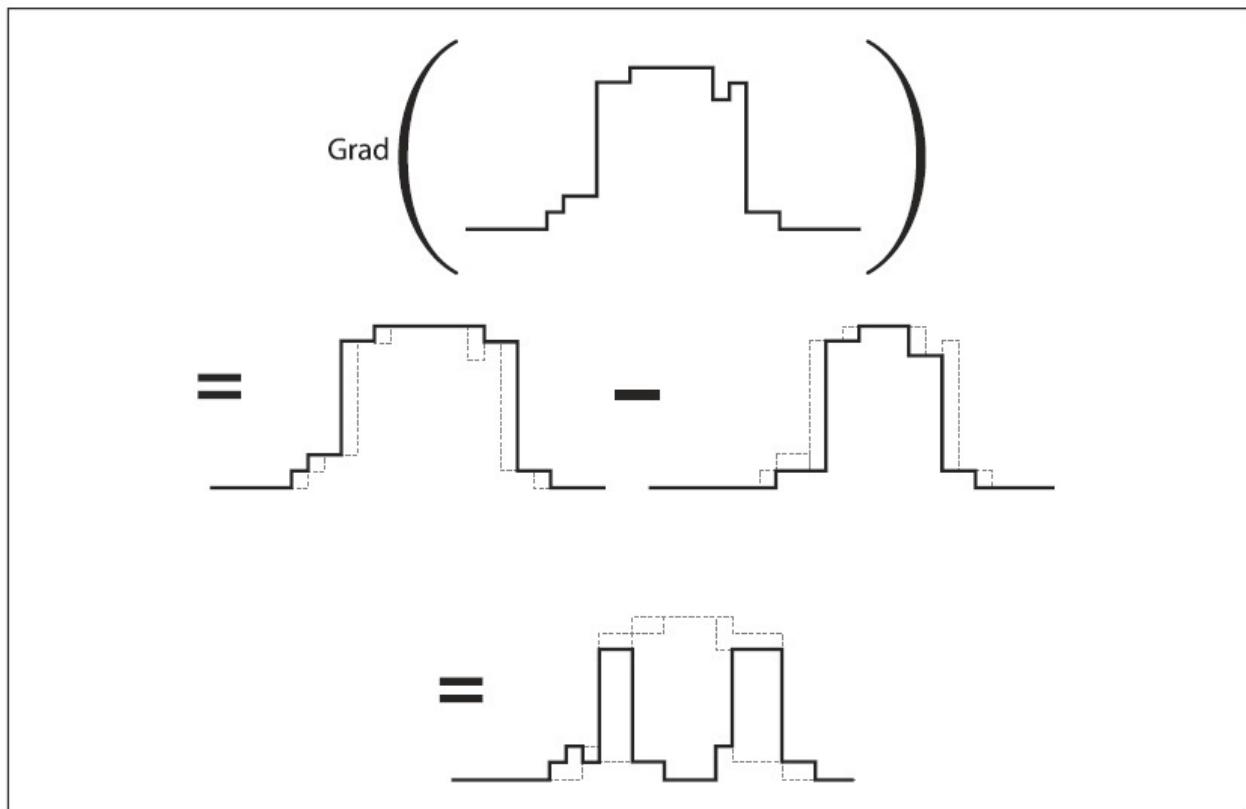


Рисунок 5-14. Применение оператора морфологического градиента к черно-белому изображению: как и ожидалось оператор получает наивысшие значения в наиболее быстро меняющихся местах изображения

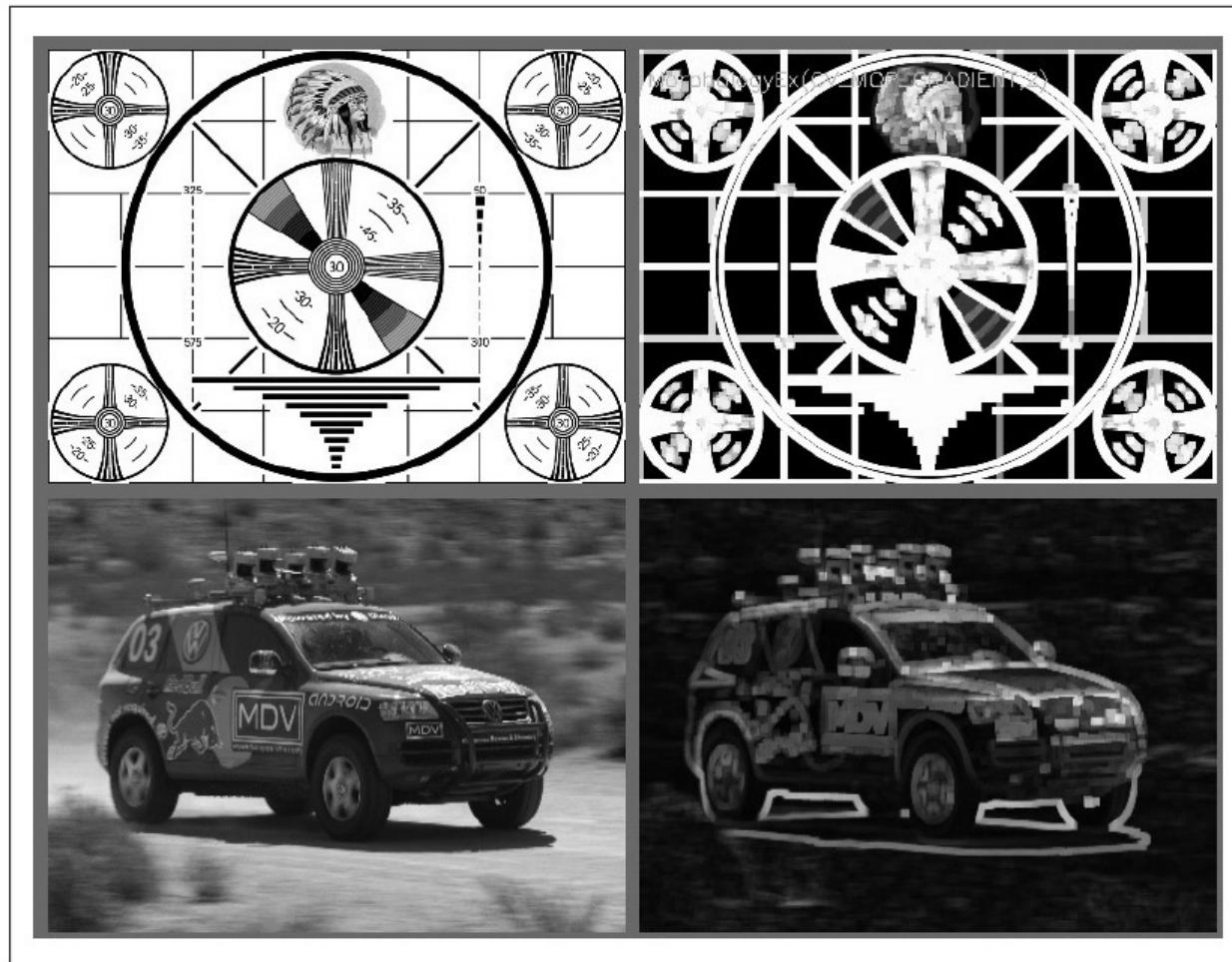


Рисунок 5-15. Результат операции мифологического градиента: определение ярких краев по периметру

На чёрно-белом изображении данный оператор будет показывать, как быстро меняется яркость, поэтому он и называется "морфологический градиент". Морфологический градиент зачастую используется в тех случаях, когда требуется изолировать периметры ярких областей для возможности рассмотреть их как целые объекты (или как целевые части объектов).

Изоляция ярких и темных регионов

Последние два оператора **Top Hat** и **Black Hat** используются для изоляции более ярких и более темных регионов, чем их соседи. Данные операторы могут быть использованы для изоляции частей объекта, которые проявляют изменение яркости по отношению к объекту, к которому они присоединены. Это зачастую встречается, например, при обработке изображений клеток под микроскопом. Обе операции основаны на более примитивных операторах:

$$\text{TopHat(src)} = \text{src} - \text{open(src)}$$

$$\text{BlackHat(src)} = \text{close(src)} - \text{src}$$

Оператор *Top Hat* вычитает результат выполнения операции открытия на исходном изображении из исходного изображения. Данная операция должна выявить области, которые светлее остальных (рисунок 5-16); и наоборот оператор *Black Hat* показывает области, которые темнее остальных (рисунок 5-17). Результаты всех морфологических преобразований, обсуждаемых в данной главе, собраны на рисунке 5-18.

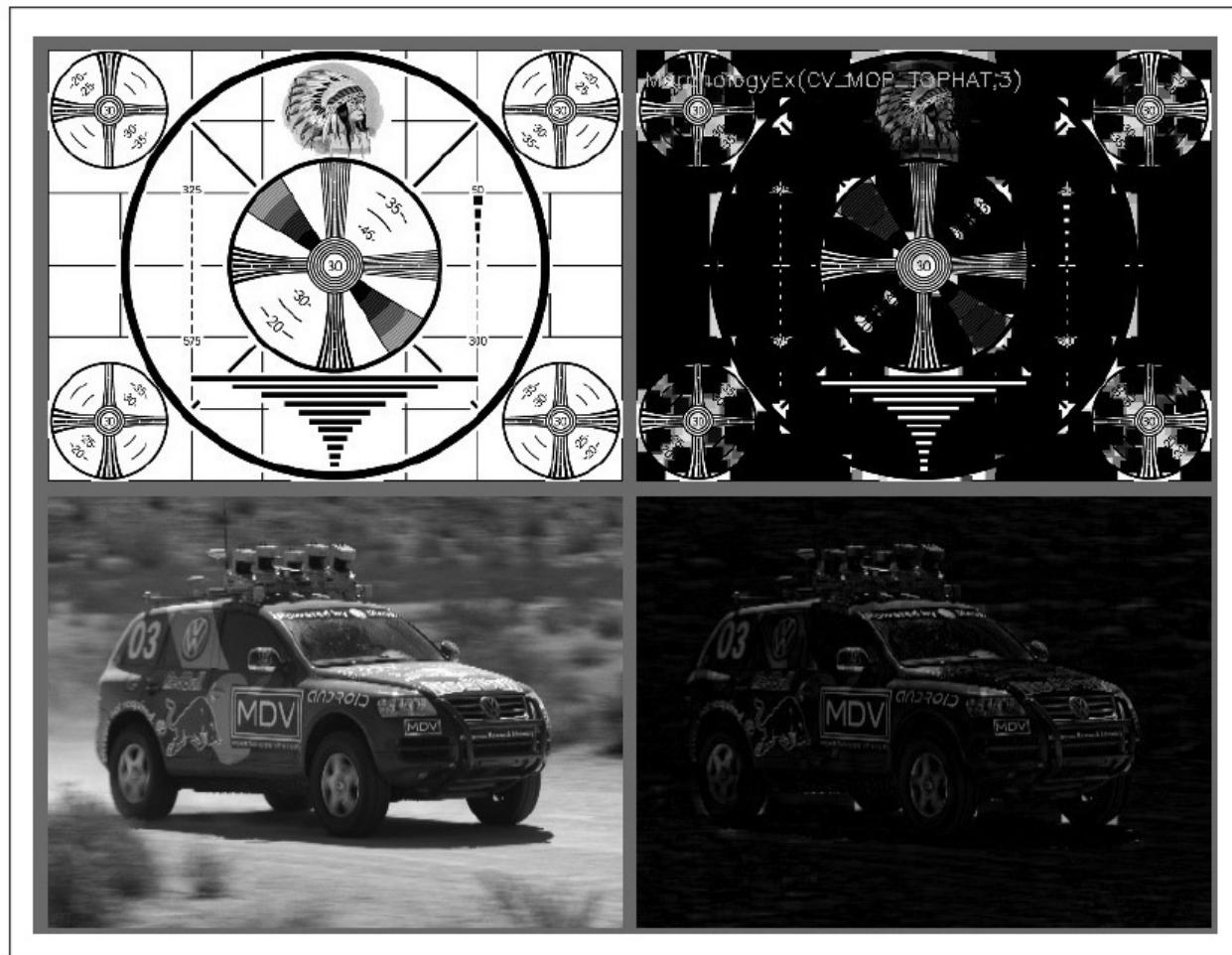


Рисунок 5-16. Результат работы операции морфологический Топ Нат: изоляция пиковых ярких мест



Рисунок 5-17. Результат работы операции морфологический Black Hat: изоляция пиковых темных мест

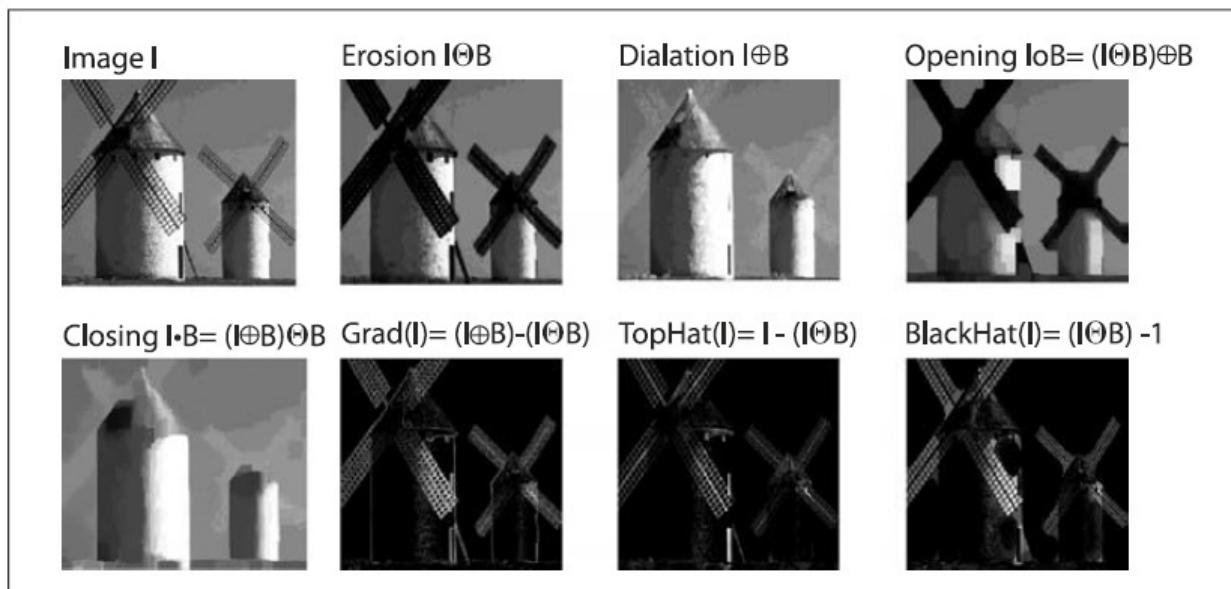


Рисунок 5-18. Результаты всех морфологических преобразований

[П]||[РС]||(РП) Заливка

Заливка - чрезвычайно полезная функция, которая зачастую используется для того, чтобы отметить или выделить часть изображения для дальнейшей обработки или анализа. Заливка может быть использована для получения маски из исходного изображения для использования в последующих процедурах, чтобы ускорить обработку или ограничить обработку только тех пикселей, которые указаны в маске. Между тем, сама функция `cvFloodFill()` принимает в качестве необязательного аргумента маску, которая в дальнейшем может быть использована для управления процессом заливки (например, для выполнения нескольких заливок на одном изображении).

В OpenCV функция заливки имеет более обобщенную идею, нежели чем в стандартных программах для рисования. Для обоих случаев выбирается начальная точка на изображении, а затем все подобные соседние точки окрашиваются в однородный цвет. Разница здесь заключается в том, что не все соседние пиксели должны быть одинакового цвета. Результатом операции заливки всегда будет одна непрерывная область. Функция `cvFloodFill()` будет закрашивать соседний пиксель, если он находится в пределах указанного диапазона (*loDiff* и *upDiff*) текущего пикселя или если (в зависимости от настроек флагов) соседний пиксель находится в пределах указанного диапазона для точки *seedPoint*. Заливка также может быть ограничена за счет дополнительного аргумента маски. Прототип функции заливки:

```
void cvFloodFill(
    IplImage*           img
    ,CvPoint            seedPoint
    ,CvScalar           newVal
    ,CvScalar           loDiff  = cvScalarAll(0)
    ,CvScalar           upDiff  = cvScalarAll(0)
    ,CvConnectedComp*  comp    = NULL
    ,int                flags   = 4
    ,CvArr*             mask    = NULL
);
```

img - исходное изображение, которое может быть 8-битным или вещественным и одно- или трехканальным. *seedPoint* - начальная точка для заливки, *newVal* - значение цвета заливки. Пиксель будет раскрашен, если его интенсивность не меньше, чем интенсивность закрашенного соседа минус *loDiff* и не больше, чем интенсивность закрашенного соседа плюс *upDiff*. Если *flags* содержит `CV_FLOODFILL_FIXED_RANGE`, то пиксель будет сравниваться с начальной точкой, а

не с соседними пикселями. Если `comp != NULL`, то в структуре `CvConnectedComp` будет содержаться статистика заполненных областей. Первый пример заливки показан на рисунке 5-19.

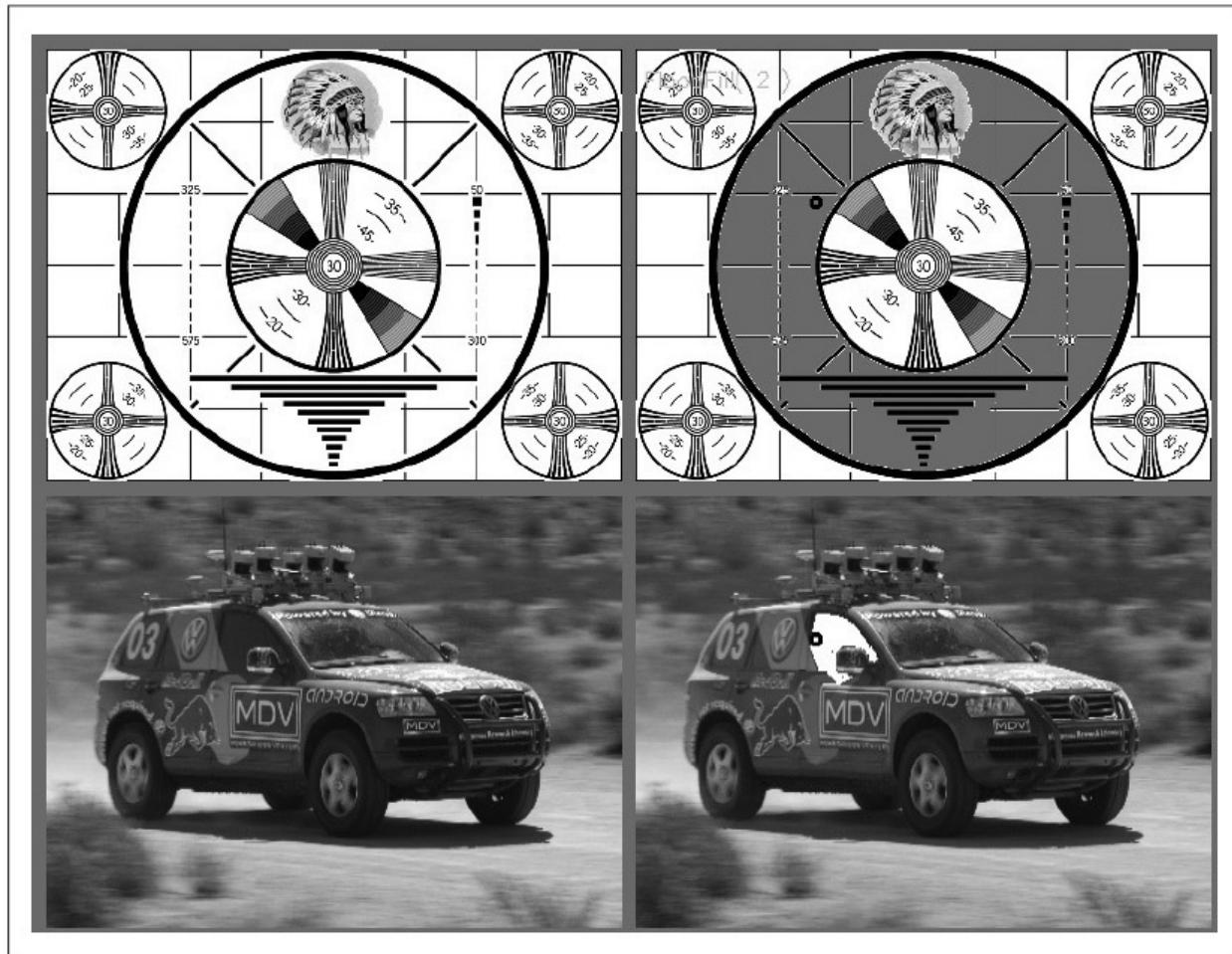


Рисунок 5-19. Результат заливки тестового изображения (верхнее изображение залито серым, а нижнее белым). Параметры `IoDiff` и `upDiff` равны 7.0 для каждого изображения

Аргумент `mask` устанавливает маску, которая может быть использована в качестве входных (ограничивает рабочую область) или выходных (указывает на заполненные регионы) данных для `cvFloodFill()`. Если `mask != NULL`, то маска должна быть одноканальной, 8-битной и по размеру на 2 пикселя больше по ширине и высоте, чем исходное изображение (данний факт ускоряет внутренний алгоритм обработки).

Пиксель $(x+1, y+1)$ маски соответствует пикселию (x, y) на исходном изображении.

Функция `cvFloodFill()` не будет закрашивать пиксели на исходном изображении, соответствующие ненулевым элементам маски.

При использовании маски, её значения должны быть заданы средним битом (8-15) значения флага. Если эти биты не установлены, то значения маски устанавливаются значениями по умолчанию - 1.

Аргумент *flags* довольно таки сложен, т.к. состоит из трех частей. Его младшие 8 бит (0-7) содержат значение связности и могут равняться 4 или 8. Если установлено значение 4, то в процессе заливки участвуют соседние пиксели по горизонтали и по вертикали, а если 8, то участвуют ещё и соседние пиксели по диагонали. Старшие 8 бит (16-23) могут быть установлены флагом *CV_FLOODFILL_FIXED_RANGE* (используется разница между текущим и начальным пикселями, в противном случае - текущего и соседних) и/или *CV_FLOODFILL_MASK_ONLY* (заполняется только область, соответствующая маске). При этом во втором случае маска обязательно должна быть задана. Средний бит (8-15) хранит значение для маски. Если средний бит установлен в 0s, то значения маски будут установлены в 1s. Флаги могут быть скомбинированы с помощью операции *OR*. Например, если требуется 8-связная заливка, фиксированного диапазона, ограниченная маской, заполненная значением 47, то необходимо написать следующее:

```
flags = 8
| CV_FLOODFILL_MASK_ONLY
| CV_FLOODFILL_FIXED_RANGE
| (47<<8);
```

На рисунке 5-20 изображено тестовое изображение после заливки. Следует отметить что *newVal*, *loDiff* и *upDiff* имеют тип *CvScalar*, поэтому они могут быть установлены для трёх каналов изображения одновременно. Например, если *lowDiff* = *CV_RGB(20,30,40)*, то будут установлены *lowDiff* пороги - 20 для красного, 30 для зелёного и 40 для синего.

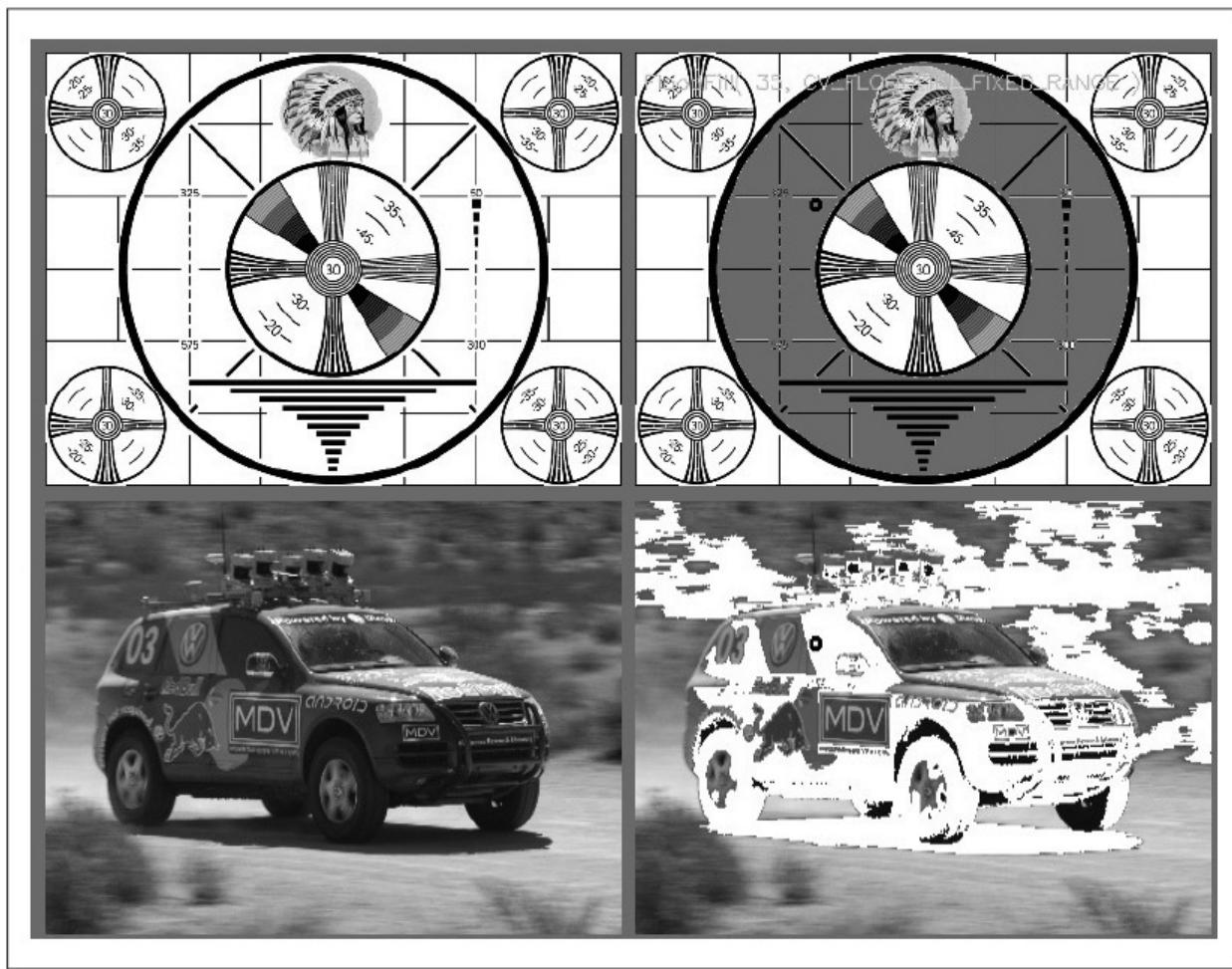


Рисунок 5-20. Результаты заливки тестового изображения (верхнее изображение залито серым, а нижнее белым). В данном случае заливка была выполнена с фиксированным диапазоном. Параметры *loDiff* и *upDiff* равны 25.0

[П]||[РС]||(РП) Изменение размера

Зачастую приходиться обрабатывать изображения неподходящего размера. Поэтому возникает потребность в функции, которая могла бы уменьшать или увеличивать исходное изображение. Для решения подобного рода задач в OpenCV есть функция `cvResize()`. Эта функция подгоняет исходное изображение под конечное изображение. Если задан ROI (регион интереса) на исходном изображении, то оно будет промасштабировано и подогнано под конечное изображение. Точно также, если в конечном изображении установлен ROI, тогда исходное изображение будет перемасштабировано под размеры этого региона.

```
void cvResize(
    const CvArr*   src
    ,CvArr*        dst
    ,int           interpolation = CV_INTER_LINEAR
);
```

Последний аргумент - это метод интерполяции (по умолчанию - линейная интерполяция). Другие доступные опции показаны в Таблице 5-4.

Таблица 5-4. Опции интерполяции

Интерполяция	Значение
CV_INTER_NN	Ближайшие соседи
CV_INTER_LINEAR	Билинейная
CV_INTER_AREA	re-sampling по области пикселя
CV_INTER_CUBIC	Бикубическая

В общем случае требуется получить наиболее сглаженное преобразование исходного изображения. Аргумент *interpolation* контролирует методику преобразования. Интерполяция возникает в момент уменьшения исходного изображения и пиксель на конечном изображении как бы попадает между пикселями на исходном изображении. Интерполяция так же возникает в момент увеличения исходного изображения. В любом случае, существует несколько вариантов для вычисления значений таких пикселей. Простейший подход - получить значение масштабированного пикселя за счет ближайшего пикселя на исходном изображении - это соответствует опции `CV_INTER_NN`. Другой вариант - линейное взвешивание пикселей блоками по 2x2 пикселя вокруг исходного пикселя учитывая их близость к пикслю-приемнику (опция `CV_INTER_LINEAR`). Так же возможно мысленно поместить масштабный пиксель

поверх старых пикселей, а затем усреднить значения в них (опция *CV_INTER_AREA*). И последний вариант - подгонка кубического сплайна под блок 4x4 пикселя в исходном изображении с последующим рассчётом соответствующего значения из подогнанного сплайна (опция *CV_INTER_CUBIC*).

[П]||[РС]||(РП) Пирамиды изображений

Пирамиды изображений активно используются в разнообразных приложениях компьютерного зрения. Пирамида изображения - это коллекция изображений, получаемая из исходного изображения путём последовательного сжатия, пока не будет достигнута точка останова (естественно, конечной точкой может быть один пиксель).

Существует два вида пирамид, которые наиболее часто можно встретить в литературе и приложениях: пирамиды Гаусса и Лапласа. Пирамида Гаусса используется для сжатия изображения, а пирамида Лапласа для восстановления изображения с повышенной дискретизацией из слоя в пирамиде.

Для получения слоя $(i+1)$ в Гауссовой пирамиде (пусть будет G_{i+1}) из слоя G_i , необходимо сначала свернуть этот слой с помощью Гауссово ядра, а затем удалить все четные строки и столбцы. Из этого следует, что каждое последующее изображение будет занимать четверть площади предшественника. Итерационный процесс на исходном изображении G_0 дает ту самую пирамиду. OpenCV предоставляет метод генерации каждого слоя на основе предыдущего изображения:

```
void cvPyrDown(
    IplImage* src
    ,IplImage* dst
    ,IplFilter filter = IPL_GAUSSIAN_5x5
);
```

На момент написания книги, параметр *filter* поддерживал только Гауссово ядро 5x5.

Аналогичным образом, можно преобразовать существующее изображение, получив в два раза большее в обоих направлениях изображение, подобной (но не обратной) операцией:

```
void cvPyrUp(
    IplImage* src
    ,IplImage* dst
    ,IplFilter filter = IPL_GAUSSIAN_5x5
);
```

В данном случае, изображение сначала преобразуется в два раза большее в обоих направлениях изображение, с новыми строками, заполненными 0s. Затем выполняется свертка по заданному фильтру (на самом деле, фильтр в два раза больше для каждого измерения, чем указано) для аппроксимации значений "отсутствующих" пикселей.

Ранее было отмечено, что функция *PyrUp()* не является обратной *PyrDown()*. Это очевидно, т.к. *PyrDown()* приводит к потери данных. Для восстановления оригинального изображения (с более высоким разрешением), требуется та самая, теряемая при сжатии изображения, информация. Эти данные формируют пирамиду Лапласа. i -ый слой пирамиды Лапласа определяется следующим образом:

$$L_i = G_i - \text{UP}(G_{i+1}) \otimes \mathcal{G}_{5 \times 5}$$

Оператор *UP()* конвертирует каждый пиксель (x, y) исходного изображения в пиксель $(2x + 1, 2y + 1)$ в конечном изображении; символ \otimes обозначает свертку; $\mathcal{G}_{5 \times 5}$ Гауссово ядро размером 5×5 . Конечно, формула является определением оператора *PyrUp()*. Следовательно, оператор Лапласа можно определить следующим образом:

$$L_i = G_i - \text{PyrUp}(G_{i+1})$$

Пирамиды Гаусса и Лапласа схематично изображены на рисунке 5-21, на котором также показан обратный процесс для восстановления исходного изображения по субизображению. На самом деле приближение Лапласиана использует разницу Гауссианов, что собственно отражает предыдущее уравнение и что схематично показано на рисунке.

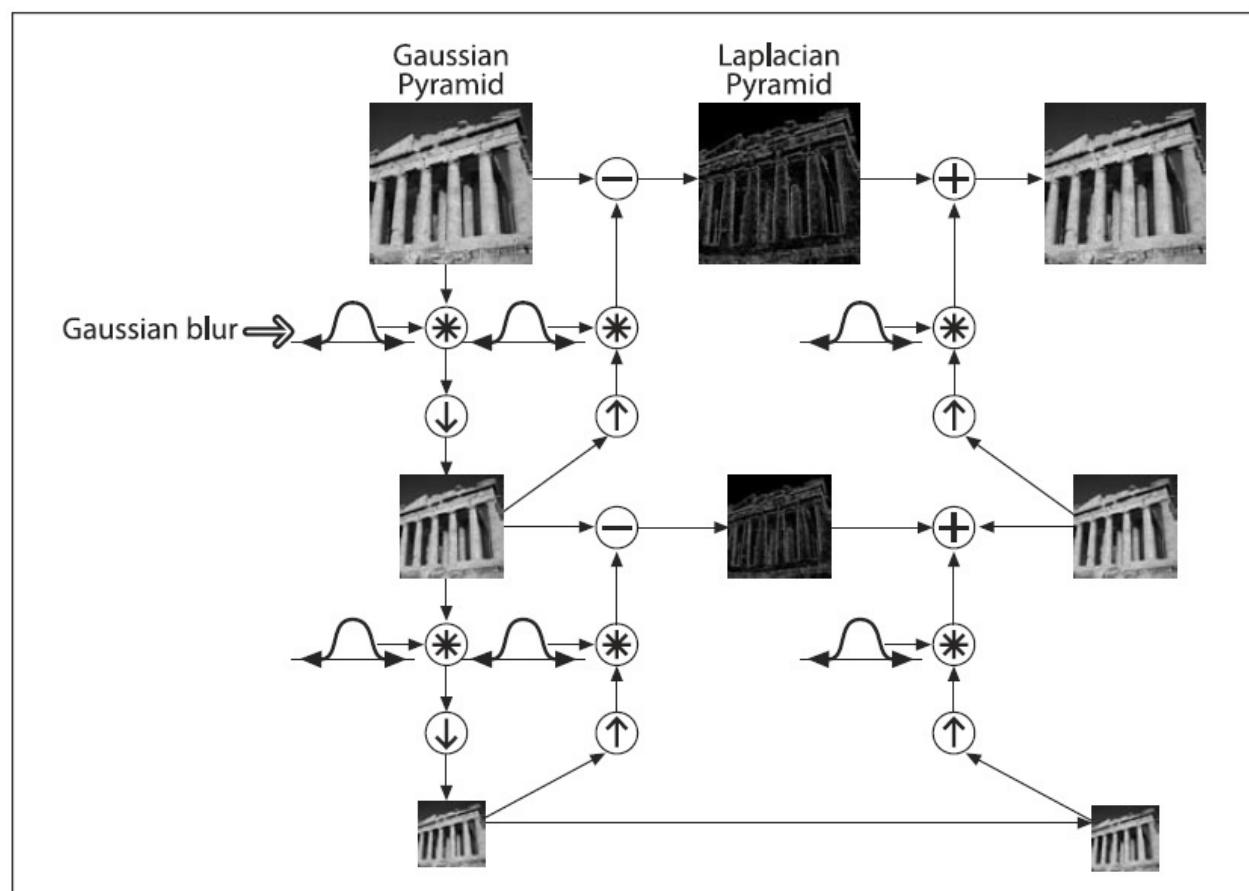


Рисунок 5-21. Пирамида Гаусса и ее противоположность - пирамида Лапласа

Существует множество операций, в которых могут быть задействованы пирамиды Гаусса и Лапласа, но наиболее важной является операция сегментации изображения (рисунок 5-22). В этом случае сначала строиться пирамида изображения, а затем выстраивается связь родитель-потомок между пикселями уровня G_{l+1} и соответствующими пикселями уровня G_l . Таким образом, быстрая начальная сегментация может быть выполнена на изображениях с низким разрешением, с последующим уточнением и дальнейшим дифференцированием уровень за уровнем.

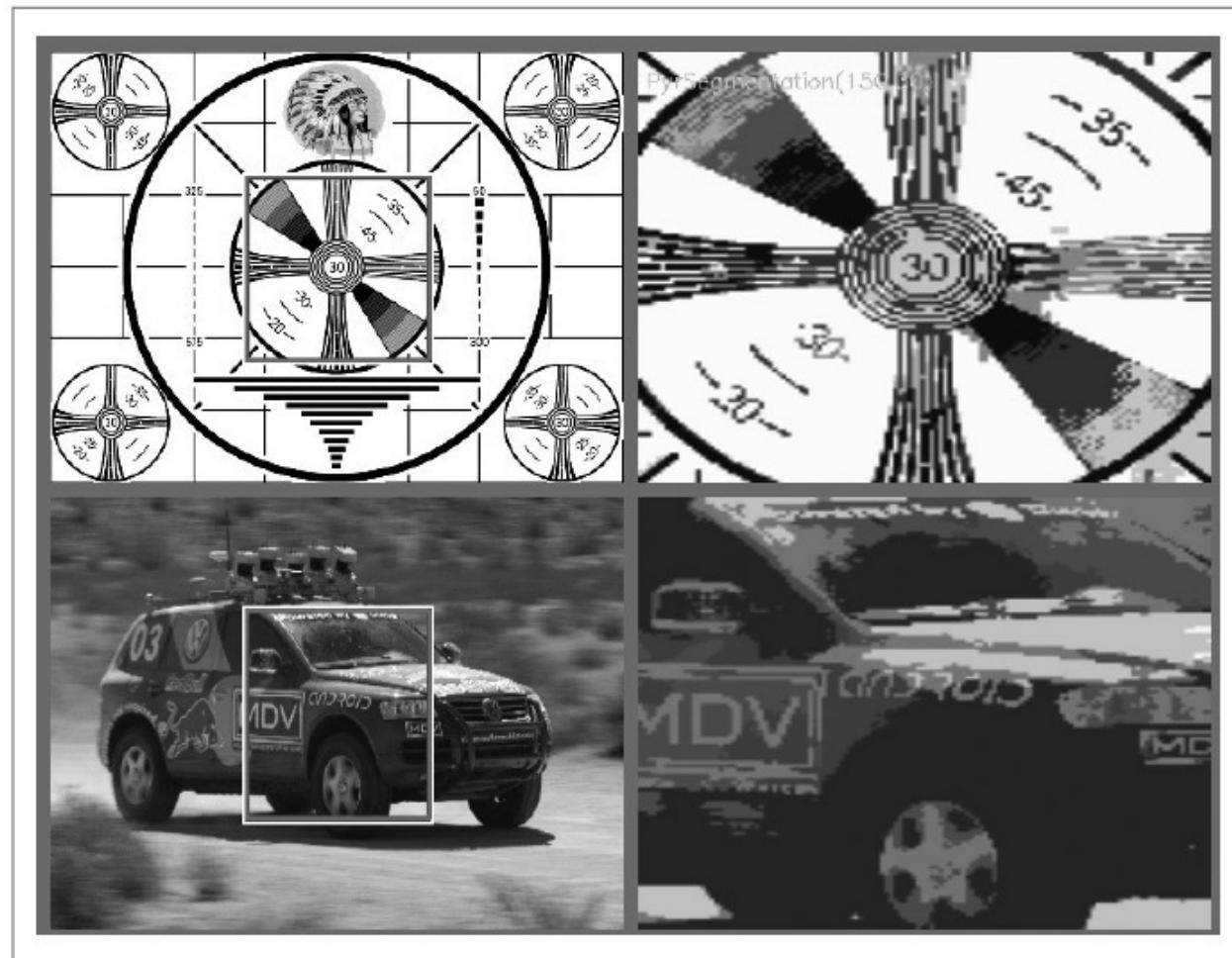


Рисунок 5-22. Сегментация с $threshold1 = 150$ и $threshold2 = 30$; изображение справа содержит только часть левого изображения, т.к. сегментация требует изображение, которое N раз делится на 2, где N - вычисляемое число слоев пирамиды (в данном случае 512x512 от оригинального изображения)

В OpenCV этот алгоритм представлен функцией `cvPyrSegmentation()`:

```

void cvPyrSegmentation(
    IplImage*      src
    ,IplImage*      dst
    ,CvMemStorage* storage
    ,CvSeq**        comp
    ,int            level
    ,double         threshold1
    ,double         threshold2
);

```

Как правило *src* и *dst* это исходное и конечное изображения, которые должны быть 8-битными, одинакового размера и с одинаковым количеством каналов (1 или 3). Может возникнуть вопрос: "Что еще за конечное изображение?". На самом деле, конечное изображение *dst* используется как рабочее пространство для алгоритма, а также возвращающее визуализацию сегментации. Если взглянуть на это изображение, то можно увидеть, что каждый сегмент одноцветный. Т.к. это изображение рабочее пространство алгоритма, то его нельзя установить в NULL. Даже если результат не нужен, всё равно необходимо предоставить функции это изображение. Ещё одно важное замечание по поводу *src* и *dst*: все уровни пирамиды должны иметь целочисленные размеры в обоих направлениях, исходное изображение должно делиться на 2 столько раз, сколько есть уровней в пирамиде. Например, для четырехуровневой пирамиды изображения с шириной и высотой равными 80 (2x2x2x5) пикселей было бы достаточно, а равными 90 (2x3x3x5) пикселей нет.

Аргумент *storage* это указатель на область хранения памяти. В главе 8 будет более детально рассказано о данной области хранения, а на данный момент будет вполне достаточно знать, что выделение данной области осуществляется следующим образом:

```
CvMemStorage* storage = cvCreateMemStorage();
```

Аргумент *comp* это место для хранения дополнительной информации о результатах сегментации: последовательность связных компонент, выделенные из области хранения. О том, как это работает будет рассказано в главе 8, но для удобства работы с *cvPyrSegmentation()* некоторые детали все таки будут рассмотрены сейчас.

Во-первых, последовательность представлена списком структур особого рода. Имея последовательность, можно получить количество элементов в ней, а также, зная тип и номер элемента в последовательности, получить сам элемент. В примере 5-1 представлен подход в получении доступа к последовательности.

Пример 5-1. Сделать что-то с каждым элементом из последовательности связных компонент, возвращаемая функцией *cvPyrSegmentation()*

```

void f( IplImage* src, IplImage* dst ) {
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* comp = NULL;

    cvPyrSegmentation( src, dst, storage, &comp, 4, 200, 50 );

    int n_comp = comp->total;

    for( int i = 0; i < n_comp; i++ ) {
        CvConnectedComp* cc = (CvConnectedComp*) cvGetSeqElem( comp, i );
        do_something_with( cc );
    }

    cvReleaseMemStorage( &storage );
}

```

Есть несколько вещей в этом примере, на которые стоит обратить внимание. Во-первых, на необходимость выделения памяти с помощью `cvCreateMemStorage()` под хранение связных компонент. Затем на создание указателя `comp` типа `CvSeq` и инициализацию его `NULL`, т.к. начальное значение в данном случае ничего не значит. Затем, на выполнение сегментации при помощи функции `cvPyrSegmentation()` и заполнение ранее созданной последовательности `comp`. После всего этого появляется возможность получать количество элементов в последовательности через поле `total`, а благодаря функции `cvGetSeqElem()` можно ещё получать и доступ к конкретному элементу из последовательности. Однако, т.к. функция является обобщенной и возвращает только указатель `void*`, необходимо выполнить приведение типов (в данном случае к `CvConnectedComp*`).

И в заключение, необходимо помнить о том, что связная компонента является одним из основных типов в OpenCV. Этот тип можно рассматривать как описание "капли" на изображении. Данный тип имеет следующее определение:

```

typedef struct CvConnectedComponent {
    double      area;
    CvScalar    value;
    CvRect      rect;
    CvSeq*     contour;
};

```

Аргумент `area` это область компоненты. Аргумент `value` это среднее значение цвета над областью компоненты. Аргумент `rect` это ограничительная рамка для компоненты (определенная в координатах родительского изображения). Аргумент `contour` это

указатель на другую последовательность, которая может быть использована для хранения представления границ компоненты и которая, как правило, представлена в виде последовательности точек типа *CvPoint*.

В некоторых случаях параметр *contour* *cvPyrSegmentation()* не задаётся. В результате, если необходимо получить конкретные представления пикселей компоненты, то необходимо будет вычислить их самим. Используемый для этого способ зависит, конечно, от требуемого представления. Зачастую используется булевая маска с ненулевыми элементами везде, где расположен компонент. Этого можно с легкостью добиться за счёт выделения компоненты прямоугольником (маска), а затем при помощи *cvFloodFill()* выбрать нужные пиксели внутри данного прямоугольника.

[П]||[РС]||(РП) Пороговое преобразование

Зачастую после обработки изображения, возникает потребность, чтобы в конечном изображении остались только те пиксели, которые выше или ниже определенного значения. В OpenCV для решения данной проблемы существует функция `cvThreshold()`. Основная идея алгоритма, заложенного в функцию, заключается в том, чтобы в конечный массив попали только те пиксели, которые выше или ниже определенного значения.

```
double cvThreshold(
    CvArr* src
    ,CvArr* dst
    ,double threshold
    ,double max_value
    ,int threshold_type
);
```

Как показано в таблице 5-5 каждому типу порога соответствует своя операция сравнения между i -ым пикселям исходного изображения (src_i) и порогом (обозначен как T). В зависимости от соотношения между значением пикселя на исходном изображении и значением порога, значение пикселя на конечном изображении dst_i может быть установлено в 0, src_i или max_value (обозначено как M).

Таблица 5-5. Значения `threshold_type` функции `cvThreshold()`

Тип порога	Операция
CV_THRESH_BINARY	$\text{dst}_i = (\text{src}_i > T) ? M : 0$
CV_THRESH_BINARY_INV	$\text{dst}_i = (\text{src}_i > T) ? 0 : M$
CV_THRESH_TRUNC	$\text{dst}_i = (\text{src}_i > T) ? M : \text{src}_i$
CV_THRESH_TOZERO_INV	$\text{dst}_i = (\text{src}_i > T) ? 0 : \text{src}_i$
CV_THRESH_TOZERO	$\text{dst}_i = (\text{src}_i > T) ? \text{src}_i : 0$

Рисунок 5-23 должен прояснить последствия каждого из типов преобразования.

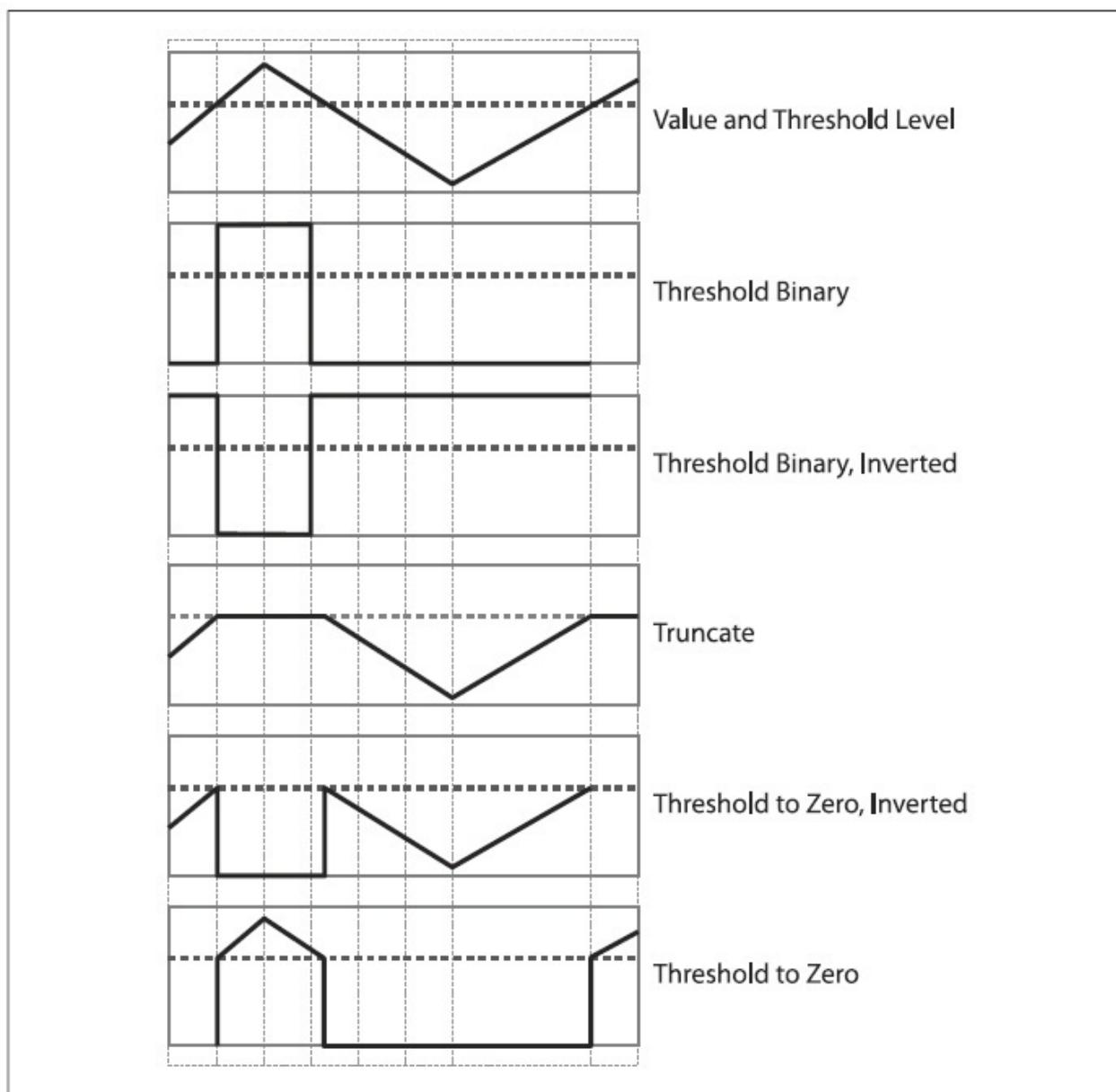


Рисунок 5-23. Результаты применения типов порогового преобразования.

Горизонтальная линия показывает значение порога, на первой диаграмме представлено исходное изображение, а на 5 последующих влияние применения типов порогового преобразования.

Рассмотрим небольшой пример (пример 5-2). В данном примере представлен процесс суммирования трех каналов исходного изображения с последующим удалением всех значений выше 100.

Пример 5-2. Пример использования функции *cvThreshold()*

```
#include <stdio.h>
#include <cv.h>
#include <highgui.h>

void sum_rgb( IplImage* src, IplImage* dst ) {
```

```
// Создание изображений для каждого канала
//
IplImage* r = cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, 1 );
IplImage* g = cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, 1 );
IplImage* b = cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, 1 );

// "Разделение" исходного изображения на составляющие
//
cvSplit( src, r, g, b, NULL );

// Временное изображение
//
IplImage* s = cvCreateImage( cvGetSize(src), IPL_DEPTH_8U, 1 );

// Средневзвешенная сумма
//
cvAddWeighted( r, 1./3., g, 1./3., 0.0, s );
cvAddWeighted( s, 2./3., b, 1./3., 0.0, s );

// Усечение значений до 100
//
cvThreshold( s, dst, 100, 100, CV_THRESH_TRUNC );

cvReleaseImage( &r );
cvReleaseImage( &g );
cvReleaseImage( &b );
cvReleaseImage( &s );
}

int main( int argc, char** argv ) {
// Создание окна
//
cvNamedWindow( argv[1], 1 );

// Создание исходного и конечного изображения
//
IplImage* src = cvLoadImage( argv[1] );
IplImage* dst = cvCreateImage( cvGetSize(src), src->depth, 1 );

// Обработка исходного изображения
//
sum_rgb( src, dst);

// Вывод конечного изображения на экран
//
cvShowImage( argv[1], dst );

// Ожидание нажатия клавиши "Esc"
//
while( 1 ) {
    if( (cvWaitKey( 10 )&0x7f) == 27 ) {
        break;
    }
}
```

```

}

// Освобождение ресурсов
//
cvDestroyWindow( argv[1] );
cvReleaseImage( &src );
cvReleaseImage( &dst );
}

```

В данном примере показано несколько довольно таки важных идей. Во-первых, к 8-битному массиву ничего не добавляется, иначе произойдет переполнение. Вместо этого используется средневзвешенное суммирование трех каналов с последующим усечением результата по насыщенности до 100. Во-вторых, функция *cvThreshold()* работает только с черно-белыми, 8-битными или вещественными изображениями. В-третьих, конечное изображение должно соответствовать исходному изображению или быть 8-битным. Кроме того, функция позволяет использовать входное изображение и как исходное и как конечное изображение. В примере в качестве промежуточного изображения использовано изображение вещественного типа для того, чтобы можно было применить код с альтернативным методом обработки (пример 5-3). Стоит обратить внимание на функцию *cvAcc()*, которая может складывать изображения целого и вещественного типа, чего в свою очередь не может делать функция *cvADD()*.

Пример 5-3. Альтернативный метод

```

IplImage* s = cvCreateImage(cvGetSize(src), IPL_DEPTH_32F, 1);
cvZero(s);
cvAcc(b,s);
cvAcc(g,s);
cvAcc(r,s);
cvThreshold( s, s, 100, 100, CV_THRESH_TRUNC );
cvConvertScale( s, dst, 1, 0 );

```

Адаптивное пороговое преобразование

Существует еще модифицированный метод порогового преобразования, у которого пороговый уровень - это переменная величина.

```

void cvAdaptiveThreshold(
    CvArr* src
    ,CvArr* dst
    ,double max_val
    ,int adaptive_method = CV_ADAPTIVE_THRESH_MEAN_C
    ,int threshold_type = CV_THRESH_BINARY
    ,int block_size = 3
    ,double param1 = 5
);

```

Функция реализует два варианта адаптивного порогового преобразования за счет параметра *adaptive_method*. В обоих случаях адаптивный порог $T(x, y)$ устанавливает попиксельно средневзвешенное значение, вычисляемое в регионе $b \times b$ вокруг каждого пикселя минус константа, где b это аргумент *block_size*, а константа это *param1*. Если установлен метод *CV_ADAPTIVE_THRESH_MEAN_C*, тогда все пиксели области взвешиваются одинаково. Если установлен метод *CV_ADAPTIVE_THRESH_GAUSSIAN_C*, тогда пиксели в регионе вокруг (x, y) взвешиваются в соответствии с Гауссовой функцией их расстояния от центральной точки.

Аргумент *threshold_type* может принимать любое значение из таблицы 5-5.

Адаптивный порог полезен в тех случаях, когда изображение засвеченено или содержит отражённые градиенты, а также, если нужно, чтобы порог отработал по отношению к общей интенсивности, а не с каждым пикселием в отдельности. Функция работает только с одноканальными, 8-битными и вещественными изображениями и требует, чтобы исходное и конечное изображения были различными.

Исходный код сравнения *cvAdaptiveThreshold()* и *cvThreshold()* приведён в примере 5-4. На рисунке 5-24 показан результат работы функции с изображением, которое имеет сильную градиентную засветку. В левой нижней части рисунка показан результат использования *cvThreshold()*, а в нижней правой части показан результат адаптивного порогового преобразования с использованием функции *cvAdaptiveThreshold()*. Как видно из рисунка, с помощью адаптивного порогового преобразования получилось "увидеть" всё шахматное поле, чего, в свою очередь, не было достигнуто при использовании обычного порогового преобразования. Примечание: в примере 5-4 для адаптивного преобразования были использованы следующие параметры:

```
./adaptThresh 15 1 1 71 15 ../Data/cal3-L.bmp
```

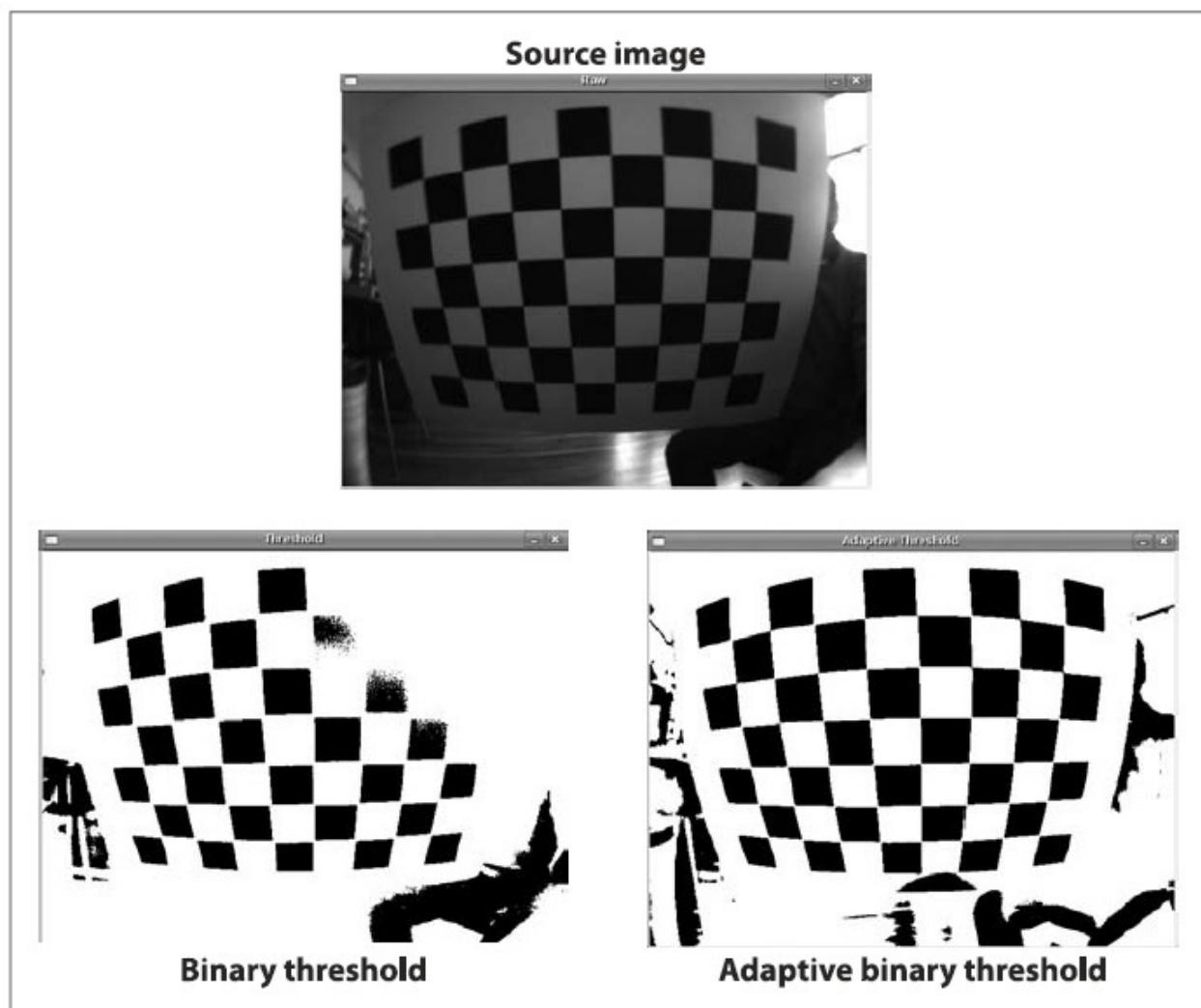


Рисунок 5-24. Обычное пороговое преобразование против адаптивного порогового преобразования: исходное изображение (сверху) было преобразовано в двоичное изображение с помощью глобального порога (внизу слева) и адаптивного порога (внизу справа); изображение предоставлено Kurt Konolidge

Пример 5-4. Обычное пороговое преобразование против адаптивного порогового преобразования

```
#include <cv.h>
#include <highgui.h>
#include <math.h>

IplImage *Igray=0, *It = 0, *Iat;

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

    if( 7 != argc ) {
        return -1;
    }

    // Обработка ключей командной строки
```

```
//  
double threshold = (double)atof(argv[1]);  
int threshold_type = atoi(argv[2]) ? CV_THRESH_BINARY  
                                : CV_THRESH_BINARY_INV;  
int adaptive_method = atoi(argv[3]) ? CV_ADAPTIVE_THRESH_MEAN_C  
                                : CV_ADAPTIVE_THRESH_GAUSSIAN_C;  
int block_size = atoi(argv[4]);  
double offset = (double)atof(argv[5]);  
  
// Загрузка изображения в чёрно-белом формате  
//  
if((Igray = cvLoadImage( argv[6], CV_LOAD_IMAGE_GRAYSCALE)) == 0) {  
    return -1;  
}  
  
// Создание выходных чёрно-белых изображений  
//  
It = cvCreateImage(cvSize(Igray->width,Igray->height), IPL_DEPTH_8U, 1);  
Iat = cvCreateImage(cvSize(Igray->width,Igray->height), IPL_DEPTH_8U, 1);  
  
// Пороговые преобразования  
//  
cvThreshold( Igray, It,threshold, 255, threshold_type );  
cvAdaptiveThreshold( Igray, Iat, 255, adaptive_method, threshold_type, block_size, of  
  
// Создание окон для вывода результатов  
//  
cvNamedWindow("Raw", 1);  
cvNamedWindow("Threshold", 1);  
cvNamedWindow("Adaptive Threshold", 1);  
  
// Вывод результатов  
//  
cvShowImage("Raw", Igray);  
cvShowImage("Threshold", It);  
cvShowImage("Adaptive Threshold", Iat);  
cvWaitKey(0);  
  
// Освобождение занимаемых ресурсов  
//  
cvReleaseImage( &Igray );  
cvReleaseImage( &It );  
cvReleaseImage( &Iat );  
cvDestroyWindow( "Raw" );  
cvDestroyWindow( "Threshold" );  
cvDestroyWindow( "Adaptive Threshold" );  
  
return(0);  
}
```

[П]|[РС]|(РП) Упражнения

1. Загрузите любое изображение и размойте его с помощью `cvSmooth()` с `smoothtype = CV_GAUSSIAN`.
 - a. Используйте симметричную область сглаживания 3x3, 5x5, 9x9, 11x11 и выведите результаты на экран.
 - b. Однаковы ли результаты двойного сглаживания изображения с областью 5x5 и с областью 11x11. Почему да или почему нет?
2. Создайте одноканальное изображение 100x100, очистите его и установите центральный пиксель равным 255.
 - a. Размойте данное изображение с помощью Гауссова ядра размером 5x5 и отобразите результат.
 - b. Сделайте тоже самое с фильтром 9x9.
 - c. Что будет, если исходное изображение сгладить дважды фильтром 5x5? Сравните этот результат с результатом сглаживания с использованием фильтра 9x9. Являются ли они почти одинаковыми? Почему да или почему нет?
3. Загрузите любое изображение и размойте его с помощью `cvSmooth()`.
 - a. Установите `param1 = param2 = 9`, и попробуйте изменять значение `param3` (например, 1, 4, 6). Отобразите результат в трех разных окнах.
 - b. Теперь установите `param1 = param2 = 0`, и попробуйте изменять значение `param3` (1, 4, 6). Отобразите результат в трех разных окнах. Отличаются ли они? Почему?
 - c. Опять установите `param1 = param2 = 0`, только теперь `param3 = 1`, а `param4 = 9`. Отобразите результат.
 - d. Опять установите `param1 = param2 = 0`, только теперь `param3 = 9`, а `param4 = 1`. Отобразите результат.
 - e. Теперь последовательно примените операцию сглаживания сначала с настройками из пункта c, а затем с настройками из пункта d. Отобразите результат.
 - f. Сравните результаты операции сглаживания из пункта e, с результатом, при условии, что `param3 = param4 = 9` и `param3 = param4 = 0`. Результаты те же? Почему да или почему нет?

4. Используя камеру, сделайте две фотографии одной и той же сцены, стараясь не двигать камеру. Загрузите эти изображения как `src1` и `src2`.
 - a. Вычислите абсолютное значение $src1 - src2$ (вычитание изображений), сохраните результат в `diff12` и отобразите результат. Если все было сделано верно, то изображение `diff12` будет состоять из пикселей только черного цвета. Почему?
 - b. Создайте `cleandiff` с помощью `cvErode()`, а затем примените `cvDilate()` к `diff12`, отобразите результат.
 - c. Создайте `dirtydiff` с помощью `cvDilate()`, а затем примените `cvErode()` к `diff12`, отобразите результат.
 - d. Объясните разницу между `cleandiff` и `dirtydiff`.
5. Сделайте изображение любой сцены. Затем, не перемещая камеру, поставьте какой-нибудь предмет и сделайте второй снимок. Загрузите эти изображения и конвертируйте в 8-битные в градациях серого изображения.
 - a. Вычислите абсолютное значение их разности (`cvAbsDiff()`). Выведите результат, который должен содержать зашумленную маску предмета.
 - b. Выполните двоичное пороговое преобразование в результате которого предмет остается, но некоторая степень шума удаляется. Отобразите результат.
 - c. Выполните `CV_MOP_OPEN` над изображением, чтобы ещё больше устраниТЬ шум.
6. Создание маски, очищенной от шума. После выполнения упражнения 5, найдите самый большой регион на изображении. Установите указатель в верхнем левом углу изображения и пройдитесь по всему изображению. Когда будут встречаться значения 255, используйте функцию `cvFloodFill()` со значением 100 в этой точке. Извлеките связную компоненту, которую вернет функция и сохраните её. Найдите самую большую область и залейте её значением 255. Если последующая область больше предыдущей, то предыдущую залейте 0, а текущую значением 255. И так до тех пор, пока не останется самая большая область. Отобразите результат. В результате должна получиться довольно таки точная маска объекта.
7. Для выполнения этого упражнения используйте маску из предыдущего упражнения или создайте другую маску (например, простую маску в виде квадрата). Создайте новое изображение сцены и поместите в него, с помощью функции `cvCopy()`, маску объекта.

8. Создайте низко дисперсное случайное изображение (используйте случайно генерируемое число из диапазона от 0 до 3). Загрузите полученное изображение в любой графический редактор и нарисуйте на нем множество линий, сходящихся в одной точке. В завершение примените двухстороннюю фильтрацию на полученном изображении и объясните полученный результат.
9. Загрузите изображение сцены в градациях серого.
 - a. Выполните морфологическую операцию Top Hat на изображении сцены и отобразите результат.
 - b. Конвертируйте полученное изображение в 8-битное
 - c. Скопируйте значения в оттенках серого в части Top Hat и отобразите результат.
10. Загрузите изображение, содержащее множество объектов.
 - a. Используя функцию `cvResize()` уменьшите изображение в 2 раза по каждому измерению (следовательно изображение будет уменьшено в 4 раза). Проделайте это еще три раза и отобразите результат.
 - b. Теперь возьмите оригинальное изображение и, используя функцию `cvPyrDown()`, уменьшите его трижды и отобразите результат.
 - c. В чем различия между результатами из пункта a и b?
11. Загрузите изображение сцены. Примените к изображению функцию `cvPyrSegmentation()` и отобразите результат.
12. Загрузите изображение с достаточно "богатой" сценой. Примените к изображению функцию `cvThreshold()` с порогом 128. Воспользуйтесь всеми параметрами из таблицы 5-5 и отобразите результаты. Осознание принципов работы порогового преобразования принесет еще не мало пользы в дальнейшем.
 - a. Повторите упражнение, но уже с использованием функции `cvAdaptiveThreshold()`. Установите `param1 = 5`.
 - b. Повторите предыдущий пункт, используя сначала параметр `param1 = 0`, а затем к результату вновь примените функцию `cvAdaptiveThreshold()`, но уже с параметром `param1 = -5`.

Преобразования изображений

[П]|[РС]|(РП) Краткий обзор

В предыдущей главе уже было рассмотрено довольно-таки большое количество различных действий, которые можно производить над изображениями. Большинство из представленных операций используются для улучшения и модификации изображения.

В этой главе будут рассмотрены *трансформации изображений*, которые являются методами преобразования изображения в изображение с полностью другим представлением данных. Возможно наиболее общим представлением такого преобразования является преобразование Фурье, в котором изображение преобразовывается в альтернативное представление данных исходного изображения. Хотя результат данной операции и сохраняется в структуре "изображение", однако, "пиксель" в новом изображении представляет спектральную составляющую от оригинального изображения, а не пространственную составляющую, в терминах которой до сих пор велась речь.

Существует большой набор полезных преобразований, которые периодически применяются в компьютерном зрении. OpenCV предоставляет полную реализацию некоторых обобщенных функций в качестве "строительных блоков", на основе которых в последующем можно создавать собственные преобразования над изображением.

[П]||[РС]||(РП) Свертка

Свертка - это основа большинства преобразований, о которых будет идти речь в этой главе. В общем значении, этот термин означает некоторое действие, которые выполняется над каждой частью изображения. Так, большинство операций из главы 5 могут быть представлены как особый случай свертки. Что "делает" свертка определяется по форме **ядра свертки**. Это ядро, по существу, является массивом фиксированного размера, состоящий из числовых коэффициентов, с якорной точкой (в дальнейшем просто якорь), расположенной, как правило, в центре. Размер массива именуется *support* ядра (по техническим причинам, *support* ядра состоит только из ненулевых частей массива ядра).

На рисунок 6-1 представлено ядро свертки размером 3x3 с якорем в центре. Значение свертки в конкретной точке вычисляется в следующем порядке. В начале якорь помещается поверх пикселя конкретной точки, при этом принимая во внимание тот факт, что остальная часть ядра накладывается на соответствующие рядом стоящие пиксели изображения. В результате имеем два значения для пикселей под ядром: исходное значение пикселя и значение ядра. Далее эти значения перемножаются и суммируются; результат помещается в место размещения якоря. Этот процесс повторяется для всех пикселей изображения путем "скольжения" ядра по изображению.

1	-2	1
2	-4	2
1	-2	1

Рисунок 6-1. Ядро свертки 3x3 для производной Собеля. Якорь расположен в центре

Разумеется, все ранее описанные действия можно описать математическим уравнением. Пусть изображение это $I(x, y)$, ядро $G(i, j)$ (где $0 < i < M_I - 1$ и $0 < j < M_J - 1$) и ядро, размещенное в (a_I, a_J) , тогда свертка $H(x, y)$ будет определяться следующим выражением:

$$H(x, y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j) G(i, j)$$

На первый взгляд может показаться, что число операций равно количеству пикселей на изображении помноженное на количество пикселей в ядре (Так же возможно производить свертку в частотной области. В этом случае для изображения размером $N \times N$ и ядра $M \times M$, при том, что $N > M$, время вычисления будет пропорционально $N^2 \log(N)$, а не $N^2 M^2$ как ожидается при вычислениях в пространственной области).

Поскольку вычисления в частотной области не зависят от размера ядра, оно более эффективно для больших ядер. OpenCV автоматически решает, когда выполнять свертку в частотной области, учитывая размер ядра.). Это может привести к большому объему вычислений, так что такая операция, возможно, не лучший кандидат на реализацию в виде цикла *for* со множеством разыменований указателей. В этом случае лучше всего позволить OpenCV выполнить эту работу за вас путем вызова *cvFilter2D()* (вместе с тем получить преимущество от использования оптимизаций, заложенных в OpenCV):

```
void cvFilter2D(
    const CvArr*   src
    ,CvArr*        dst
    ,const CvMat*  kernel
    ,CvPoint        anchor = cvPoint(-1, -1)
);
```

На вход функции передается матрица ядра с заполненными коэффициентами, исходное и конечное изображение. Так же возможно указать необязательное положение якоря (по умолчанию *cvPoint(-1, -1)* - центр ядра). Ядро может быть четного размера, если задать положение якоря; иначе оно обязательно должно быть нечетного размера.

Аргументы изображений *src* и *dst* должны быть одинакового размера. Изначально можно подумать, что исходное изображение должно быть чуть больше, чем конечное изображение, чтобы обработать избыток по ширине и длине ядра свертки. Но *src* и *dst* могут быть одинакового размера, потому что по умолчанию, до того, как происходит свертка, OpenCV создает виртуальные пиксели, лежащие за границей изображения, путем репликации пограничных пикселей *src*, чтобы пограничные пиксели *dst* можно было заполнить. Репликация выполняется как $input(-dx, y) = input(0, y)$, $input(w + dx, y) = input(w - 1, y)$ и так далее. Есть несколько альтернативных способов в противовес данному поведению по умолчанию; эти способы будут рассмотрены в следующем разделе.

И в заключение, стоит отметить, что коэффициенты ядра свертки должны всегда быть вещественными числами. Это означает, что при создании матрицы ядра необходимо использовать флаг `CV_32FC1`.

Края свертки

Вполне естественно, что может возникнуть вопрос: как обрабатывать границы. Например, что произойдет с точкой на краю изображения, если применить ранее описанное ядро свертки? Многие встроенные функции OpenCV используют `cvFilter2D()`, при этом решают данную проблему по разному. Собственно, когда пишется самописная функция свертки, данная проблема так же должна быть обработана эффективно.

Решением является функция `cvCopyMakeBorder()`, которая копирует исходное изображение в другое, чуть большее изображение, а затем автоматически заполняет границу тем или иным образом:

```
void cvCopyMakeBorder(
    const CvArr* src
    ,CvArr* dst
    ,CvPoint offset
    ,int bordertype
    ,CvScalar value = cvScalarAll(0)
);
```

Аргумент `offset` указывает функции, где размещать копию исходного изображения в конечном изображении. Обычно, если ядро $N \times N$ (N нечетное число), то ширина краев равна $(N - 1)/2$, т.е. изображение, которое $N - 1$ шире и выше оригинала. В этом случае лучше всего установить `offset = cvPoint((N - 1)/2, (N - 1)/2)`, так толщина границы будет одинаковой со всех сторон (Разумеется, случай ядра $N \times N$ с нечетным N и ядром в центре - это простейший случай). Если ядро имеет размер $N \times M$ и ядро (a_x, a_y) , тогда конечное изображение должно быть на $N - 1$ пикселей шире и на $M - 1$ пикселей выше исходного изображения. Тогда смещение будет (a_x, a_y) .

Аргумент `bordertype` может быть `IPL_BORDER_CONSTANT` или `IPL_BORDER_REPLICATE` (Рисунок 6-2). В первом случае, параметр `value` будет трактоваться как значение, которым будут заполнены все пиксели края. Во втором случае, строки или столбцы с каждого края исходного изображения реплицируются, чтобы увеличить изображение. Обратите внимание, что граница тестового изображения едва различима (присмотритесь к верхнему правому изображению на рисунке 6-2); на узорном изображении есть черная граница толщиной в один пиксель, кроме места, где круг близко подходит к краю изображения, там контур становится

белым. Аргумент *bordertype* поддерживает еще два типа, *IPL_BORDER_REFLECT* и *IPL_BORDER_WRAP*, которые пока не реализованы в OpenCV (по крайней мере в последней версии на момент написания книги).



Рисунок 6-2. Расширение краев изображения. В левой колонке показано действие *IPL_BORDER_CONSTANT*, где *value* = 0. В правой колонке показано действие *IPL_BORDER_REPLICATE*, где краевые пиксели реплицируются в горизонтальном и вертикальном направлениях

Ранее уже было отмечено, что функции, которые используют свертку, скрыто вызывают функцию *cvCopyMakeBorder()*, чтобы выполнить свою работу. В большинстве случаев используется *IPL_BORDER_REPLICATE*, но иногда может понадобиться поменять это поведение. Помимо этого существуют другие случаи, при которых может понадобиться функция *cvCopyMakeBorder()*. Например, при создании изображения немного большего размера, с нужным краем, с последующей обработкой этого изображения и вырезанием той части изображения, которая необходима. Таким образом, автоматическое увеличение краев не испортит важные пиксели на крае исходного изображения.

[П] | [РС] | (РП) Градиенты и производная Собеля

Вычисление производных (или их аппроксимации) - это одна из наиболее важных базовых сверток. Есть множество способов сделать это, но далеко не все из них хорошо подходят для конкретной ситуации.

В общем, самым распространенным оператором, используемый для представления дифференцирования, является производная Собеля (рисунок 6-3 и 6-4). Существуют операторы Собеля для любого порядка производной, а также для смешанных частных производных ($\frac{\partial^2}{\partial x \partial y}$).

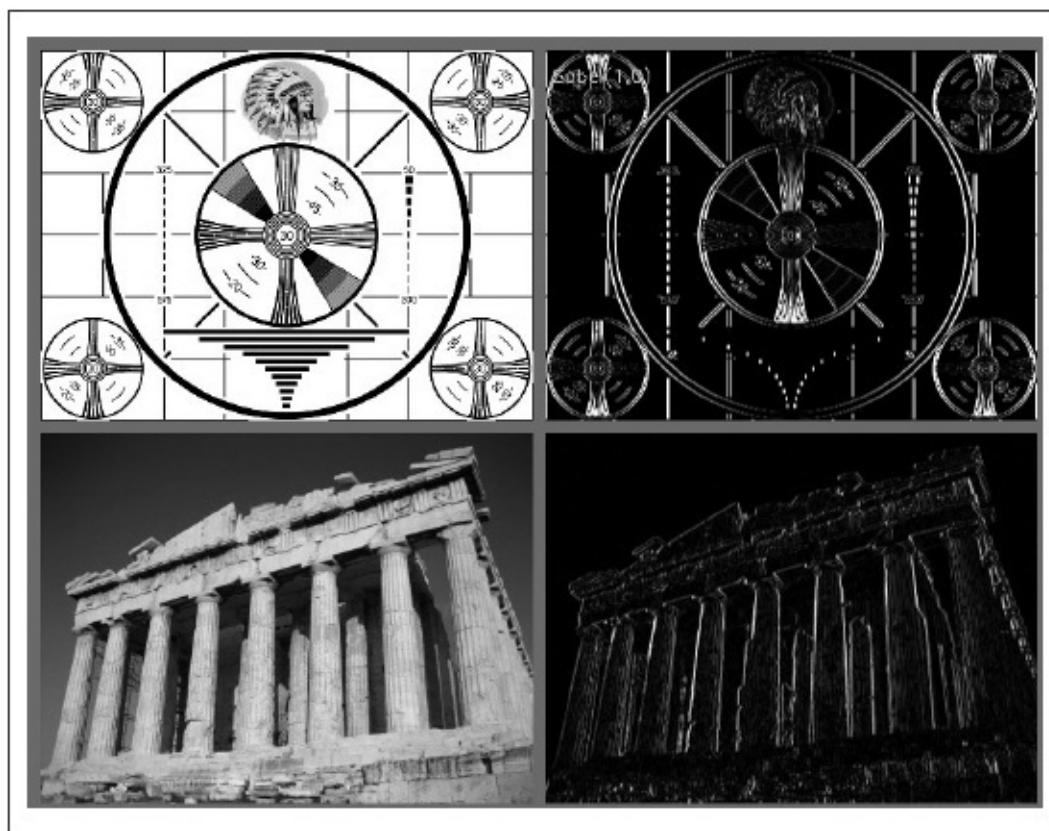


Рисунок 6-3. Результат применения оператора Собеля, используемый для аппроксимации производной по оси x

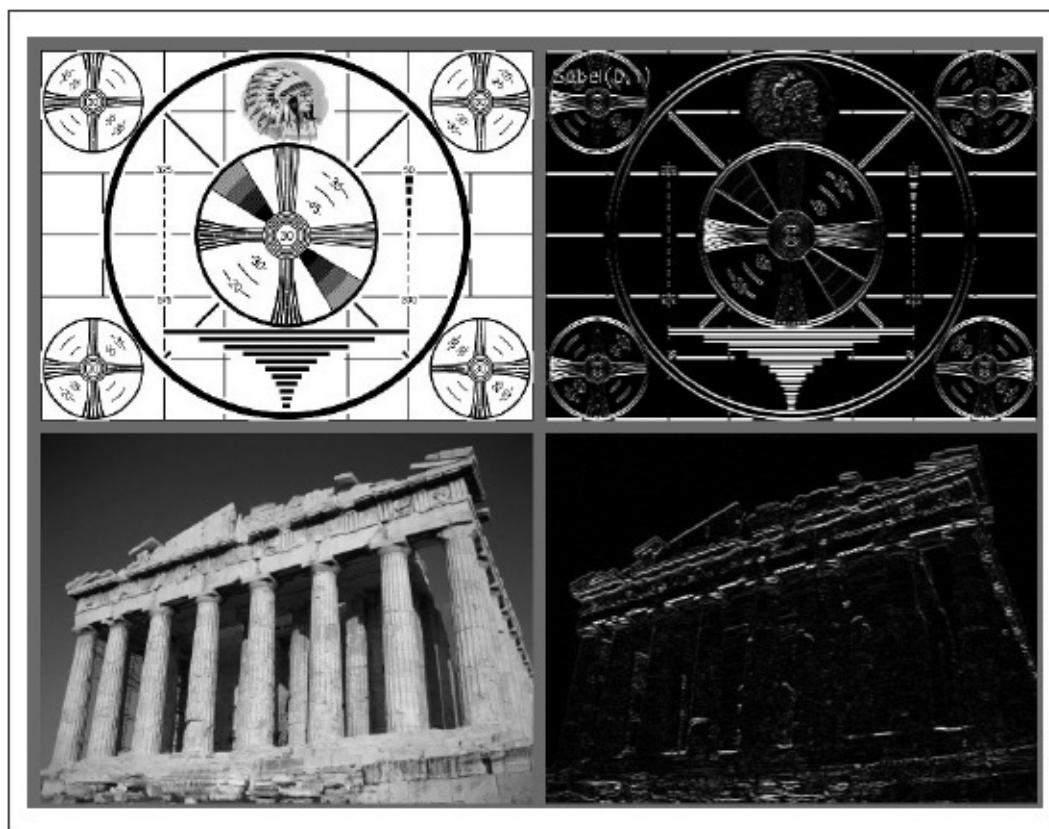


Рисунок 6-4. Результат применения оператора Собеля, используемый для аппроксимации производной по оси у

```
cvSobel(
    const CvArr*   src
    ,CvArr*        dst
    ,int            xorder
    ,int            yorder
    ,int            aperture_size = 3
);
```

Аргументы *src* и *dst* - исходное и конечное изображения, а аргументы *xorder* и *yorder* - порядки производной. Обычно используется 0, 1 или максимум 2 производная; значение 0 означает, что производная в указанном направлении браться не будет (при этом хотя бы одно из направлений должно быть не нулевым). Аргумент *aperture_size* должен быть нечетным числом и задавать ширину (и высоту) квадратного фильтра. На момент написания книги поддерживаемые размеры апертуры 1, 3, 5 и 7. Если исходное изображение 8-битное, тогда конечное изображение должно быть глубиной *IPL_DEPTH_16S*, чтобы избежать переполнения.

Производные Собеля обладают хорошим свойством в том смысле, что могут быть определены для ядер любого размера, которые в свою очередь могут быть быстро построены. Более крупные ядра дают лучшую аппроксимацию производной, потому что мелкие ядра очень чувствительны к шуму.

Что бы это осознать, нужно понимать, что производная Собеля в действительности производной не является. Это все потому, что оператор Собеля определен в дискретном пространстве. Что в действительности представляет собой оператор Собеля, так это подгонку под полином. Так что, производная Собеля второго порядка по оси x на самом деле не вторая производная, а локальная подгонка под параболическую функцию. Это объясняет необходимость использования ядра большего размера, т.к. большое ядро производит подгонку над большим количеством пикселей.

Фильтр Щарра

В действительности существует множество способов аппроксимировать производную для случая дискретной сетки. Недостатком аппроксимации с использованием оператора Собеля является то, что она менее точна для маленьких ядер. Для больших ядер, где используется больше точек для аппроксимации, эта проблема менее значительна. Эти неточности не видны для X и Y фильтров, используемые в `cvSobel()`, потому что они точно совпадают с осями x и y. Трудности появляются при проведении замеров по изображению с использованием аппроксимации производной по направлению (например, по направлению градиента изображения путем использования результата фильтра по арктангенсу y/x).

Чтобы прояснить смысл, рассмотрим реальный пример замера изображения такого вида - процесс получения информации о форме объекта путем сбора гистограммы градиентных углов вокруг объекта. Подобные гистограммы - это выходные данные и основа для множества обычных классификаторов форм (как для их обучения, так и для работы). В этом случае, неточные измерения угла наклона снижит производительность распознавания классификатора.

Для фильтра Собеля размером 3x3 неточность возрастает с увеличением градиентного угла по горизонтали или вертикали. OpenCV заботится о маленьких (но быстро растущих) неточностях фильтров Собеля размером 3x3 при помощи "закулисных" дополнений внутри функции `cvSobel()` за счет использования значения `CV_SCHARR` для аргумента `aperture_size`. Фильтр Щарра так же быстр, но более точен, чем фильтр Собеля, поэтому именно фильтр Щарра должен быть использован для измерений изображений с фильтром размера 3x3. Коэффициенты Щарра показаны на рисунке 6-5.

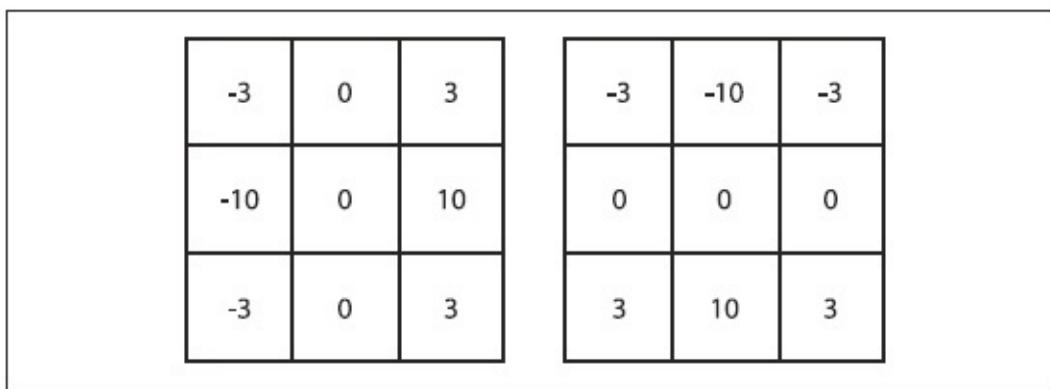


Рисунок 6-5. Фильтр Щарра размером 3x3 с использованием флага CV_SCHARR

[П]||[РС]||(РП) Лапласиан

Функция Лапласа (впервые используемая в компьютерном зрении Marr) реализует дискретный аналог оператора Лапласа (при этом не стоит путать с пирамидой Лапласа):

$$\text{Laplace}(f) \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Поскольку оператор Лапласа может быть определен в терминологии второй производной, то можно предположить, что дискретная реализация работает подобно производной Собеля второго порядка. В действительности так и есть, при реализации оператора Лапласа OpenCV использует оператор Собеля.

```
void cvLaplace(
    const CvArr*   src
    ,CvArr*        dst
    ,int           apertureSize = 3
);
```

Функция *cvLaplace()* принимает в качестве входных параметров исходное и конечное изображения, а также размер апертуры. Исходное изображение должно быть 8-битным (без знаковым) или 32-битным (вещественным). Конечное изображение должно быть 16-битным (знаковым) или 32-битным (вещественным). Апертура — это тот же самый аргумент, как и в производных Собеля и задает размер региона, по которому обрабатываются пиксели при вычислении вторых производных.

Оператор Лапласа может быть применен в различных контекстах. Наиболее общее применение - обнаружение "капель". Оператор Лапласа выражается в виде суммы вторых производных по осям x и y. Это означает, что единичная точка или любая маленькая капля (меньше апертуры), которые обнаружены большими значениями приведет к возрастанию значения выходной функции. И наоборот, точка или маленькая капля, которая окружена меньшими значениями будет стремиться максимизировать отрицательное значение данной функции.

Принимая это во внимание, оператор Лапласа может быть использован в качестве детектора для подсветки (выделения) краев. Что бы осознать, как это происходит, представьте первую производную функции, которая будет большой в местах, где функция быстро изменяется. В то же время, она будет скачкообразно возрастать в местах приближения к краеобразной неоднородности и скачкообразно уменьшаться при выходе из неоднородности. Получается производная будет иметь локальный

максимум где-то в этом диапазоне. Таким образом для поиска таких локальных максимумов необходимо найти нули второй производной. Границы в исходном изображении будут нулями для Лапласиана. При этом нулями Лапласиана будут как более существенные, так и менее значимые края, но это не проблема, т.к. можно просто отфильтровать те пиксели, которые имеют наибольшие значения для первой производной Собеля. На рисунке 6-6 приведен пример применения Лапласиана над изображением вместе с деталями о первой и второй производной и их пересечениях нулевых значений.

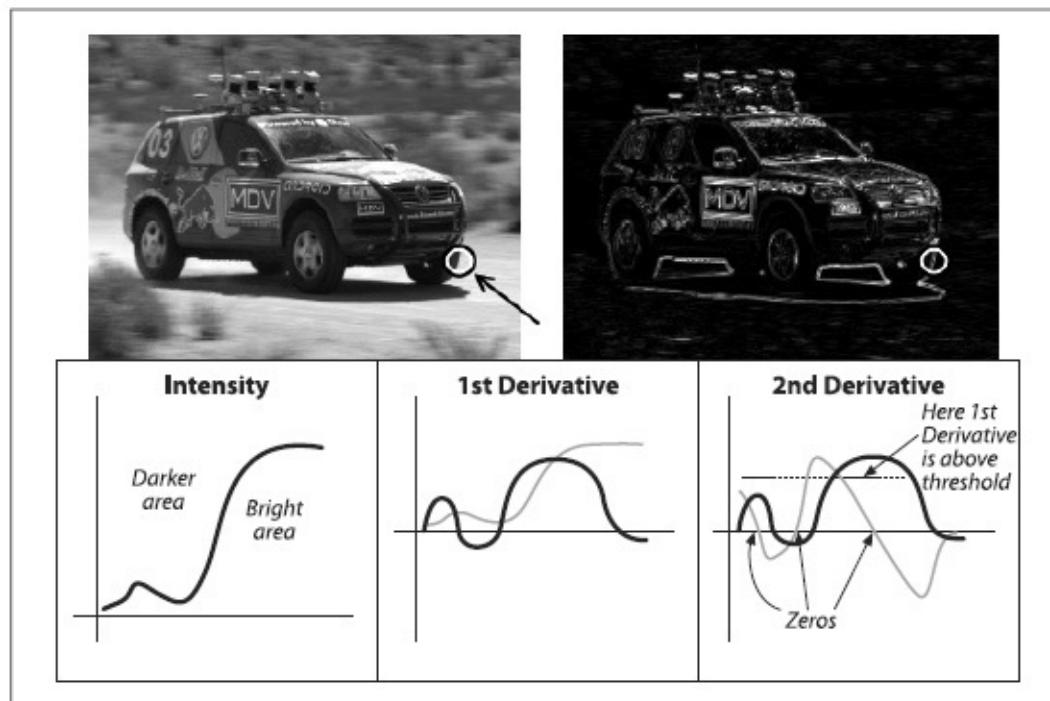


Рисунок 6-6. Применение преобразования Лапласа (сверху справа) на изображение гоночной машины: масштабирование в области шины (выделено белым кругом) и, рассматривая только ось x, отображение (качественное) представления яркости, а также значений первой и второй производной (три нижних рисунка); нули второй производной соответствуют краям на изображении, а нули первой производной соответствуют четким краям

[П]|[РС]|(РП) Canny

Только что описанный метод поиска краев в 1986 году был улучшен J. Canny и именуется как детектор границ Canny. Отличие алгоритма Canny от более простого алгоритма, основанного на преобразованиях Лапласа, в том, что в алгоритме Canny первые производные вычисляются по осям x и y, а затем объединяются в четырех направлениях производных. Точки, в которых эти производные достигают локального максимума затем рассматриваются в качестве кандидатов на группировку в край.

Однако, наиболее значимым дополнением алгоритма Canny является то, что он пытается собрать кандидатов в целые контуры (Контуры будут рассмотрены чуть позже, а пока необходимо понимать, что `cvCanny()` в действительности не возвращает объектов типа `CvContour`; при необходимости этот объект можно получить применив `cvFindContours()`. Более подробно контуры будут рассмотрены в главе 8). Эти контуры формируются путем применения порога гистерезиса к пикселям. Это означает наличие двух порогов - верхнего и нижнего. Если значение градиента пикселя больше, чем верхний порог, то он включается в край; если значение градиента пикселя меньше нижнего порога, то данный пиксель исключается из рассмотрения. Если значение градиента пикселя между этими порогами, то он будет частью края только в том случае, если он соединен с пикселием, который больше верхнего порога. Согласно рекомендациям автора, соотношение порогов `высокий:низкий` должно быть между 2:1 и 3:1. На рисунке 6-7 и рисунке 6-8 показан результат применения `cvCanny()` к тестовому шаблону и фотографии с пороговыми соотношениями 5:1 и 3:2.

```
void cvCanny(  
    const CvArr* img  
    ,CvArr* edges  
    ,double lowThresh  
    ,double highThresh  
    ,int apertureSize = 3  
) ;
```

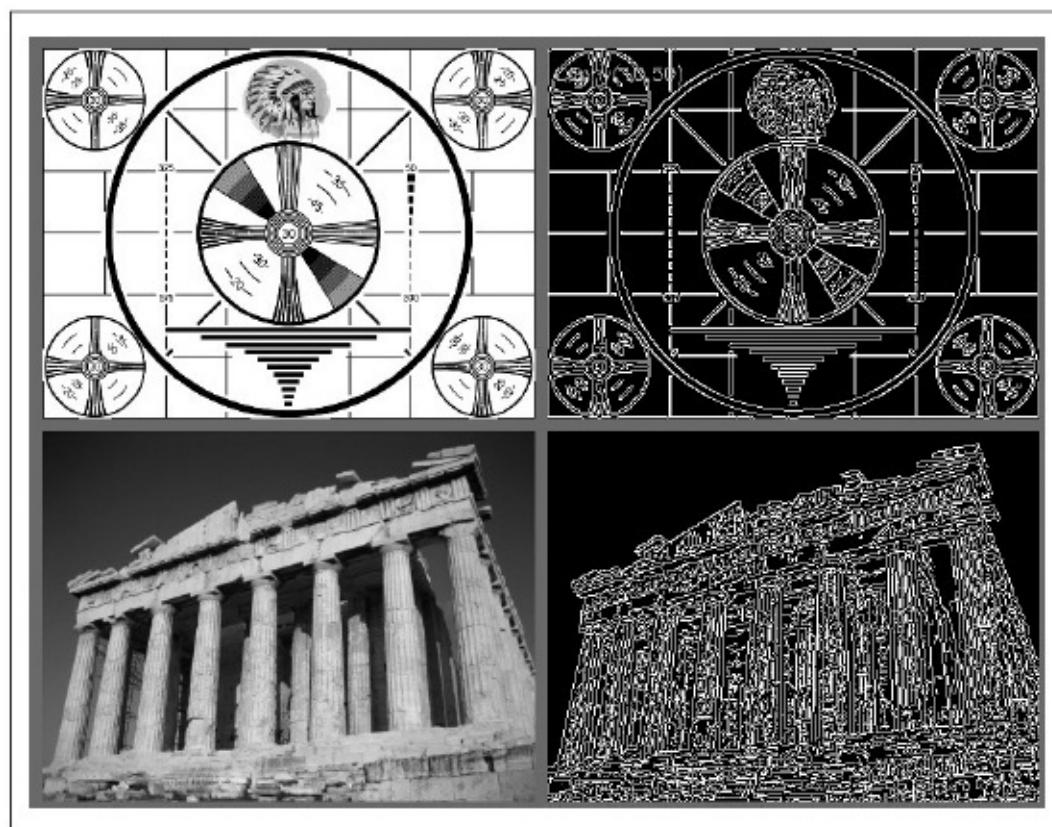


Рисунок 6-7. Результат применения cvCanny для двух совершенно разных изображений, для которых верхний и нижний пороги установлены равными 50 и 10 соответственно

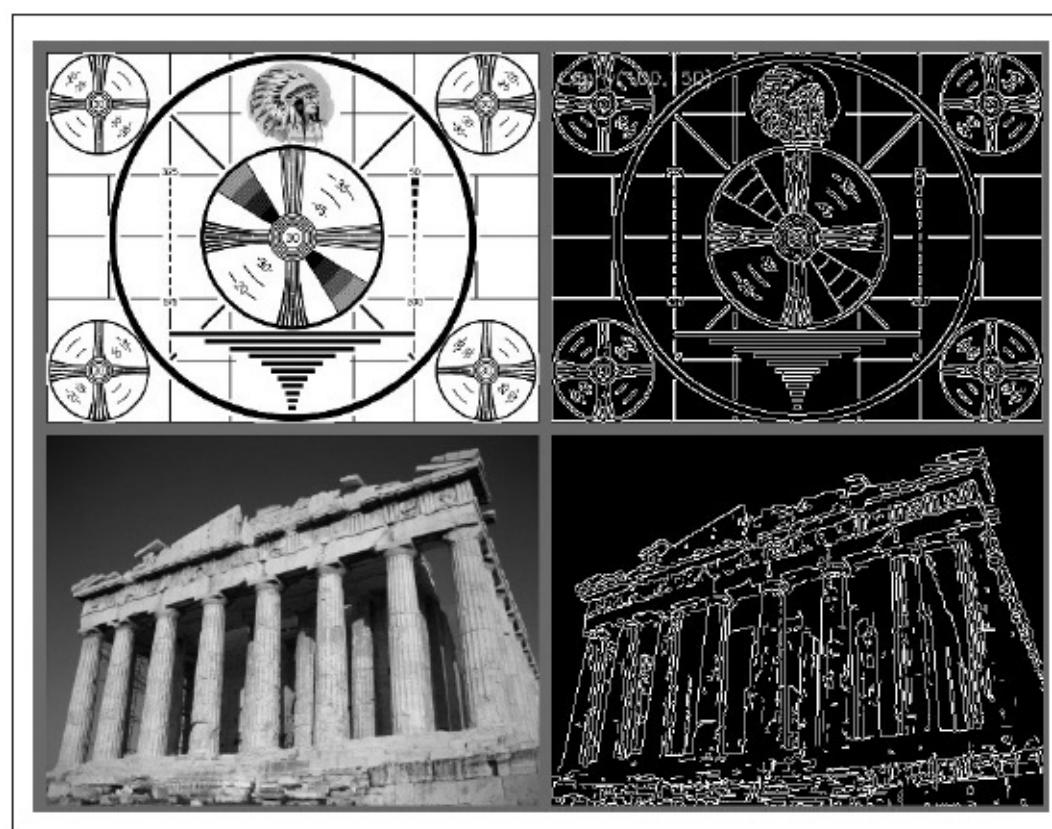


Рисунок 6-8. Результат применения cvCanny для двух совершенно разных изображений, для которых верхний и нижний пороги установлены равными 150 и 100 соответственно

Функция *cvCanny()* на вход принимает исходное изображение в оттенках серого и конечное изображение, которое также должно быть в оттенках серого (в действительности оно будет бинарным - для краев пиксели будут белого цвета, для остальных черного). Следующие два аргумента - это верхний и нижний порог, последний аргумент - размер апертуры (по умолчанию, применяется матрица размера 3x3, но возможны и другие значения - 5x5, 7x7 и т.д.). Апертура поступает на вход оператору Собеля, который вызывается внутри функции *cvCanny()*.

[П] | [РС] | (РП) Преобразования Хафа

Преобразования Хафа - это метод нахождения линий, кругов и других простых форм на изображении (Хаф разработал данное преобразование для применения в физических экспериментах. Внедрение преобразования для решения задач компьютерного зрения осуществили Duda и Hart). Первоначально преобразование Хафа применялось для нахождения линий, т.к. является относительно быстрым способом нахождения прямых линий на бинарном изображении. В дальнейшем данное преобразование обобщили для более сложных задач, чем просто поиск линий.

Преобразование Хафа для линий

Теоретической подоплекой преобразования Хафа для линий является то, что любая точка на бинарном изображении, возможно, является частью некоторого множества линий. Если описать каждую линию по её наклону a и сдвигу b {Roman: наклон и пересечение (или наклон-сдвиг) - параметры уравнения прямой $y = k \cdot x + b$, где k - угловой коэффициент прямой, вычисляемый через тангенс, а пересечение b - коэффициент сдвига прямой по оси u так, чтобы прямая пересекала ось u на высоте b }, то точка на исходном изображении преобразуется во множество точек на плоскости (a, b) , соответствующие всем линиям, проходящих через эту точку (рисунок 6-9). Если преобразовать каждый ненулевой пиксель во входном изображении в такой набор точек в выходном изображении и просуммировать все подобные внесения, тогда линия, появившаяся на входном изображении (то есть на плоскости (x, y)) будет соответствовать локальному максимуму в выходном изображении (то есть на плоскости (a, b)). Поскольку суммируется вклад от каждой точки, плоскость (a, b) принято называть накопительной плоскостью (в дальнейшем просто накопитель).

Может оказаться, что плоскость вида наклон-сдвиг не лучший способ представления всех линий, проходящих через точку (из-за значительно различной плотности линий в зависимости от наклона и от того, что интервал возможных наклонов лежит в диапазоне от $-\infty$ до $+\infty$) {Roman - бесконечности из-за того, что наклон рассчитывается через тангенс}. По этой причине, в численных расчетах используется другая параметризация преобразования изображения. Предпочтительная параметризация представляет каждую линию как точку в полярных координатах (ρ, θ) , при этом эта линия проходит через указанную точку, но линия должна быть перпендикулярной к радиус-вектору от начала координат до этой точки на линии (рисунок 6-10). Уравнение для такой прямой:

$$\rho = x \cos \theta + y \sin \theta$$

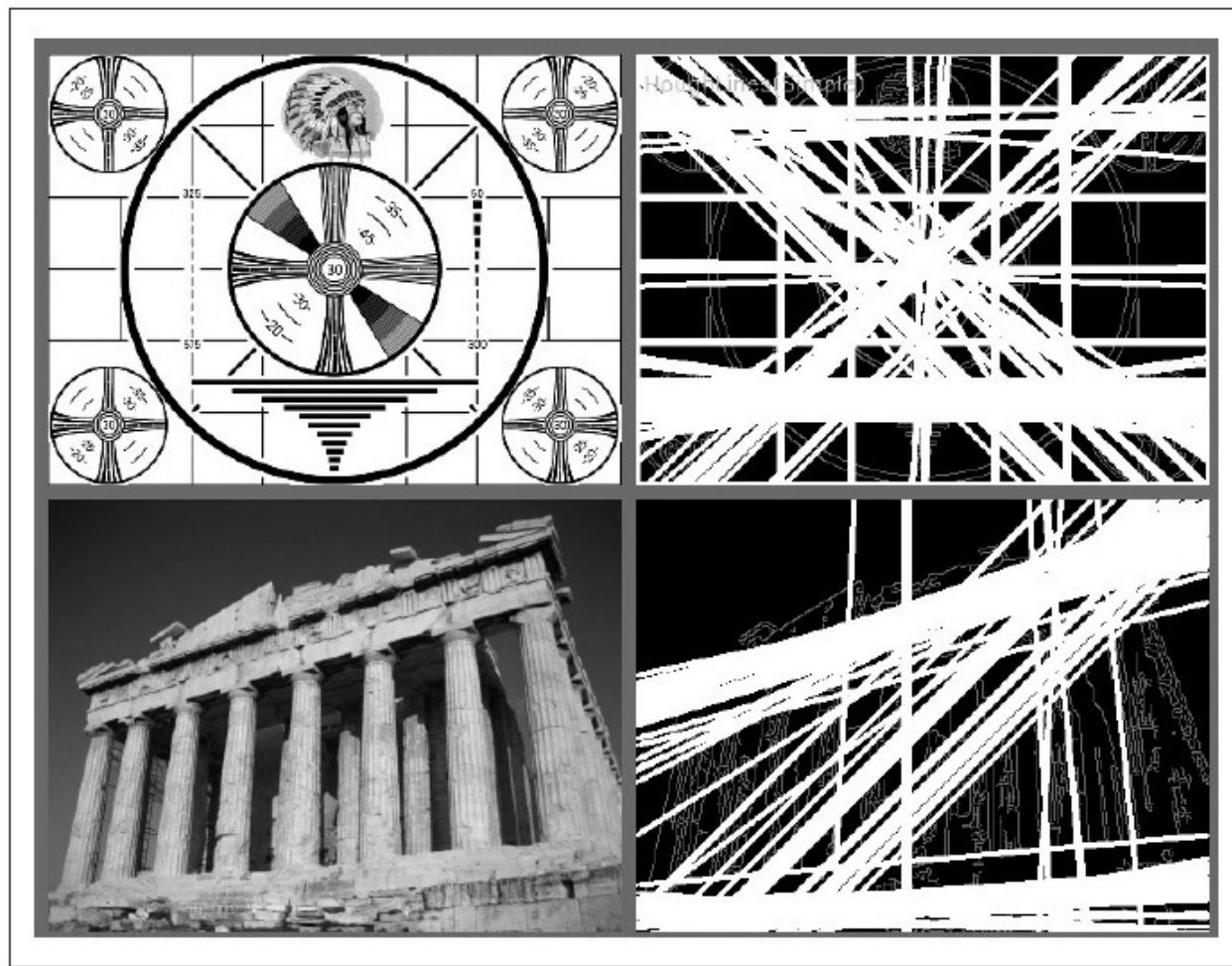


Рисунок 6-9. Преобразование Хафа для линий находит множество линий на каждом изображении. Некоторые из них ожидаемые, но другие могут не быть таковыми

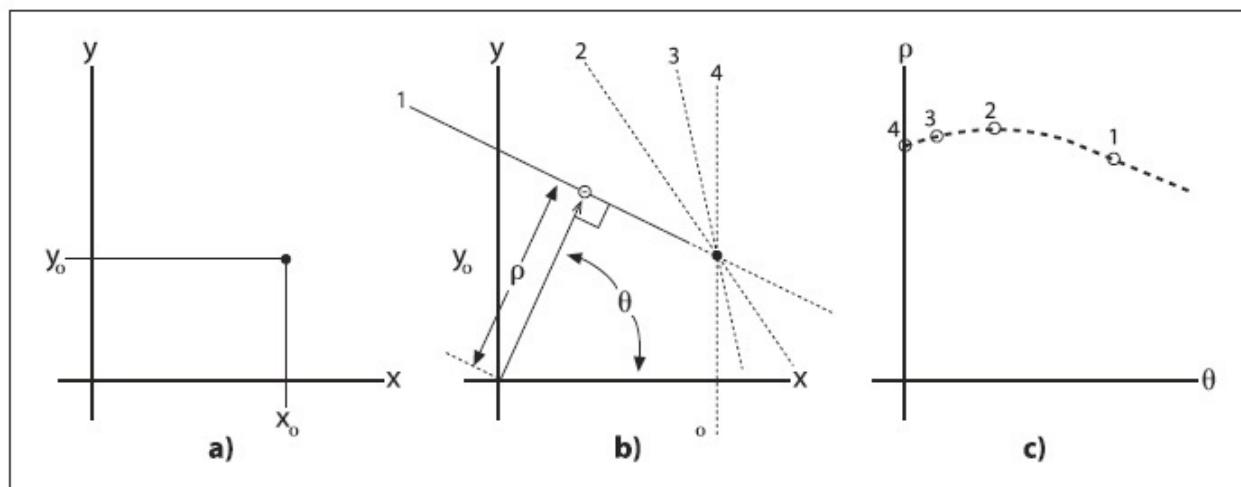


Рисунок 6-10. Точка (x_0 , y_0) на плоскости изображения (график а) соответствует множеству линий, каждая из которых параметризована разными ρ и θ (график б); каждая из этих линий соответствует точкам на плоскости (ρ, θ) , которые, будучи собранными вместе, образуют кривую характеристической формы (график с)

Алгоритм преобразования Хафа не является явным для пользователя. Вместо этого он просто возвращает локальный максимум на плоскости (ρ, θ) . Однако, необходимо понимать процесс преобразования, чтобы осознавать назначения входных аргументов функции, выполняющая преобразования Хафа для линий.

OpenCV поддерживает два вида преобразований Хафа для линий: *обычное преобразование Хафа* (SHT) и *прогрессивное (улучшенное) вероятностное преобразование Хафа* (PRHT). Ранее уже был рассмотрен алгоритм SHT. PRHT является его разновидностью и также вычисляет протяжённость каждой линии в дополнение к их наклону (рисунок 6-11). Алгоритм назван "вероятностным", т.к. вместо добавления каждой возможной точки на накопительную плоскость, он добавляет только часть из них. Идея состоит в том, что если существует максимум на плоскости, то, в любом случае, хотя бы частичное попадание в этот максимум будет достаточным условием, чтобы обнаружить линию; как результат - существенное снижение времени выполнения вычислений. Для обоих случаев в OpenCV существует одна функция, которая, в зависимости от входных параметров, принимает решение о выполнении того или иного метода.

```
CvSeq* cvHoughLines2(
    CvArr* image
    ,void* line_storage
    ,int method
    ,double rho
    ,double theta
    ,int threshold
    ,double param1 = 0
    ,double param2 = 0
);
```

Первый аргумент - это исходное изображение. Оно должно быть 8-битным, но при этом будет трактоваться как бинарное (т.е. все ненулевые значения будут восприниматься как равные единицы). Второй аргумент - указатель на место, где будет храниться результат, который может быть либо хранилищем в памяти (*CvMemoryStorage* из главы 8), либо матрицей размера $N \times 1$ (при этом количество строк N будет также еще ограничивать максимальное число возвращаемых линий). Следующий аргумент *method* может быть *CV_HOUGH_STANDARD*, *CV_HOUGH_PROBABLISTIC*, *CV_HOUGH_MULTI_SCALE* для SHT, PRHT и многопараметрического варианта SHT (MSHT) соответственно.

Следующие два аргумента, *rho* и *theta*, устанавливают желательное разрешение для линий (т.е. разрешение накопительной плоскости). Параметр *rho* вычисляется в пикселях, а параметр *theta* в радианах, поэтому накопительную плоскость можно рассматривать как двухмерную гистограмму с ячейками размерностью *rho* пикселей на

theta радиан. Значение параметра *threshold* определяет величину, при достижении которой сообщается о нахождении линии. Этот последний параметр на практике несколько мудрён, при чём он не нормализуется, поэтому ожидается, что сам разработчик будет его масштабировать с учетом роста размерности входного изображения для алгоритма SHT. Помните, этот параметр, в действительности, определяет количество точек, которое должна содержать линия, чтобы быть добавленной в возвращаемый список линий.

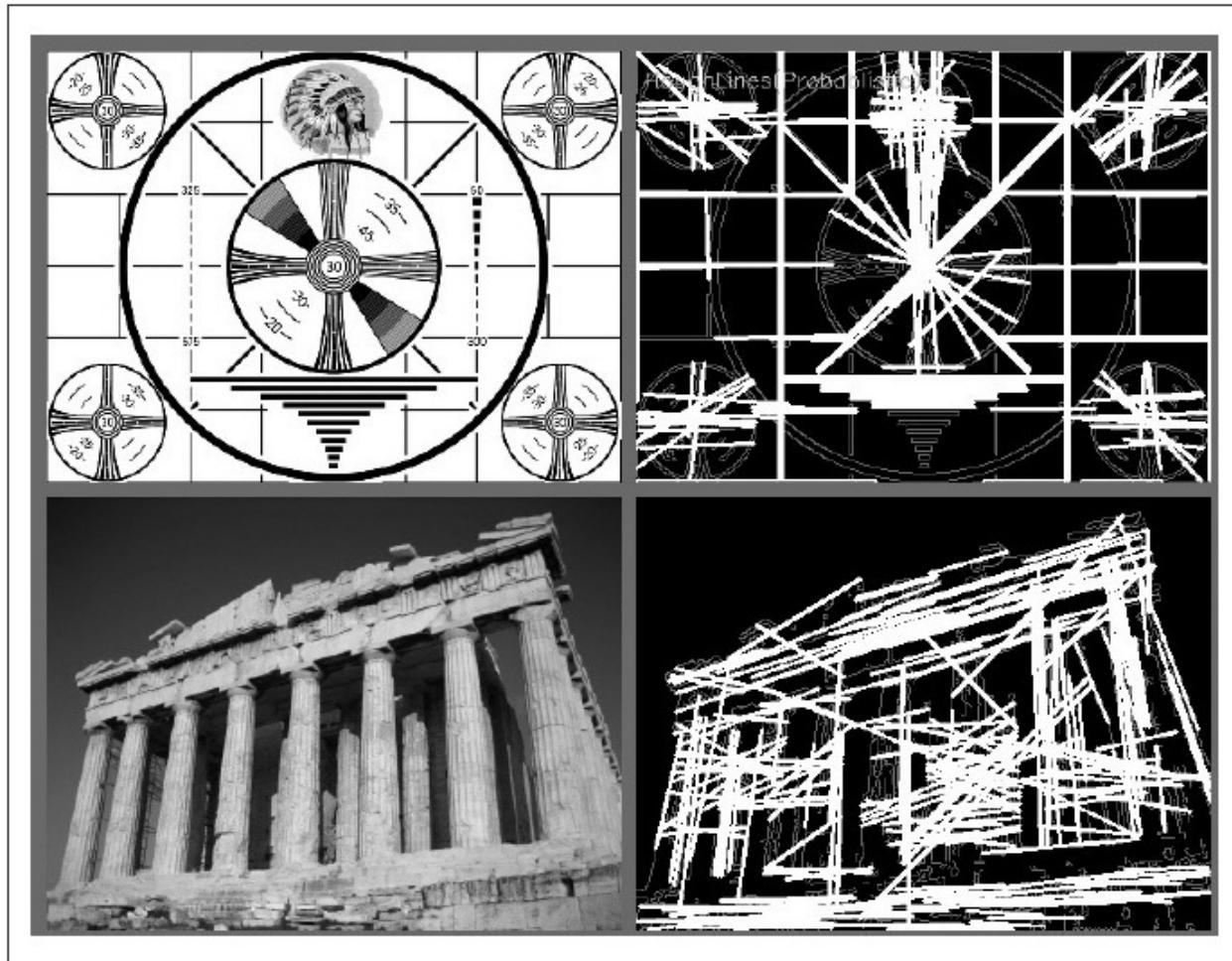


Рисунок 6-11. Сначала выполнен проход при помощи детектора границ Canny (*param1* = 50, *param2* = 150), результат показан в оттенках серого. Затем произведено прогрессивное вероятностное преобразование Хафа (*param1* = 50, *param2* = 10), результат показан белым. Заметьте, что преобразование Хафа в основном правильно обнаруживает чёткие линии

Аргументы *param1* и *param2* алгоритм SHT не использует. Для алгоритма PPHT *param1* задает минимальную длину для возвращаемого сегмента линии, а аргумент *param2* задает расстояние между коллинеарными сегментами, необходимое для того, чтобы алгоритм не склеил их вместе в один сегмент большей длины. Для многопараметрического варианта SHT эти два параметра применяются для указания наивысшего разрешения до которого параметры возвращаемых линий должны быть

вычислены. Многопараметрический SHT сначала вычисляет положение линий с учетом разрешений *rho* и *theta*, а затем начинает уточнение результатов до степени параметров *param1* и *param2* соответственно (т.е. конечное разрешение *rho* – это *rho* делённое на *param1*, а конечное разрешение для *theta* равно *theta* делённому на *param2*).

Возвращаемое значение функции зависит от входных параметров. Если параметр *line_storage* матрица, тогда возвращаемое значение будет NULL. В этом случае, матрица должна быть типа CV_32FC2 для SHT или многопараметрического SHT и CV_32SC4 для PPHT. В первых двух случаях величины ρ и θ для каждой линии будут размещены в двух каналах массива. В случае PPHT, четыре канала будут содержать значения x и y для начальной и конечной точек возвращаемого сегмента. И для всех этих случаев, количество строк в массиве будет обновлено функцией *cvHoughLines2()* до количества возвращаемых линий.

Если параметр *line_storage* содержит указатель на хранилище в памяти (более подробно об этом будет рассказано в главе 8), тогда возвращаемым значением будет указатель на структуру последовательности CvSeq. В этом случае, можно получить каждую линию или сегмент линии из последовательности при помощи подобной команды:

```
float* line = (float*) cvGetSeqElem( lines, i );
```

где *lines* это возвращаемое значение функции *cvHoughLines2()*, а *i* - индекс запрашиваемой линии. В этом случае, *line* будет указателем на данные запрашиваемой линии, при чем *line[0]* и *line[1]* будут действительными числами, соответствующие значениям ρ и θ (для SHT и MSHT), либо указателем на структуру из парных значений CvPoint для начальной и конечной точек сегмента (для алгоритма PPHT).

Преобразование Хафа для окружностей

Преобразование Хафа для окружностей (рисунок 6-12) работает почти аналогично только что описанному преобразованию Хафа для линий. "Почти" только лишь потому, что - если выполнить точно аналогичные действия - накопительная плоскость будет замещена плоскостью объема с тремя измерениями: одно для x , одно для y и последнее для радиуса круга r . Это приводит к значительно большим затратам памяти и значительно меньшей скорости выполнения. Реализация преобразования Хафа для окружности в OpenCV уходит от этой проблемы, применяя несколько более хитрый метод, называемый *градиентным методом Хафа*.

Градиентный метод Хафа работает следующим образом. Вначале изображение проходит фазу поиска краев (использование функции `cvCanny()`). Затем для каждой ненулевой точки краев изображения ищется локальный градиент (вычисляется путем расчета производных Собеля первого и второго порядка для x и y при помощи функции `cvSobel()`). Используя этот градиент, каждая точка линии обозначается как наклон - от установленного минимума до указанного максимального расстояния - итерационно изменяя накопитель. В тоже время, запоминается расположение каждой ненулевой точки на изображении краев. Центры-кандидаты затем выбираются из тех точек (двухмерного) накопителя, величины которых выше заданного порога и, одновременно, больше всех их непосредственных соседей. Эти центры-кандидаты сортируются в порядке убывания их величины в накопителе, так, что центры с наибольшим количеством пикселей выбираются первыми. Далее, для каждого центра, анализируются все ненулевые пиксели. Эти пиксели сортируются в соответствии с их расстоянием от центра. Обрабатывая от наименьших расстояний до наибольших радиусов, выбирается единственный радиус, который лучше всего подходит ненулевым пикселям. Центр сохраняется, если он имеет достаточное количество ненулевых пикселей на краях изображения и если расстояние от любого ранее выбранного центра удовлетворяет заданному значению.

Эта реализация позволяет алгоритму выполняться намного быстрее и, что еще более важно, позволяет обойти проблему разрастания трехмерного накопителя, что приводит к значительно большому зашумлению и нестабильно отражаемому результату. С другой стороны, этот алгоритм имеет несколько недостатков.

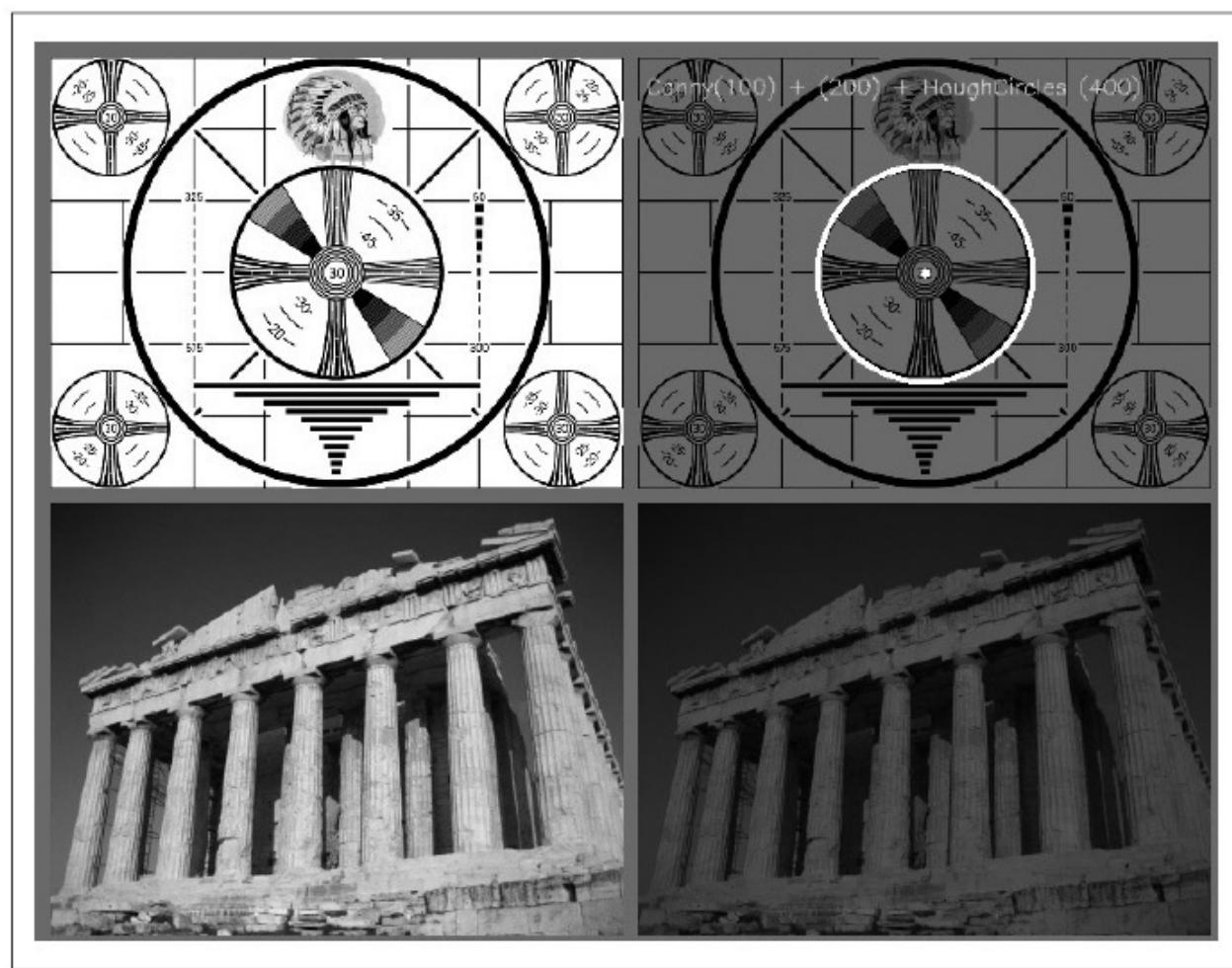


Рисунок 6-12. Преобразование Хафа для круга находит несколько кругов на тестовом шаблоне и (правильно) не находит ничего на фотографии

Во-первых, применение производных Собеля для вычисления локального градиента - и предположение, что он может быть рассмотрен как локальный тангенс - численно не стабильное решение. Он может давать верный результат "большую часть времени", при этом на выходе не исключено присутствие некоторого количества шума.

Во-вторых, каждый ненулевой пиксель на гранях изображения рассматривается как центр-кандидат, поэтому для малого значения порога накопителя время выполнения алгоритма резко возрастет. В-третьих, поскольку для каждого центра выбирается только одна окружность, то в случае существования концентрических окружностей, используется только одна из них.

И наконец, поскольку центры рассматриваются в возрастающем порядке значений из накопителя и к тому же новые центры не добавляются, если они слишком близки к уже добавленным центрам, в случае концентрических кругов (либо около концентрических), предпочтение может отдаваться тем, у которых наибольший размер. (Указано "может" потому что существует некоторый шум, исходящий от производных Собеля; на гладком изображении с бесконечным расширением алгоритм будет определенным)

Принимая во внимание все выше сказанное, рассмотрим функцию OpenCV, которая реализует алгоритм Хафа для окружностей:

```
CvSeq* cvHoughCircles(
    CvArr* image
    ,void* circle_storage
    ,int method
    ,double dp
    ,double min_dist
    ,double param1      = 100
    ,double param2      = 300
    ,int   min_radius   = 0
    ,int   max_radius   = 0
);
```

Функция преобразования Хафа для окружностей *cvHoughCircles()* имеет параметры, схожие с параметрами функции для линий. Исходное изображение должно быть 8-битным. Значительное отличие между *cvHoughCircles()* и *cvHoughLines2()* в том, что последняя функция запрашивает бинарное изображение. Функция *cvHoughCircles()* внутренне (автоматически) сама вызывает функцию *cvSobel()* (Внутренне вызывается функция *cvSobel()*, вместо *cvCanny()*). Это связано с тем, что *cvHoughCircles()* необходимо определить направление градиента для каждого пикселя, а это сложно выполнить для бинарной карты краев), поэтому вполне хватает и изображения в оттенках серого.

Аргумент *circle_storage* может быть либо массивом, либо хранилищем в памяти - зависит от предпочтений разработчика. Если массив, то он должен иметь одну колонку типа *CV_32FC3*; три канала используются для хранения значений положения и радиуса круга. Если хранилище в памяти, то окружности будут преобразованы в последовательности OpenCV, а функция *cvHoughCircles()* возвратит указатель на эту последовательность. (При указании в *circle_storage* указателя на массив, возвращаемым значением функции будет *NULL*). Аргумент *method* всегда должен быть равен *CV_HOUGH_GRADIENT*.

Аргумент *dp* является разрешением накопителя. Этот аргумент позволяет создавать накопитель более низких разрешений, чем исходное изображение. (Есть смысл так делать, т.к. нет никаких оснований для существования окружностей, которые точно бы подходили под количество категорий, такие как высота или ширина изображения, сами по себе). Если аргумент *dp* равен 1, тогда разрешение будет таким же, как и у исходного изображения; если аргумент *dp* больше 1, тогда разрешение будет в *dp* раз меньше (в случае *dp* = 2, будет половинным). Значение *dp* не может быть меньше 1.

Аргумент *min_dist* задает минимальное расстояние между двумя окружностями, чтобы алгоритм рассматривал их как две разные окружности.

Для применяемого (и требуемого на данный момент) метода *CV_HOUGH_GRADIENT*, следующие два параметра, *param1* и *param2*, являются порогами для краев (Canny) и накопителя соответственно. В данном случае может смутиТЬ, что Canny сам по себе запрашивает два разных параметра для порогов. На самом деле внутренний вызов функции *cvCanny()* устанавливает в качестве верхнего порога значение равное *param1*, а нижний порог устанавливает равным половине величине *param1*. Параметр *param2* применяется к накопителю в точно таком же смысле, как и параметр *threshold* для *cvHoughLines()*.

Последние два параметра - это минимально и максимально возможные значения радиуса для окружностей, которые ищутся. Это означает, что они являются радиусами окружностей, для которых накопитель имеет представление. Пример 6-1 отражает пример использования функции *cvHoughCircles()*.

Пример 6-1. Использование *cvHoughCircles()* для получения последовательности найденных окружностей на изображение в оттенках серого

```

#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv) {
    IplImage* image = cvLoadImage(
        argv[1]
        , CV_LOAD_IMAGE_GRAYSCALE
    );

    CvMemStorage* storage = cvCreateMemStorage( 0 );
    cvSmooth( image, image, CV_GAUSSIAN, 5, 5 );

    CvSeq* results = cvHoughCircles(
        image
        ,storage
        ,CV_HOUGH_GRADIENT
        ,2
        ,image->width/10
    );

    for( int i = 0; i < results->total; i++ ) {
        float* p = (float*) cvGetSeqElem( results, i );
        CvPoint pt = cvPoint( cvRound( p[0] ), cvRound( p[1] ) )
        cvCircle(
            image
            ,pt
            ,cvRound( p[2] )
            ,CV_RGB( 0xff, 0xff, 0xff )
        );
    }

    cvNamedWindow( "cvHoughCircles", 1 );
    cvShowImage( "cvHoughCircles", image );
    cvWaitKey( 0 );
}

```

Стоит заострить внимание на том, что независимо от все возможных применяемых приемов, описать окружности можно только при помощи трех степеней свободы (x , y и r), в то время как для линий достаточно только двух (ρ и θ). Как результат, алгоритм поиска окружностей, по сравнению с алгоритмом поиска линий, затрачивает на выполнение больше времени и памяти. Принимая во внимание данный факт, неплохо было бы ограничивать радиус окружности насколько это возможно в той или иной ситуации (Хотя `cvHoughCircles()` улавливает центры достаточно точно, он иногда не способен найти правильный радиус. Поэтому в приложениях, где требуется только нахождение центра (либо там, где для нахождения радиуса можно применить другие

механизмы), величину радиуса, возвращаемую *cvHoughCircles* можно игнорировать). Преобразование Хафа было расширено произвольными формами Балларда в 1981, в основном рассматривая формы, как наборы градиентных граней.

[П]|[РС]|(РП) Remap

"Под капотом", многие преобразования имеют некий общий элемент. В частности, они выбирают пиксели, находящиеся в одном положении и отображают их в другое положение. В этом случае, всегда будет некоторое гладкое отображение, которое будет выполнять запрашиваемое действие, при этом, не всегда выполняющийся перенос пикселей будет один к одному.

Иногда, возникает необходимость программно произвести такую интерполяцию; т.е. применить какой-нибудь известный алгоритм, который будет определять отображение. Однако, в других случаях, нужно провести отображение по-своему. Прежде, чем перейти к более углубленному изучению методов осуществления такого отображения, рассмотрим функцию, отвечающую за применение отображений и составляющую основу для остальных методов. В OpenCV эта функция именуется как *cvRemap()*:

```
void cvRemap(
    const CvArr*    src
    ,CvArr*        dst
    ,const CvArr*   mapx
    ,const CvArr*   mapy
    ,int            flags = CV_INTER_LINEAR | CV_WARP_FILL_OUTLIERS
    ,CvScalar       fillval = cvScalarAll(0)
);
```

Первые два параметра *cvRemap()* - это исходное и конечное изображения. Очевидно, что они должны быть одинакового размера и с равным количеством каналов, но при этом иметь произвольный формат данных. Стоит отметить, что исходное и конечное изображения это два разных изображения (Немного поразмыслив, можно разобраться, почему наиболее эффективная стратегия отображения не совместима с записью в исходное изображение. В конце концов, если переместить пиксель А в положение В, а затем, когда дойдет очередь до перемещения В в положение С, окажется, что исходное значение В переписано значением А). Следующие два параметра, *mapx* и *mapy*, указывают положение в которое должны быть перемещены пиксели. Они должны быть такого же размера, как исходное и конечное изображения, но при этом одноканальными и, как правило, вещественного типа (*IPL_DEPTH_32F*). Нецелочисленные отображения являются допустимыми и *cvRemap()* производит необходимые интерполяционные вычисления автоматически. Одно из наиболее общих применений *cvRemap()* - подправить (устранить искажения) калибровку и стереоизображение. Эти варианты применения функции будут рассмотрены в главах

11 и 12. Следующие два параметра содержат флаги, точно указывающие `cvRemap()` каким образом производить интерполяцию. Все возможные значения указаны в таблице 6-1.

Таблица 6-1. Дополнительные значения флагов `cvWarpAffine()`

Флаг	Значение
<code>CV_INTER_NN</code>	Ближайшие соседи
<code>CV_INTER_LINEAR</code>	Билинейная (по умолчанию)
<code>CV_INTER_AREA</code>	Перебор области пикселя
<code>CV_INTER_CUBIC</code>	Бикубическая интерполяция

Интерполяция - это крайне важный момент. Пиксели в исходном изображении находятся на целочисленной сетке; так, например, можно обратиться к пикселью с координатами (20, 17). При отображении целочисленных координат в новое изображение, могут появиться зазоры - либо, поскольку целочисленные координаты пикселей на исходном изображении отображаются в вещественные координаты на конечном изображении и должны быть округлены до ближайшей целой координаты пикселя, либо из-за существования нескольких локаций, в которые пиксель не отображается (например, при увеличении изображения, путем растягивания; тогда каждый второй пиксель будет пустым). Эти проблемы обычно обобщаются как проблемы *прямого проецирования*. Чтобы разобраться с подобного рода проблемами округления и проблемами в конечном изображении, как правило, необходимо решить обратную задачу: совершить проход по каждому пикселью в конечном изображении и ответить на вопрос: "Какие пиксели исходного изображения необходимо перенести в пиксели конечного изображения?". Эти исходные пиксели почти всегда будут расположены в дробных координатах, так что потребуется интерполировать исходные пиксели, чтобы получить корректную величину для конечного пикселя. По умолчанию, применяется линейная интерполяция (таблица 6-1).

Флаги можно комбинировать (с использованием оператора OR); так добавление флага `CV_WARP_FILL_OUTLIERS` позволит заполнить те пиксели в конечном изображении, для которых нет пикселей в исходном, в соответствии со значением, указанным в последнем параметре *fillval*. В этом случае, при переносе изображения на круг, размещенного в центре, область вне круга будет автоматически заполнена черным цветом (либо любым указанным цветом).

[П] | [РС] | (РП) Растигивание, сжатие, деформация и поворот

В данном разделе речь пойдет о геометрических манипуляциях над изображениями (об этих преобразованиях довольно таки подробно будет рассказано в этом разделе и в главе 11, когда они будут использованы в контексте трехмерного компьютерного зрения). Такие манипуляции включают в себя растигивания в различных направлениях, при этом речь может идти как об однородных, так и об неоднородных изменениях размера (последнее более известно, как *деформация*). Есть множество причин, чтобы выполнить эти операции: например, деформируя и поворачивая изображение так, чтобы уместить его в существующей сцене или для искусственного увеличения набора шаблонных изображений, используемых для распознавания объектов (это может показаться хитроумным; в конце концов, почему бы просто не использовать метод распознавания, инвариантный к локальным аффинным искажениям? Тем не менее, этот метод имеет долгую историю и до сих пор может быть весьма полезным на практике). Функции, которые могут растигивать, сжимать, деформировать и/или поворачивать изображение именуются *геометрическими трансформациями*. Для плоскости есть две разновидности геометрических преобразований: преобразование, которое использует матрицу 2x3 и именуется *аффинным преобразованием*; и преобразование, которое использует матрицу 3x3 и именуется *перспективным преобразованием* или *гомографией*. Для последнего преобразования нужно представить, что оно основано на методе вычисления пути, по которому в трехмерном измерении направлен взор конкретного наблюдателя, при условии, что прямо взглянуть на эту плоскость он не может.

Аффинные преобразования - это любые преобразования, которые могут быть выражены в виде последовательного перемножения матриц и сложения с некоторым вектором. В OpenCV стандартной для таких преобразований является матрица размера 2x3.

$$\mathbf{A} \equiv \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad \mathbf{B} \equiv \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \quad \mathbf{T} \equiv \begin{bmatrix} A & B \end{bmatrix} \quad \mathbf{X} \equiv \begin{bmatrix} x \\ y \end{bmatrix} \quad \mathbf{X}' \equiv \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Нетрудно заметить, что результатом аффинных преобразований является $\mathbf{AX} + \mathbf{B}$, что в точности эквивалентно расширению вектора \mathbf{X} до вектора \mathbf{X}' и последующего левого умножения на вектор \mathbf{T} .

Аффинные преобразования могут быть представлены следующим образом: любой параллелограмм ABCD на плоскости может быть отражен в любой другой параллелограмм A'B'C'D'; если площади этих параллелограммов не равны нулю, тогда подразумеваемое аффинное преобразование однозначно определено для трех вершин обоих параллелограммов. Так же для понимания аффинного преобразования можно представить изображение в виде большого резиного листа, который в результате деформации нажатием или растягиванием (вплоть до переворота параллелограмма) за углы может быть преобразован в разнообразные параллелограммы.

Имея несколько изображений одного объекта с разных ракурсов, можно применить фактическое преобразование для связывания различных ракурсов. В этом случае, зачастую для моделирования ракурсов используются аффинные преобразования, потому что задействуется малое количество параметров, что облегчает процесс вычисления. При этом недостатком является то, что реальные перспективные преобразования могут быть смоделированы только при помощи гомографии ("гомография" - это математический термин для отображения точек одной поверхности к точкам на другой поверхности. В контексте компьютерного зрения, под гомографией почти всегда имеется ввиду отображение между точками двух плоскостей изображений, которые соответствуют расположению объектов на плоскости в реальном мире). Такое преобразование представляется ортогональной матрицей 3x3 (подробнее об этом в главе 11)), так как аффинные преобразования дают представления, которые не могут вместить все возможные связи между ракурсами. С другой стороны, для небольших изменений положения наблюдателя, результирующее искажение будет аффинным, поэтому в некоторых случаях аффинных преобразований вполне достаточно.

За счет аффинного преобразования прямоугольник может быть преобразован в параллелограмм. При этом, в результате преобразований (вращения и/или масштабирования), стороны должны сохранять параллельность. Перспективные преобразования обеспечивают большую гибкость; прямоугольник может быть преобразован в трапецию. Так как параллелограмм можно рассматривать как частный случай трапеции, то и аффинное преобразование можно рассматривать как подмножество перспективного преобразования. На рисунке 6-13 представлены примеры различных аффинных и перспективных преобразований.

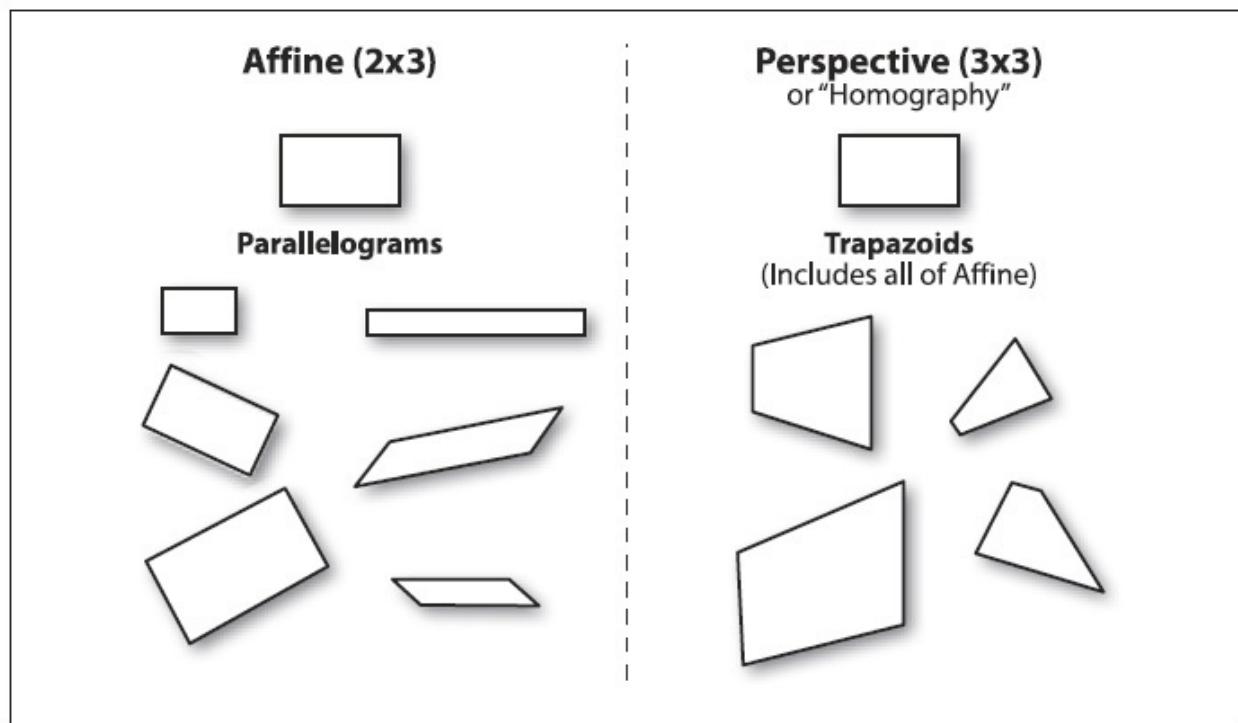


Рисунок 6-13. Аффинные и перспективные преобразования

Аффинные преобразования

Есть две ситуации, при которых используются аффинные преобразования. В первом случае для выполнения преобразования над изображением (или его частью); во втором случае для выполнения преобразования по списку точек.

Плотные аффинные преобразования

В первом случае очевидно, что входные и выходные форматы - это изображения, и для преобразования явно требуется, чтобы пиксели были *плотным представлением* нижележащего изображения. Это значит, что преобразователь изображения должен производить интерполяцию, так что конечное изображение будет выглядеть натурально и сглажено. Функция OpenCV, отвечающая за аффинные преобразования, именуется `cvWarpAffine()`.

```
void cvWarpAffine(
    const CvArr*    src
    ,CvArr*         dst
    ,const CvMat*   map_matrix
    ,int            flags = CV_INTER_LINEAR | CV_WARP_FILL_OUTLIERS
    ,CvScalar       fillval = cvScalarAll(0)
);
```

Параметры *src* и *dst* указывают на массив или изображение, которые могут быть одно- или трехканальными произвольного типа (при этом оба обязательно одного типа и размера) (Так как вращение изображения приводит к расширению описывающего его прямоугольника, результатом станет отсеченное изображение. Обойти эту проблему можно либо за счет сжатия изображения, либо за счет копирования первого изображения в область интереса ROI целевого изображения большего размера, которое затем используется как исходное изображение для проведения преобразования). Параметр *map_matrix* - это матрица размера 2x3 по которой вычисляется требуемое преобразование. Предпоследний параметр *flags* определяет метод интерполяции, ровно, как и следующие вспомогательные опции (комбинирование всех возможных значений флага производиться при помощи **OR**).

CV_WARP_FILL_OUTLIERS - зачастую, в результате преобразований, изображение *src* не вписывается в изображение *dst*, поэтому на конечное изображение нужно *перенести* пиксели исходного изображения, которых в действительности нет. Если данный флаг выбран, то эти потерянные значения заполняются значением параметра *fillval*.

CV_WARP_INVERSE_MAP. Этот флаг предназначен для обратного преобразования *dst* в *src*, вместо стандартного *src* в *dst*.

Производительность cvWarpAffine

Важно знать, что использование *cvWarpAffine()* приводит к значительным накладным расходам. Альтернативой является использование *cvGetQuadrangleSubPix()*. Эта функция функционально ограничена, однако, имеет ряд преимуществ. В частности, она имеет меньшие накладные расходы и обрабатывает частный случай, когда исходное изображение 8 битное, а конечное изображение вещественное. Так же эта функция может обрабатывать и многоканальные изображения.

```
void cvGetQuadrangleSubPix(
    const CvArr*    src
    ,CvArr*        dst
    ,const CvMat*   map_matrix
);
```

cvGetQuadrangleSubPix() вычисляет все точки *dst* путем (интерполированного) их отображения из точек *src*, за счет применения аффинного преобразования, что в свою очередь подразумевает умножение на матрицу *map_matrix* размером 2x3. (Преобразование расположения пикселей в *dst* к однородным координатам для выполнения умножения производиться автоматически)

Особенность `cvGetQuadrangleSubPix()` заключается в том, что функция производит дополнительное отображение. В частности, результат, указанный в `dst` вычисляется по следующей формуле:

$$dst(x, y) = \text{src}(a_{00}x'' + a_{01}y'' + b_0, a_{10}x'' + a_{11}y'' + b_1)$$

где

$$M_{\text{map}} \equiv \begin{bmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} x - \frac{(\text{width(dst)}-1)}{2} \\ y - \frac{(\text{height(dst)}-1)}{2} \end{bmatrix}$$

Стоит обратить внимание, что отображение (x, y) в (x'', y'') имеет следующий эффект - даже если преобразование M единичное преобразование - точки в центре конечного изображения будут соответствовать точкам, взятым из начала отсчета исходного изображения. Если `cvGetQuadrangleSubPix()` требуется точка вне изображения, будет задействована репликация для восстановления этих значений.

Вычисление матрицы аффинного преобразования

OpenCV предоставляет две функции для вычисления матрицы `map_matrix`. Первая используется, когда уже есть два изображения, о которых известно, что они соотносятся неким аффинным преобразованием или могут быть аппроксимированы следующим образом:

```
CvMat* cvGetAffineTransform(
    const CvPoint2D32f* pts_src
    , const CvPoint2D32f* pts_dst
    , CvMat* map_matrix
);
```

`src` и `dst` - это массивы, содержащие три двухмерные (x, y) точки, а `map_matrix` - вычисляемая из этих точек аффинная матрица.

Аргументы `pts_src` и `pts_dst` функции `cvGetAffineTransform()` - это всего лишь массивы из трех точек, определяющие два параллелограмма. Простейший способ определить аффинное преобразование - установить указатель `pts_src` на три (вполне достаточно трех точек, т.к. в аффинном преобразовании используется параллелограмм. Четыре точки будут задействованы только в случае с трапецией общего вида в перспективном преобразовании) угла исходного изображения - например, на верхний и нижний левый и на верхний правый. Отображение исходного изображения в конечное изображение полностью определяется указанием в `pts_dst` места, в котором по трем точкам будет

отображено конечное изображение. После произведения отображения этих трех независимых углов (которые на самом деле "представляют" параллелограмм), все остальные точки могут быть преобразованы соответственно.

В примере 6-2 приведен код, использующий эти функции. В данном примере получение параметров матрицы функции *cvWarpAffine()* производиться следующим образом: в начале происходит построение двух трехкомпонентных массивов точек (для углов параллелограмма), а затем преобразование их в действительную матрицу преобразований за счет вызова *cvGetAffineTransform()*. Потом выполняются аффинные преобразования с последующим вращением изображения. Для исходного изображения создается массив точек именуемый *srcTri[]* с заданными точками: (0, 0), (0, *height* - 1) и (*width* - 1, 0). В завершении указывается в какое соответствующее расположение будут отображены точки из массива *srcTri[]* в массив *dstTri[]*.

Пример 6-2. Аффинные преобразования

```
// Пример использования: warp_affine <файл_изображения>
//  

#include <cv.h>  

#include <highgui.h>  

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

    CvPoint2D32f    srcTri[3], dstTri[3];  

    CvMat*          rot_mat = cvCreateMat( 2, 3, CV_32FC1 );  

    CvMat*          warp_mat = cvCreateMat( 2, 3, CV_32FC1 );  

    IplImage        *src, *dst;  

    if( argc == 2 && ((src=cvLoadImage(argv[1], 1)) != 0) ) {  

        dst = cvCloneImage( src );  

        dst->origin = src->origin;  

        cvZero( dst );  

        // Вычисление матрицы преобразования  

        //  

        srcTri[0].x = 0;                                //Верхний левый угол источника  

        srcTri[0].y = 0;  

        srcTri[1].x = src->width - 1;                  //Верхний правый угол источника  

        srcTri[1].y = 0;  

        srcTri[2].x = 0;                                //Сдвиг нижнего левого угла источника  

        srcTri[2].y = src->height - 1;  

        dstTri[0].x = src->width * 0.0;                 //Верхний левый угол приемника  

        dstTri[0].y = src->height * 0.33;  

        dstTri[1].x = src->width * 0.85;                //Верхний левый угол приемника  

        dstTri[1].y = src->height * 0.25;  

        dstTri[2].x = src->width * 0.15;                //Сдвиг нижнего левого угла приемника  

        dstTri[2].y = src->height * 0.7;  

        cvGetAffineTransform( srcTri, dstTri, warp_mat );  

        cvWarpAffine( src, dst, warp_mat );  

    }
```

```

cvCopy( dst, src );

// Вычисление матрицы вращения
//
CvPoint2D32f center = cvPoint2D32f(
    src->width/2
    ,src->height/2
);

double angle = -50.0;
double scale = 0.6;
cv2DRotationMatrix( center, angle, scale, rot_mat );

// Выполнение преобразования
//
cvWarpAffine( src, dst, rot_mat );

// Отображение результата
//
cvNamedWindow( "Affine_Transform", 1 );
cvShowImage( "Affine_Transform", dst );
cvWaitKey();

}

cvReleaseImage( &dst );
cvReleaseMat( &rot_mat );
cvReleaseMat( &warp_mat );

return 0;
}

```

Второй функцией вычисления матрицы *map_matrix* является *cv2DRotationMatrix()*, которая вычисляет матрицу преобразований при помощи вращения вокруг заданной точки и необязательного масштабирования. Это всего лишь один из возможных вариантов аффинных преобразований, однако, очень важный, т.к. имеет альтернативное (и более интуитивное) представление, которое легче понимать и использовать:

```

CvMat* cv2DRotationMatrix(
    CvPoint2D32f    center
    ,double         angle
    ,double         scale
    ,CvMat*        map_matrix
);

```

Первый аргумент *center* - точка вращения. Следующие два аргумента задают величину вращения и масштабный коэффициент. Последний аргумент - матрица *map_matrix* вещественного типа размерностью 2x3.

Если определить $\alpha = \text{scale} \cdot \cos(\text{angle})$ и $\beta = \text{scale} \cdot \sin(\text{angle})$, тогда функция будет производить вычисление *map_matrix* по следующей формуле:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \text{center}_x - \beta \cdot \text{center}_y \\ -\beta & \alpha & \beta \cdot \text{center}_x + (1-\alpha) \cdot \text{center}_y \end{bmatrix}$$

Эти методы получения *map_matrix* можно комбинировать для получения, например, повернутого, масштабированного и деформированного изображения.

Разряженное аффинное преобразование

Ранее уже было сказано, что для обработки плотных преобразований используется *cvWarpAffine()*. Для разряженного преобразования (т.е. преобразования набора независимых точек) лучше всего использовать *cvTransform()*:

```
void cvTransform(
    const CvArr*    src
    ,CvArr*         dst
    ,const CvMat*   transmat
    ,const CvMat*   shiftvec = NULL
);
```

Аргумент *src* - это массив $N \times 1$ с D_s каналами, где N - количество точек для преобразования, а D_s - размерность этих точек. Аргумент *dst* должен быть того же размера, но может иметь отличное количество каналов D_d . Матрица преобразований *transmat* - это матрица размерностью $D_s \times D_d$, которая затем применяется к каждому элементу из *src*, с последующим размещением результата в *dst*. Если необязательный вектор *shiftvec* не NULL, то он должен быть массивом размерностью $D_s \times 1$, элементы которого добавляются к результату до его размещения в *dst*.

В случае аффинного преобразования, существует два способа использования *cvTransform()*, выбор зависит от представления преобразования. В первом случае, преобразование раскладывается на части размером 2×2 (которая выполняет вращение, масштабирование и деформацию) и размером 2×1 (которая выполняет преобразование). Входные данные: массив $N \times 1$ с двумя каналами, *transmat* локальное гомогенное преобразование, а *shiftvec* содержит все необходимые сдвиги. Во втором случае, применяется обычное представление матрицы аффинного преобразования размерностью 2×3 . В этом случае, *src* - это трехканальный массив, для которого нужно установить все элементы третьего канала в 1 (т.е. точки должны быть представлены в однородных координатах). Разумеется, выходной массив будет двухканальным.

Перспективные преобразования

Для получения большей гибкости при помощи перспективных преобразований (гомографии), необходима новая функция, которая позволит выразить этот широкий класс преобразований. Прежде всего стоит отметить что, хотя перспективная проекция задается полностью при помощи одной матрицы, проекция на самом деле это не линейное преобразование. Это связано с тем, что преобразование требует деление на последнее измерение (обычно на Z, Глава 11), тем самым в процессе преобразования одно измерение теряется.

Как и в аффинных преобразованиях, операции над изображениями (плотные преобразования) и точечные преобразования (разряженные преобразования) обрабатываются различными функциями.

Плотное перспективное преобразование

При плотном перспективном преобразовании используются аналогичные функции OpenCV, что и при плотных аффинных преобразованиях. В частности `cvWarpPerspective()` имеет такой же набор аргументов, как и `cvWarpAffine()`, с одним лишь существенным отличием - матрица отображения должна быть размера 3x3.

```
void cvWarpPerspective(
    const CvArr*    src
    ,CvArr*         dst
    ,const CvMat*   map_matrix
    ,int            flags = CV_INTER_LINEAR + CV_WARP_FILL_OUTLIERS
    ,CvScalar       fillval = cvScalarAll(0)
);
```

Флаги те же, что и в случае с аффинным преобразованием.

Вычисление матрицы перспективного преобразования

Как и в случае аффинных преобразований, для заполнения матрицы `map_matrix` функции `cvWarpPerspective()` есть функция, которая может вычислить матрицу преобразования из списка точечных соответствий:

```
CvMat* cvGetPerspectiveTransform(
    const CvPoint2D32f* pts_src
    ,const CvPoint2D32f* pts_dst
    ,CvMat*           map_matrix
);
```

Аргументы `pts_src` и `pts_dst` теперь массивы четырех точек (а не трех), поэтому теперь контролировать отображение углов (обычно) прямоугольника `pts_src` в некий ромб `pts_dst` можно самостоятельно. Преобразование полностью определяется указанием четырех точек назначения. Ранее уже было отмечено, что для перспективного

преобразования необходимо выделять массив *map_matrix* размера 3x3; в примере 6-3 показан пример применения перспективного преобразования. Иная размерность матрицы отображения, 3x3, и переход от трех контрольных точек к четырем - это все, что отличает перспективное преобразование от аффинного.

Пример 6-3. Пример применения перспективного преобразования

```

// Пример использования: warp <файл_изображения>
//
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv) {
    CvPoint2D32f    srcQuad[4], dstQuad[4];
    CvMat*          warp_matrix = cvCreateMat( 3, 3, CV_32FC1 );
    IplImage        *src, *dst;

    if( argc == 2 && ((src = cvLoadImage(argv[1], 1)) != 0) ) {
        dst = cvCloneImage( src );
        dst->origin = src->origin;
        cvZero(dst);

        srcQuad[0].x = 0;           // src Top left
        srcQuad[0].y = 0;
        srcQuad[1].x = src->width - 1; // src Top right
        srcQuad[1].y = 0;
        srcQuad[2].x = 0;           // src Bottom left
        srcQuad[2].y = src->height - 1;
        srcQuad[3].x = src->width - 1; // src Bottom right
        srcQuad[3].y = src->height - 1;

        dstQuad[0].x = src->width*0.05; // dst Top left
        dstQuad[0].y = src->height*0.33;
        dstQuad[1].x = src->width*0.9; // dst Top right
        dstQuad[1].y = src->height*0.25;
        dstQuad[2].x = src->width*0.2; // dst Bottom left
        dstQuad[2].y = src->height*0.7;
        dstQuad[3].x = src->width*0.8; // dst Bottom right
        dstQuad[3].y = src->height*0.9;

        cvGetPerspectiveTransform(
            srcQuad
            ,dstQuad
            ,warp_matrix
        );

        cvWarpPerspective( src, dst, warp_matrix );
        cvNamedWindow( "Perspective_Warp", 1 );
        cvShowImage( "Perspective_Warp", dst );
        cvWaitKey();
    }

    cvReleaseImage( &dst );
    cvReleaseMat( &warp_matrix );

    return 0;
}

```

Разряженное перспективное преобразование

В OpenCV есть специальная функция `cvPerspectiveTransform()`, которая выполняет точечное перспективное преобразование; использование функции `cvTransform()` недопустимо, т.к. она предназначена только для выполнения линейных операций. В случае же с перспективным преобразованием требуется деление на третью координату однородного представления ($x = f*X/Z$, $y = f*Y/Z$). Функция `cvPerspectiveTransform()` решает все эти задачи за разработчиков самостоятельно.

```
void cvPerspectiveTransform(
    const CvArr*   src
    ,CvArr*        dst
    ,const CvMat*  mat
);
```

Как правило, `src` и `dst` это массивы исходных и конечных точек соответственно; массивы должны быть трехканальными вещественного типа. Размерность матрицы `mat` может быть либо 3x3, либо 4x4. Если 3x3, то проекция не меняет размерность (два); если 4x4, то проекция меняет размерность с четырех до трех.

В текущем контексте происходит преобразование установленных точек на исходном изображении в другие установленные точки на конечном изображении, о котором говорят как об отображении из двумерного в двумерное пространство. Но это не совсем точно, так как на самом деле при перспективном преобразовании происходит отображение точек на двумерную плоскость,строенную в трехмерное пространство, в обратное (другое) двухмерное подпространство. Представляйте это как тоже самое, что делает камера (более подробно об этом будет рассказано в последующих главах с описанием работы камеры). Камера снимает точки трехмерного пространства и отображает их в двухмерном пространстве камеры. По сути, имеется ввиду, что точки источника должны быть взяты в "однородных координатах". Вводится измерение Z и все значения этого измерения заполняются 1. Затем выполняется прямое преобразование проекции, и обратное в двумерном пространстве конечного изображения.

Результаты выполнения аффинных и перспективных преобразований из примера 6-3 показаны на рисунке 6-14. Сравнивая этот результат с диаграммами на рисунке 6-13 можно увидеть, как это работает с реальными изображениями. На рисунке 6-14 преобразованию подверглось всё изображения. Но можно преобразовывать не всё изображение; можно просто в `src_pts` определить меньшую (или большую!) область преобразования. К тому же можно задействовать ROI в исходном или конечном изображении для ограничения преобразования.

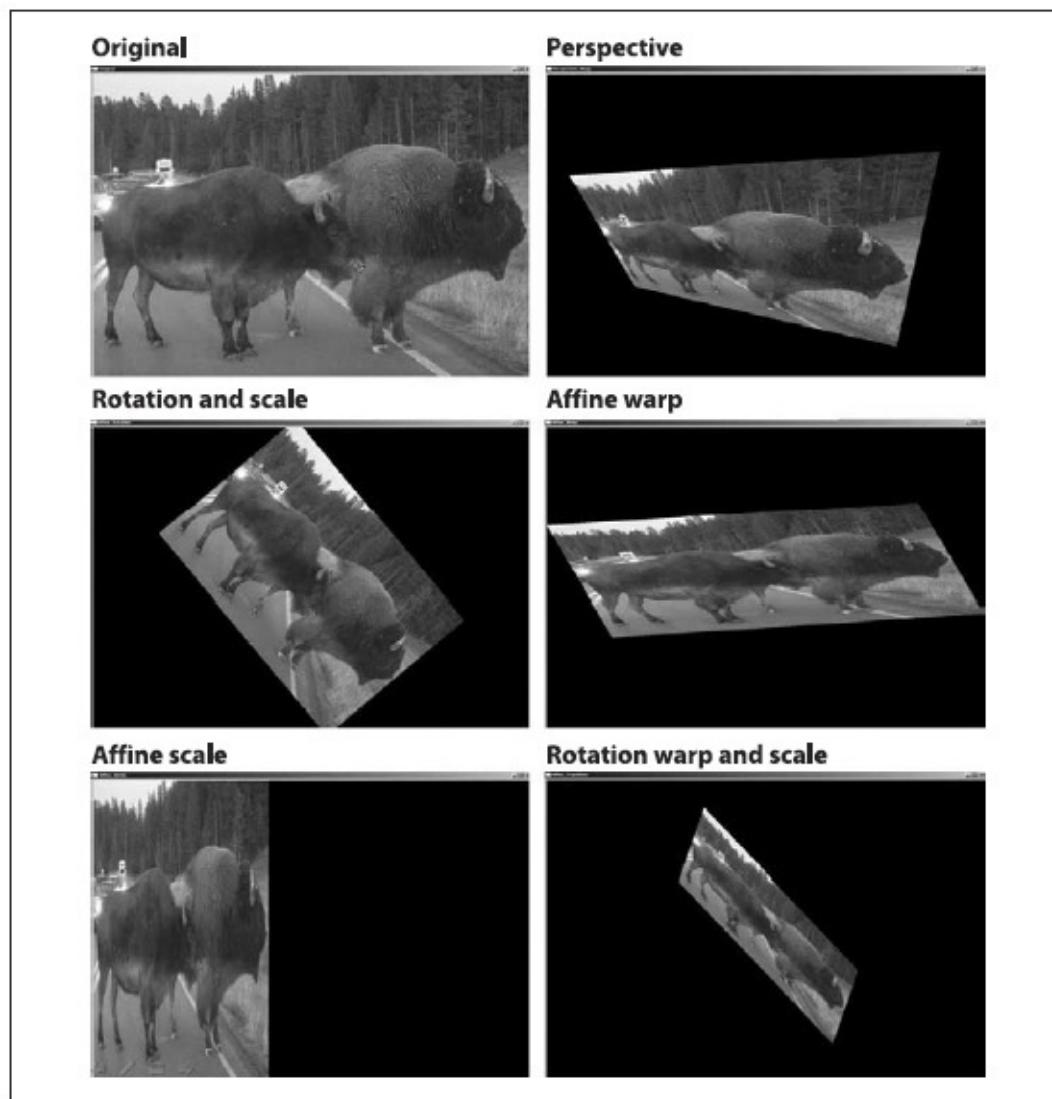


Рисунок 6-14. Пример выполнения аффинных и перспективных преобразований

[П]||[РС]||(РП) CartToPolar и PolarToCart

Функции `cvCartToPolar()` и `cvPolarToCart()`, как правило, используются более сложными функциями, такими как `cvLogPolar()`, но и сами по себе тоже весьма полезны. Эти функции преобразуют значения отображения между декартовым (x, y) и полярным (r, θ) пространствами (т.е. из декартовых в полярные координаты и наоборот).

```
void cvCartToPolar(
    const CvArr*    x
    ,const CvArr*    y
    ,CvArr*          magnitude
    ,CvArr*          angle = NULL
    ,int             angle_in_degrees = 0
);

void cvPolarToCart(
    const CvArr*    magnitude
    ,const CvArr*    angle
    ,CvArr*          x
    ,CvArr*          y
    ,int             angle_in_degrees = 0
);
```

В обоих функциях первые два аргумента это исходные двухмерные массивы или изображения, а следующие два аргумента это конечные. Если какой-либо из конечных указателей установлен в `NULL`, тогда он не будет участвовать в вычислениях. Эти массивы должны быть вещественного типа (`float` или `double`) и соотноситься друг с другом (по размеру, количеству каналов и типу). Последний параметр указывает с какими углами, в градусах (0, 360) или в радианах (0, 2π), осуществляется работа.

Для примера, чтобы осознать назначение этих функций, предположим, что уже получены производные от изображения по x и y , либо при помощи `cvSobel()`, либо при помощи функции свертки `cvDFT()` или `cvFilter2D()`. Если сохранить x -производные в изображение `dx_img`, а y -производные в изображение `dy_img`, тогда можно создать гистограмму распознавания углов краев. То есть, можно отобрать все углы согласно заданным значениям или силе (магнитуде) краев, которая больше некоторого порога. Для выполнения данных расчетов, необходимо наличие двух уже созданных изображений одного типа (целочисленного или вещественного) в качестве производных изображений и, например, назвать их `img_mag` и `img_angle`. Тогда для получения результата в градусах необходимо применить функцию `cvCartToPolar()`

dx_img, dy_img, img_mag, img_angle, 1). Заполнение гистограммы значениями *img_angle* происходит до тех пор, пока значение соответствующего "пикселя" *img_mag* выше установленного порога.

[П]||[РС]||(РП) LogPolar

Для двухмерных изображений логарифмически-полярное (далее лог-полярное) преобразование - перевод из декартовых в полярные координаты: $(x, y) \leftrightarrow re^{i\theta}$, где $r = \sqrt{x^2 + y^2}$ и $\exp(i\theta) = \exp(i \cdot \arctan(y/x))$. Для выделения полярных координат в пространство (ρ, θ) , которое соотносится с некоторой центральной точкой (x_c, y_c) , необходимо взять логарифм так, чтобы $\rho = \log(\sqrt{(x-x_c)^2 + (y-y_c)^2})$ и x_0, \dots, x_{N-1} . Применимельно к изображению - чтобы "уместить" интересные части изображения в памяти - необходимо применять масштабный коэффициент m к ρ . На рисунке 6-15 показан квадрат (слева) и его перекодирование в лог-полярное пространство.

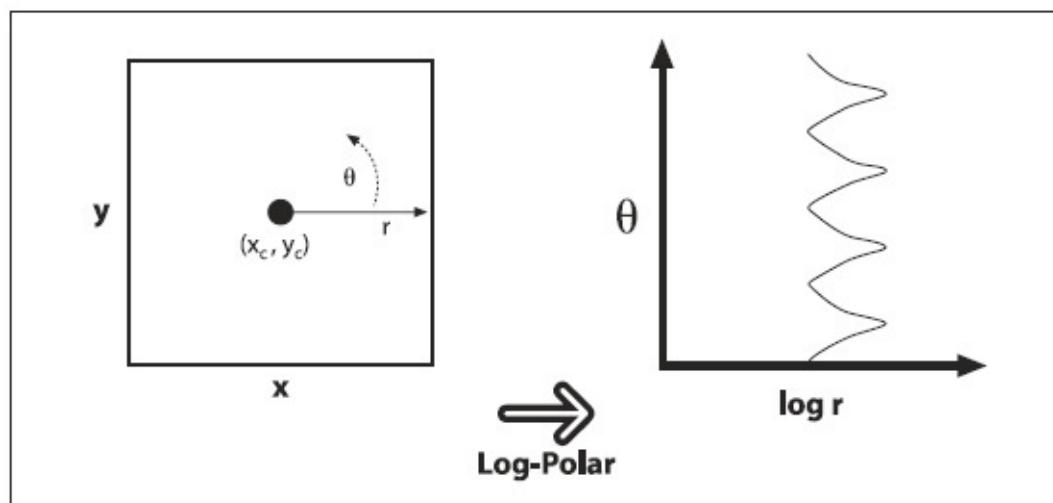


Рисунок 6-15. Лог-полярное отображение (x, y) в $(\log(r), \theta)$

Несомненно, возникает вопрос "Зачем все это делать?". Лог-полярное преобразование берет свои корни от зрительной системы человека. Глаз имеет небольшой, но плотно покрытый фоторецепторами центр (ямка), при этом при отдалении от него плотность фоторецептором резко сокращается (экспоненциально). Попробуйте смотреть на некоторую точку на стене и держать ваш палец на расстоянии вытянутой руки на линии поля вашего зрения. Затем, продолжайте смотреть на точку и медленно отводите оттуда палец, - обратите внимание, как быстро снижается детальность при отдалении изображения вашего пальца от ямки. Эта структура также имеет хорошие математические свойства (выходящие за рамки этой книги), согласно которым происходит сохранение углов пересечения линий.

Наиболее важным является то, что лог-полярное преобразование может быть использовано для создания двумерного инвариантного представления вида объекта путем сдвига центра масс трансформированного изображения в фиксированную точку на лог-полярной плоскости (рисунок 6-16). Слева имеется три формы, которые

необходимо распознать как "квадрат". Проблема в том, что они выглядят по-разному. Один намного больше остальных, а один повернут. Лог-полярное преобразование показано на рисунке 6-16 справа. Стоит обратить внимание на то, что отличие в размере на плоскости (x, y) преобразуется в сдвиг по оси $\log(r)$ лог-полярной плоскости, а отличия в повороте преобразуются в сдвиг по оси θ на лог-полярной плоскости.

Если взять преобразованный центр каждого преобразованного квадрата на лог-полярной плоскости, а затем перецентрировать в некоторое фиксированное положение, тогда все квадраты будут выглядеть одинаково на лог-полярной плоскости. Это приводит к инвариантности двумерного поворота и масштабирования (В главе 13 речь пойдет об распознавании. Сейчас достаточно будет знать, что получение лог-полярного преобразования для всего объекта не является хорошей идеей, т.к. подобные преобразования достаточно чувствительны к подбору их центральной точки. Более работоспособным вариантом является последовательное определение набора ключевых точек (таких как положение углов и впадин) вокруг объекта, отсечение лишнего по краю вокруг этих точек и использование центров ключевых точек в роли лог-полярных центров. Эти местные лог-полярные преобразования могут быть затем использованы для того, чтобы создать местные признаки, которые (частично) инвариантны (не зависимы) от вращения и масштабирования, и которые могут быть ассоциированы с видимым объектом).

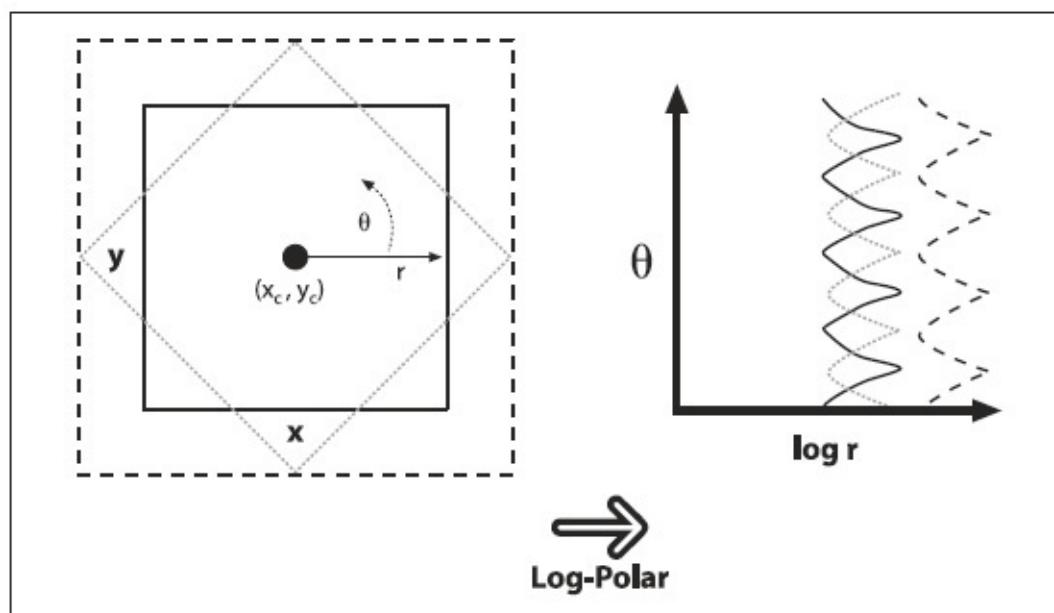


Рисунок 6-16. Лог-полярное преобразование повернутого и масштабированного квадрата: размер влияет на сдвиг по оси $\log(r)$, а поворот - на сдвиг по оси θ .

`cvLogPolar()` - функция OpenCV для лог-полярных преобразований:

```
void cvLogPolar(  
    const CvArr*   src  
    ,CvArr*        dst  
    ,CvPoint2D32f  center  
    ,double        m  
    ,int           flags = CV_INTER_LINEAR | CV_WARP_FILL_OUTLIERS  
)
```

src и *dst* - это одноканальные или трехканальные цветные или серые изображения.

Параметр *center* это центральная точка (x_c , y_c) лог-полярного преобразования; *m* коэффициент масштабирования, который должен быть выставлен таким образом, чтобы главные черты изображения доминировали в доступной области изображения. Параметр *flags* разрешает использование различных методов интерполяции. Можно использовать методы интерполяции из того же набора стандартных методов интерполяции, доступных в OpenCV (таблица 6-1). Методы интерполяции могут быть комбинированы с одним или обоими флагами *CV_WARP_FILL_OUTLIERS* (чтобы заполнить точки, которые иначе были бы неопределенными) или *CV_WARP_INVERSE_MAP* (чтобы вычислить обратное преобразование из лог-полярных в прямоугольные координаты).

Пример лог-полярного кодирования показан в примере 6-4, который демонстрирует прямое и обратное (инверсное) лог-полярное преобразование. Результаты обработки фотографии показаны на рисунке 6-17.

Пример 6-4. Пример лог-полярного преобразования

```
// logPolar.cpp : Определяет точку входа для консольного приложения
//
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv) {
    IplImage* src;
    double M;

    if( argc == 3 && (( src=cvLoadImage(argv[1],1) != 0 )) {
        M = atof(argv[2]);

        IplImage* dst      = cvCreateImage( cvGetSize(src), 8, 3 );
        IplImage* src2     = cvCreateImage( cvGetSize(src), 8, 3 );

        cvLogPolar(
            src
            ,dst
            ,cvPoint2D32f(src->width/4,src->height/2)
            ,M
            ,CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS
        );

        cvLogPolar(
            dst
            ,src2
            ,cvPoint2D32f(src->width/4, src->height/2)
            ,M
            ,CV_INTER_LINEAR | CV_WARP_INVERSE_MAP
        );

        cvNamedWindow( "log-polar", 1 );
        cvShowImage( "log-polar", dst );

        cvNamedWindow( "inverse log-polar", 1 );
        cvShowImage( "inverse log-polar", src2 );
        cvWaitKey();
    }

    return 0;
}
```



Рисунок 6-17. Пример лог-полярного преобразования над изображением лося с центром показанным белым кругом на левом изображении. Результат преобразования - справа

[П] | [РС] | (РП) Дискретное преобразование Фурье (ДПФ)

Для произвольного набора значений, которые проиндексированы дискретным (целочисленным) параметром, возможно определить дискретное преобразование Фурье (ДПФ), аналогичное по виду преобразованию Фурье для непрерывной функции (Джозеф Фурье был первым, кто обнаружил, что некоторые функции могут быть разложены в бесконечный ряд других функций - это привело к появлению раздела, названного анализ Фурье. Некоторые ключевые моменты, связанные с разложением функций в ряд Фурье, можно найти у Morse для физиков в частности и у Papoulis в общем. Быстрое преобразование Фурье было изобретено Cooley и Tukeye в 1965, хотя основные моменты самой идеи были предложены еще Карлом Гауссом в 1805. Первое упоминание об использовании разложения в ряд Фурье в компьютерном зренении можно найти у Ballard и Brown). Для N комплексных чисел x_0, \dots, x_{N-1} одномерное ДПФ определяется по следующей формуле (где $i = \sqrt{-1}$):

$$f_k = \sum_{n=0}^{N-1} x_n \exp\left(-\frac{2\pi i}{N} kn\right), \quad k=0, \dots, N-1$$

Подобное преобразование может быть определено для двумерного массива чисел (разумеется существуют аналоги и для массивов более высших порядков):

$$f_{k_x k_y} = \sum_{n_x=0}^{N_x-1} \sum_{n_y=0}^{N_y-1} x_{n_x n_y} \exp\left(-\frac{2\pi i}{N_x} k_x n_x\right) \exp\left(-\frac{2\pi i}{N_y} k_y n_y\right)$$

В общем, можно ожидать, что на вычисление N разных членов f_k потребуется $O(N^2)$ операций. На самом деле, существует несколько алгоритмов *быстрого преобразования Фурье* (БПФ), способных вычислять эти величины за время $O(N \log N)$. Функция OpenCV `cvDFT()` реализует один из таких алгоритмов БПФ. Функция `cvDFT()` вычисляет БПФ для одномерных и двумерных исходных массивов. Для последнего случая, может быть вычислено двумерное преобразование, либо, если указано, одномерное преобразование для каждого независимого ряда (эта операция намного быстрее, чем вызов `cvDFT()` для каждого отдельного случая).

```
void cvDFT(
    const CvArr*   src
    ,CvArr*        dst
    ,int           flags
    ,int           nonzero_rows = 0
);
```

Исходный и конечный массивы могут быть вещественного типа одно- или двухканальными. В случае одноканального (исходного) массива, элементами являются действительные значения, а выходные значения будут упакованы в специальный, компактный формат (унаследованный от старой библиотеки IPL, так же, как и структура *IplImage*). В случае двухканального (исходного) массива, два канала будут интерпретироваться как действительные и мнимые компоненты входных данных. В этом случае не будет никакого особенного упаковывания результатов; тем самым будет потеряно некоторое пространство, заполненное множеством нулей как в исходном, так и в конечном массивах (При использовании этого метода необходимо явно устанавливать мнимые компоненты в нули в двухканальном представлении). Простой способ сделать это - создать матрицу заполненную нулями при помощи *cvZero()* для мнимой части, а затем вызвать *cvMerge()* вместе с матрицей действительных чисел, чтобы сформировать временный массив комплексных чисел и выполнить *cvDFT()*. Эта процедура приведет к выводу полноразмерной, неупакованной, комплексной матрицы спектра).

Форматы специально упакованных выходных данных для случая с одноканальным выходным массивом.

Для одномерного массива:

$\text{Re } Y_0$	$\text{Re } Y_1$	$\text{Im } Y_1$	$\text{Re } Y_2$	$\text{Im } Y_2$...	$\text{Re } Y_{(N/2-1)}$	$\text{Im } Y_{(N/2-1)}$	$\text{Re } Y_{(N/2)}$
------------------	------------------	------------------	------------------	------------------	-----	--------------------------	--------------------------	------------------------

Для двумерного массива:

$\text{Re } Y_{00}$	$\text{Re } Y_{01}$	$\text{Im } Y_{01}$	$\text{Re } Y_{02}$	$\text{Im } Y_{02}$...	$\text{Re } Y_{0(Nx/2-1)}$	$\text{Im } Y_{0(Nx/2-1)}$	$\text{Re } Y_{0(Nx/2)}$
$\text{Re } Y_{10}$	$\text{Re } Y_{11}$	$\text{Im } Y_{11}$	$\text{Re } Y_{12}$	$\text{Im } Y_{12}$...	$\text{Re } Y_{1(Nx/2-1)}$	$\text{Im } Y_{1(Nx/2-1)}$	$\text{Re } Y_{1(Nx/2)}$
$\text{Re } Y_{20}$	$\text{Re } Y_{21}$	$\text{Im } Y_{21}$	$\text{Re } Y_{22}$	$\text{Im } Y_{22}$...	$\text{Re } Y_{2(Nx/2-1)}$	$\text{Im } Y_{2(Nx/2-1)}$	$\text{Re } Y_{2(Nx/2)}$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\text{Re } Y_{(Ny/2-1)0}$	$\text{Re } Y_{(Ny-3)1}$	$\text{Im } Y_{(Ny-3)1}$	$\text{Re } Y_{(Ny-3)2}$	$\text{Im } Y_{(Ny-3)2}$...	$\text{Re } Y_{(Ny-3)(Nx/2-1)}$	$\text{Im } Y_{(Ny-3)(Nx/2-1)}$	$\text{Re } Y_{(Ny-3)(Nx/2)}$
$\text{Im } Y_{(Ny/2-1)0}$	$\text{Re } Y_{(Ny-2)1}$	$\text{Im } Y_{(Ny-2)1}$	$\text{Re } Y_{(Ny-2)2}$	$\text{Im } Y_{(Ny-2)2}$...	$\text{Re } Y_{(Ny-2)(Nx/2-1)}$	$\text{Im } Y_{(Ny-2)(Nx/2-1)}$	$\text{Re } Y_{(Ny-2)(Nx/2)}$
$\text{Re } Y_{(Ny/2)0}$	$\text{Re } Y_{(Ny-1)1}$	$\text{Im } Y_{(Ny-1)1}$	$\text{Re } Y_{(Ny-1)2}$	$\text{Im } Y_{(Ny-1)2}$...	$\text{Re } Y_{(Ny-1)(Nx/2-1)}$	$\text{Im } Y_{(Ny-1)(Nx/2-1)}$	$\text{Re } Y_{(Ny-1)(Nx/2)}$

Стоит заострить внимание на индексах этих массивов. Проблема в том, что некоторые значения гарантированно нулевые (если более точно, то некоторые значения f_k гарантированно только действительные). При этом стоит отметить, что последний ряд таблицы будет присутствовать только если N_y нечетно, а последняя колонка будет присутствовать только если N_x нечетно. (В случае двумерного массива, рассматриваемого как одномерных массивов, вместо полноценного двумерного преобразования, все выходные строки будут аналогичны единственной строке одномерного массива).

Третий аргумент *flags* определяет тип операции. Прямое преобразование задаётся флагом *CV_DXT_FORWARD*. Обратное преобразование определяется схожим образом, за исключением смены знака у экспоненты и масштабного коэффициента (При обратном преобразовании, входные данные упаковываются в специальный формат. Это имеет смысл, т.к. если вызывать прямое ДПФ, а затем обратное ДПФ, то результатом должны быть исходные данные - это произойдёт только в том случае, если используется флаг *CV_DXT_SCALE*). Для выполнения обратного преобразования без масштабного коэффициента, используется флаг *CV_DXT_INVERSE*. Флаг *CV_DXT_SCALE* отвечает за масштабный коэффициент - это приводит к масштабированию результата на величину $1/N$ (или $1/(N_x N_y)$ для двумерного преобразования). Этот флаг используется, если в результате последовательности прямого и обратного преобразования над исходными данными необходимо получить исходные данные в том же виде. Т.к. довольно-таки часто приходиться использовать сочетание флагов *CV_DXT_INVERSE* и *CV_DXT_SCALE*, существует несколько сокращенных записей для операций такого рода. В дополнение к простому комбинированию этих флагов при помощи операции *OR*, можно воспользоваться флагом *CV_DXT_INV_SCALE* (или *CV_DXT_INVERSE_SCALE*, если короткие записи "не по вкусу"). И последний вариант флага, который может быть задействован, *CV_DXT_ROWS* указывает *cvDFT()* обрабатывать двумерный массив как набор одномерных, каждый из которых должен быть преобразован независимо, так будто имеется N_y независимых векторов длиною N_x . Это значительно снижает накладные расходы на выполнение множества одновременных преобразований (особенно при использовании библиотеки Intel IPP). Использование флага *CV_DXT_ROWS* позволяет осуществить трехмерное (или выше) ДПФ.

Прежде, чем осознать назначение последнего аргумента *nonzero_rows*, необходимо немного отвлечься. В общем, алгоритмы ДПФ сильно зависят от длин векторов или размерностей массивов. В большинстве алгоритмах ДПФ, предпочтаемые размеры должны быть степени двойки (т.е. 2^n для целого n). В случае алгоритма, используемого в OpenCV, предпочтение отдается векторам длиною, или массивам размерностью, равными $2^p 3^q 5^r$ для некоторых целых p , q и r . Поэтому, как правило, создается несколько больший массив (для этих целей существует удобная функция *cvGetOptimalDFTSize()*, которая принимает длину вектора, а возвращает первый приемлемый размер, больший или равный длине вектора) с последующим использованием *cvGetSubRect()* для копирования исходного массива в более вместительный массив с нулями в конце. Помимо необходимости занулять окончание, *cvDFT()* также можно указать, что эти строки не нужно преобразовывать, а просто добавлять после реальных данных (или, если выполняется обратное преобразование,

то указать, что эти строки не нужны). В любом случае, параметр *nonzero_rows* указывает какое количество строк нужно проигнорировать. Это немного экономит вычислительное время.

Перемножение спектров

В большинстве приложений, которые используют ДПФ, также требуется вычисление поэлементного произведения двух спектров. Поскольку результат обычно упаковывается в специальный уплотненный формат и, как правило, значения - это комплексные числа, было бы утомительно распаковывать их и выполнять произведение при помощи "обычных" матричных операций. К счастью, OpenCV предоставляет удобную функцию *cvMulSpectrums()* для выполнения этих и ещё нескольких других полезных операций.

```
void cvMulSpectrums(
    const CvArr*   src1
    ,const CvArr*   src2
    ,CvArr*         dst
    ,int            flags
);
```

Первые два параметра - это обычные массивы, хотя на самом деле это спектры, полученные в результате вызова *cvDFT()*. Третьим параметром должен быть указатель на массив - того же типа и размера, что и исходные два массива - в него будет помещен результат. Последний аргумент *flags* указывает *cvMulSpectrums()*, что конкретно необходимо выполнить. В частности, он может быть установлен в 0 (*CV_DXT_FORWARD*) для выполнения перемножения исходных массивов или в *CV_DXT_MUL_CONJ* для выполнения перемножения элемента первого массива на соответствующее комплексное сопряженное значение второго массива. Флаги допускается комбинировать с *CV_DXT_ROWS* для двумерного случая, при условии, что каждый ряд массива должен трактоваться как отдельный спектр (помните, при создании массивов спектров с флагом *CV_DXT_ROWS* упаковка данных несколько отличается от той, что была бы, если бы этот флаг не использовался, так что следует быть последовательными в методе вызова *cvMulSpectrums()*).

Свертка и ДПФ

Существует возможность существенно увеличить скорость свертки, используя ДПФ по теореме свертки, которая соотносит свертку в пространственном пространстве с произведением в пространстве Фурье (Алгоритм ДПФ в OpenCV использует БПФ в тех случаях, когда размеры данных позволяют это сделать). Для выполнения этого

сначала потребуется выполнить преобразование Фурье над изображением, а затем преобразование Фурье над сверточным фильтром. После этого свертка может быть выполнена в пространстве преобразования за линейное время по отношению к количеству пикселей на изображении. Пример 6-5, взятый из справки по OpenCV, отображает вычисление подобной свертки.

Пример 6-5. Использование *cvDFT()* для ускоренного вычисления свертки

```
// Использование ДПФ для ускоренного вычисления свертки массива A ядром B.
// Результат размещается в массиве V.
//

void speedy_convolution(
    const CvMat*     A    // Размер: M1xN1
    ,const CvMat*   B    // Размер: M2xN2
    ,CvMat*         C    // Размер: (A->rows+B->rows-1)x(A->cols+B->cols-1)
) {
    int dft_M = cvGetOptimalDFTSize( A->rows+B->rows-1 );
    int dft_N = cvGetOptimalDFTSize( A->cols+B->cols-1 );

    CvMat* dft_A = cvCreateMat( dft_M, dft_N, A->type );
    CvMat* dft_B = cvCreateMat( dft_M, dft_N, B->type );
    CvMat tmp;

    // Копирование A в dft_A и дополнение dft_A нулями в конце
    //
    cvGetSubRect( dft_A, &tmp, cvRect(0,0,A->cols,A->rows));
    cvCopy( A, &tmp );
    cvGetSubRect(
        dft_A
        ,&tmp
        ,cvRect( A->cols, 0, dft_A->cols-A->cols, A->rows )
    );
    cvZero( &tmp );

    // Нет необходимости дополнять нижнюю часть dft_A нулями, т.к.
    // используется параметр nonzero_rows в вызове cvDFT()
    //
    cvDFT( dft_A, dft_A, CV_DXT_FORWARD, A->rows );

    // Выполнение тех же операций над вторым массивом
    //
    cvGetSubRect( dft_B, &tmp, cvRect(0, 0, B->cols, B->rows) );
    cvCopy( B, &tmp );
    cvGetSubRect(
        dft_B
        ,&tmp
        ,cvRect( B->cols, 0, dft_B->cols-B->cols, B->rows )
    );
    cvZero( &tmp );

    // Нет необходимости дополнять нижнюю часть dft_B нулями, т.к.
}
```

```

// используется параметр nonzero_rows в вызове cvDFT()
//
cvDFT( dft_B, dft_B, CV_DXT_FORWARD, B->rows );

// Или CV_DXT_MUL_CONJ, чтобы получить корреляцию вместо свертки
//
cvMulSpectrums( dft_A, dft_B, dft_A, 0 );

// Вычисление только верхней части
//
cvDFT( dft_A, dft_A, CV_DXT_INV_SCALE, C->rows );
cvGetSubRect( dft_A, &tmp, cvRect(0, 0, conv->cols, C->rows) );

cvCopy( &tmp, C );

cvReleaseMat( dft_A );
cvReleaseMat( dft_B );
}

```

В примере 6-5 можно увидеть, что исходные массивы сначала создаются, а затем инициализируются. После создаются два новых массива, размерность которых оптимальна для алгоритма ДПФ. Исходные массивы копируются в эти новые массивы, а затем выполняется преобразование. В заключении, спектры перемножаются и к перемножению применяется обратное преобразование. Преобразование - наиболее медленная часть (под "наиболее медленным" подразумевается "асимптотически наиболее медленный" - иными словами, эта часть алгоритма требует больше времени для очень больших N). Это важное отличие. На практике, как было показано в предыдущей секции о свертке, не всегда оптимально расходовать время при преобразовании в пространство Фурье. В общем, выполняя свертку с малым ядром, не стоит производить подобное преобразование всей операции; изображение $N \times N$ требует времени $O(N^2 \log N)$, что соответствует всему времени вычислений (при условии, что $N > M$ для сверточного ядра размером $M \times M$). Это намного быстрее, чем $O(N^2 M^2)$ затрачиваемое на более простой не-ДПФ метод.

[П]|[РС]|(РП) Дискретно косинусное преобразование

Зачастую для вычислений с участием вещественных данных достаточно только половины дискретного преобразования Фурье. Дискретно косинусное преобразование (ДКП) определяется аналогично полному ДПФ по следующей формуле:

$$c_k = \sum_{n=0}^{N-1} \sqrt{n} = \begin{cases} \frac{1}{N} & \text{if } n=0 \\ \frac{2}{N} & \text{else} \end{cases} \cdot x_n \cdot \cos\left(-\pi \frac{(2k+1)n}{N}\right)$$

Стоит отметить тот факт, что по соглашению, коэффициент нормализации применяется как к косинусному, так и к обратному преобразованию. И конечно, есть подобное преобразование и для более высших размерностей.

Основная идея ДПФ применима и к ДКП, только все коэффициенты вещественного типа. Внимательный читатель может возразить, что косинусное преобразование применяется к вектору, который явно не является функцией. Однако, алгоритм `cvDCT()` обрабатывает вектор так, как если бы он был продлен до отрицательных показателей зеркально.

```
void cvDCT(
    const CvArr*   src
    ,CvArr*        dst
    ,int           flags
);
```

Функция `cvDCT()` ожидает аргументы, такие же как и для `cvDFT()` за исключением того, что поскольку результат состоит из действительных чисел, то нет необходимости в специальной упаковке результирующего массива (или исходного массива в случае обратного преобразования). Аргумент `flags` может быть установлен в `CV_DXT_FORWARD` или `CV_DXT_INVERSE`, которые могут быть скомбинированы с `CV_DXT_ROWS` с тем же эффектом, как у `cvDFT()`. Из-за различных соглашений по нормализации, прямое и обратное косинусное преобразование всегда содержит соответствующие вклады в общую нормализацию преобразования; следовательно `CV_DXT_SCALE` не играет никакой роли в `cvDCT()`.

[П] | [РС] | (РП) Интегральное изображение

OpenCV позволяет легко рассчитывать интегральное изображение с помощью функции `cvIntegral()`. *Интегральное изображение* - это структура данных, которая позволяет быстро суммировать области. Такое суммирование полезно во многих приложениях; одним из значимых является вычисление *вейвлетов Хаара*, которые используются при распознавании лиц и подобных алгоритмах.

```
void cvIntegral(
    const CvArr*    image
    ,CvArr*         sum
    ,CvArr*         sqsum = NULL
    ,CvArr*         tilted_sum = NULL
);
```

Аргументами `cvIntegral()` являются оригинальное изображение, а так же указатели на результаты. Аргумент `sum` обязательен; остальные, `sqsum` и `tilted_sum`, могут быть использованы по необходимости. (На самом деле, аргументы могут быть не изображениями, они могут быть и матрицами, хотя на практике, как правило, это изображения). Если исходное изображение 8-битное беззнаковое, то `sum` или `tilted_sum` могут быть 32-битными целыми или вещественными массивами. Для всех остальных случаев, `sum` или `tilted_sum` должны быть вещественными (32 или 64 битными). Результат всегда должен быть вещественного типа. Если исходное изображение имеет размер $W \times H$, то конечное изображение должно иметь размер $(W + 1) \times (H + 1)$ (потому что необходимо поместить в буфер нулевые значения по осям x и y для более эффективного процесса вычисления).

Интегральное изображение `sum` имеет форму:

$$\text{sum}(X, Y) = \sum_{x \leq X} \sum_{y \leq Y} \text{image}(x, y)$$

Необязательное изображение `sqsum` представляет собой сумму квадратов:

$$\text{sum}(X, Y) = \sum_{x \leq X} \sum_{y \leq Y} (\text{image}(x, y))^2$$

`tilted_sum` схоже с `sum` за исключением того, что оно повернуто на 45 градусов:

$$\text{tilt_sum}(X, Y) = \sum_{y \leq Y} \sum_{\text{abs}(x-X) \leq y} \text{image}(x, y)$$

Используя эти интегральные изображения можно вычислить суммы, среднее значение и стандартное отклонение по произвольной вертикале или "наклонной" прямоугольной области изображения. В качестве простого примера просуммируем простую прямоугольную область, заключенную в угловых точках (x_1, y_1) и (x_2, y_2) , где $x_2 > x_1$ и $y_2 > y_1$; вычисления производятся по следующей формуле:

$$\sum_{x_1 \leq x \leq x_2} \sum_{y_1 \leq y \leq y_2} [image(x, y)] = [\text{sum}(x_2, y_2) - \text{sum}(x_1-1, y_2) - \text{sum}(x_2, y_1-1) + \text{sum}(x_1-1, y_1-1)]$$

Таким образом можно выполнить быстрое размытие и аппроксимацию градиента, вычислить среднее значение и стандартное отклонение, а так же выполнить быструю настройку переменных, отвечающих за размеры окна. Что бы стало более понятнее, рассмотрим изображение 7×5 , показанное на изображение 6-18; взятый регион показан в виде гистограммы, в которой высота столбиков отображает значения яркости пикселей. Эта же информация показана на рисунке 6-19, численно слева и интегрально справа. Интегральное изображение (I') вычисляется путем прохода по строкам, строка за строкой, с использованием ранее вычисленных интегральных значений вместе с текущей не обработанной строкой (I) значений $I(x, y)$ для вычисления следующего целого значения изображения:

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

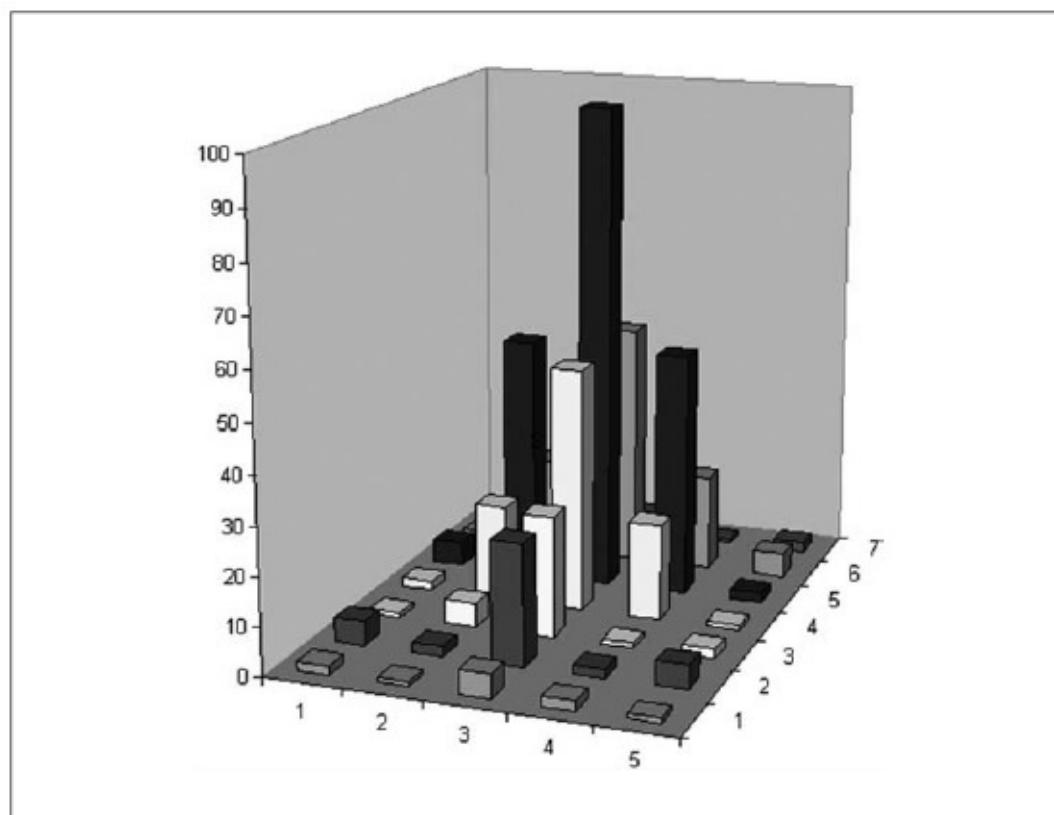


Рисунок 6-18. Изображение 7×5 , показанное в виде гистограммы

Последнее слагаемое необходимо чтобы избежать двойного учета при добавлении второго и третьего слагаемого. Это можно проверить, протестирував некоторые значения на рисунке 6-19.

При использовании интегрального изображения для вычисления региона, как показано на рисунке 6-19, и для вычисления центральной прямоугольной области, ограниченной 20-ю, необходимо $398 - 9 - 10 + 1 = 380$. Таким образом, прямоугольник любого размера может быть вычислен с использованием четырех измерений (в результате чего вычислительная сложность равна $O(1)$).

1	2	5	1	2
2	20	50	20	5
5	50	100	50	2
2	20	50	20	1
1	5	25	1	2
5	2	25	2	5
2	1	5	2	1

1	3	8	9	11
3	25	80	101	108
8	80	235	306	315
10	102	307	398	408
11	108	338	430	442
16	115	370	464	481
18	118	378	474	492

Рисунок 6-19. Изображение 7x5 численно показано слева и интегрально справа

[П]|[РС]|(РП) Дистанционные преобразования

Дистанционное преобразование для изображения определяется как новое изображение, в котором каждый пиксель установлен в значение, равное расстоянию до ближайшего нулевого пикселя в исходном изображении. Должно быть очевидным, что типичным входом дистанционного преобразования должен быть край изображения. В большинстве приложений входом дистанционного преобразования является выход от детектора краев, такого как Canny, который в свою очередь был ещё и инвертирован (так что края имеют нулевые значения, а не края ненулевые).

На практике, дистанционное преобразование осуществляется при помощи маски: матрицы 3x3 или 5x5. Каждая точка в массиве определяет "расстояние", связанное с точкой в конкретной позиции по отношению к центру маски. Увеличение расстояния строится (и таким образом аппроксимируется) в виде последовательности "ходов", определяемых в маске. Это означает, что использование большой маски дает более точные расстояния.

В зависимости от желаемых показателей расстояния, соответствующая маска автоматически выбирается из набора, известного OpenCV. Помимо этого, можно запросить вычислить "точные" расстояния по некоторой формуле, соответствующей выбранным показателям, однако это заметно замедляет процесс вычислений.

Дистанционный показатель может быть любым из доступных типов, включая классический L2 (декартовый) дистанционный показатель; полный список в таблице 6-2. В добавок к этому можно определить пользовательские показатели и связать с ними собственную маску.

Таблица 6-2. Возможные значения для аргумента *distance_type* *cvDistTransform()*

Значение <code>distance_type</code>	Метрика
<code>CV_DIST_L2</code>	$\rho(r) = \frac{r^2}{2}$
<code>CV_DIST_L1</code>	$\rho(r) = r$
<code>CV_DIST_L12</code>	$\rho(r) = 2\left[\sqrt{1 + \frac{r^2}{2}} - 1\right]$
<code>CV_DIST_FAIR</code>	$\rho(r) = C^2 \left[\frac{r}{C} - \log\left(1 + \frac{r}{C}\right) \right], C = 1.3998$
<code>CV_DIST_WELSCH</code>	$\rho(r) = \frac{C^2}{2} \left[1 - \exp\left(-\left(\frac{r}{C}\right)^2\right) \right], C = 2.9846$
<code>CV_DIST_USER</code>	Пользовательская метрика

При вызове функции дистанционного преобразования, конечное изображение должно быть 32-битным вещественным (т.е. `IPL_DEPTH_32F`).

```
void cvDistTransform(
    const CvArr* src
    ,CvArr* dst
    ,int distance_type = CV_DIST_L2
    ,int mask_size = 3
    ,const float* kernel = NULL
    ,CvArr* labels = NULL
);
```

При вызове `cvDistTransform()` можно использовать несколько дополнительных параметров. Первый, `distance_type`, указывает на тип дистанционной метрики. Возможные значения для этого аргумента были определены в Borgefors (1986).

Далее аргумент `mask_size`, который может быть 3 (`CV_DIST_MASK_3`) или 5 (`CV_DIST_MASK_5`); в альтернативе, расчеты можно произвести и без ядра (`CV_DIST_MASK_PRECISE`). Аргумент `kernel` - это расстояние маски, которое будет использовано в случае пользовательской метрики. Эти ядра строятся по методу Gunilla Borgefors, и два примера таких ядер показаны на рисунке 6-20. Последний аргумент `labels` указывает, что ассоциации должны быть выполнены между отдельными точками и ближайшими связными компонентами, состоящих из нулевых точек. Если `label !=`

NULL, то это указатель на массив целых значений того же размера, что и исходное и конечное изображения. Когда функция возвращает результат, по этому изображению можно определить какой из объектов был ближе всего к конкретной точке на стадии рассмотрения. На рисунке 6-21 показано конечное дистанционное преобразование на тестовом шаблоне и фотографии.

User defined 3x3 mask (a=1, b=1.5)								User defined 5x5 mask (a=1, b=1.5, c=2)							
4.5	4	3.5	3	3.5	4	4.5		4.5	3.5	3	3	3	3.5	4.5	
4	3	2.5	2	2.5	3	4		3.5	3	2	2	2	3	3.5	
3.5	2.5	1.5	1	1.5	2.5	3.5		3	2	1.5	1	1.5	2	3	
3	2	1	0	1	2	3		3	2	1	0	1	2	3	
3.5	2.5	1.5	1	1.5	2.5	3.5		3	2	1.5	1	1.5	2	3	
4	3	2.5	2	2.5	3	4		3.5	3	2	2	2	3	3.5	
4.5	4	3.5	3	3.5	4	4.5		4	3.5	3	3	3	3.5	4	

Рисунок 6-20. Две маски пользовательского дистанционного преобразования

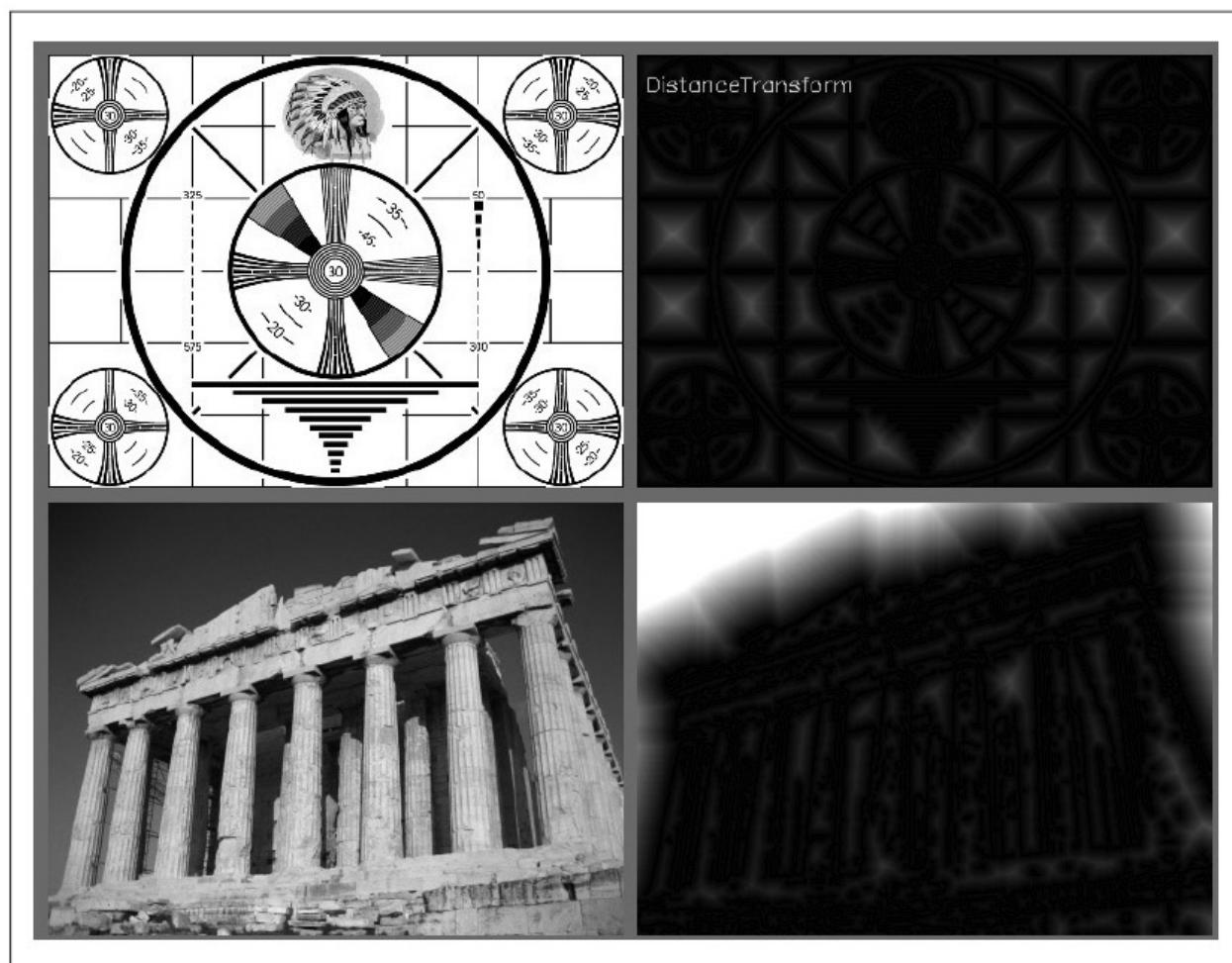


Рисунок 6-21. Используется детектор границ Кенни с параметрами $\text{param1} = 100$ и $\text{param2} = 200$; поэтому для повышения видимости дистанционное преобразование производится с масштабным коэффициентом 5

[П] | [РС] | (РП) Коррекция гистограмм

Камеры и детекторы изображений должны, как правило, заниматься обработкой не только контраста, но и света от датчиков изображения. В стандартной камере затвор и диафрагма объектива попеременно подают на датчики либо слишком много, либо слишком мало света. Часто диапазон контрастов слишком велик для датчиков; следовательно, существует компромисс между сбором данных темных областей (например, теней), которые требуют увеличения времени экспозиции, и светлых областей, которые требуют малого времени экспозиции, чтобы избежать "пересвета".

После того, как снимок сделан, уже не возможно изменить то, что записывается датчиком; однако, все еще возможно попытаться расширить динамический диапазон изображения. Наиболее часто используемым методом для этого является коррекция гистограмм (Может возникнуть вопрос, почему данная тема не рассматривается в главе 7; причина в том, что коррекция гистограмм не использует явно какой-либо тип гистограмм данных. Даже с учетом того, что гистограмма зашита внутри, функция (с точки зрения разработчика) не запрашивает гистограмму в явном виде. Коррекция гистограмм - это старый математический метод; его применение в обработке изображений было описано в различных учебниках, на научных конференциях и даже в биологическом зрении). На рисунке 6-22 видно, что левое изображение бедно, т.к. диапазон изменений значений невелик. Это видно по гистограмме интенсивности справа. Т.к. представлено 8-битное изображение, то его значения интенсивности могут изменяться в диапазоне от 0 до 255, однако, гистограмма показывает, что фактические значения интенсивности сгруппированы вблизи середины доступного диапазона.

Коррекция гистограммы - это метод растяжки этого диапазона.

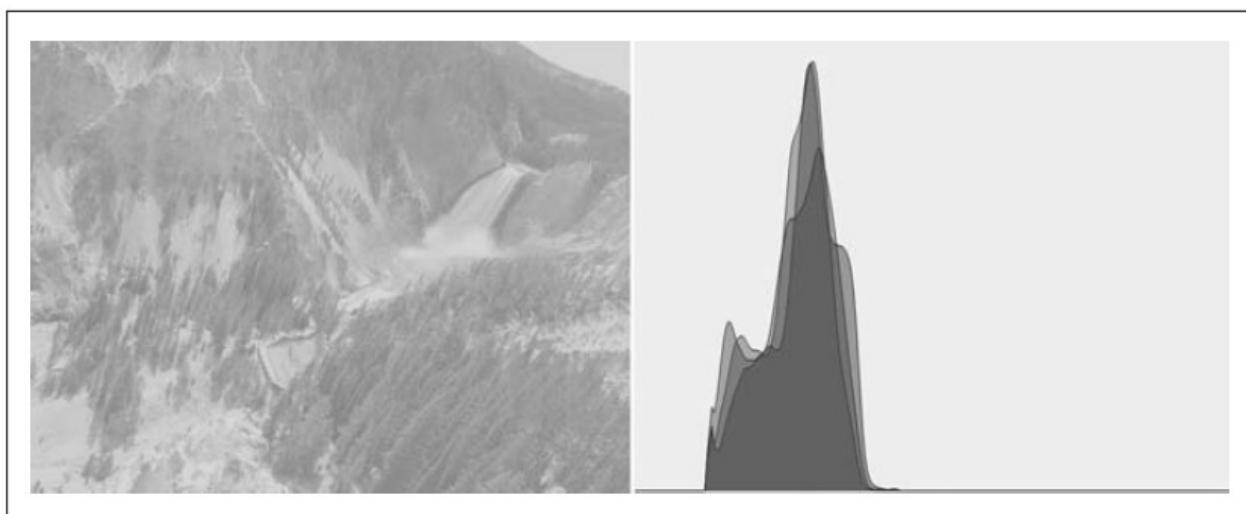


Рисунок 6-22. Изображение слева имеет плохую контрастность, что подтверждает гистограмма значений интенсивности справа

Лежащая в основе коррекции гистограмм математика включает в себя отображение одного распределения (учитывающее значения интенсивности гистограммы) на другое распределение (в ширину и, в идеале, равномерно распределяя значения интенсивности). Т.е. требуется раскидать у-значения первоначального распределения как можно равномернее в новом распределении. Оказывается, что есть хорошее решение проблемы распространения распределения значений: функция повторного отображения должна быть *интегральной функцией распределения*. Пример интегральной функции плотности показан на рисунке 6-23 для нескольких идеализированного случая распределения, которое изначально было чисто Гауссовым. Тем не менее, совокупная плотность может быть применена к любому распределению; это просто текущая сумма первоначального распределения от отрицательного до положительного предела.

Функцию распределения можно использовать для переназначения первоначального распределения в качестве равномерного распространения распределения (рисунок 6-24), просто просматривая каждое у-значение в первоначальном распределении и отслеживая направление равномерного распределения.

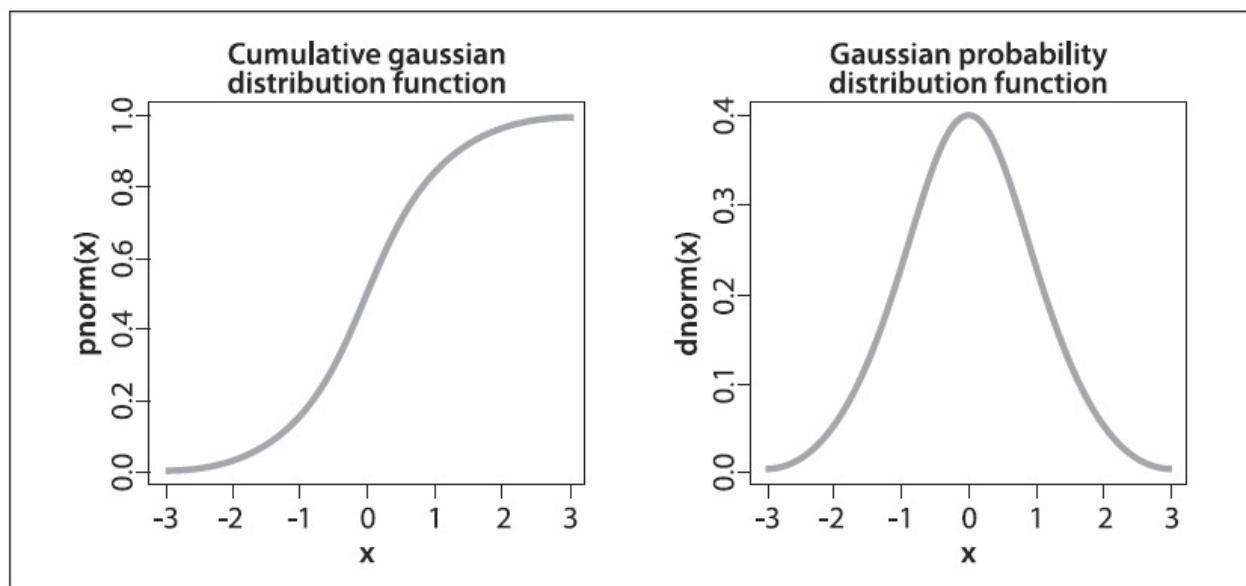


Рисунок 6-23. Интегральная функция распределения (слева) и Гауссово распределение (справа)

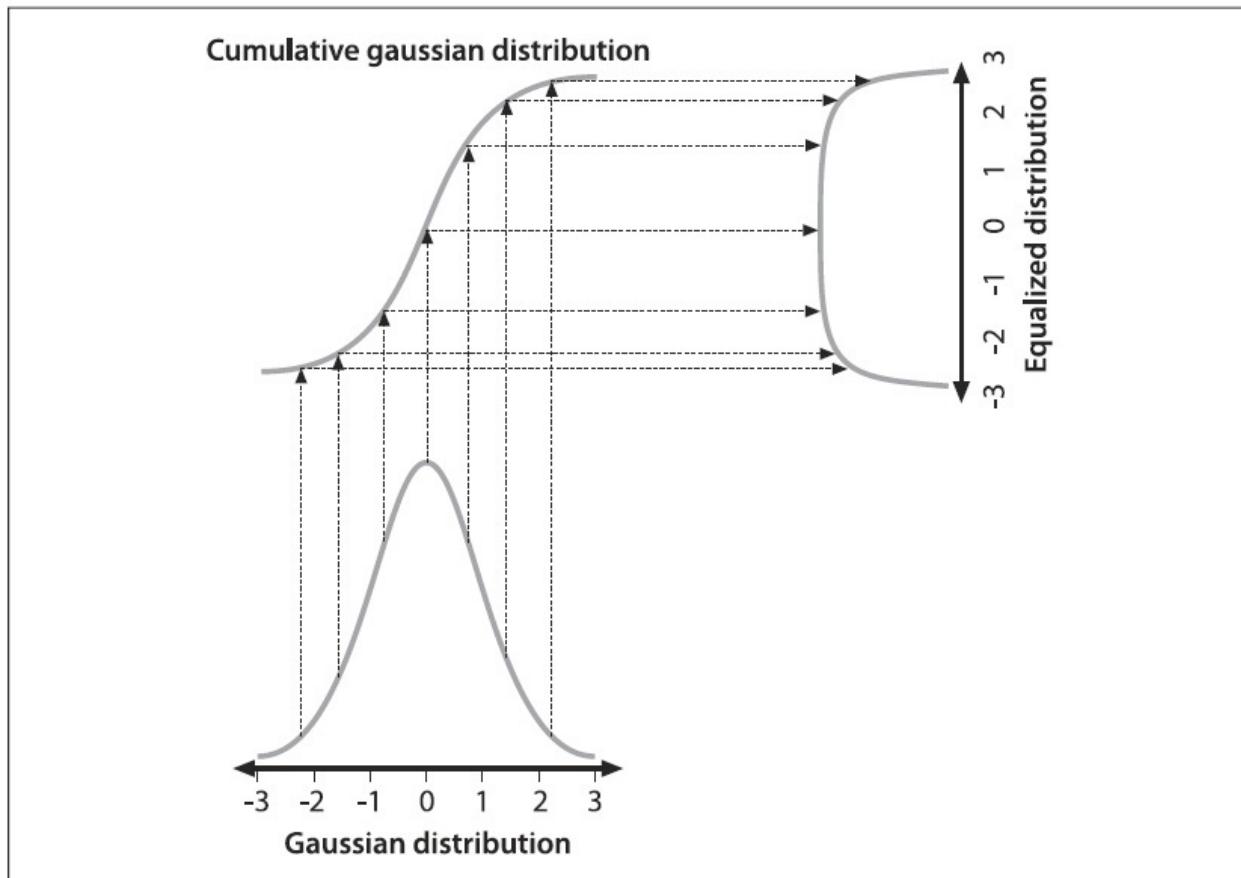


Рисунок 6-24. Использование интегральной функции плотности для выравнивания распределения Гаусса

Для непрерывного распределения результат будет точным, но для цифровых/дискретных распределений результаты могут быть далеко не однородными.

В результате применения процесса выравнивания к рисунку 6-22 происходит выравнивание гистограммы распределения интенсивности; результирующее изображение на рисунке 6-25. Весь этот процесс "зашит" в одной функции:

```
void cvEqualizeHist(
    const CvArr* src
    ,CvArr* dst
);
```

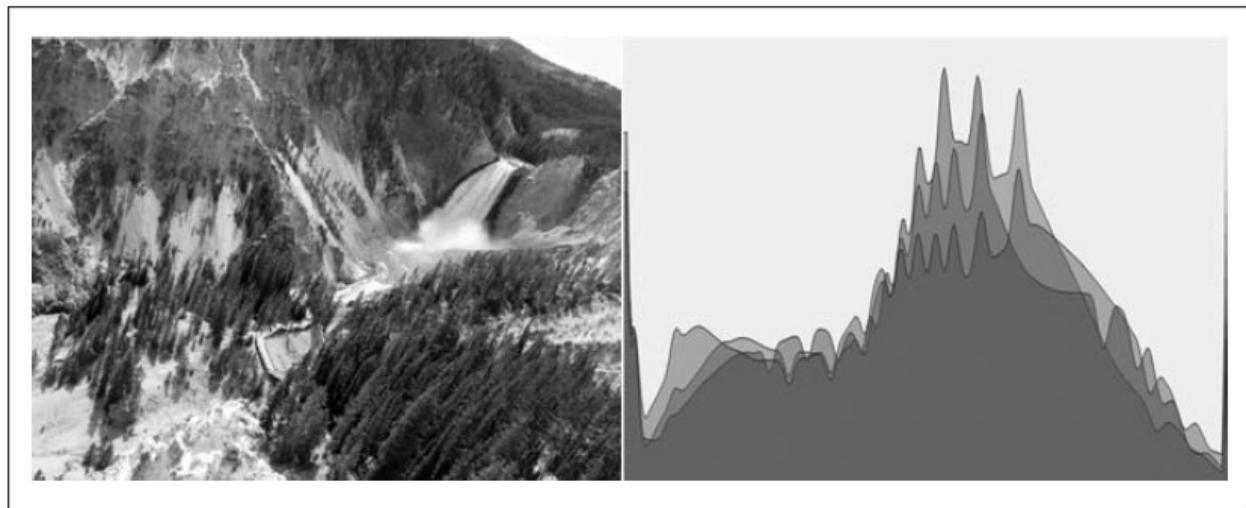


Рисунок 6-25. Результат выравнивания гистограммы

В `cvEqualizeHist()` исходное и конечное изображения должны быть одноканальными, 8-битными, одного и того же размера. В случае цветных изображений, каждый канал обрабатывается по отдельности.

[П]|[РС]|(РП) Упражнения

1. Используя `cvFilter2D()`, создайте фильтр, который определит только линии под 60 градусов к горизонту. Отобразите довольно-таки интересный результат.
2. Разделяемые ядра. Создайте ядро Гаусса 3x3, используя следующие значения по строкам [(1/16, 2/16, 1/16), (2/16, 4/16, 2/16), (1/16, 2/16, 1/16)], с якорем по середине.
 - a. Используйте это ядро на любом изображении и отобразите результат
 - b. Теперь создайте два одномерных ядра с якорем по центру: матрица строка (1/4, 2/4, 1/4) и матрица столбец (1/4, 2/4, 1/4). Загрузите любое изображение и примените к нему `cvFilter2D()`, чтобы выполнить свертку дважды, один раз с первым одномерным ядром и один раз со вторым одномерным ядром. Опишите результат.
 - c. Опишите порядок сложности (количество операций) для ядра из пункта a и ядра из пункта b. Разницу в преимуществах от использования разделяемых ядер и всего класса фильтров Гаусса - и любого линейно нестабильного фильтра, который разделим, а свертка является линейной операцией.
3. Можно ли сделать разъемное ядро из фильтра с рисунка 6-5? Если да, то покажите, как оно выглядит.
4. Нарисуйте серию концентрических окружностей в форме, напоминающей мишени в таких видах спорта, как стрельба из лука или дартс (например, в PowerPoint)
 - a. Нарисуйте серию линий, сходящихся в центре окружностей.
 - b. Воспользуйтесь апертурой размера 3x3 и получите производные первого порядка по осям x и y. Затем повторите с увеличенной апертурой 5x5, 9x9 и 13x13. Опишите результаты.
5. Создайте изображение, состоящие из чередующихся черных и белых линий, расположенных под углом 45 градусов к горизонту. Для получения нескольких размеров апертур, сначала получите производную первого порядка по оси x (`dx`), а затем производную первого порядка по оси y (`dy`). Затем снимите мерку для этих линий следующим образом: `dx` и `dy` будут представлять градиент для входного изображения, магнитуда точки (i,j) будет вычисляться по формуле
$$\text{mag}(i,j)=\sqrt{dx^2(i,j)+dy^2(i,j)}$$
, а угол по формуле $\theta(i,j) = \arctan(dy(i,j)\times dx(i,j))$.

Просканируйте изображение и найдите места где магнитуда равна или близка к максимуму. Зафиксируйте значение угла в этих местах. Вычислите средний угол и зафиксируйте его как угол для линий.

- a. Проделайте все это по новой с апертурой 3x3 фильтра Собеля.
 - b. Проделайте все это по новой с апертурой 5x5.
 - c. Проделайте все это по новой с апертурой 9x9.
 - d. Изменяется ли результат? Если да, то почему?
6. Найдите и загрузите фотографию лица, где лицо фронтально, глаза открыты и лицо занимает большую часть изображения. Напишите код, который найдет зрачки глаз.
- Лапласиане "нравиться" яркая центральная точка, окруженная темнотой.
Зрачок как раз-таки соответствует противоположному варианту.
Инвертируйте и найдите свертку для достаточно большого Лапласиана*
7. В этом упражнение мы научимся подбирать хорошие значения *lowThresh* и *highThresh* для *cvCanny()*. Загрузите любое изображение с довольно таки необычной структурой линий. В данном упражнении используйте три различных соотношения высокий:низкий порог: 1.5:1, 2.75:1 и 4:1.
 - a. Зафиксируйте результат с величиной высокого порога меньшего, чем 50.
 - b. Зафиксируйте результат с величиной высокого порога между 50 и 100.
 - c. Зафиксируйте результат с величиной высокого порога между 100 и 150.
 - d. Зафиксируйте результат с величиной высокого порога между 150 и 200.
 - e. Зафиксируйте результат с величиной высокого порога между 200 и 250.
 - f. Дайте оценку полученным результатам и объясните какой из указанных вариантом подходит лучше всего.
 8. Загрузите изображение, содержащее четкие линии и окружности (например, вид велосипеда сбоку). Воспользуйтесь линиями и окружностями Хафа и оцените получаемые результаты.
 9. Придумайте способ, если сможете, использования преобразования Хафа для выявления каких-либо фигур с ярко выраженными краями.
 10. Взгляните на диаграмму на которой функция лог-полярного преобразования превращает квадрат в волнистую линию.

- a. Какой результат будет получен в результате размещения центра функции лог-полярного преобразования в одном из углов квадрата?
 - b. Что бы круг после лог-полярного преобразования выглядел как круг, центральная точка должна быть внутри или вблизи края круга?
 - c. Что будет если разместить центральную точку за пределами круга?
11. Лог-полярное преобразование может обрабатывать фигуры различных размеров и разворотов в пространстве, со сдвигами по осям θ и $\log(r)$. Преобразование Фурье является инвариантным. Как можно использовать эти факты, чтобы автоматически получать из фигур различных размеров и разворотов эквивалентные представления в лог-полярном представлении?
12. Нарисуйте несколько изображений, состоящих из: больших квадратов, маленьких квадратов, из больших развернутых квадратов, из маленьких развернутых квадратов. Выполните лог-полярное преобразование для каждого изображения в отдельности. Добавьте бегунок к каждой картинке и "свяжите" его с центральной точкой. Сдвигайте центральные точки до тех пор, пока результаты не будут максимально идентичными.
13. Возьмите преобразование Фурье небольшого распределения Гаусса и преобразование Фурье изображения. Перемножьте их и возьмите обратное преобразование от результата перемножения. Что получилось?
14. Возьмите любое интересное изображение, конвертируйте его в черно-белое и получите интегральное изображение. Теперь найдите вертикальные и горизонтальные края при помощи свойств интегрального изображения.
- Используйте длинные узкие прямоугольники*
15. Объясните, как при помощи дистанционных преобразований можно выправить известную фигуру при помощи тестовой фигуры, при условии, что масштабный коэффициент известен и зафиксирован. Как это можно сделать при помощи нескольких изменений масштабного коэффициента?
 16. Попрактикуйтесь в корректировке гистограмм любых интересных вам изображений.
 17. Загрузите любое изображение, примените перспективное преобразование и разверните. Можно ли выполнить данное преобразование за один шаг?

[П]|[РС]|(РП) Гистограммы и сопоставления

Зачастую, в ходе анализа изображений, объектов и видеинформации, требуется представление в виде *гистограммы*. Гистограммы могут быть использованы для представления таких вещей, как цветовое распределение объекта, шаблона градиента краев объекта и распределение вероятностей, представляющие текущую гипотезу о местонахождении объекта. Рисунок 7-1 отображает использование гистограмм для быстрого распознавания жестов. Градиенты краев были собраны из жестов: "вверх", "вправо", "влево" и "OK". При помощи этих жестов возможно управлять видеопотоком с веб-камеры. В каждом кадре, цвет региона интересов определяется с поступающего видео; затем вычисляются значения краев градиента вокруг этих регионов и в завершении эти значения представляются в виде гистограмм. Гистограммы сопоставляются с моделями жестов и жест распознается. Вертикальные полосы на рисунке 7-1 отображают сопоставление уровней различных жестов. Серая горизонтальная линия представляет порог для принятия решения о соответствующем жесте.

Гистограммы находят применения во многих приложениях компьютерного зрения. Гистограммы могут использоваться для обнаружения переходов между сценами видеопотока, отмечая заметные изменения краев и цвета от кадра к кадру. Они используются для идентификации интересующих точек на изображении, назначив каждой точке "метку", состоящей из гистограмм близлежащих признаков. Гистограммы краев, цвета, углов и т.д. формируют общий тип признака, который в последующем передается классификаторам для распознавания объекта. Последовательности цвета или краев гистограмм используют для определения, например, было ли видео скачано с интернета. Гистограммы являются классическим инструментом компьютерного зрения.

Гистограммы - это просто коллекция *counts*, представляющая основу данных и организованная в набор заранее определенных *контейнеров*. Они могут быть представлены такими вычисляемыми признаками, как градиент магнитуды и направления, цвета или какими-либо другими характеристиками. В любом случае, они используются для получения статистического представления основного распределения данных. Гистограммы, как правило, по размерам меньше, чем исходные данные. На рисунке 7-2 показана типичная ситуация. Показано двумерное распределение точек (вверху слева); наложенная сетка (вверху справа) и количество точек в каждой ячейке сетки, в итоге получается одномерная гистограмма (справа)

внизу). Поскольку вместо точек могут быть любые другие данные, то гистограмма является удобным средством представления всего, что можно получить из изображения.

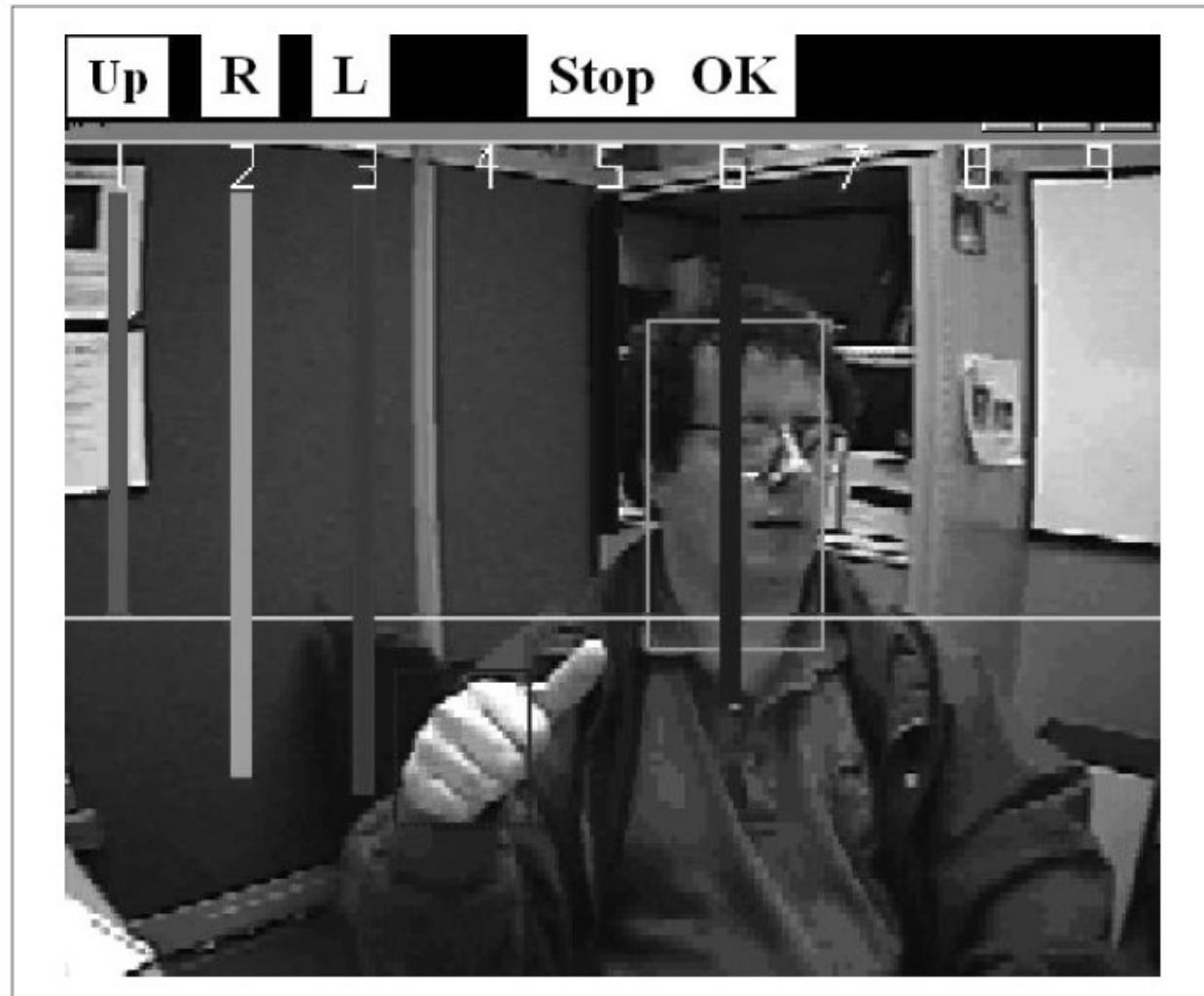


Рисунок 7-1. Гистограммы используются для распознавания жестов руки

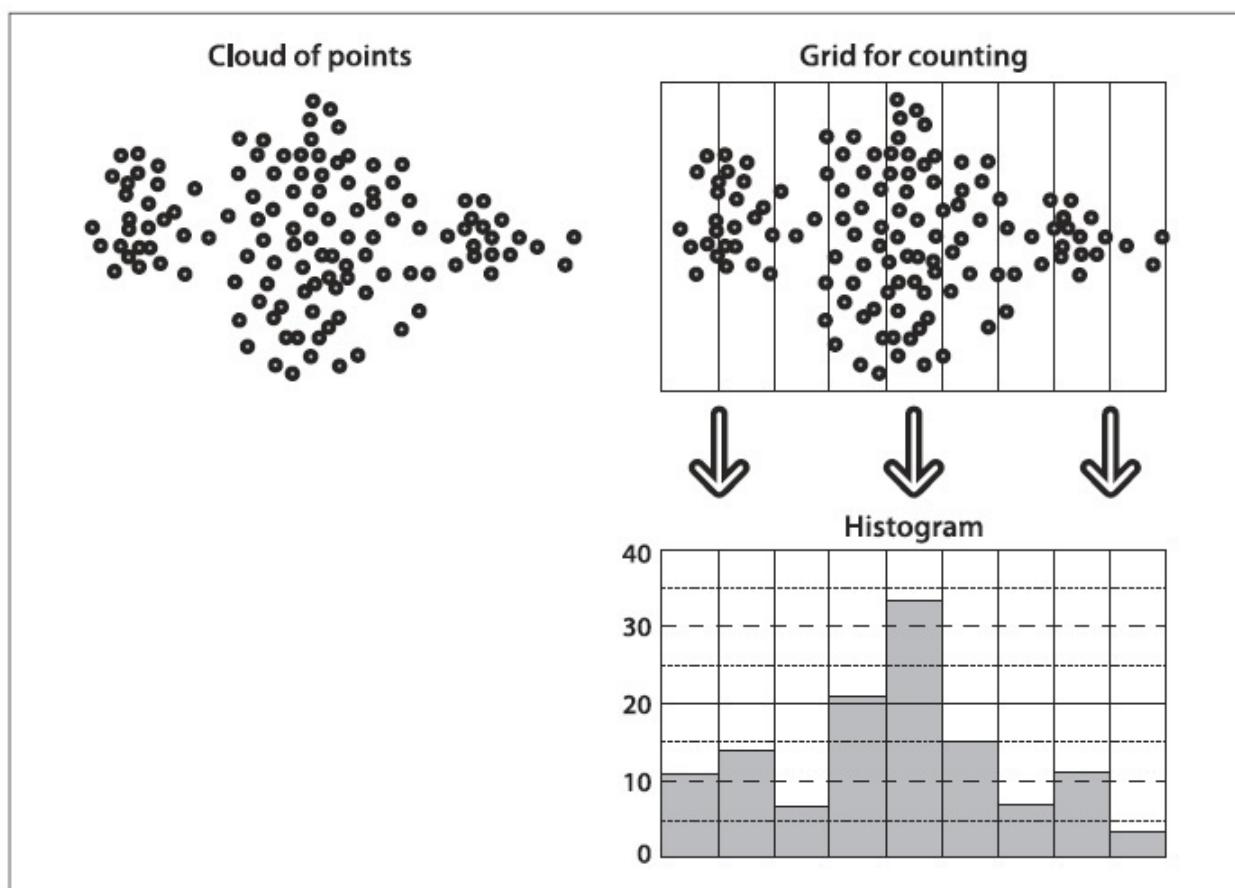


Рисунок 7-2. Типичный пример гистограмм

Гистограммы, которые представляют непрерывное распределение, строятся таким образом, чтобы неявно содержать среднее количество точек в каждой ячейке сетки. Это то место, где могут возникнуть проблемы (рисунок 7-3). Если сетка слишком широка (вверху слева), то получиться большое усреднение и структура распределения будет утеряна. Если сетка слишком узкая (вверху справа), то получиться не достаточное усреднение для представления точного распределения и в итоге получатся небольшие, "заостренные" ячейки.

В OpenCV есть тип данных для представления гистограмм. Данная структура способна представлять гистограммы в одном или нескольких измерениях и содержит все необходимые данные для отслеживания контейнеров однородного или неоднородного размера. Для всего этого имеется множество полезных функций, которые позволяют с легкостью выполнять типовые операции над гистограммами.

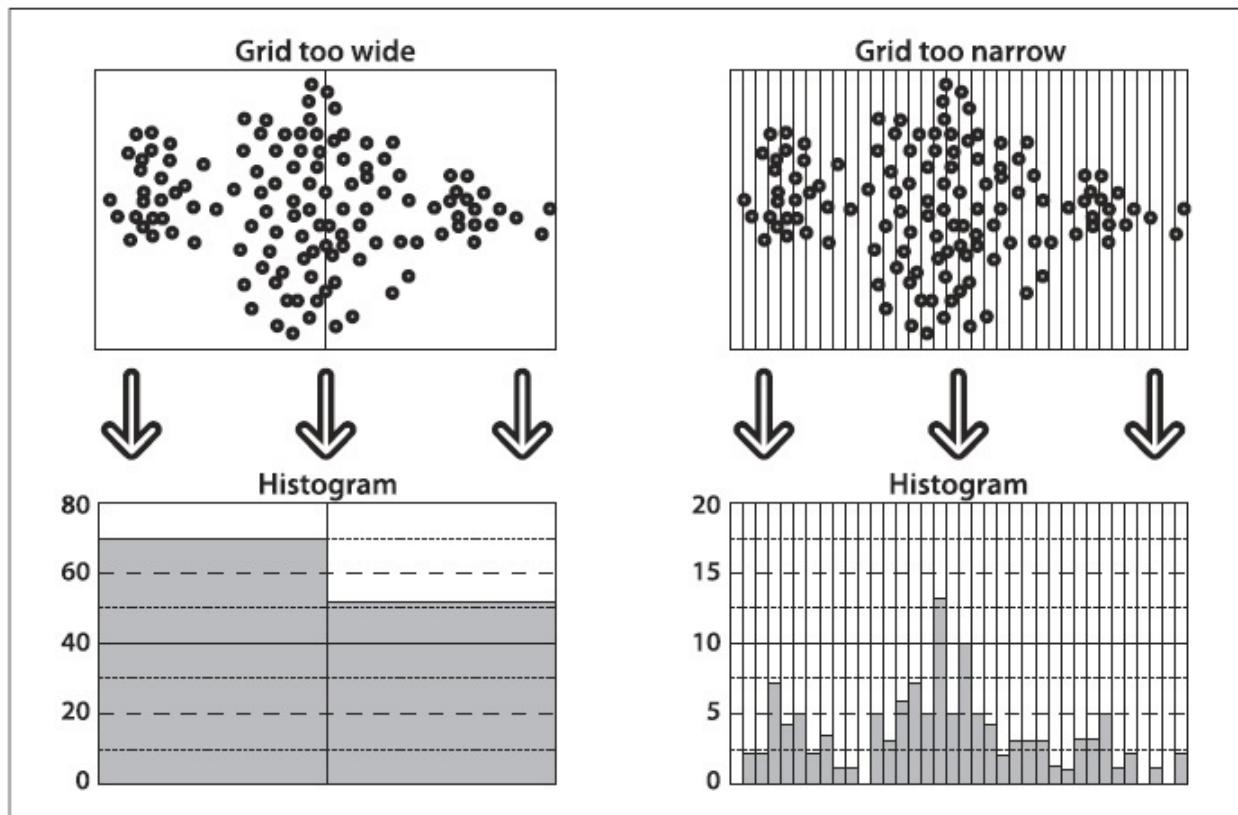


Рисунок 7-3. Точность гистограмм зависит от размера сетки: сетки, у которых широкие ячейки имеют слишком большое пространственное усреднение на гистограмме counts (слева); сетка со слишком маленькими "заостренными" ячейками и одиночным результатом имеют слишком малое пространственное усреднение (справа)

[П]|[РС]|(РП) Базовый тип Histogram

Для начала необходимо разобраться со структурой *CvHistogram*:

```
typedef struct CvHistogram {
    int      type;
    CvArr*  bins;
    float   thresh[CV_MAX_DIM][2]; // для однородных гистограмм
    float** thresh2;             // для неоднородных гистограмм
    CvMatND mat;                // встроенный заголовок матрицы
                                // для массива гистограмм
} CvHistogram;
```

Данное объявление обманчиво простое, потому что большая часть внутренних данных гистограммы храниться внутри структуры *CvMatND*. Создать гистограмму можно при помощи следующей функции:

```
CvHistogram* cvCreateHist(
    int      dims
, int*   sizes
, int     type
, float** ranges = NULL
, int     uniform = 1
);
```

Аргумент *dims* задает размерность гистограммы. Аргумент *sizes* должен быть массивом целых чисел размера *dims*. Каждое целое число в этом массиве задает какое количество контейнеров должно быть отнесено к соответствующей размерности. Аргумент *type* может быть либо *CV_HIST_ARRAY* для многомерных гистограмм, хранящиеся с использованием плотной многомерной матричной структуры (т.е. *CvMatND*), либо *CV_HIST_SPARSE* (от старых версий осталась поддержка *CV_HIST_TREE*, идентичная *CV_HIST_SPARSE*), если данные будут храниться с помощью разреженных матриц представления (*CvSparseMat*). Аргумент *ranges* может принимать одну из двух форм. Для однородных гистограммы, *ranges* - массив пар (эти "пары" всего лишь С-массивы с двумя записями) значений с плавающей точкой, где число пар равно размерности. Для неоднородных гистограмм, пары, использующиеся для однородных гистограмм, заменяются массивами, содержащие значения, разделенные неоднородными контейнерами. Если имеется N контейнеров, тогда в каждом из этих подмассивов будет N+1 записей. Каждый массив значений начинается от нижней границы первого контейнера и заканчивается верхней границей последнего (Для уточнения: в случае однородной гистограммы, если нижний и верхний диапазоны

установлены в 0 и 10 соответственно и если есть два контейнера, то контейнерам будут назначены соответствующие интервалы [0, 5) и [5, 10]. В случае неоднородной гистограммы, если размерность равна 4 и есть соответствующие диапазоны значений (0, 2, 4, 9, 10), то контейнерам будут назначены следующие (неоднородные) интервалы: [0, 2), [2,4), [4, 9) и [9, 10]). Аргумент *uniform* указывает на то, должны ли у гистограммы быть однородные контейнеры и таким образом указывать как будут интерпретироваться диапазоны значений; если установлено значение отличное от нуля, контейнеры будут однородными (при этом не стоит обращать внимание на то, что этот аргумент имеет тип int). Существует возможность устанавливать аргумент *ranges* в NULL, в этом случае диапазон просто будет "неизвестен" (можно будет установить потом при помощи *cvSetHistBinRanges()*). Должно быть очевидно, что значения диапазона необходимо задавать перед использованием гистограмм.

```
void cvSetHistBinRanges(
    CvHistogram* hist
    , float** ranges
    , int uniform = 1
);
```

Аргументы *cvSetHistRanges()* в точности совпадают с аргументами *cvCreateHist()*. После того, как гистограмма создана, её можно очистить (т.е. установить все контейнеры в 0):

```
void cvClearHist(
    CvHistogram* hist
);
```

Чтобы освободить память, которую занимает гистограмма можно воспользоваться функцией:

```
void cvReleaseHist(
    CvHistogram** hist
);
```

Как и всегда для вызова подобного рода функций передается двойной указатель на объект. После освобождения памяти, указатель на гистограмму устанавливается в NULL.

Есть еще одна полезная функция, которая позволяет создавать гистограммы из уже существующих данных:

```
CvHistogram* cvMakeHistHeaderForArray(
    int          dims
    ,int*        sizes
    ,CvHistogram* hist
    ,float*      data
    ,float**     ranges = NULL
    ,int         uniform = 1
);
```

В этом случае *hist* является указателем на структуру *CvHistogram*, а *data* указателем на область размером *sizes[0]×sizes[1]×...×sizes[dims-1]* для хранения контейнеров. *data* это указатель на float, т.к. внутреннее представление данных в гистограмме имеет тип float. Возвращаемое значение это одно из значений *hist*. В отличие от *cvCreateHist()*, данная функция не имеет аргумента *type*. Все гистограммы, созданные при помощи *cvMakeHistHeaderForArray()* являются плотными гистограммами. И в заключении: так, как, скорее всего, разработчик сам будет выделять пространство под *data* для хранения контейнеров, не будет происходить вызов функции *cvReleaseHist()* внутри структуры *CvHistogram*. И потому, разработчик должен сам позаботиться об очистке заголовка структуры (если она не была размещена в стеке) и данных.

[П]|[РС]|(РП) Доступ к Histogram

Есть несколько способов получить доступ к данным гистограмм. Наиболее простой: использовать функции OpenCV:

```
double cvQueryHistValue_1D(
    CvHistogram*    hist
    ,int            idx0
);

double cvQueryHistValue_2D(
    CvHistogram*    hist
    ,int            idx0
    ,int            idx1
);

double cvQueryHistValue_3D(
    CvHistogram*    hist
    ,int            idx0
    ,int            idx1
    ,int            idx2
);

double cvQueryHistValue_nD(
    CvHistogram*    hist
    ,int*           idxN
);
```

Каждая из этих функций возвращает вещественное значение для соответствующего контейнера. Кроме того, при помощи этих функций можно установить (или получить) значение контейнера за счет указателя на него (а не самого объекта):

```

float* cvGetHistValue_1D(
    CvHistogram* hist
    , int         idx0
);

float* cvGetHistValue_2D(
    CvHistogram* hist
    , int         idx0
    , int         idx1
);

float* cvGetHistValue_3D(
    CvHistogram* hist
    , int         idx0
    , int         idx1
    , int         idx2
);

float* cvGetHistValue_nD(
    CvHistogram* hist
    , int*        idxN
);

```

Эти функции похожи на семейство функций **cvGetReal*D** и **cvPtr*D** и фактически они делают почти тоже самое. По сути эти функции работают с теми же матрицами, которые они получают при помощи *hist->bins*. Кроме того, функции, работающие с разрежёнными гистограммами, наследуют поведение соответствующих функций для работы с разрежёнными матрицами. При попытке обращения к несуществующему контейнеру при помощи функции **GetHist()** в разреженной гистограмме, контейнер будет создан автоматически и его значение будет установлено в 0. В свою очередь, функции **QueryHist()** не создают недостающих контейнеров.

Во многих случаях, при работе с *плотными гистограммами* необходимо получать прямой доступа к контейнерам. Конечно, можно было бы сделать это при помощи прямого доступа к данным гистограмм. Например, можно получить прямой доступ ко всем элементам плотной гистограммы последовательно или непосредственно к контейнерам гистограммы для увеличения производительности; в этом случае можно воспользоваться *hist->mat.data.fl* (только для плотных гистограмм). Причин, по которым можно использовать представленные далее примеры, много, одни из возможных - это поиск размерности гистограммы, определение какие регионы представляют отдельные контейнеры и т.п. Для получения этой информации можно использовать следующие приемы (получение доступа либо к фактическим данным в структуре *CvHistogram*, либо к информации, вложенной в структуру *CvMatND* известной как *mat*):

```
int n_dimension = histogram->mat.dims;
int dim_i_nbins = histogram->mat.dim[ i ].size;

// для плотных гистограмм
int dim_i_bin_lower_bound = histogram->thresh[ i ][ 0 ];
int dim_i_bin_upper_bound = histogram->thresh[ i ][ 1 ];

// для разрежённых гистограмм
int dim_i_bin_j_lower_bound = histogram->thresh2[ i ][ j ];
int dim_j_bin_j_upper_bound = histogram->thresh2[ i ][ j+1 ];
```

Как можно было бы заметить, многое чего происходит внутри структуры *Histogram*.

[П]|[РС]|(РП) Базовые манипуляции над гистограммами

Теперь, разобрав структуру *Histogram*, можно переходить к разбору некоторых более интересных вещей. Для начала необходимо рассмотреть основы основ, которые будут использоваться снова и снова. А затем уже можно будет перейти к более сложным функциям, которые можно будет задействовать при выполнении более сложных задач.

При работе с гистограммами, как правило, необходимо просто накапливать информацию в различных контейнерах. При этом, зачастую желательно, чтобы гистограмма была нормализована, т.е. когда каждый контейнер представлял долю от общего числа событий:

```
cvNormalizeHist( CvHistogram* hist, double factor );
```

Аргумент *hist* - это гистограмма, а *factor* - число на которое необходимо нормализовать гистограмму (обычно он равен 1). Как можно было заметить, аргумент *factor* имеет тип *double*, хотя внутри функции *CvHistogram()* он всегда *float* - это ещё одно доказательство того, что OpenCV постоянно развивается!

Следующая удобная функция - функция порогового преобразования:

```
cvThreshHist( CvHistogram* hist, double factor );
```

Аргумент *factor* - это пороговое значение. В результате порогового преобразования над гистограммой, все контейнеры, у которых стоимость ниже порогового значения, устанавливаются в 0. Если вспомнить функцию *cvThreshold()*, можно сказать, что функция порогового преобразования над гистограммами аналогична вызову функции порогового преобразования над изображениями с установленным значением аргумента *threshold_type* в *CV_THRESH_TOZERO*. К сожалению, не существует функций, обеспечивающих преобразования, аналогичные другим функциям пороговых преобразований. Однако, на практике, функции *cvThreshHist()* вполне достаточно, т.к. при работе с реальными данными некоторые контейнеры содержат малое количество точек, которые в свою очередь являются шумами и должны быть, как правило, обнуляться.

cvCopyHist() - ещё одна удобная функция, которая копирует данные одной гистограммы в другую.

```
void cvCopyHist( const CvHistogram* src, CvHistogram** dst );
```

Эту функцию можно использовать двумя способами. Если конечная гистограмма ***dst** имеет тот же размер, что и исходная гистограмма, то все данные **src** будут скопированы в ***dst**. Другой способ использовать *cvCopyHist()* - установить ***dst** в NULL, в этом случае под ***dst** будет выделено столько памяти, сколько занимает **src**, с последующим копированием данных (эту функцию можно сравнить с аналогичной ей *cvCloneImage()*). Если при вызове *cvCopyHist()* ***dst** = NULL, то ***dst** будет задан указателем на вновь выделенную гистограмму только после того, как функция вернет значение.

Следующей удобной функцией является *cvGetMinMaxHistValue()*, которая возвращает максимальное и минимальное значения в гистограмме.

```
void cvGetMinMaxHistValue(
    const CvHistogram* hist
    , float*           min_value
    , float*           max_value
    , int*             min_idx = NULL
    , int*             max_idx = NULL
);
```

Аргумент *hist* - это гистограмма. Функция возвращает максимальное и минимальное значения в ***min_value** и ***max_value** соответственно. Если одно из значений (или оба) не нужны, тогда нужно установить соответствующий аргумент в NULL. Следующие два аргумента не обязательны: если они установлены в значения по умолчанию (NULL), то они не используются. Однако, если указатели *int != NULL, тогда эти аргументы будут содержать индексы минимального и максимального значений. Если гистограмма многомерная, то аргументы *min_idx* и *max_idx* (если они не NULL) должны указывать на массивы целых чисел размером равным размерности гистограммы. Если более чем один контейнер имеет одинаковое минимальное (или максимальное) значение, то будет возвращен наименьший индекс (в лексографическом порядке для многомерных гистограмм).

Зачастую, после сбора данных и формирования гистограммы используется функция *cvGetMinMaxHistValue()* для поиска минимального значения, с последующим поиском порога в районе этого минимума при помощи *cvThreshHist()* и окончательной нормализацией гистограммы при помощи *cvNormalizeHist()*.

Последней, но не менее значимой, удобной функцией является *cvCalcHist()*. Она автоматически вычисляет гистограмму по изображению:

```

void cvCalcHist(
    IplImage** image
    , CvHistogram* hist
    , int accumulate = 0
    , const CvArr* mask = NULL
);

```

Аргумент *image* - это указатель на массив указателей на *IplImage* (также возможно использование указателя на матрицу *CvMat**). Это позволяет обрабатывать несколько плоскостей изображения. В случае с многоканальными изображениями (например, HSV или RGB), перед вызовом *cvCalcHist()* необходимо разделить это изображение на плоскости при помощи *cvSplit()*. Правда это немного мучительно, однако, зачастую, требуется "пройтись" по многоканальному изображению, которое содержит различные отфильтрованные версии изображения - например, плоскость градиента или U- и V-плоскости YUV. Это может повлечь за собой бардак при попытке передачи изображений с различным числом каналов (а то, что кто-то в определенный момент времени захочет получить лишь часть изображения сомневаться не стоит!). Что бы избежать всей этой путаницы, передаваемые в *cvCalcHist()* изображения должны быть одноканальными. При заполнении гистограмм, контейнеры будут идентифицироваться как кортежи, образованные на основе многоканальных изображений. Аргумент *hist* должен быть гистограммой соответствующего размера (т.е. размерности, равной числу плоскостей изображения передаваемых изображений). Последние два аргумента не обязательны. Если накопительный аргумент не нулевой, то он указывает на то, что гистограмма *hist* не должна быть очищена прежде чем изображение будет прочитано; стоит обратить внимание на то, что накопитель позволяет вызывать *cvCalcHist()* несколько раз во время циклического сбора данных. Последний аргумент *mask* - это необязательная булева маска; если *mask != NULL*, тогда пиксели, соответствующие ненулевым элементам в маске будут включены в гистограмму.

Сравнение двух гистограмм

Ещё одним незаменимым инструментом при работе с гистограммами, впервые введённый Swain и Ballard и обобщённый в дальнейшем Schiele и Crowley, является возможность сравнить две гистограммы в терминологии некоторых специфичных критериев подобия. В OpenCV для этого существует функция *cvCompareHist()*:

```

double cvCompareHist(
    const CvHistogram* hist1
    , const CvHistogram* hist2
    , int method
);

```

Первые два аргумента - это гистограммы для сравнения, которые должны быть одинакового размера. Третий аргумент задает дистанционную метрику. Доступны четыре варианта этой метрики.

Корреляция (method = CV_COMP_CORREL)

$$d_{\text{correl}}(H_1, H_2) = \frac{\sum_i H'_1(i) \cdot H'_2(i)}{\sqrt{\sum_i H'^2_1(i) \cdot H'^2_2(i)}}$$

где $H'_k(i) = H_k(i) - (1/N) \left(\sum_j H_k(j) \right)$, а N равно числу контейнеров гистограммы.

Для корреляции большая оценка представляет лучшее совпадение, чем меньшая. Идеальное совпадение 1, а максимальное несовпадение -1; значение 0 указывает на отсутствие корреляции (случайная ассоциация).

Хи-квадрат (method = CV_COMP_CHISQR)

$$d_{\text{chi-square}}(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) + H_2(i)}$$

Для хи-квадрата (хи-квадрат был изобретён Karl Pearson, который основал область математической статистики) более низкая оценка означает лучшее совпадение, чем более высокая. Идеальное совпадение равно 0, а максимальное несовпадение не ограничено (зависит от размера гистограммы)

Пересечение (method = CV_COMP_INTERSECT)

$$d_{\text{intersection}}(H_1, H_2) = \sum_i \min(H_1(i), H_2(i))$$

Для пересечения гистограмм высокая оценка указывает на хорошее совпадение, а низкая на плохое. Если обе гистограммы нормированы на 1, то идеальное совпадение это 1, а полное несовпадение - 0.

Расстояние Bhattacharyya (method = CV_COMP_BHATTACHARYYA)

$$d_{\text{Bhattacharyya}}(H_1, H_2) = \sqrt{1 - \sum_i \frac{\sqrt{H_1(i) \cdot H_2(i)}}{\sqrt{\sum_i H_1(i) \cdot \sum_i H_2(i)}}}$$

Для соответствия Bhattacharyya низкие показатели свидетельствуют о хорошем совпадении, а высокие о плохом. Идеальное совпадение 0, а полное несовпадение 1.

При помощи CV_COMP_BHATTACHARYYA осуществляется нормализация исходной гистограммы. В целом, однако, нормализация гистограмм должна происходить перед их сравнением, потому что такие понятия, как пересечение гистограмм имеют мало

смысла без нормализации.

Простейший случай изображенный на рисунке 7-4 должен прояснить ситуацию. На самом деле, речь идет о наиболее простейшем случае, который возможно представить: одномерная гистограмма с двумя контейнерами. Данная модель гистограммы имеет значение 1.0 для левого контейнера и 0.0 для правого контейнера. Последние три строки показывают сравнение гистограмм и значения, сгенерированные различными метриками (метрика EMD будет описана позже).

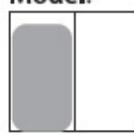
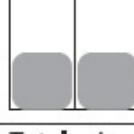
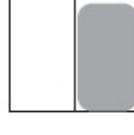
Histograms:	Matching measures:				
Model:	Correlation:	Chi square:	Intersection:	Bhattacharyya:	EMD:
					
Exact match: 	1.0	0.0	1.0	0.0	0.0
Half match: 	0.7	0.67	0.5	0.55	0.5
Total mis-match: 	-1.0	2.0	0.0	1.0	1.0

Рисунок 7-4. Сопоставление гистограмм

Рисунок 7-4 предоставляет наглядное представление сравнения различных типов сопоставления, но есть в этом рисунке кое-что, что может смутить. Если совместить все контейнеры гистограммы в один слот - например, гистограммы из первого и третьего графика сравнения - то все эти методы сопоставления (за исключением EMD) дают максимальное несоответствие, даже при условии, что эти две гистограммы имеют аналогичную "форму". В крайне правом столбце на рисунке 7-4 указаны значения, возвращаемые EMD. Согласно третьей модели, мера EMD количественно изменяет ситуацию, а именно: третья гистограмма смещается вправо на одну единицу. Про данное поведение будет более подробно рассказано в следующем разделе "Earth Mover's Distance".

Согласно опыту авторов метрик, пересечение работает хорошо для быстрых и грязных соответствий, а хи-квадрат или Bhattacharyya работает хорошо для медленных, но более точных совпадений. EMD дает более интуитивно понятные совпадения, но при

этом работает медленнее.

Пример использования гистограмм

Программа из примера 7-1 показывает, как можно использовать некоторые из ранее рассмотренных функций. Эта программа вычисляет гистограмму тон-насыщенность для изображения и рисует эту гистограмму как освещенную сетку.

Пример 7-1. Вычисление и отображение гистограммы

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv ) {
    IplImage* src;

    if( argc == 2 && (src = cvLoadImage(argv[1], 1)) != 0) {
        // Конвертация изображения в формат HSV и раскладывание его на каналы
        //
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        cvCvtColor( src, hsv, CV_BGR2HSV );

        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );

        // Построение гистограммы
        //
        int h_bins = 30, s_bins = 32;
        CvHistogram* hist;

        {
            int hist_size[] = { h_bins, s_bins };
            float h_ranges[] = { 0, 180 }; // hue is [0,180]
            float s_ranges[] = { 0, 255 };
            float* ranges[] = { h_ranges, s_ranges };

            hist = cvCreateHist(
                2
                ,hist_size
                ,CV_HIST_ARRAY
                ,ranges
                ,1
            );
        }

        cvCalcHist( planes, hist, 0, 0 ); // Вычисление гистограммы
        cvNormalizeHist( hist[1], 1.0 ); // Нормализация

        // Создание изображения для визуализации гистограммы
    }
}
```

```

//  

int scale = 10;  

IplImage* hist_img = cvCreateImage(  

    cvSize( h_bins * scale, s_bins * scale )  

    ,8  

    ,3  

);  

cvZero( hist_img );  

// Заполнение изображения небольшими серыми квадратами  

//  

float max_value = 0;  

cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );  

for( int h = 0; h < h_bins; h++ ) {  

    for( int s = 0; s < s_bins; s++ ) {  

        float bin_val = cvQueryHistValue_2D( hist, h, s );  

        int intensity = cvRound( bin_val * 255 / max_value );  

        cvRectangle(  

            hist_img  

            ,cvPoint( h*scale, s*scale )  

            ,cvPoint( (h+1)*scale - 1, (s+1)*scale - 1 )
            ,CV_RGB( intensity,intensity,intensity )
            ,CV_FILLED
        );
    }
}  

cvNamedWindow( "Source", 1 );  

cvShowImage( "Source", src );  

cvNamedWindow( "H-S Histogram", 1 );  

cvShowImage( "H-S Histogram", hist_img );  

cvWaitKey(0);
}
}

```

В данном примере было проделано не мало работы по подготовке аргументов функции *cvCalcHist()*. Так же было принято решение о нормализации цвета, а не самой гистограммы, хотя обратный вариант, в случае с некоторыми изображениями, может дать более лучший результат. Выбрать какой порядок использовать, можно с помощью *cvGetMinMaxHistValue()*.

Теперь давайте рассмотрим более практический пример: цветовая гистограмма человеческой руки под различным освещением. В левой колонке на рисунке 7-5 представлены фотографии руки в помещении, на улице в тени и на солнце. В средней колонке показаны синие, зеленые и красные (BGR) гистограммы отслеживаемых тонов руки. В правом столбце представлены соответствующие HSV гистограммы, где

вертикальная ось это V (значение), радиус это S (насыщенность), а угол это H (оттенок). Стоит отметить тот факт, что цветовое смещение вызвано изменением освещения света.

В качестве тестовой гистограммы сравнения можно рассмотреть часть одного из ранее представленных изображений (например, верхнюю половину ладони в помещении) и сравнить гистограммы представляющие цвета в этом изображении или гистограммы представляющие цвета в нижней половине взятого изображения; либо сравнить гистограммы представлений двух других изображений руки. Телесные цвета легче распознать, если преобразовать изображение в цветовое пространство HSV.

Оказывается, ограничившись тоном и насыщенностью, можно распознавать признаки телесного цвета по всем этническим группам. Результаты сравнения для описанного эксперимента показаны в таблице 7-1, подтверждают тот факт, что освещение может вызвать серьезные несоответствия в цвете. Иногда в контексте изменения освещения, нормализованные BGR изображения дают результат лучше, чем HSV.

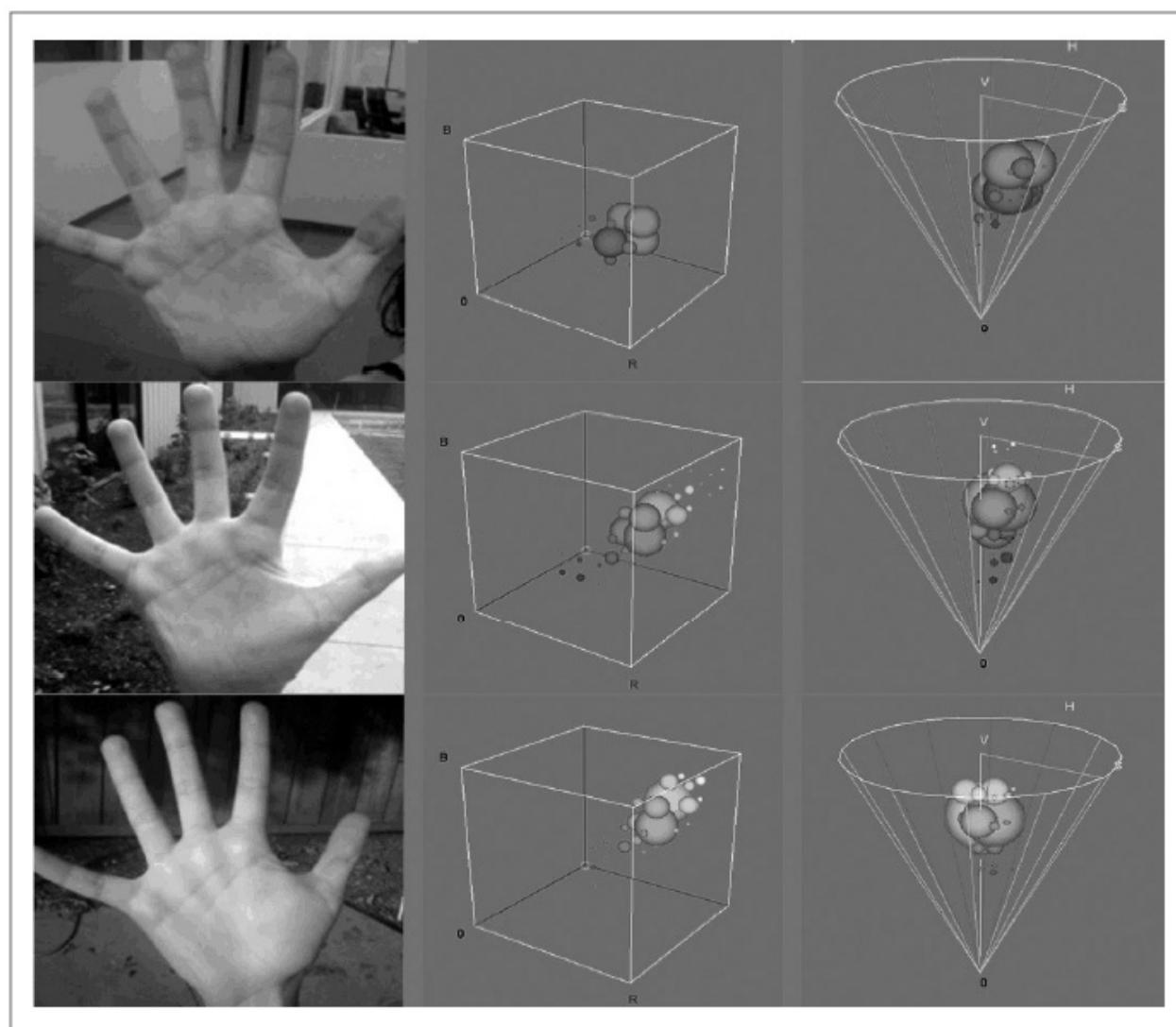


Рисунок 7-5. Гистограммы фотографий руки, сделанные в разных местах (в помещении, в тени и на солнце); по середине гистограмма BGR, а справа HSV

Таблица 7-1. Гистограммы сравнения фотографий руки (в помещении, в тени и на солнце), построенные при помощи четырех методов сопоставления

Объект сравнения	CORREL	CHISQR	INTERSECT	BHATTACHARYYA
В помещении	0.96	0.14	0.82	0.2
В тени	0.09	1.57	0.13	0.8
На солнце	0.0	1.98	0.01	0.99

[П]|[РС]|(РП) Более сложные манипуляции

Всё, что было рассмотрено до сих пор, было достаточно простым. Каждая из функций несла очевидную необходимость. В совокупности, они образуют хорошую основу для всего того, что можно сделать с гистограммами в контексте компьютерного зрения (или же в каких-либо других контекстах). В данном разделе речь пойдет о более сложных процедурах, которые имеются в OpenCV и которые чрезвычайно полезны в некоторых приложениях. Эти процедуры включают в себя более сложные методы сравнения двух гистограмм, а так же инструменты для вычисления и/или визуализации частей изображения, которые соответствуют заданной части гистограммы.

Earth Mover's Distance

Изменение освещения может привести к сдвигу цветовых значений (рисунок 7-5), хотя такой сдвиг, как правило, не изменяет форму гистограммы значений цвета, но при изменении положения цветовых значений, схемы сравнения гистограмм обречены на провал. Однако если вместо гистограмм *сравнения* использовать гистограммы *расстояния*, то можно сравнить две гистограммы, несмотря на сдвиг одной гистограммы относительно другой на малые расстояния. Earth mover's distance (EMD) является метрикой; по сути это мера работы необходимая для преобразования одной или нескольких гистограмм в новое положение. Она работает с любым количеством измерений.

Если снова взглянуть на рисунок 7-4, то можно увидеть характер изменения расстояния EMD. Точное совпадение соответствует расстоянию равное 0. Совпадение на половину это половина "полного перебора" - то количество, которое потребуется, чтобы перенести половину левой гистограммы в соседний слот. И наконец, для перемещения всей гистограммы на один шаг вправо потребуется вся единица расстояния (например, для изменения гистограммы в "совершенно несовпадающую" гистограмму).

Алгоритм EMD имеет весьма общий характер; это позволяет пользователям устанавливать собственную дистанционную метрику или собственную матрицу стоимости перемещения. Можно зафиксировать, где гистограмма "material" изменяется при переходе от одной гистограммы к другой или можно воспользоваться нелинейными показателями расстояния, полученные из предварительной информации о данных. В OpenCV EMD соответствует функция `cvCalcEMD2()`:

```

float cvCalcEMD2(
    const CvArr*      signature1
    ,const CvArr*      signature2
    ,int               distance_type
    ,CvDistanceFunction distance_func = NULL
    ,const CvArr*      cost_matrix = NULL
    ,CvArr*            flow        = NULL
    ,float*            lower_bound = NULL
    ,void*             userdata   = NULL
);

```

Функция `cvCalcEMD2()` имеет достаточно много аргументов. Можно подумать, что это решение довольно таки сложное для такой интуитивной функции, но сложность проистекает от возможности тонкой настройки алгоритма (если есть необходимость в знании всех "кровавых" подробностей, то необходимо изучить статью S. Peleg, M. Werman и H. Rom 1989 года "A Unified Approach to the Change of Resolution: Space and Gray-Level", а затем заглянуть в руководство OpenCV и изучить файл `...\\opencv\\docs\\refopencvref_cv.htm`). К счастью, функция может быть использована в более обобщенной и интуитивно понятной форме без большинства аргументов (на это указывает `= NULL` в определении функции). В примере 7-2 показана упрощённая версия этой функции.

Пример 7-2. Упрощённая версия функции `cvCalcEMD2()`

```

float cvCalcEMD2(
    const CvArr*      signature1
    ,const CvArr*      signature2
    ,int               distance_type
);

```

Параметр `distance_type` для упрощённой версии `cvCalcEMD2()` может быть либо *расстоянием Manhattan* (`CV_DIST_L1`), либо *расстоянием Euclidean* (`CV_DIST_L2`). Хотя EMD и используется для гистограмм, интерфейс упрощенной версии функции предполагает использование массивов для первых двух аргументов.

Эти массивы всегда типа `float` и состоят из строк, содержащие количество контейнеров гистограммы и их координаты. Для одномерной гистограммы (рисунок 7-4) сигнатуры (перечислимые строки массива) левой колонки гистограммы (пропускная модель) будет выглядеть следующим образом: верхняя [1, 0; 0, 1]; средняя [0.5, 0; 0.5, 1]; нижняя [0, 0; 1, 1]. Если взять контейнер трехмерной гистограммы с 537 контейнерами (x, y, z), то сигнтура строки с индексом (7, 43, 11) будет [537, 7; 43, 11]. Это необходимый шаг преобразования гистограмм в сигнатуры.

В качестве примера, предположим, что имеется две гистограммы *hist1* и *hist2*, которые необходимо преобразовать в сигнатуры *sig1* и *sig2*. Для усложнения задачи, будут использованы двумерные гистограммы размерностью *h_bins* × *s_bins*. Пример 7-3 отображает процесс преобразования этих двух гистограмм в две сигнатуры.

Пример 7-3. Создание сигнатур гистограмм для EMD сравнения

```
// Преобразование гистограмм в сигнатуры для EMD сравнения
// На входе имеются двумерные гистограммы hist1 и hist2
// обе имеют размерности h_bins × s_bins (хотя для EMD,
// гистограммы не должны совпадать по размеру)
//

CvMat* sig1,sig2;
int numrows = h_bins*s_bins;

// Создание матриц для хранения сигнатур
//
sig1 = cvCreateMat(numrows, 3, CV_32FC1); // 1 count + 2 coords = 3
sig2 = cvCreateMat(numrows, 3, CV_32FC1); // sig1 и sig2 имеют тип float

// Заполнение сигнатур для двух гистограмм
//
for( int h = 0; h < h_bins; h++ ) {
    for( int s = 0; s < s_bins; s++ ) {
        float bin_val = cvQueryHistValue_2D( hist1, h, s );
        cvSet2D(sig1,h*s_bins + s,0,cvScalar(bin_val)); // значение контейнера
        cvSet2D(sig1,h*s_bins + s,1,cvScalar(h)); // Coord 1
        cvSet2D(sig1,h*s_bins + s,2,cvScalar(s)); // Coord 2

        bin_val = cvQueryHistValue_2D( hist2, h, s );
        cvSet2D(sig2,h*s_bins + s,0,cvScalar(bin_val)); // значение контейнера
        cvSet2D(sig2,h*s_bins + s,1,cvScalar(h)); // Coord 1
        cvSet2D(sig2,h*s_bins + s,2,cvScalar(s)); // Coord 2
    }
}
```

В этом примере функция *cvSet2D()* (использование *cvSetReal2D()* или *cvmSet()*) было бы более компактным и эффективным решением, но пример проясняет выбранный вариант: дополнительные накладные расходы малы по сравнению с фактическими расчетами расстояния в EMD) принимает *CvScalar()*, чтобы задать значения массива, хотя каждая запись в данной матрице имеет тип *float*. Для удобства используется встроенный макрос *CvScalar()*. После получения сигнатур можно вычислять меру расстояния. В примере 7-4 показано вычисление расстояния *Euclidean*.

Пример 7-4. Использование EMD для измерения сходства между распределениями

```
float emd = cvCalcEMD2( sig1, sig2, CV_DIST_L2 );
printf( "%f", emd );
```

Обратная проекция

Обратная проекция это способ записи пикселей (*cvCalcBackProject()*) или участка пикселей (*cvCalcBackProjectPatch()*), соответствующие распределению пикселей в модели гистограммы. Например, имея гистограмму телесного цвета можно воспользоваться обратной проекцией, чтобы найти цвета на изображении, соответствующие цвету кожи. В OpenCV функция для выполнения подобного рода поиска выглядит следующим образом:

```
void cvCalcBackProject(
    IplImage**          image
    ,CvArr*             back_project
    ,const CvHistogram* hist
);
```

Ранее уже было рассказано о массиве одноканальных изображений *IplImage*** при разборе функции *cvCalcHist()* (раздел "Базовые манипуляции над гистограммами"). Число изображений в этом массиве точно такое же - и в том же порядке - что использовалось при построении модели гистограммы *hist*. В примере 7-1 показано, как преобразовать изображение в одноканальные кадры и сделать из них массив. Изображение или массив *back_project* это одноканальное 8-битное или вещественное изображение того же размера, что и входное изображение. Значения *back_project* устанавливаются в значения контейнеров *hist*. Если гистограмма нормализована, тогда эти значения могут быть связаны с условной вероятностью значения (т.е. вероятность того, что пиксели в изображении - члены типа, характеризуемые гистограммой *hist*). На рисунке 7-6 используется гистограмма телесных цветов для получения изображения вероятностей телесных цветов.

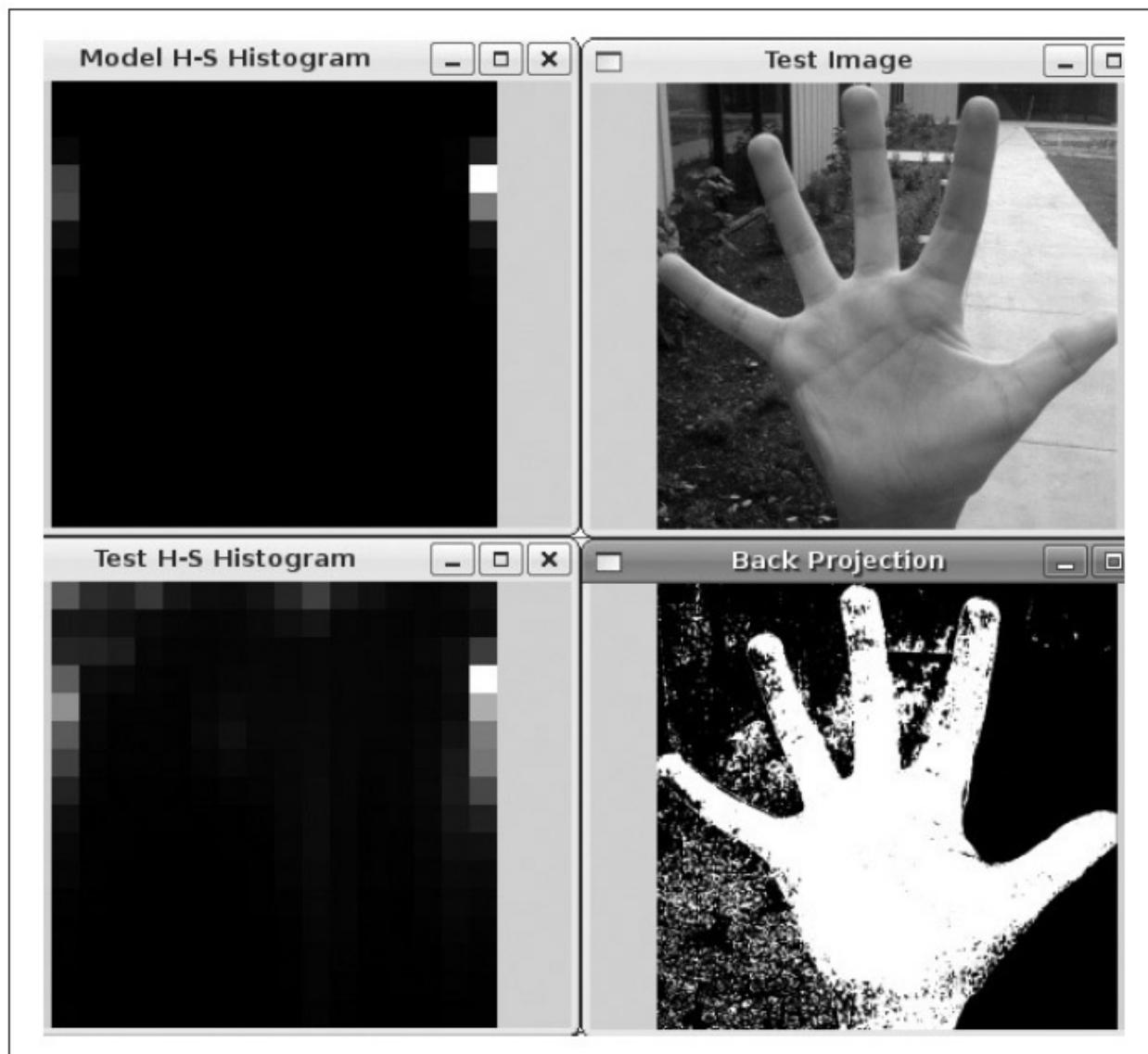


Рисунок 7-6. Обратная проекция гистограммы оценивает каждый пиксель, основываясь на значение его цвета: гистограмма телесного цвета HSV (сверху слева) используется для преобразования изображения руки (сверху справа) в изображение вероятностей телесных цветов (внизу справа); нижнее левое изображение - это гистограмма изображения руки

В частности, в случае гистограммы H-S, если C это цвет пикселя и F это вероятность того, что пиксель соответствует телесному цвету, тогда имеем вероятность $p(C|F)$ того, что цвет пикселя действительно соответствует телесному цвету. Но это не значит, что $p(F|C)$ - это вероятность того, что телесный цвет соответствует цвету пикселя. Тем не менее, эти две вероятности связаны теоремой Байеса и если в целом известна вероятность встречи объекта телесного цвета на сцене, а так же общая вероятность попадания в диапазон телесного цвета, тогда можно вычислить $p(F|C)$ от $p(C|F)$. В частности, теорема Байеса устанавливает следующее соответствие:

$$p(F|C) = \frac{p(F)}{p(C)} p(C|F)$$

Когда *back_project* - это изображение типа byte, а не float, то либо над гистограммой не нужно выполнять нормализацию, либо это изображение необходимо масштабировать перед использованием. Причина этого в том, что максимально возможное значение нормализованной гистограммы 1, так что, все, что меньше будет округлено до 0 в 8-битном изображении. Так же, возможно, потребуется масштабировать *back_project* для того, чтобы можно было увидеть собственными глазами разницу в высоте контейнеров гистограммы.

Обратная проекция, основанная на patch

Базовый метод обратной проекции может быть использован для моделирования - является ли пиксель членом определенного типа объекта (когда данный тип объекта был смоделирован гистограммой). Это не совсем так же, как вычисление вероятности наличия конкретного объекта. Альтернативный метод предполагает рассмотрение субрегиона изображения и признака (например, цвета) гистограммы в этом субрегионе и проверку на соответствие гистограммы субрегиона гистограмме модели; потом можно связать с каждым субрегионом вероятность того, что моделируемый объект присутствует в данном субрегионе.

Таким образом, так же, как *cvCalcBackProject()* позволяет нам вычислять является ли пиксель частью заданного объекта, *cvCalcBackProjectPatch()* позволяет узнавать, является ли *patch* частью заданного объекта. Как показано на рисунке 7-7, функция *cvCalcBackProjectPatch()* использует окно выборки по всему исходному изображению. В каждом месте исходного массива изображения, все пиксели *patch* используются для установки одного пикселя в конечном изображении. Это важно, поскольку многие свойства изображения, например, такое как текстура, не могут быть определены на уровне отдельных пикселей, только на уровне группы пикселей.

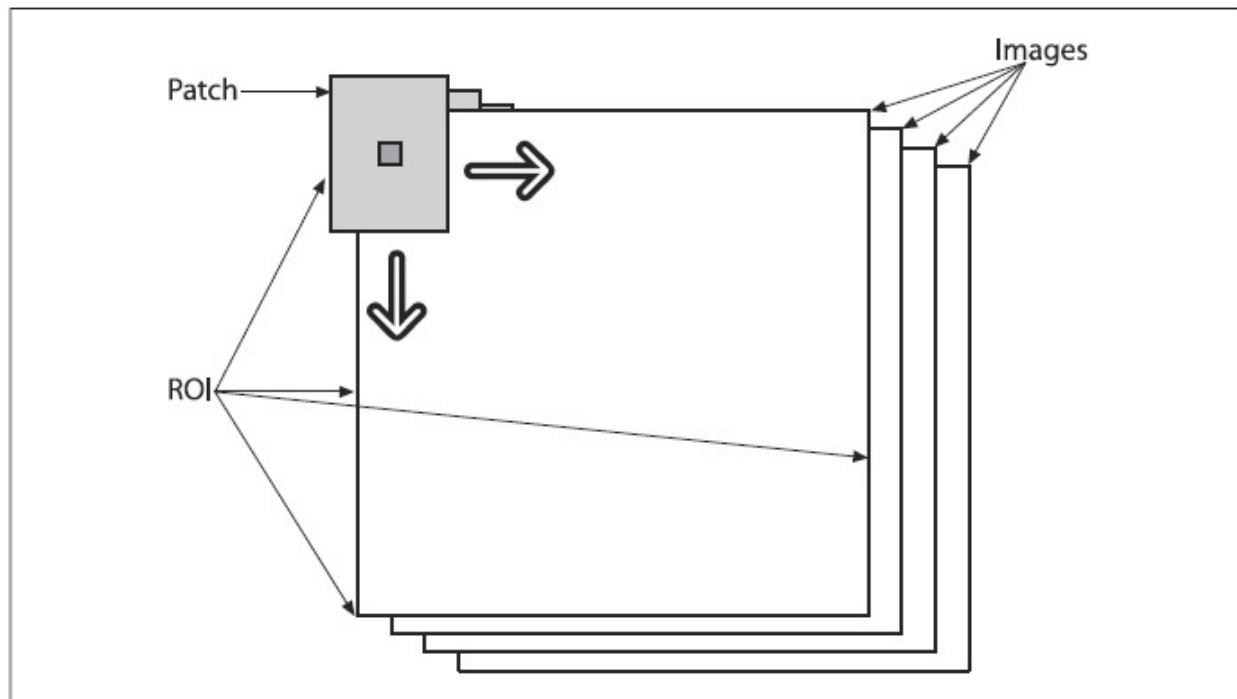


Рисунок 7-7. Обратная проекция: скользящее окно по поверхности исходного изображения используется для установки соответствующего пикселя в конечном изображении; для нормализованной гистограммы изображение может быть интерпретировано как карта вероятности с указанием вероятного места присутствия объекта

Для упрощения описанных примеров, цвет был подобран для создаваемой модели гистограммы. Таким образом, на рисунке 7-6 вся рука "светится", потому что пиксели соответствуют модели гистограммы телесного цвета. Использование *patch* помогает при обнаружении статистических свойств, которые появляются в локальных областях при изменении локальной интенсивности, при этом текстура формируется до конфигурирования свойств, формирующие весь объект. Существует два варианта использования *cvCalcBackProjectPatch()*: в качестве обнаружения региона, когда окно выборки меньше объекта и в качестве детектора объекта, когда окно выборки соизмеримо с объектом. Рисунок 7-8 отображает использование *cvCalcBackProjectPatch()* для поиска региона. Все начинается с модели гистограммы телесных цветов и передвижения небольшого окна вдоль изображения так, что в каждый пиксель обратной проекции записывается вероятность того, что текущий пиксель соответствует телесному цвету при учете всех окружающих окна пикселей в оригинальном изображении. На рисунке 7-8 рука намного больше сканирующего окна, что является преимуществом при распознавании региона ладони. На рисунке 7-9 показана гистограмма, соответствующая распознанной синей кружке. В отличие от рисунка 7-8, где был обнаружен регион, рисунок 7-9 показывает как *cvCalcBackProjectPatch()* может быть использован для обнаружения объекта. Когда размеры окна примерно такого же размера, как объект, который требуется найти на

изображении, то искомый объект "светится" на изображении обратной проекции. Поиск пиковых значений на изображении обратной проекции соответствует поиску местоположения объекта (на рисунке 7-9 это кружка).

```
void cvCalcBackProjectPatch(  
    IplImage** images  
    , CvArr* dst  
    , CvSize patch_size  
    , CvHistogram* hist  
    , int method  
    , float factor  
) ;
```

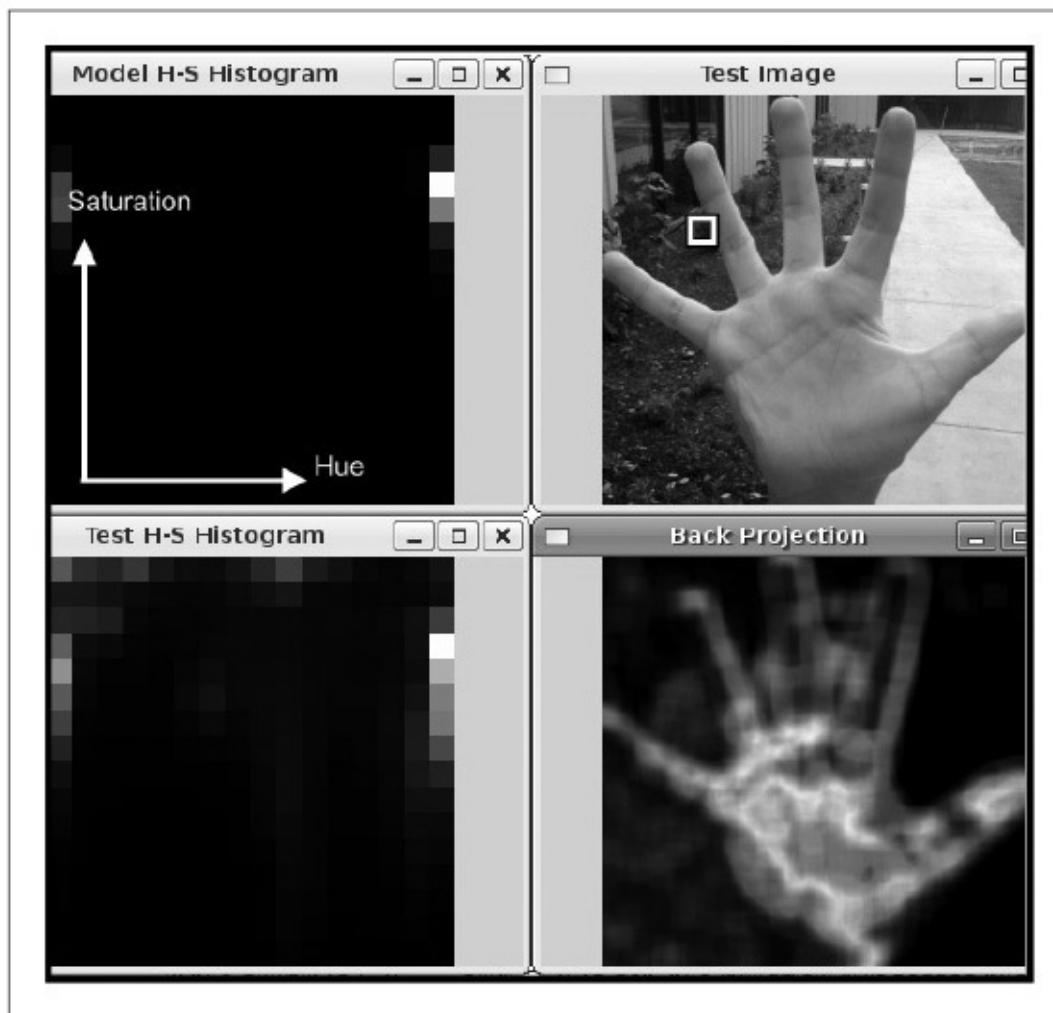


Рисунок 7-8. Обратная проекция используемая для построения гистограммы объекта телесного цвета, где окно (маленький белый квадрат в верхней правой части окна) намного меньше руки; гистограмма соответствует распределению телесных цветов с пиковыми значениями преимущественно в центре руки

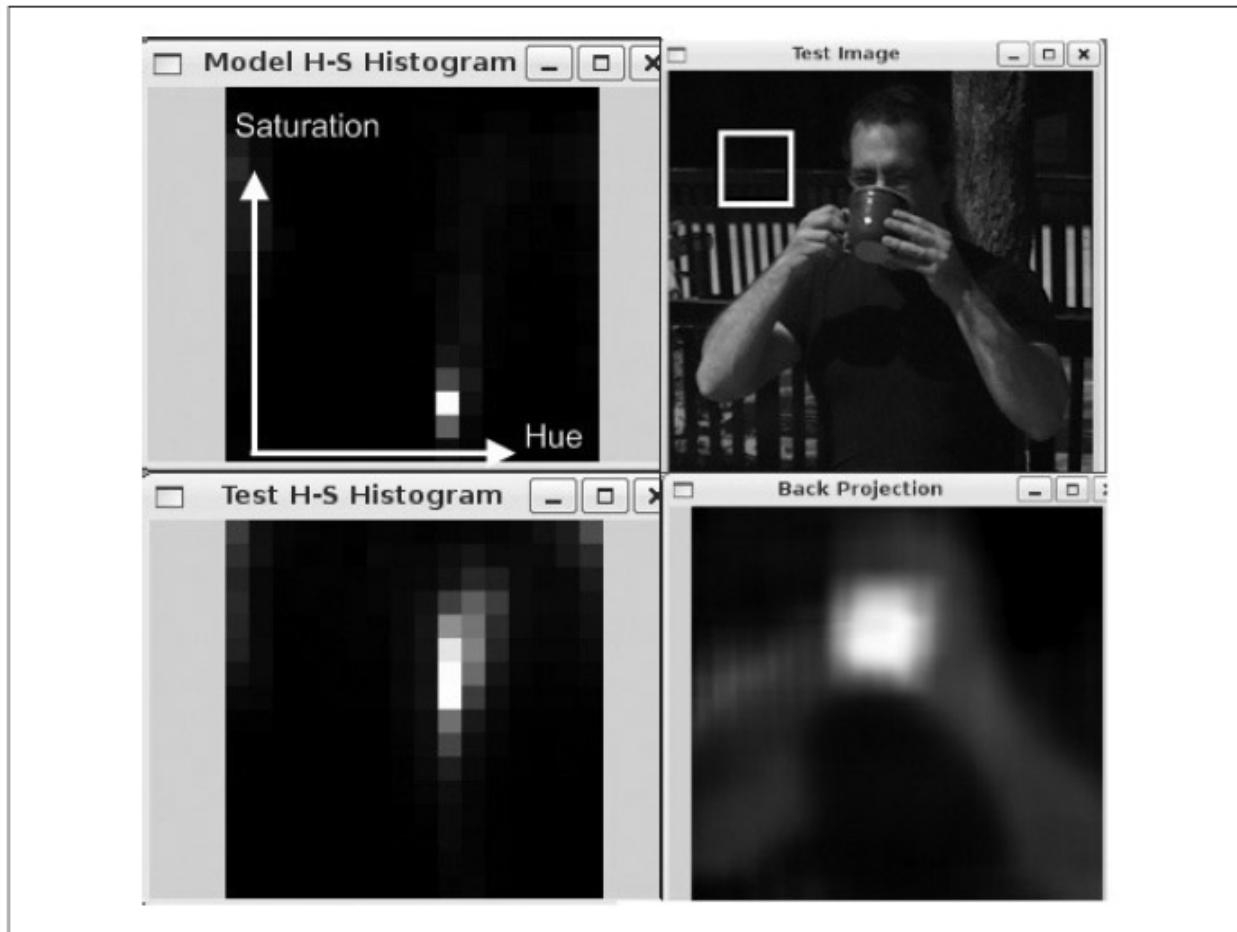


Рисунок 7-9. Использование `cvCalcBackProjectPatch()` для поиска объекта (чашки кофе), размер которого примерно соответствует размеру *patch* (белый квадрат в верхнем правом изображении): гистограмма оттенок-насыщенность для искомого объекта (левый верхний угол), которая может быть сопоставлена с HS гистограммой для изображения в целом (нижний левый угол); результат (нижний правый угол) показывает, что объект удалось довольно таки легко определить в силу своего цвета

Аргумент *images* - это тот же массив одноканального изображения, который был использован в `cvCalcHist()` при создании гистограммы. Тем не менее, конечное изображение *dst* отличается: оно может быть только одноканальным вещественным изображением размера (*images[0][0].width - patch_size.x + 1, images[0][0].height - patch_size.y + 1*). Указанный размер объясняется тем, что центральный пиксель в *patch* используется для указания места для результата в конечном изображении *dst*, поэтому теряется половина размера *patch* по краям изображения со всех сторон.

Параметр *patch_size* задает размер *patch* и может быть установлен с помощью удобного макроса `cvSize(width, height)`. Параметр гистограмма *hist* был уже рассмотрен ранее. Параметр метода сравнения *method* принимает все те же значения, что и в случае с `cvCompareHist()` (раздел "Сравнение двух гистограмм") (Необходимо соблюдать осторожность при выборе метода, потому что некоторые отображают наилучший результат как 1, а некоторые как 0). Последний аргумент *factor* задает

уровень нормализации; этот параметр так же был описан при рассмотрении `cvNormalizeHist()`. По умолчанию он соответствует 1, любое более высокое значение помогает при визуализации. Благодаря этому параметру существует возможность нормализовать *hist* перед использованием `cvCalcBackProjectPatch()`.

И в заключении остался один не решенный вопрос: после того, как найдено изображение вероятности объекта, как этим изображением можно воспользоваться, что бы найти искомый объект? Для поиска можно воспользоваться функцией `cvMinMaxLoc()`, о которой шла речь в главе 3. Максимальные значения (если при этом еще и сгладить изображение), скорее всего, укажут на искомый объект.

Шаблон соответствия

Шаблон соответствия, используемый в `cvMatchTemplate()`, основан не на гистограммах; функция сравнивает текущее изображение *patch* с исходным изображением при помощи "скольжения" *patch* над исходным изображением, при использовании одного из методов сравнения, описанных в конце данного раздела.

Если, как показано на рисунке 7-10, имеется *patch*, содержащий изображение лица, то можно "скользить" этим изображением лица по исходному изображению для поиска совпадений, которые указывали бы, что на исходном изображении присутствует лицо. Вызов функции аналогичен вызову `cvCalcBackProjectPatch()`:



Рисунок 7-10. Использование cvMatchTemplate() для поиска лица при помощи шаблона

```
void cvMatchTemplate(
    const CvArr* image
    ,const CvArr* templ
    ,CvArr* result
    ,int method
);
```

В отличие от *cvCalcBackProjectPatch()*, на вход подается поток данных в виде 8-битного или вещественного типа или цветное изображение. Шаблон для сравнения *templ* это просто *patch* от исходного изображения, содержащее искомый объект поиска. Конечное изображение объекта будет размещено в *result*; это одноканальное типа *byte* или *float* изображение размера (*image->width - patch_size.x + 1, image->height - patch_size.y + 1*), аналогичное ранее рассмотренному в функции *cvCalcBackProjectPatch()*. Методы сравнения шаблонов *method* несколько сложнее, что будет показано далее. Обозначения: *I* - исходное изображение, *T* - шаблон, *R* - результат.

Метод сравнения - квадрат разности (method = CV_TM_SQDIFF)

Для данного метода идеальное соответствие равно 0, а плохое более высокому значению:

$$R_{\text{sq_diff}}(x, y) = \sum_{x', y'} [T(x', y') - I(x+x', y+y')]^2$$

Метод сравнения - корреляция (method = CV_TM_CCORR)

Это метод мультипликативного соответствия шаблона изображению, при этом плохое соответствие близко или равно 0, а идеальное равно довольно таки большому значению:

$$R_{\text{ccorr}}(x, y) = \sum_{x', y'} [T(x', y') \cdot I(x+x', y+y')]^2$$

Метод сравнения - коэффициент корреляции (method = CV_TM_CCOEFF)

Идея метода состоит в сравнении шаблона относительно его среднего и изображения относительно его среднего; так идеальное соответствие равно 1, худшее -1, а значение 0 означает, что корреляции нет (случайное выравнивание).

$$R_{\text{coeff}}(x, y) = \sum_{x', y'} [T'(x', y') \cdot I'(x+x', y+y')]^2$$

$$T'(x', y') = T(x', y') - \frac{1}{(w \cdot h) \sum_{x'', y''} T(x'', y'')}$$

$$I'(x+x', y+y') = I(x+x', y+y') - \frac{1}{(w \cdot h) \sum_{x'', y''} I(x+x'', y+y'')}$$

Метод нормализации

Для каждого из трех ранее описанных методов есть нормализованная версия, впервые разработанная Galton и описанная Rodgers. Как уже было сказано ранее нормализованный метод может помочь уменьшить влияние освещения на различия между шаблоном и изображением. В каждом из трех случаев коэффициент нормализации один и тот же:

$$Z(x, y) = \sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x+x', y+y')^2}$$

Значения для *method*, которые дают нормализованный расчет, приведены в таблице 7-2.

Таблица 7-2. Значения параметра *method* для нормализации шаблона сравнения

Значение параметра	Формула расчета
CV_TM_SQDIFF_NORMED	$R_{\text{sq_diff_normed}}(x,y) = \frac{R_{\text{sq_diff}}(x,y)}{Z(x,y)}$
CV_TM_CCORR_NORMED	$R_{\text{cor_normed}}(x,y) = \frac{R_{\text{cor}}(x,y)}{Z(x,y)}$
CV_TM_CCOEFF_NORMED	$R_{\text{coeff_normed}}(x,y) = \frac{R_{\text{coeff}}(x,y)}{Z(x,y)}$

Как обычно, более точные соответствия получаются (за счет большего числа вычислений) при переходе от простых методов (квадрат разности) к более сложным (коэффициент корреляции). Для выбора наилучшего в той или иной ситуации необходимо проводить небольшое тестирование каждого и сравнивать их в соотношении точность - скорость.

Опять же, необходимо соблюдать осторожность при интерпретации результатов. Метод квадрат разности дает лучший результат с минимальным количеством точек, в то время как методы корреляция и коэффициент корреляции дают лучшие результаты с максимальным количеством точек.

Как и в случае с *cvCalcBackProjectPatch()* при использовании *cvMatchTemplate()* чтобы получить необходимое изображение *result* можно воспользоваться *cvMinMaxLoc()* для поиска точки наилучшего соответствия. Опять же, необходимо гарантировать хорошую площадь совпадений вокруг этой точки во избежания случайного выравнивания шаблона, что порою случается для хорошего выполнения работы. Хорошее совпадение должно иметь хорошие совпадения рядом, потому что незначительные смещения шаблона не должны сильно изменять результаты реальных совпадений. В поисках лучшего соответствия может быть сделано незначительное сглаживание изображения *result* перед поиском максимума (для корреляции и коэффициента корреляции) или минимума (для квадрата разности). Морфологические операции также могут быть полезны при решении данной задачи.

Пример 7-5 дает хорошее представление того, как ведут себя различные методы шаблона соответствия. Эта программа в начале считывает сопоставляемые шаблон и изображение, а затем выполняет сравнение при помощи ранее рассмотренных методов.

Пример 7-5. Шаблон сопоставления

```

#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
#include <stdio.h>

int main( int argc, char** argv ) {
    IplImage *src, *templ, *ftmp[6]; // в ftmp будет помещен результат
    int i;

    if( argc == 3 ) {
        // Загрузка исходного изображения (где искать)
        //
        if((src=cvLoadImage(argv[1], 1))== 0) {
            printf("Error on reading src image %s\n", argv[i]);
            return(-1);
        }

        // Загрузка шаблона (что искать)
        //
        if((templ=cvLoadImage(argv[2], 1))== 0) {
            printf("Error on reading template %s\n", argv[2]);
            return(-1);
        }

        // Создание конечного изображения
        //
        int iwidth = src->width - templ->width + 1;
        int iheight = src->height - templ->height + 1;

        for(i=0; i < 6; ++i) {
            ftmp[i] = cvCreateImage(cvSize(iwidth, iheight), 32, 1);
        }

        // Сравнение шаблона с изображением
        //
        for(i=0; i<6; ++i) {
            cvMatchTemplate( src, templ, ftmp[i], i );
            cvNormalize(ftmp[i], ftmp[i], 1, 0, CV_MINMAX);
        }

        // Отображение результатов
        //
        cvNamedWindow( "Template", 0 );
        cvShowImage( "Template", templ );
        cvNamedWindow( "Image", 0 );
        cvShowImage( "Image", src );
        cvNamedWindow( "SQDIFF", 0 );
        cvShowImage( "SQDIFF", ftmp[0] );
        cvNamedWindow( "SQDIFF_NORMED", 0 );
        cvShowImage( "SQDIFF_NORMED", ftmp[1] );
        cvNamedWindow( "CCORR", 0 );
        cvShowImage( "CCORR", ftmp[2] );
    }
}

```

```
cvNamedWindow("CCORR_NORMED", 0);
cvShowImage("CCORR_NORMED", ftmp[3]);
cvNamedWindow("CCOEFF", 0);
cvShowImage("CCOEFF", ftmp[4]);
cvNamedWindow("CCOEFF_NORMED", 0);
cvShowImage("CCOEFF_NORMED", ftmp[5]);

// Позволить пользователю увидеть результаты
//
cvWaitKey(0);
}

else {
    printf("Call should be: matchTemplate image template \n");
}
}
```

В данном примере стоит обратить внимание на функцию `cvNormalize()`, которая позволяет отображать результаты на постоянной основе (некоторые из этих методов могут возвращать отрицательные значения). При нормализации используется флаг `CV_MINMAX`; это велит функции смещать и масштабировать току на изображении так, что все возвращаемые значения будут принимать значения из диапазона от 0 до 1. Рисунок 7-11 отображает результаты перемещения шаблона лица поверх исходного изображения (рисунок 7-10) при использовании всех возможных методов функции `cvMatchTemplate()`. Для наружного формирования изображения, почти всегда лучше использовать один из нормализованных методов. Среди всех возможных методов, коэффициент корреляции дает наиболее четкое очертание соответствия - но, как и ожидалось, обладает большой вычислительной стоимостью. Для конкретного приложения, таких, как автоматическая проверка частей или отслеживание признаков в видео, необходимо перебирать все метода, что бы найти баланс между скоростью и точностью.

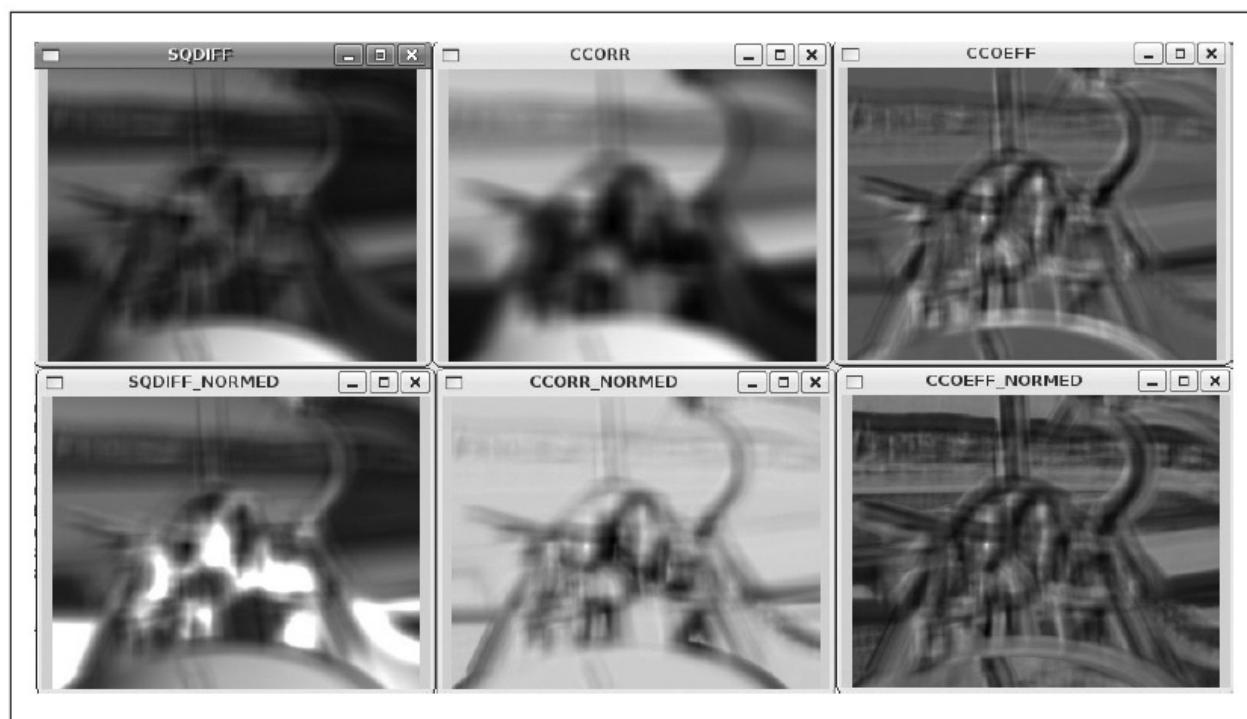


Рисунок 7-11. Результаты применения шести методов поиска по шаблону к рисунку 7-10: лучшее совпадение для метода квадрат разности равен 0, а для других методов это точка максимума; таким образом, совпадения в левой колонке соответствует темным областям, а для двух других колонок светлым

[П]|[РС]|(РП) Упражнения

1. Сгенерируйте 1000 случайных чисел r_i между 0 и 1. Определите размер контейнера гистограммы как $1/r_i$.
 - a. Присутствуют ли в полученной гистограмме контейнеры со схожим (разброс ± 10) количеством записей
 - b. Предложите способ борьбы с нелинейным распределением, так, чтобы каждый контейнер имел схожее количество данных
2. Возьмите три изображения руки в трех различных сценах (как было описано в тексте главы). Используйте `cvCalcHist()`, чтобы создать RGB гистограмму телесного цвета для руки в помещении.
 - a. Попробуйте использовать только несколько больших контейнеров (например, 2), среднее количество контейнеров (например, 16) и малое количество контейнеров (например, 256). Затем запустите процедуру сопоставления (используя все методы сопоставления). Опишите результаты
 - b. Теперь добавьте 8, а затем 32 контейнера и попытайтесь сопоставить при различном освещении (в помещении и на улице). Опишите результаты.
3. Как и в упражнении 2, создайте RGB гистограмму телесного цвета руки. Примите первую гистограмму руки в помещении (внутреннюю) за модель и измерьте EMD в отношении второй внутренней, первой открытой затененной и первой открытой освещенной гистограммах. Используйте эти измерения для установки расстояния порога.
 - a. Используя этот EMD порог, оцените насколько хорошо был обнаружен телесный цвет на третьей внутренней, второй открытой затененной и второй открытой освещенной гистограммах. Опишите результат.
 - b. Выберете случайным образом *patch* не телесного цвета, чтобы увидеть существенную разницу. Возможно ли удаление фона при подборе правдивой гистограммы телесного цвета?
4. Используя коллекцию изображений рук, смоделируйте гистограмму, по которой можно будет определить в каком из трех условий освещения данное изображение было сделано. Для выполнения этой задачи, вам необходимо создать признаки – возможно, это будет часть сцены, значения яркости и/или относительной яркости (например, от верхнего к нижнему *patch* в кадре) или градиенты от центра к краям.

5. Соберите модели гистограмм телесного цвета при трех условиях освещения.
 - a. Используйте первые во внутреннем, открытом затененном и открытом освещенном гистограммы как модели. Сопоставьте каждую из них со вторыми изображениями в каждой из соответствующей категории, чтобы увидеть насколько хорошо работает оценка совпадения плоти.
 - b. Используйте "детектор сцены" из задания а, чтобы создать модель "переключения гистограмм". Сначала используйте детектор сцены, чтобы определить, какие модели гистограмм задействовать: внутреннюю, открытую затененную, открытую освещенную. Затем используйте соответствующую телесную модель, чтобы принять или отклонить второй телесный *patch* в рамках всех трех условий. Насколько хорошо работает данная модель переключения?
6. Создание региона интереса телесного цвета
 - a. Используйте несколько образцов рук и лица телесного цвета в помещении, чтобы создать гистограмму RGB.
 - b. Используйте *cvCalcBackProject()* для поиска областей телесного цвета
 - c. Используйте *cvErode()* из главы 5, чтобы избавиться от шумов, а затем примените *cvFloodFill()* (из этой же главы) для поиска больших участков телесного цвета на изображении. Это и будут регионы интересов.
7. Попробуйте распознать жест руки. Сфотографируйте руку в 2 футах (~ 60 см) от камеры, создайте несколько (неподвижных) жестов руки: большой палец вверх, влево и вправо.
 - a. Используйте регион интереса из упражнения 6, возьмите изображение градиентов в области руки и создайте модели гистограмм для каждого из трех жестов. Так же создайте гистограмму лица (если лицо есть на изображении). Так же можно получить гистограммы некоторых аналогичных, но не жестовых положений рук.
 - b. Тест распознавания с использованием веб-камеры: используйте регион интереса телесного цвета для поиска "потенциальной руки"; получите градиенты для каждого региона телесного цвета; используйте гистограмму сопоставления превышающая порог для обнаружения жеста. Если две модели выше порога, возьмите лучшее соответствие.
 - c. Переместите руки на 1-2 фута дальше и посмотрите может ли еще распознавать жесты градиент гистограммы.

8. Повторите упражнение 7, но с EMD для сравнения. Что происходит с EMD, когда перемещается рука?
9. Для тех же изображений используйте `cvMatchTemplate()` вместо гистограмм соответствия. Что будет происходить с шаблоном сопоставления, когда вы будете перемещать вашу руку назад так, что она будет становиться меньше?

[П]|[РС]|(РП) Контуры

Хотя такие алгоритмы, как детектор границ Canny могут быть использованы для поиска пикселей границы, отделяющие различные сегменты на изображении, однако, они не могут по этим краям оперировать отдельными элементами. Соответственно следующим логическим шагом будет сбор этих пикселей края в контуры. В OpenCV для этого существует удобная функция `cvFindContours()`. В начале данной главы речь пойдет о некоторых основах, которые необходимы для использования этой функции. В частности, будет введено понятие *memory storages* (хранилище памяти), которое необходимо функциям OpenCV для получения доступа к памяти при динамическом построении объекта; затем речь пойдет об основах *sequences* (последовательностей), которые являются объектами, используемые для отображения контуров. Имея эти знания, можно будет перейти к деталям поиска контура. И в заключении будут рассмотрены некоторые вещи, которые можно выполнять над контурами после их вычисления.

[П]|[РС]|(РП) Хранилище памяти

OpenCV использует объект именуемый ***memory storage*** (хранилище памяти) в качестве метода управления выделением памяти для динамического объекта. *Memory storage* - это связанные списки блоков памяти, которые допускают быстрое распределение и перераспределение непрерывного набора блоков. Функции OpenCV, требующие выделение памяти для их нормального функционирования запрашивают доступ к *memory storage*, благодаря которому можно получить необходимую память (как правило, это функции, результаты которых имеют переменный размер)

Для работы с *memory storage* в OpenCV имеются следующие четыре функции:

```
CvMemStorage* cvCreateMemStorage(
    int block_size = 0
);

void cvReleaseMemStorage(
    CvMemStorage** storage
);

void cvClearMemStorage(
    CvMemStorage* storage
);

void* cvMemStorageAlloc(
    CvMemStorage* storage
    , size_t           size
);
```

Для создания *memory storage* используется функция *cvCreateMemStorage()*. В качестве исходного значения функции передается размер блока, который устанавливает размер блока памяти в хранилище. Если этот аргумент установлен в 0, тогда по умолчанию будет использован блок размера 64kB. Функция возвращает указатель нового хранилища памяти.

Функция *cvReleaseMemStorage()* использует указатель хранилища памяти для освобождения занимаемой памяти. Это, по сути, соответствует освобождению изображений, матриц и других структур.

Для освобождения памяти так же можно воспользоваться *cvClearMemStorage()*, которая тоже принимает указатель на существующее хранилище. Однако необходимо знать одну особенность этой функции: это единственный способ освободить (для повторного использования этого хранилища) память, выделенную под хранилище. Это

может показаться не столь важным, но есть процедуры, которые удаляют объекты внутри хранилища, а занимаемую память не освобождают. Короче говоря, только `cvClearMemStorage()` (и, конечно, `cvReleaseMemStorage()`) могут освобождать занимаемую память. (На самом деле функция `cvRestoreMemStoragePos()` может восстановить память в хранилище. Однако эта функция, прежде всего для внутреннего пользования библиотеки и выходит за рамки данной книги). Удаление любой динамической структуры (`CvSeq`, `CvSet` и т.д.) никогда не возвращает память обратно в хранилище (хотя структуры в состоянии снова использовать память, однажды взятую из хранилища под собственные данные).

Так же возможно выделение собственных блоков из хранилища при помощи функции `cvMemStorageAlloc()` по аналогии с тем, как `malloc()` выделяет память из кучи. В этом случае необходимо будет указать лишь указатель на хранилище и число требуемых байтов. Функция возвратит указатель типа `void*` (подобно `malloc()`).

[П]|[РС]|(РП) Последовательности

Одним из типов объектов, который может хранить хранилище является *последовательность*. Последовательности – это связанные списки различных структур. OpenCV может сделать последовательность из множества различных объектов. В этом смысле можно думать о последовательности как о нечто похожим на общий контейнер классов (или шаблон контейнера классов), который существует в различных языках программирования. Конструкция последовательности в OpenCV выглядит в виде симметричной очереди, так что это позволяет быстро получать произвольный доступ и добавлять/удалять объекты с любого конца очереди, но при этом немного медленнее в случае добавления и удаления объектов в середину очереди.

Непосредственно внутри самой структуры (пример 8-1) есть некоторые важные элементы, о которых необходимо знать. Во-первых, чаще всего будет использоваться элемент *total*. Это общее количество элементов в последовательности. Следующими четырьмя важными элементами являются указатели на другие последовательности: *h_prev*, *h_next*, *v_prev* и *v_next*. Эти четыре указателя являются частью так называемого элемента *CV_TREE_NODE_FIELDS*; они используются не для указания на элемент последовательности, они нужны для соединения различных последовательностей друг с другом. Другие объекты OpenCV так же содержат эти поля узлов дерева. Любые такие объекты могут быть собраны при помощи этих указателей в более сложные супер структуры такие, как списки, деревья и графы. Переменные *h_prev* и *h_next* могут быть использованы для создания простого связного списка. Другие две переменные *v_prev* и *v_next* могут быть использованы для создания более сложной топологии, которая связывает узлы друг с другом. Именно с помощью этих четырех указателей функция *cvFindContours()* способна представить все найденные контуры, составленные в сложную структуру в виде дерева контуров.

Пример 8-1. Внутренняя организация структуры последовательности CvSeq

```

typedef struct CvSeq {
    int          flags;           // различные флаги
    int          header_size;     // размер заголовка последовательности
    CvSeq*      h_prev;          // горизонтально-предыдущая последовательность
    CvSeq*      h_next;          // горизонтально-следующая последовательность
    CvSeq*      v_prev;          // вертикально-предыдущая последовательность
    CvSeq*      v_next;          // вертикально-следующая последовательность
    int          total;           // общее количество элементов
    int          elem_size;       // размер элемента последовательности в байтах
    char*       block_max;        // максимальный предел последнего блока
    char*       ptr;              // текущий указатель записи
    int          delta_elems;     // сколько элементов выделять при росте последовательность
    CvMemStorage* storage;        // где хранить последовательность
    CvSeqBlock* free_blocks;     // список свободных блоков
    CvSeqBlock* first;           // указатель на первый блок последовательности
}

```

Создание последовательности

Как уже было сказано ранее, многие функции OpenCV могут возвращать последовательности. В добавок к этому существует возможность создавать собственные последовательности. Как и для большинства объектов OpenCV, существует функция, создающая последовательность и возвращающая указатель на эту последовательность. Эта функция называется `cvCreateSeq()`:

```

CvSeq* cvCreateSeq(
    int          seq_flags
    ,int          header_size
    ,int          elem_size
    ,CvMemStorage* storage
);

```

Эта функция запрашивает некоторые дополнительные флаги, которые указывают, какая именно последовательность создается. Кроме того необходимо указывать размер заголовка последовательности (всегда `sizeof(CvSeq)`) и размеры объектов, которые должны содержать последовательность. И наконец, необходимо хранилище, которое будет использовано для выделения памяти под последовательность при добавлении нового элемента в последовательность.

Флаги `flags` бывают трех различных категорий и могут быть объединены с помощью оператора побитового `OR`. Первая категория определяет типы объектов, из которых будет создана последовательность. (Типы из данной категории используются редко. Чтобы создать последовательность, элементами которой являются кортежи чисел, нужно использовать `CV_32SC2`, `CV_32FC4` и т.д. Для создания последовательности

элементов собственного типа, необходимо передать 0 и указать правильного размера `elem_size`). Многие из этих типов могут быть не знакомы, а некоторые в первую очередь предназначены для других функций OpenCV. К тому же некоторые флаги имеет смысл использовать только с определенными типами последовательностей (например, `CV_SEQ_FLAG_CLOSED` имеет смысл использовать только для последовательностей, представляющие многоугольники)

```
CV_SEQ_ELTYPE_POINT
(x, y)

CV_SEQ_ELTYPE_CODE
Код Freeman: 0..7

CV_SEQ_ELTYPE_POINT
Указатель на точку: &(x, y)

CV_SEQ_ELTYPE_INDEX
Целочисленный индекс точки: #(x, y)

CV_SEQ_ELTYPE_GRAPH_EDGE
&next_o, &next_d, &vtx_o, &vtx_d

CV_SEQ_ELTYPE_GRAPH_VERTEX
first_edge, &(x, y)

CV_SEQ_ELTYPE_TRIAN_ATR
Вершина бинарного дерева

CV_SEQ_ELTYPE_CONNECTED_COMP
Связная компонента

CV_SEQ_ELTYPE_POINT3D
(x, y, z)
```

Вторая категория указывает на характер последовательности, который может быть любым из ниже перечисленных.

```
CV_SEQ_KIND_SET
Множество объектов

CV_SEQ_KIND_CURVE
Кривая, определенная объектами

CV_SEQ_KIND_BIN_TREE
Бинарное дерево объектов

CV_SEQ_KIND_GRAPH
Граф с объектами в узле
```

Третья категория состоит из флагов, которые указывают на некоторые дополнительные свойства последовательности

CV_SEQ_FLAG_CLOSED	Закрытая последовательность (многоугольники)
CV_SEQ_FLAG_SIMPLE	Простая последовательность (многоугольники)
CV_SEQ_FLAG_CONVEX	Выпуклая последовательность (многоугольники)
CV_SEQ_FLAG_HOLE	Вогнутая последовательность (многоугольники)

Удаление последовательности

```
void cvClearSeq(
    CvSeq* seq
);
```

Для удаления последовательности можно использовать функцию `cvClearSeq()`, которая очистит все элементы последовательности. Однако эта функция не возвращает хранилища или системе выделяемые под них блоки; память, выделенная последовательностью может быть повторно использована только этой же последовательностью. Если необходимо использовать память для других целей, необходимо очистить память хранилища при помощи `cvClearMemStore()`.

Прямой доступ к элементам последовательности

Зачастую будет возникать необходимость прямого доступа к конкретному элементу последовательности. Есть несколько способов сделать это, однако, самый прямой и правильный путь получения доступа к случайно выбранному элементу использовать `cvGetSeqElem()`.

```
char* cvGetSeqElem( seq, index )
```

Так же будет возникать необходимость приводить возвращаемый указатель к любому из типов, из которых может состоять последовательность. Следующий пример показывает как с помощью `cvGetSeqElem()` распечатать элементы последовательности точек (они могут быть возвращены функцией `cvFindContours()`, которая будет рассмотрена чуть позже):

```

for( int i = 0; i < seq->total; ++i ) {
    CvPoint* p = (CvPoint*)cvGetSeqElem( seq, i );
    printf("(%.d,%.d)\n", p->x, p->y );
}

```

В добавок к этому при помощи функции `cvSeqElemIdx()` можно узнать, где в последовательности расположен конкретный элемент:

```

int cvSeqElemIdx(
    const CvSeq*    seq
    ,const void*   element
    ,CvSeqBlock**  block = NULL
);

```

Выполнение данной функции занимает некоторое время, так что это не очень эффективная операция (время поиска пропорционально количеству элементов в последовательности). Стоит обратить внимание на то, что `cvSeqElemIdx()` принимает в качестве аргумента указатель на последовательность и указатель на элемент, который ищется. (Правильнее было бы сказать, что `cvSeqElemIdx()` принимает разыскиваемый указатель. Так происходит, потому что `cvSeqElemIdx()` не ищет элемент в последовательности который равен `element`, а скорее ищет элемент, который находится на месте `element`). Так же можно передать указатель на блок памяти последовательности. Если `block != NULL`, тогда будет возвращено расположение блока, в котором был найден элемент последовательности.

Деление на части, копирование и перемещение данных

Последовательности можно копировать при помощи функции `cvCloneSeq()`, которая выполняет глубокое копирование последовательности, создавая совершенно новую последовательность.

```

CvSeq* cvCloneSeq(
    const CvSeq*    seq
    ,CvMemStorage* storage = NULL
)

```

На самом деле эта функция является "оберткой" для более общей функции `cvSeqSlice()`. Эта функция может "вытащить" часть массива, выполнить глубокое копирование или же просто создать новый заголовок для создания альтернативного "представления" на те же данные.

```
CvSeq* cvSeqSlice(
    const CvSeq* seq
    ,CvSlice slice
    ,CvMemStorage* storage = NULL
    ,int copy_data = 0
);
```

Как можно было бы заметить аргумент *slice* имеет тип *CvSlice*. Фрагмент (*slice*) может быть определен с помощью вспомогательной функции *cvSlice(a, b)*, либо с помощью макроса *CV_WHOLE_SEQ*. В первом случае, в копию будут включены только элементы, начиная с *a* и до *b* (*b* так же может быть установлено в *CV_WHOLE_SEQ_END_INDEX*, чтобы указать конец массива). Аргумент *copy_data* определяет будет ли выполняться "глубокое" копирование (т.е. необходимо ли копировать все элементы данных в новую последовательность).

Фрагменты могут быть использованы для удаления части элементов последовательности с помощью функции *cvSeqRemoveSlice()*, либо для вставки в последовательность новых элементов при помощи *cvSeqInsertSlice()*.

```
void cvSeqRemoveSlice(
    CvSeq* seq
    ,CvSlice slice
);
void cvSeqInsertSlice(
    CvSeq* seq
    ,int before_index
    ,const CvArr* from_arr
);
```

С введением функции сравнения так же появляется возможность сортировки и поиска (в отсортированной) последовательности. Функция сравнения должна иметь следующий прототип:

```
typedef int (*CvCmpFunc)(const void* a, const void* b, void* userdata );
```

a и *b* являются указателями на элементы сортируемого типа, *userdata* - указатель на дополнительные структуры данных, которые вызывают сортировку или поиск во время выполнения. Функция сравнения возвращает -1, если *a* больше *b*, +1, если *a* меньше *b* и 0, если *a* и *b* равны.

Для данной функции сравнения последовательность может быть отсортирована с помощью *cvSeqSort()*. Найти элемент *elem* последовательности (или указатель на него) можно при помощи *cvSeqSearch()*. Поиск занимает O(log n) времени, если

последовательность уже отсортирована (*is_sorted* = 1). Если последовательность не отсортирована, тогда функция сравнения не нужна и поиск займет $O(n)$ времени. По завершении поиска *elem_idx** будет установлен в индекс найденного элемента (если он был найден) и функция вернет указатель на этот элемент. Если элемент не найден, тогда функция вернет NULL.

```
void cvSeqSort(
    CvSeq*      seq
    ,CvCmpFunc   func
    ,void*       userdata = NULL
);

char* cvSeqSearch(
    CvSeq*      seq
    ,const void* elem
    ,CvCmpFunc   func
    ,int         is_sorted
    ,int*        elem_idx
    ,void*       userdata = NULL
);
```

Последовательность может быть инвертирована с помощью функции *cvSeqInvert()*. Эта функция не изменяет сами данные, но она реорганизует последовательность так, что элементы в ней будут переставлены в обратном порядке.

```
void cvSeqInvert(
    CvSeq*  seq
);
```

OpenCV так же поддерживает метод разделения последовательности на основе пользовательского критерия через функцию *cvSeqPartition()*. Для разделения используется та же функция сравнения, описанная выше, но с предположением, что функция будет возвращать ненулевое значение, если два аргумента равны и 0, если они не равны (т.е. в противоположность функции сортировки и поиска).

```
int cvSeqPartition(
    const CvSeq*  seq
    ,CvMemStorage* storage
    ,CvSeq**       labels
    ,CvCmpFunc     is_equal
    ,void*         userdata
);
```

Для обеспечения процесса разделения выделяется память хранилища. Аргумент *labels* должен быть указателем на последовательность указателей. После выполнения *cvSeqPartition()* результатом будет то, что *labels* будет указывать на последовательность целых чисел, которые имеют взаимно-однозначное соответствие с элементами разделённой последовательности *seq*. Значения этих чисел начинаются с нуля и увеличиваются, "имена" разделов те же, что и у указателя *seq*, к которым они присоединены. Указатель *userdata* прозрачно передается функции сравнения.

На рисунке 8-1, группа из 100 точек хаотично распределена на холсте размером 100x100. Для них вызывается функция сравнения *cvSeqPartition()*, основанная на Евклидовом расстоянии. Функция возвращает *true* (1), если расстояние меньше 5 и *false* (0) в противном случае. Полученные кластеры помечаются целыми порядковыми номерами из *labels*.

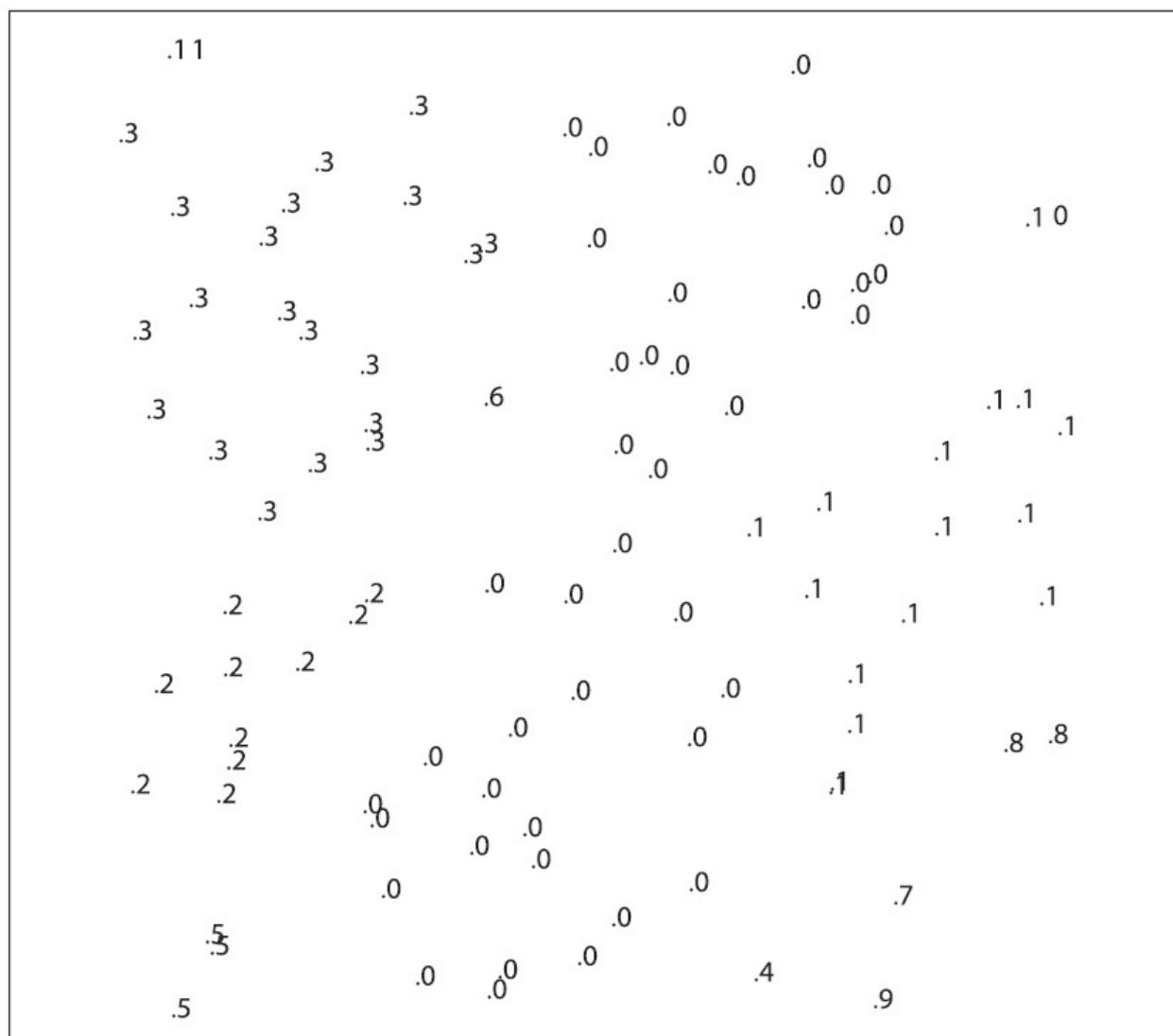


Рисунок 8-1. Последовательность из 100 точек на холсте 100x100, разделенная расстоянием $D \leq 5$

Использование последовательности как стек

Как отмечалось ранее, последовательность в OpenCV - это связный список. Это означает, что последовательность может быть одинаково эффективно доступна с обоих концов. Как результат, последовательность может быть использована в качестве стека. Следующие шесть функций в сочетании со структурой CvSeq могут реализовать поведение, необходимое для использования последовательности в качестве стека (а если точнее, то в качестве очереди, т.к. функции позволяют получать доступ к обоим концам списка).

```

char* cvSeqPush(
    CvSeq* seq
    ,void* element = NULL
);

char* cvSeqPushFront(
    CvSeq* seq
    ,void* element = NULL
);

void cvSeqPop(
    CvSeq* seq
    ,void* element = NULL
);

void cvSeqPopFront(
    CvSeq* seq
    ,void* element = NULL
);

void cvSeqPushMulti(
    CvSeq* seq
    ,void* elements
    ,int count
    ,int in_front = 0
);

void cvSeqPopMulti(
    CvSeq* seq
    ,void* elements
    ,int count
    ,int in_front = 0
);

```

Основные режимы доступа к последовательности осуществляются через `cvSeqPush()`, `cvSeqPushFront()`, `cvSeqPop()` и `cvSeqPopFront()`. Поскольку все эти функции работают с окончанием последовательности, на выполнение затрачивается $O(1)$ времени (т.е. нет зависимости от размера последовательности). `Push` функции возвращают аргумент

элементу, помещенному в последовательность, а *Pop* функции дополнительно сохраняют "выброшенный" элемент, если предоставлен указатель расположения, куда объект может быть скопирован.

Функции *cvSeqPushMulti()* и *cvSeqPopMulti()* записывают или извлекают из последовательности одновременно несколько элементов. Обе функции принимают дополнительный аргумент, чтобы различать начало и конец; можно установить *in_front* либо в *CV_FRONT* (1) либо в *CV_BACK* (0), тем самым определить откуда будут извлекаться или куда будут записываться элементы.

Добавление и удаление элементов

```
char* cvSeqInsert(
    CvSeq* seq
    ,int before_index
    ,void* element = NULL
);

void cvSeqRemove(
    CvSeq* seq
    ,int index
);
```

Объекты могут быть добавлены в и удалены из середины последовательности при помощи функций *cvSeqInsert()* и *cvSeqRemove()*, соответственно, при этом стоит не забывать о том, что это не очень быстро. В среднем это занимает время, пропорциональное общему размеру последовательности.

Размер блока последовательности

cvSetSeqBlockSize() это одна из функций, цель которой может быть не очевидной на первый взгляд. Эта функция в качестве аргумента принимает последовательность и новый размер блока, под блоки будет выделена память из хранилища при появлении новых элементов последовательности. Увеличение размера блока может привести к разрыву последовательности поперек разъединенных блоков памяти; уменьшение размера блока может привести к утечке памяти. По умолчанию размер блока составляет 1000 байтов, но это значение может быть изменено в любое время. (Начиная с 5 beta версии OpenCV, этот размер автоматически увеличивается, если последовательность увеличивается, поэтому не стоит беспокоиться об этом при обычных условиях)

```
void cvSetSeqBlockSize(
    CvSeq* seq
    ,int    delta_elems
);
```

Чтение и запись последовательности

При работе с последовательностями требуется высокая производительность, поэтому существуют специальные методы для доступа и изменения последовательности, что (хотя и требует некоторого внимания во время использования) позволяет выполнять требуемые действия с минимальными накладными расходами. Эти функции используют специальные структуры для отслеживания состояния того, что они делают; благодаря этому имеется возможность для выполнения многих действий над последовательностью, которые в конечном счёте выполняются после завершения выполнения последнего действия.

Для записи контролируемая структура именуется *CvSeqWriter*. Структура инициализируется при помощи *cvStartWriteSeq()*, а "закрывается" при помощи *cvEndWriteSeq()*. Когда последовательность является "открытой", новые элементы могут быть добавлены в последовательность при помощи макроса *CV_WRITE_SEQ()*. Стоит обратить внимание на то, что при записи элемента с помощью макроса не происходит вызов функции, что экономит накладные расходы на заход и выход из функции. Данный подход работает быстрее, чем при использовании *cvSeqPush()*; однако, этот макрос обновляет заголовки последовательности не сразу, поэтому добавляемый элемент будет невидим до тех пор, пока процесс записи не будет завершен. Элемент становится видимым, когда структура будет полностью обновлена при помощи *cvEndWriteSeq()*.

При необходимости структура может быть обновлена без вызова *cvEndWriteSeq()* при помощи *cvFlushSeqWriter()*.

```

void cvStartWriteSeq(
    int           seq_flags
, int           header_size
, int           elem_size
, CvMemStorage* storage
, CvSeqWriter*  writer
);

void cvStartAppendToSeq(
    CvSeq*        seq
, CvSeqWriter*  writer
);

CvSeq* cvEndWriteSeq(
    CvSeqWriter*  writer
);

void cvFlushSeqWriter(
    CvSeqWriter*  writer
);

CV_WRITE_SEQ_ELEM( elem, writer )
CV_WRITE_SEQ_ELEM_VAR( elem_ptr, writer )

```

Аргументы этих функций в значительной степени говорят сами за себя. Аргументы `cvStartWriteSeq()` `seq_flags`, `header_size` и `elem_size` идентичны соответствующим аргументам `cvCreateSeq()`. Функция `cvStartAppendToSeq()` инициализирует запись новых элементов в конце существующей последовательности `seq`. Макросу `CV_WRITE_SEQ_ELEM()` требуется элемент записи (например, `CvPoint`) и указатель на `CvSeqWriter`; новый элемент добавляется в последовательность и затем `elem` копируется в новый элемент.

Как все это работает показано в простом примере, где создается `CvSeqWriter` и добавляются сто случайных точек прямоугольника 320×240 в новую последовательность.

```

CvSeqWriter writer;
cvStartWriteSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage, &writer );

for(i = 0; i < 100; i++) {
    CvPoint pt;
    pt.x = rand()%320;
    pt.y = rand()%240;

    CV_WRITE_SEQ_ELEM( pt, writer );
}

CvSeq* seq = cvEndWriteSeq( &writer );

```

Для чтения есть аналогичный набор функций и ещё несколько связанных макросов.

```

void cvStartReadSeq(
    const CvSeq*     seq
    ,CvSeqReader*   reader
    ,int            reverse = 0
);

int cvGetSeqReaderPos(
    CvSeqReader*   reader
);

void cvSetSeqReaderPos(
    CvSeqReader*   reader
    ,int           index
    ,int           is_relative = 0
);

CV_NEXT_SEQ_ELEM( elem_size, reader )
CV_PREV_SEQ_ELEM( elem_size, reader )
CV_READ_SEQ_ELEM( elem, reader )
CV_REV_READ_SEQ_ELEM( elem, reader )

```

Структура *CvSeqReader* аналогична *CvSeqWriter* и инициализируется при помощи *cvStartReadSeq()*. Аргумент *reverse* позволяет читать последовательность в нормальном (*reverse = 0*) и обратном порядке (*reverse = 1*). Функция *cvGetSeqReaderPos()* возвращает целое число, показывающее текущее положение *reader* в последовательности. И наконец, *cvSetSeqReaderPos()* позволяет задавать произвольное место *reader* в последовательности. Если аргумент *is_relative != NULL*, тогда индекс будет интерпретирован относительно текущей позиции в последовательности. В этом случае индекс может быть и положительным и отрицательным.

Два макроса *CV_NEXT_SEQ_ELEM()* и *CV_PREV_SEQ_ELEM()* просто передвигают позицию *reader* на одну позицию вперед и назад в последовательности. Они не выполняют проверку на ошибки (например, если произошел выход за пределы последовательности). Макросы *CV_READ_SEQ_ELEM()* и *CV_REV_READ_SEQ_ELEM()* используются для чтения элементов из последовательности. Они копируют "текущий" элемент, на который *reader* указал в переменной *elem*, а затем сдвигают индекс на один шаг вперед или назад соответственно. Два последних макроса ожидают только имя переменной, куда будет скопирован элемент; адрес этой переменной будет вычисляться внутри макроса.

Последовательности и массивы

Зачастую возникает желание преобразовать последовательность (как правило, полную точек) в массив.

```
void* cvCvtSeqToArray(
    const CvSeq* seq
    ,void* elements
    ,CvSlice slice = CV_WHOLE_SEQ
);

CvSeq* cvMakeSeqHeaderForArray(
    int seq_type
    ,int header_size
    ,int elem_size
    ,void* elements
    ,int total
    ,CvSeq* seq
    ,CvSeqBlock* block
);
```

Функция `cvCvtSeqToArray()` копирует содержимое последовательности в непрерывный массив памяти. Это означает, что если имеется последовательность из 20 элементов типа `CvPoint`, то функции потребуется указатель на массив `elements` достаточный для хранения 40 целых чисел. Третий (необязательный) аргумент `slice` может быть объектом типа `CvSlice` или макросом `CV_WHOLE_SEQ` (значение по умолчанию). Если выбран макрос `CV_WHOLE_SEQ`, то копируется вся последовательность.

Противоположную `cvCvtSeqToArray()` функциональность реализует `cvMakeSeqHeaderForArray()`. В этом случае, можно собрать последовательность из существующего массива. Несколько первых аргументов данной функции идентичны аргументам `cvCreateSeq()`. В дополнение к запрашиваемым данным (`elements`) для копирования и их числа (`total`), можно предоставить заголовок последовательности (`seq`) и структуру блока памяти последовательности (`block`). Последовательность, созданная таким образом, отличается от последовательностей, созданных при помощи других методов. В частности, в дальнейшем данные данной последовательности не могут быть изменены.

[П]||[РС]||(РП) Поиск контуров

Теперь настало время поговорить о *контурах*. Для начала необходимо дать точное определение тому, что такое контур. Контур – это список точек, которые в той или иной форме представляют кривую на изображении. Это представление может быть различным в зависимости от обстоятельств. Есть множество способов для представления кривой. В OpenCV контуры представлены последовательностями, в которых каждая запись содержит информацию о нахождении следующей точки кривой. Чуть позже будут рассмотрены детали, а сейчас достаточно понять, что контур в OpenCV представлен последовательностью *CvSeq*, которая, так или иначе, является последовательностью точек.

Функция *cvFindContours()* вычисляет контуры из бинарных изображений. Она может принимать изображения, созданные при помощи *cvCanny()*, которые имеют граничные пиксели или изображения, созданные при помощи *cvThreshold()* или *cvAdaptiveThreshold()*, у которых края представлены в виде границы между положительными и отрицательными регионами.

Перед переходом к прототипу функции поиска контуров, необходимо осознать, что такое контур. При разборе работы функции будет введено понятие дерево контура для понимания того, что функция *cvFindContours()* будет возвращать в качестве результата.

На рисунке 8-2 показана функциональность функции *cvFindContours()*. В верхней части рисунка находится тестовое изображение, содержащее несколько белых регионов (обозначенные буквами от *A* до *E*) на тёмном фоне. (Для ясности на рисунке тёмные области изображены серым, так проще представить, что изображение подвергнуто пороговому преобразованию, так что серые области устанавливаются в черный цвет только при передачи в функцию *cvFindContours()*). В нижней части рисунка находится тоже изображение, но уже с выделенными контурами, которые обозначены *cX* или *hX*, где "c" расшифровывается как "contour" (контур), "h" расшифровывается как "hole" (отверстие), а "X" это некое число. Некоторые из этих контуров нарисованы пунктирной линией; они представляют *внешние границы* белых регионов (т.е. ненулевых регионов). OpenCV и *cvFindContours()* различают эти внешние границы и пунктирные линии, которые можно представить как *внутренние границы* или как *внешние границы отверстий* (т.е. нулевых регионов).

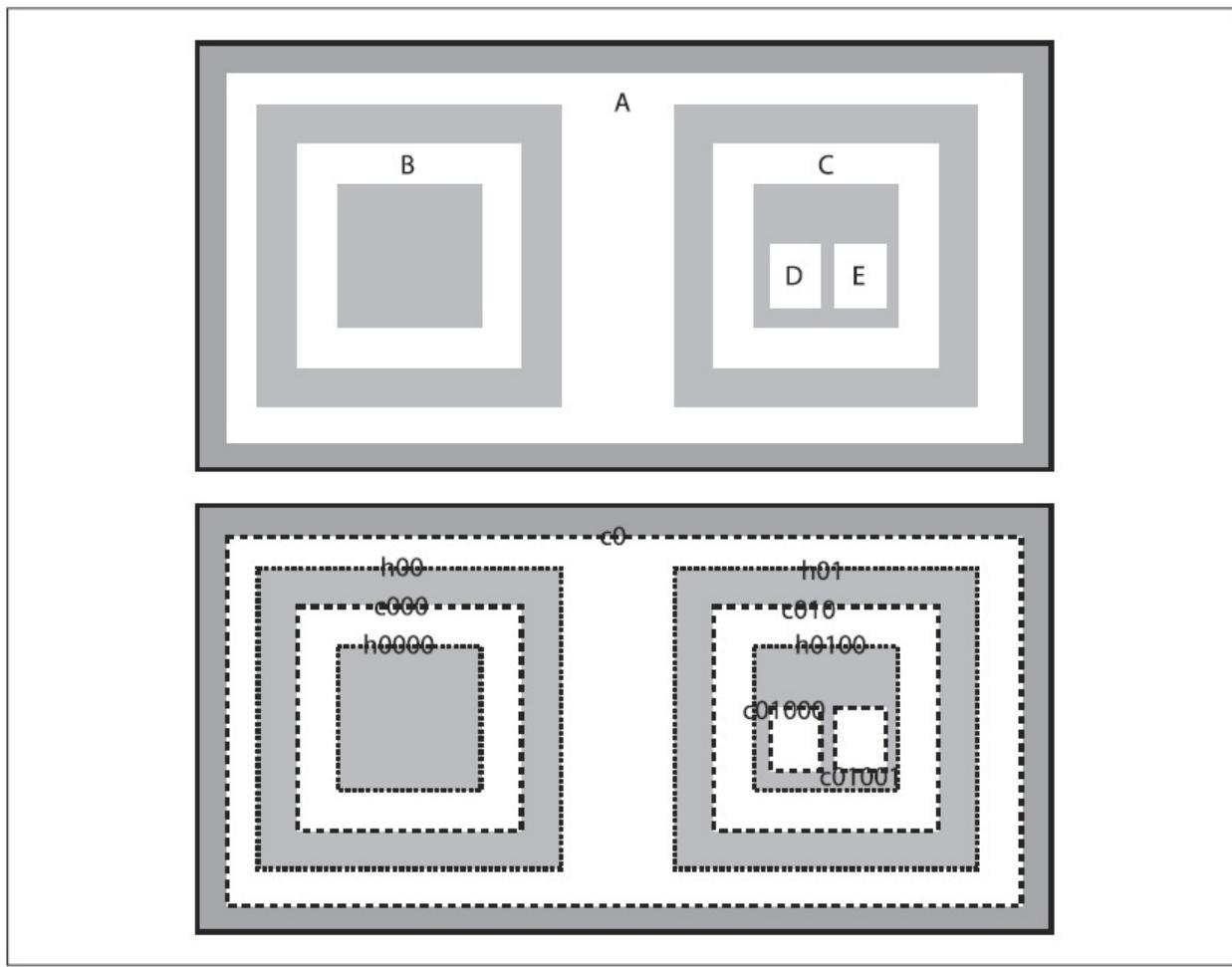


Рисунок 8-2. Тестовое изображение (сверху) переданное `cvFindContours()` (снизу): найденные контуры могут быть одного из двух типов, внешние контуры (штриховые линии) или отверстия (пунктирные линии)

Концепция вложенности играет важную роль во многих приложениях. По этой причине OpenCV может собрать найденные контуры в дерево контуров, которое отражает отношение вложенности контуров в своей структуре. Дерево контуров для тестового изображения будет иметь контур `c0` как корневой узел, с отверстиями `h00` и `h01` в качестве потомков. Эти отверстия в свою очередь так же могут иметь потомков и т.д.

Последствия использования `cvFindContours()` на изображении, генерируемое `cvCanny()` или аналогичными детекторами краев, относительны к бинарному изображению, такому как тестовое изображение на рисунке 8-1. На самом деле `cvFindContours()` ничего не знает о контурах. Это означает, что для `cvFindContours()` "край" это просто очень тонкая "белая" область. В результате для каждого внешнего контура имеется почти точно совпадающий внутренний контур. На самом деле этот внутренний контур просто находится внутри внешней границы. Можно думать об этом как о переходе от белого к черному региону, который отмечает внутренний край края.

Теперь пришло время перейти к рассмотрению самой функции `cvFindContours()`: выяснить как передать ей то что нужно и как интерпретировать результаты.

```

int cvFindContours(
    IplImage*           img
    , CvMemStorage*     storage
    , CvSeq**           firstContour
    , int               headerSize = sizeof(CvContour)
    , CvContourRetrievalMode mode      = CV_RETR_LIST
    , CvChainApproxMethod method     = CV_CHAIN_APPROX_SIMPLE
);

```

Первый аргумент - это исходное изображение; оно должно быть 8-ми битным одноканальным изображением и интерпретироваться как бинарное (т.е. все ненулевые пиксели должны быть эквивалентны друг к другу). После запуска процесса вычислений, функция *cvFindContours()* использует это изображение как рабочее пространство, поэтому, если необходимо иметь неизмененное исходное изображение, то в функцию необходимо передать копию изображения. Следующий аргумент *storage* указывает место, где функция может найти память для сохранения контуров. Это хранилище должно быть создано с помощью функции *cvCreateMemStorage()*. Следующий аргумент *firstContour* является указателем на *CvSeq**. Функция *cvFindContours()* сама инициализирует этот указатель. Поэтому достаточно только передать указатель на указатель. Операции выделения/удаления (*new/delete* или *malloc/free*) не нужны. Именно этот аргумент (*firstContour*) является указателем на корень дерева контуров. (Деревья контуров являются лишь одним из способ организации найденных контуров. В любом случае они будут организованы с помощью элементов контуров *CV_TREE_NODE_FIELDS*, про которые шла речь в самом начале знакомства с последовательностями). Функция возвращает общее количество найденных контуров.

```

vSeq* firstContour = NULL;
cvFindContours( ..., &firstContour, ... );

```

Аргумент *headerSize* сообщает *cvFindContours()* о размерах объектов, которые функция будет создавать; он может быть установлен в *sizeof(CvContour)* или в *sizeof(CvChain)* (последний используется при установленном методе аппроксимации в *CV_CHAIN_CODE*). (Фактически *headerSize* может быть любым числом, которое больше или равно перечисленным значениям). И в заключении, аргументы *mode* и *method* уточняют что должно быть вычислено и как (соответственно).

Аргумент *mode* может быть установлен в одно из следующих значений: *CV_RETR_EXTERNAL*, *CV_RETR_LIST*, *CV_RETR_CCOMP* или *CV_RETR_TREE*.

Значение *mode* указывает *cvFindContours()* какие контуры необходимо найти и в каком виде необходимо получить результат. В частности, способ использования переменных узлов дерева (*h_prev*, *h_next*, *v_prev* и *v_next*) для "соединения" найденных контуров

определяется значением *mode*. Рисунок 8-3 отображает результирующие топологии для всех четырех возможных значений *mode*. В каждом из случаев, структуры могут рассматриваться как "уровни", которые связаны "горизонтальными" связями (*h_next* и *h_prev*) и отделены друг от друга *вертикальными* связями (*v_prev* и *v_next*).

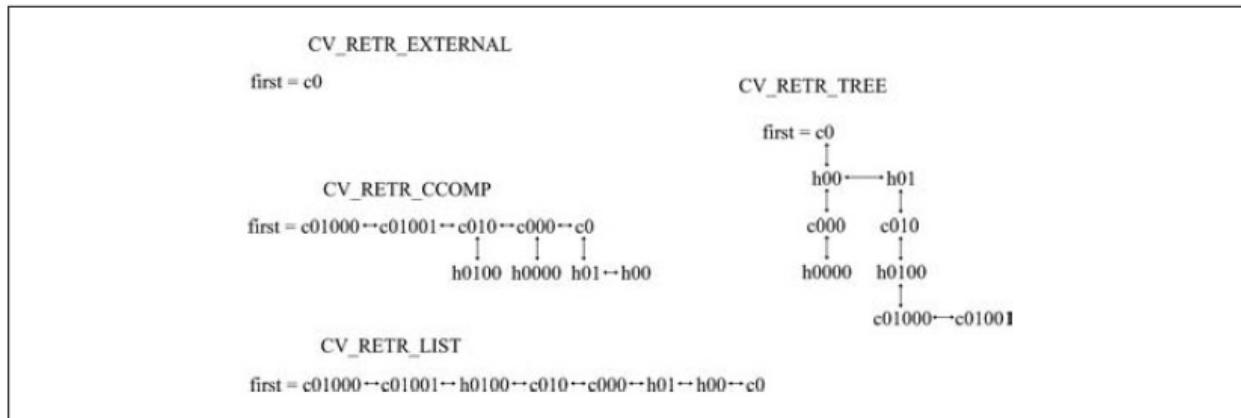


Рисунок 8-3. Способы соединения переменных вершин дерева, найденных `cvFindContours()`

CV_RETR_EXTERNAL

Извлечение только крайних внешних контуров. На рисунке 8-2 есть только один внешний контур, поэтому на рисунке 8-3 показаны точки только первого контура внешней последовательности, не имеющей никаких дальнейших связей.

CV_RETR_LIST

Извлечение всех контуров и размещение их в списке. Рисунок 8-3 отображает список, полученный в результате обработки тестового изображения с рисунка 8-2. В этом случае найдено восемь контуров, и все они связаны друг с другом *h_prev* и *h_next* (*v_prev* и *v_next* не используются).

CV_RETR_CCOMP

Извлечение всех контуров и организация их в двухуровневую иерархию, где границы верхнего уровня являются внешними границами компонентов, а границы второго уровня являются границами отверстий. По рисунку 8-3 можно видеть, что есть пять внешних границ, три из которых содержат отверстия. Отверстия связаны с соответствующими внешними границами при помощи *v_prev* и *v_next*. Внешняя граница `c0` содержит два отверстия. Т.к. *v_next* может содержать только одно значение, то узел может иметь только одного наследника. Все отверстия внутри `c0` связаны друг с другом указателями *h_prev* и *h_next*.

CV_RETR_TREE

Извлечение всех контуров и восстановление полной иерархии вложенных контуров. В рассматриваемом примере (рисунок 8-2 и 8-3) это означает, что корневой узел является внешним контуром *c0*. Следом за *c0* идет отверстие *h00*, соединенное с другим отверстием *h01* на том же уровне. Каждое из этих отверстий в свою очередь имеет наследников (контуры *c000* и *c001* соответственно), которые связаны с их родителями вертикальными связями. Так продолжается вниз по иерархии вплоть до самых внутренних контуров изображения, которые становятся листьями деревьев.

Следующие пять флагов относятся к *method* (т.е. каким образом контуры аппроксимируются).

CV_CHAIN_CODE

Внешние контуры в виде кодовой цепочки Фримана; все другие методы выводят полигоны (последовательности вершин)

CV_CHAIN_APPROX_NONE

Перевод кодовой цепочки в точки.

CV_CHAIN_APPROX_SIMPLE

Сжатие горизонтальных, вертикальных и диагональных сегментов, оставляя только их угловые точки.

CV_CHAIN_APPROX_TC89_L1 или *CV_CHAIN_APPROX_TC89_KCOS*

Применение одной из разновидностей алгоритма аппроксимации цепочек Teh-Chin.

CV_LINK_RUNS

Комплексный дифференциальный алгоритм (один из выше перечисленных), который связывает горизонтальные сегменты 1s.

Контуры последовательности

Существует огромное количество различных контуров и последовательностей.

Хорошая новость в том, что для текущих задач потребуется небольшое количество из них. В результате вызова *cvFindContours()* получается довольно таки много различных последовательностей. Все эти последовательности имеют один конкретный тип; тип зависит от аргументов, передаваемых в *cvFindContours()*. По умолчанию используется режим *CV_RETR_LIST* и метод *CV_CHAIN_APPROX_SIMPLE*.

Эти последовательности являются последовательностями точек; точнее контурами.

Главное, что нужно помнить о контурах – это частный случай последовательности.

(Тип *CvContour* не совпадает с *CvSeq*. По сути *CvContour* унаследован от *CvSeq* и

имеет несколько дополнительных членов, а именно цвет *color* и ограничительную рамку типа *CvRect*). Контур – это последовательность точек, представляющая некую кривую в пространстве. Такие цепочки точек встречаются довольно таки часто, так что существуют специальные функции, помогающие манипулировать ими.

```

int cvFindContours(
    CvArr*           image
    ,CvMemStorage*   storage
    ,CvSeq**         first_contour
    ,int             header_size = sizeof(CvContour)
    ,int             mode      = CV_RETR_LIST
    ,int             method    = CV_CHAIN_APPROX_SIMPLE
    ,CvPoint          offset    = cvPoint(0,0)
);

CvContourScanner cvStartFindContours(
    CvArr*           image
    ,CvMemStorage*   storage
    ,int             header_size = sizeof(CvContour)
    ,int             mode      = CV_RETR_LIST
    ,int             method    = CV_CHAIN_APPROX_SIMPLE
    ,CvPoint          offset    = cvPoint(0,0)
);

CvSeq* cvFindNextContour(
    CvContourScanner  scanner
);

void cvSubstituteContour(
    CvContourScanner  scanner
    ,CvSeq*           new_contour
);

CvSeq* cvEndFindContour(
    CvContourScanner* scanner
);

CvSeq* cvApproxChains(
    CvSeq*           src_seq
    ,CvMemStorage*   storage
    ,int              method        = CV_CHAIN_APPROX_SIMPLE
    ,double           parameter    = 0
    ,int              minimal_perimeter = 0
    ,int              recursive     = 0
);

```

Первую функцию *cvFindContours()* уже доводилось видеть ранее. Вторая функция *cvStartFindContours()* тесно связана с *cvFindContours()* и используется, если необходимо обрабатывать контуры по одному, в то время как они упакованы в более

высокоуровневую структуру. Функция `cvStartFindContours()` возвращает `CvSequenceScanner`. Сканер содержит информацию о том, что может быть прочитано, а что нет. (Важно не путать `CvSequenceScanner` с `CvSeqReader`. Последний используется для чтения элементов последовательности, в то время, как `CvSequenceScanner` используется для чтения того, что по сути является списком последовательностей). В последующем можно вызвать `cvFindNextContour()` и последовательно получить все найденные контуры. Если вернется `NULL`, то это будет означать, что контуров больше не осталось.

Функция `cvSubstituteContour()` позволяет заменять контур, на который в данный момент указывает сканер на какой-либо другой контур. Полезной особенностью этой функции является то, что если `new_contour = NULL`, то текущий контур будет удален из цепи или дерева, на который указывает сканер (последовательность будет обновлена соответствующим образом, так что не будет указателей на несуществующие объекты).

И в заключении, функция `cvEndFindContour()` завершает сканирование и устанавливает сканер в состояние "выполнено". При этом сканируемая последовательность не удаляется; на самом деле, значение, возвращаемое `cvEndFindContour()` является указателем на первый элемент последовательности.

Функция `cvApproxChains()` преобразует коды Фримена к полигональному представлению (точно или с некоторым приближением). Более подробно данная функция будет рассмотрена далее в этой главе (раздел "Полигон приближения").

Цепной код Фримена

Как правило, контуры, созданные при помощи `cvFindContours()` являются последовательностями вершин (т.е. точек). Альтернативное представление может быть получено путем установки `method = CV_CHAIN_CODE`. В этом случае, найденные контуры будут храниться в виде **цепи Фримена** (Рисунок 8-4). Цепь Фримена – это полигон, представляющий последовательность шагов в одном из 8-ми направлений; каждый шаг обозначается целым числом от 0 до 7. При работе с цепями Фримена можно читать их содержимое при помощи двух "вспомогательных" функций:

```
void cvStartReadChainPoints(
    CvChain*          chain
    ,CvChainPtReader* reader
);

CvPoint cvReadChainPoint(
    CvChainPtReader* reader
);
```

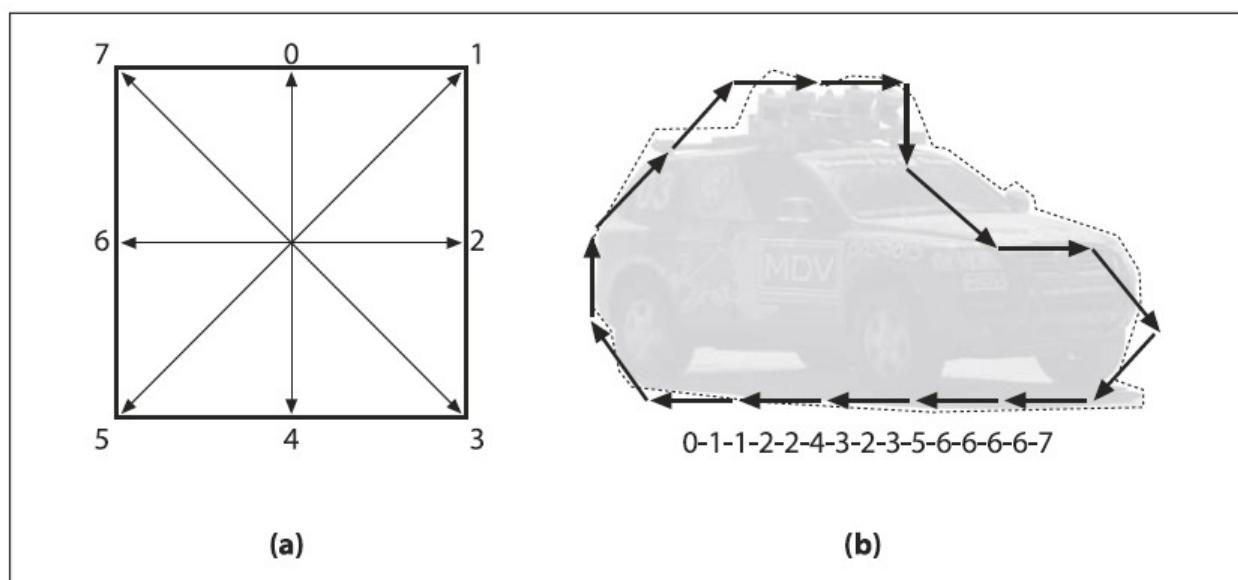


Рисунок 8-4. (а) Направления цепи Фримена, пронумерованные целыми числами 0-7; (б) Контур, конвертируемый в цепь Фримена, начиная с заднего бампера

Первая функция принимает цепь в качестве аргумента, а вторая функция цепь читает. Структура *CvChain* является формой *CvSeq*. *CvChainPtReader* перебирает один контур, представленный кодом Фримена, так же как *CvContourScanner* перебирает различные контуры. В связи с этим, итератор *CvChainPtReader* схож с более обобщенным итератором *CvSeqReader*, а *cvStartReadChainPoints* играет роль *cvStartReadSeq*. Как и следовало ожидать, *CvChainPtReader* возвращает NULL, когда читать больше нечего.

Рисование контуров

Одной из наиболее важных задач является рисование контура на экране. Для этого в OpenCV есть *cvDrawContours()*:

```
void cvDrawContours(
    CvArr*      img
    ,CvSeq*      contour
    ,CvScalar    external_color
    ,CvScalar    hole_color
    ,int         max_level
    ,int         thickness   = 1
    ,int         line_type   = 8
    ,CvPoint     offset       = cvPoint(0,0)
);
```

Первый аргумент – это изображение, на котором будет рисоваться контур. Следующий аргумент *contour* не столь прост, как кажется. В частности, он рассматривается как корневой узел дерева контуров, а другие аргументы (в первую очередь *max_level*)

определяют, что должно быть сделано с остальной часть деревом. Следующий аргумент довольно прост: цвет, которым будет рисоваться контур. Но что насчёт *hole_color*? Напомним, что OpenCV различает внешние и внутренние контуры (штриховые и пунктирные линии, представленные на рисунке 8-2). При рисовании одного контура или дерева контуров, любой контур помеченный как "отверстие" (внутренний) будет рисоваться этим альтернативным цветом.

Аргумент *max_level* сообщает *cvDrawContours()* как обращаться с любыми контурами, которые могут быть присоединены к *contour* с помощью переменных вершин дерева. Этот аргумент может быть установлен для того, чтобы указать максимальную глубину прохода на рисунке. Таким образом, *max_level* = 0 означает, что будут взяты все контуры на том же уровне что и входной уровень (точнее, входной контур и контуры, следующие за ним), *max_level* = 1 означает, что будут взяты все контуры и их потомки на том же уровне что и входной, и так далее. Если функция *cvFindContours()* вызывается с аргументом *mode* равным *CV_RETR_CCOMP* или *CV_RETR_TREE*, то *max_level* может иметь отрицательное значение. В этом случае *max_level* = -1 означает, что будет нарисован только входной контур, *max_level* = -2 означает, что будет нарисован входной контур и его прямой потомок, и так далее. Пример использования можно найти в *.../opencv/samples/c/contours.c*.

Параметры *thickness* и *line_type* имеют свои обычные значения. Можно так же задействовать *offset* для того, чтобы контур был нарисован в другом месте относительно абсолютных координат, по которым он определяется. Эта особенность весьма полезна, когда контур уже был преобразован к системе центр-масса или каким-либо другим локальным координатам.

Аргумент *offset* так же будет полезен, если использовать функцию *cvFindContours()* один или несколько раз в различных регионах изображения (ROIs), с последующим отображением полученных результатов на исходном изображении. В противоположность этому, можно использовать *offset* для извлечения контура из исходного изображения, с последующим созданием небольшой маски для этого контура.

Пример работы с контуром

Приведенный пример взят из пакета OpenCV. Вначале создается окно с изображением в нем. Ползунок задает значение порога, а контуры рисуются на изображении после порогового преобразования. Конечное изображение обновляется каждый раз при изменении положения ползунка.

Пример 8-2. Поиск контуров на изображение в зависимости от положения ползунка; изображение обновляется при изменении положения ползунка

```

#include <cv.h>
#include <highgui.h>

IplImage* g_image = NULL;
IplImage* g_gray = NULL;
int g_thresh = 100;
CvMemStorage* g_storage = NULL;

void on_trackbar(int) {
    if( g_storage == NULL ) {
        g_gray = cvCreateImage( cvGetSize(g_image), 8, 1 );
        g_storage = cvCreateMemStorage(0);
    } else {
        cvClearMemStorage( g_storage );
    }

    CvSeq* contours = 0;
    cvCvtColor( g_image, g_gray, CV_BGR2GRAY );
    cvThreshold( g_gray, g_gray, g_thresh, 255, CV_THRESH_BINARY );
    cvFindContours( g_gray, g_storage, &contours );
    cvZero( g_gray );

    if( contours ) {
        cvDrawContours(
            g_gray
            ,contours
            ,cvScalarAll(255)
            ,cvScalarAll(255)
            ,100
        );
    }
}

cvShowImage( "Contours", g_gray );
}

int main( int argc, char** argv ) {
    if( argc != 2 || !(g_image = cvLoadImage(argv[1])) ) {
        return -1;
    }

    cvNamedWindow( "Contours", 1 );

    cvCreateTrackbar(
        "Threshold"
        , "Contours"
        ,&g_thresh
        ,255
        ,on_trackbar
    );

    on_trackbar(0);
}

```

```
cvWaitKey();  
  
    return 0;  
}
```

Все самое интересное происходит внутри функции *on_trackbar()*. Если глобальная переменная *g_storage* имеет начальное значение (NULL), то *cvCreateMemStorage(0)* создает хранилище памяти, *g_gray* инициализируется пустым изображением того же размера, что и *g_image*, но только с одним каналом. Если *g_storage != NULL*, то хранилище очищается для повторного использования. Далее создается указатель *CvSeq**; он будет указывать на последовательность, созданную при помощи *cvFindContours()*.

Далее *g_image* конвертируется в серое изображение и подвергается пороговому преобразованию: все пиксели, которые ярче *g_thresh* сохраняют ненулевые значения. Функция *cvFindContours()* применяется к изображению, прошедшеме пороговое преобразование. Если контуры были найдены (т.е. если *contours != NULL*), тогда функция *cvDrawContours()* рисует (белые) контуры на сером изображении. В завершении, изображение отображается на экране, а память под структуры, выделенная в начале обратного вызова, освобождается.

[П]|[РС]||(РП) Пример поиска контура

В этом примере происходит поиск контуров на исходном изображении с последующим их рисованием друг за другом. Это хороший пример для того, чтобы поиграться с различными параметрами, в частности, посмотреть к чему приведут изменения режима поиска контуров (в примере используется `CV_RETR_LIST`) или `max_depth`, используемый для рисования контуров (в примере используется 0). Если `max_depth` будет иметь довольно таки большое значение, то возвращаемые функцией `cvFindContours()` контуры будут связаны при помощи `h_next`. Это означает, что для некоторых топологий (`CV_RETR_TREE`, `CV_RETR_CCOMP`) можно увидеть один и тот же контур несколько раз.

Пример 8-3. Поиск и рисование контуров на исходном изображении

```

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

    cvNamedWindow( argv[0], 1 );

    IplImage* img_8uc1 = cvLoadImage( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
    IplImage* img_edge = cvCreateImage( cvGetSize(img_8uc1), 8, 1 );
    IplImage* img_8uc3 = cvCreateImage( cvGetSize(img_8uc1), 8, 3 );

    cvThreshold( img_8uc1, img_edge, 128, 255, CV_THRESH_BINARY );

    CvMemStorage* storage = cvCreateMemStorage();
    CvSeq* first_contour = NULL;

    int Nc = cvFindContours(
        img_edge
        ,storage
        ,&first_contour
        ,sizeof(CvContour)
        ,CV_RETR_LIST      // Попробуйте все четыре значения и посмотрите, что получится
    );

    int n=0;
    printf( "Total Contours Detected: %d\n", Nc );

    for( CvSeq* c = first_contour; c != NULL; c = c->h_next ) {
        cvCvtColor( img_8uc1, img_8uc3, CV_GRAY2BGR );

        cvDrawContours(
            img_8uc3
            ,c
            ,CVX_RED
            ,CVX_BLUE
            ,0      // Попробуйте различные значения и посмотрите, что получится
        );
    }
}

```

```
    , 2
    , 8
);

printf("Contour #%d\n", n );
cvShowImage( argv[0], img_8uc3 );

printf(" %d elements:\n", c->total );
for( int i=0; i<c->total; ++i ) {
    CvPoint* p = CV_GET_SEQ_ELEM( CvPoint, c, i );
    printf(" (%d,%d)\n", p->x, p->y );
}

cvWaitKey(0);
n++;
}

printf("Finished all contours.\n");
cvCvtColor( img_8uc1, img_8uc3, CV_GRAY2BGR );
cvShowImage( argv[0], img_8uc3 );
cvWaitKey(0);

cvDestroyWindow( argv[0] );

cvReleaseImage( &img_8uc1 );
cvReleaseImage( &img_8uc3 );
cvReleaseImage( &img_edge );

return 0;
}
```

[П]|[РС]|(РП) Другие манипуляции с контурами

Во время анализа изображения можно производить множество различных вещей с контурами. В конце концов, большинство контуров являются тем, что интересно идентифицировать и манипулировать. Различные вариационные задачи включают в себя конкретизацию контуров различными путями, их упрощение или приближение, сопоставление их с шаблонами и так далее.

В этом разделе будут рассмотрены некоторые из этих задач и различные функции OpenCV, которые либо выполняют некоторые вещи за нас, либо предоставляют некоторый функционал для упрощения наших задач.

Полигон приближений

Если необходимо нарисовать контур или проанализировать его форму, то полигон приближений является обобщенным приближением контура, представляющий из себя многоугольник с другим контуром, имеющий меньшее количество вершин. Есть несколько способов получения такого полигона; OpenCV предоставляет один из возможных вариантов (если быть более точными, то OpenCV реализует алгоритм приближения Douglas-Peucker; другими наиболее популярными являются алгоритмы Rosenfeld-Johnson и Teh-Chin). Функция `cvApproxPoly()` реализует алгоритм, который работает с последовательностью контуров:

```
CvSeq* cvApproxPoly(
    const void*      src_seq
    ,int             header_size
    ,CvMemStorage*   storage
    ,int             method
    ,double          parameter
    ,int             recursive = 0
);
```

Данной функции можно передать последовательность контуров в виде списка или дерева, при этом обработке подвергнутся все контуры последовательности. Функция возвращает указатель на первый контур последовательности, для перехода к другим контурам можно воспользоваться `h_next` (или `v_next`).

Т.к. функция возвращает указатель на создаваемые ею объекты, то ей необходимо передавать указатель на `CvMemStorage` и размер заголовка (как правило, он равен `sizeof(CvContour)`).

Аргумент *method* всегда имеет значение *CV_POLY_APPROX_DP* (хотя существует возможность выбора иного алгоритма при их появлении). Следующие два аргумента специфичны для метода (из которых на момент написания книги существует только один). Аргумент *parameter* задает точность для алгоритма. Для понимания работы данного аргумента необходимо рассмотрение конкретного алгоритма (если это достаточно сложно, просто установите этот параметр в малую часть от общей длины кривой). Последний аргумент указывает на необходимость применения алгоритма к каждому контуру, к которым можно получить доступ с помощью указателей *h_next* или *v_next*. Если этот аргумент имеет значение 0, то только контур, на который указывает *src_seq*, подвергнется приближению.

Теперь рассмотрим, как работает алгоритм, заложенный в данную функцию. На рисунке 8-5, начиная с контура *a* (часть *b*) алгоритм выбирает две точки экстремума и соединяет их линией (часть *c*). Затем в исходном многоугольнике ищется наиболее удаленная точка от этой линии, и эта точка добавляется к приближению. Процесс повторяется (часть *d*), добавляя следующую наиболее удаленную точку к накопительному приближению, пока все точки, для которых значение расстояния меньше указанного в параметре *precision*, не будут добавлены (часть *f*). Это означает, что хорошими кандидатами для параметра будут точки на некотором отрезке контура или на ограниченном (ROI) участке контура или что-то соизмеримое с контуром.

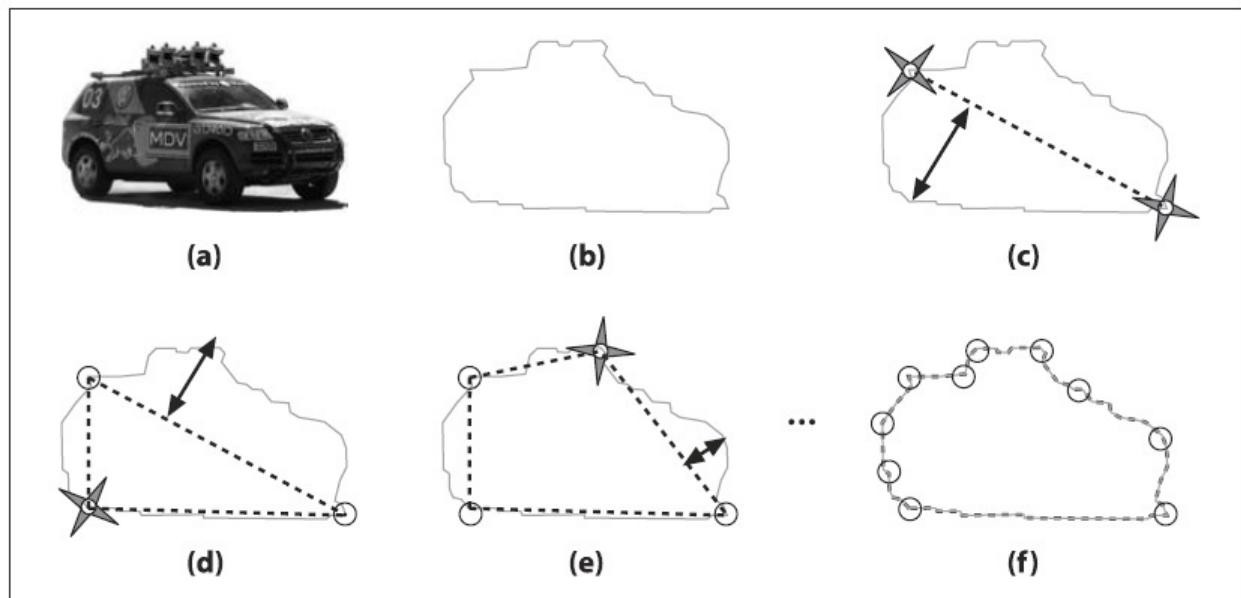


Рисунок 8-5. Наглядное представление алгоритма DP, используемого в *cvApproxPoly()*: для исходного изображения (a) выполняется аппроксимация контура (b), с последующим, начиная с первых максимально удаленных друг от друга вершин (c), отбором максимально удаленных вершин от линии, которая соединяет две вершины (d-f)

С только что описанным приближением тесно связан процесс нахождения доминирующих точек. *Доминирующая точка* определяется как точка, которая имеет наибольшее количество информации в сравнении с другими точками кривой. Доминирующие точки используются во многих ситуациях, схожих с полигоном приближений. Функция *cvFindDominantPoints()* реализует так называемый алгоритм IPAN (Image and Pattern Analysis Group, венгерская академия наук; алгоритм зачастую упоминается как IPAN99, т.к. впервые он был опубликован в 1999 году).

```
CvSeq* cvFindDominantPoints(
    CvSeq* contour
    ,CvMemStorage* storage
    ,int method      = CV_DOMINANT_IPAN
    ,double parameter1 = 0
    ,double parameter2 = 0
    ,double parameter3 = 0
    ,double parameter4 = 0
);
```

По сути, алгоритм IPAN пытается построить треугольники на внутренней кривой с помощью доступных вершин, путем сканирования вдоль контура. Треугольник характеризуется размером и углом раскрытия (рисунок 8-6). Точки с большими углами раскрытия сохраняются при условии, что их размеры меньше указанного глобального порога и меньше своих соседей.

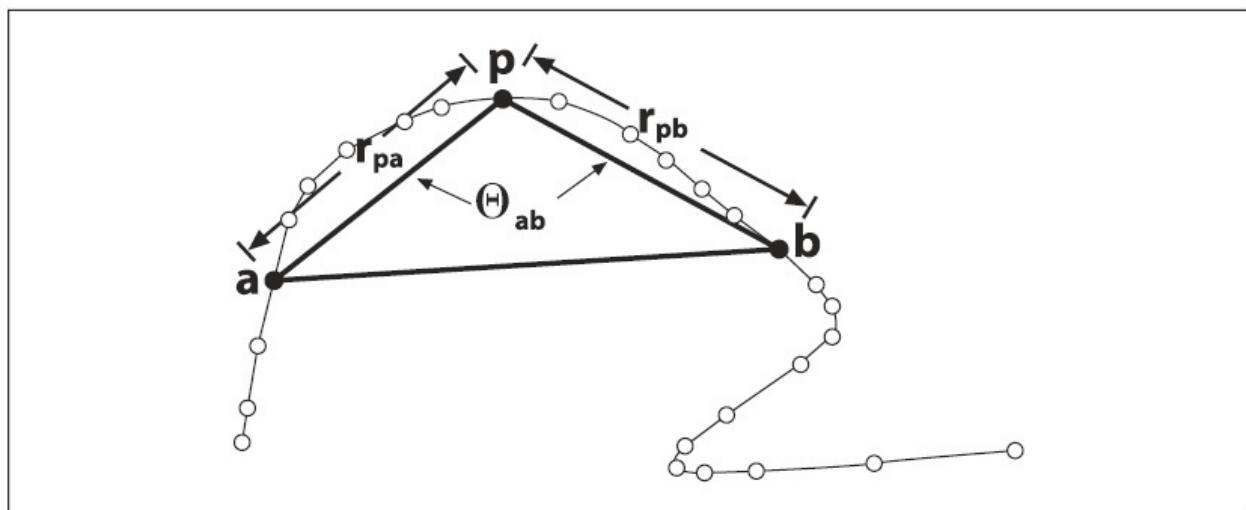


Рисунок 8-6. Алгоритм IPAN использует треугольник abp для конкретизации точки p

Первые два аргумента функции *cvFindDominantPoints()* *CvSeq** и *CvMemStorage** это последовательность контуров и хранилище соответственно. Следующий аргумент *method* на момент написания книги имеет одно единственное значение *CV_DOMINANT_IPAN*.

Следующие четыре аргумента: минимальное расстояние d_{\min} , максимальное расстояние d_{\max} , расстояние до соседей d_n и максимальный угол θ_{\max} . Как показано на рисунке 8-6 алгоритм вначале строит все треугольники для которых r_{pa} и r_{pb} находятся между d_{\min} и d_{\max} и для которых $\theta_{ab} < \theta_{\max}$. На втором проходе сохраняются только точки p с наименьшим значением связывания θ_{ab} в окрестности d_n (значение d_n никогда не должно превышать d_{\max}). Типичное значение для d_{\min} , d_{\max} , d_n и θ_{\max} : 7, 9, 9, 150 (угол измеряется в градусах).

Сводная характеристика

Другая задача, с которой часто приходиться сталкиваться при работе с контурами - это вычисление различных характеристик контура. Это может быть длина или форма контура. Другой полезной характеристикой является момент контура, который может быть использован для обобщения формы контура (об этом пойдет речь в следующем разделе)

Длина

Функция `cvContourPerimeter()` принимает контур, а возвращает его длину. На самом деле эта функция - макрос для более обобщенной функции `cvArcLength()`.

```
double cvArcLength(
    const void*   curve
    ,CvSlice      slice      = CV_WHOLE_SEQ
    ,int          is_closed  = -1
);

#define cvContourPerimeter( contour ) cvArcLength( contour, CV_WHOLE_SEQ, 1 )
```

Первый аргумент `cvArcLength()` это контур, который может быть представлен последовательностью точек (`CvContour*` или `CvSeq*`) или $nx2$ массивом точек. Далее следует аргумент `slice` и логический аргумент, указывающий должен ли быть контур закрытым (т.е. рассматривать последнюю точку как связанную с первой). Аргумент `slice` позволяет выбирать некоторое подмножество точек на кривой. (Почти всегда используется `CV_WHOLE_SEQ`. Структура `CvSlice` содержит только два элемента: `start_index` и `end_index`. Так же существует возможность создавать собственный `slice` и передать его функции при помощи вспомогательной функции `cvSlice(int start, int end)`).

С функцией `cvArcLength()` тесно связана функция `cvContourArea()`, которая (как видно из названия) вычисляет площадь контура. Функция принимает два аргумента `contour` и `slice`.

```
double cvContourArea(
    const CvArr* contour
    , CvSlice slice = CV_WHOLE_SEQ
);
```

Ограничительная рамка

Конечно, длина и площадь просто характеризует контур. Следующий уровень детализации может быть представлен ограничительной рамкой или ограничительным кругом или эллипсом. Рамку можно реализовать двумя способами, а круг или эллипс только одним.

```
CvRect cvBoundingRect(
    CvArr* points
    , int update = 0
);

CvBox2D cvMinAreaRect2(
    const CvArr* points
    , CvMemStorage* storage = NULL
);
```

Наиболее простой способ получения рамки - это использовать функцию `cvBoundingRect()`, которая возвращает структуру типа `CvRect`. Аргумент `points` может быть контуром (`CvContour*`) или *nx1* двухканальным массивом (`CvMat*`), содержащий последовательность точек. Что касается второго аргумента `update`, то необходимо помнить, что `CvContour` это не совсем то же самое, что `CvSeq`; функциональность `CvContour` шире. Одним из дополнений `CvContour` является поле типа `CvRect`, которое задает собственную ограничительную рамку. Если вызвать `cvBoundingRect()` с `update = 0`, то будет возвращено значение того самого дополнительного поля; но если `update = 1`, ограничительная рамка будет вычислена (и собственная ограничительная рамка структуры `CvContour` будет установлена в это вычисленное значение).

Однако есть одна проблема, связанная с `cvBoundingRect()` - при помощи `CvRect` можно представить только прямоугольник, стороны которого ориентированы по горизонтали и вертикали. А вот `cvMinAreaRect2()` возвращает минимальный прямоугольник, содержащий контур, и самое главное позволяет наклонять этот прямоугольник по вертикали (рисунок 8-7). Аргументы этой функции схожи с аргументами `cvBoundingRect()`. Для представления подобного рода прямоугольников в OpenCV есть соответствующий тип `CvBox2D`.

```

typedef struct CvBox2D
{
    CvPoint2D32f    center;
    CvSize2D32f     size;
    float           angle;
} CvBox2D;

```

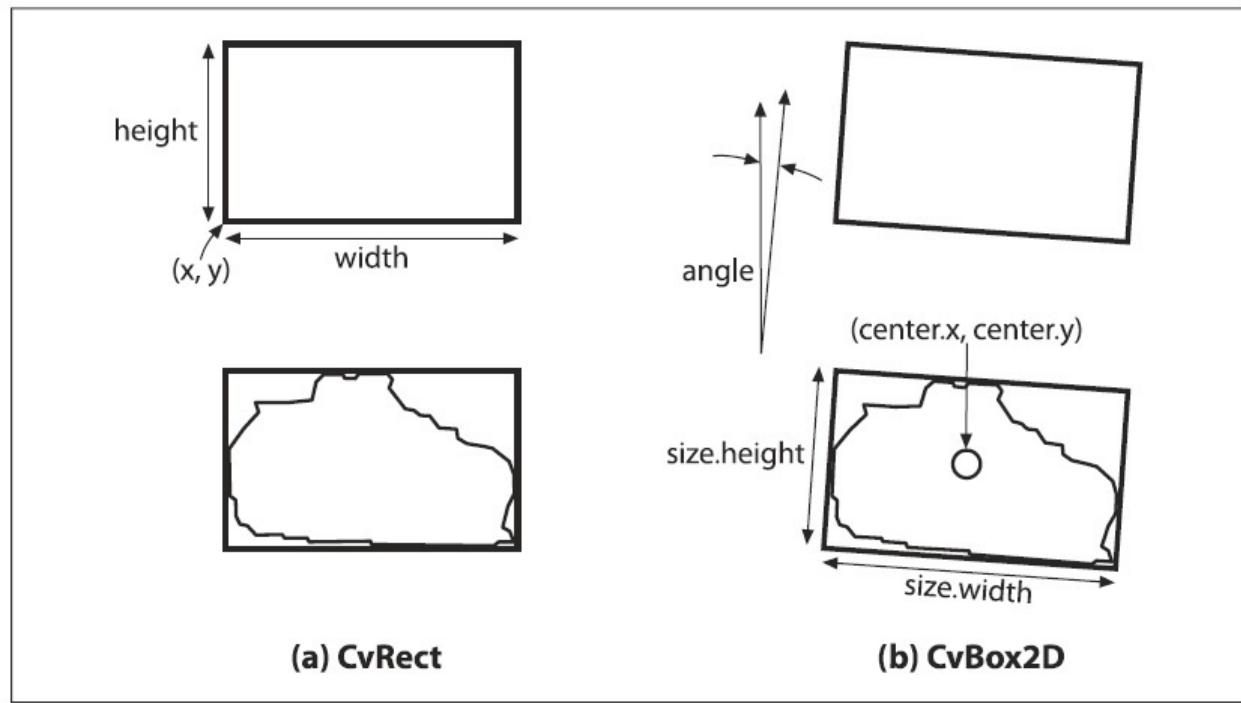


Рисунок 8-7. При помощи *CvRect* можно представить только вертикальный прямоугольник, а при помощи *CvBox2D* можно представлять прямоугольники любого наклона

Ограничительные круги и эллипсы

Теперь рассмотрим функцию *cvMinEnclosingCircle()*. Эта функция работает почти так же, как и предыдущая; множество *points* может быть последовательностью или двумерным массивом точек.

```

int cvMinEnclosingCircle(
    const CvArr*   points
    ,CvPoint2D32f* center
    ,float*        radius
);

```

Так как в OpenCV нет специальной структуры, представляющей круги, то в функцию необходимо передавать указатель центральной точки *center* и указатель типа *float** для указания радиуса.

Помимо установки ограничительного круга, OpenCV предоставляет функцию для установки ограничительного эллипса:

```
CvBox2D cvFitEllipse2(
    const CvArr*    points
);
```

Тонкая разница между *cvMinEnclosingCircle()* и *cvFitEllipse2()* состоит в том, что первая функция вычисляет наименьший круг, который полностью охватывает заданный контур, в то время как последняя использует функции подгонки и возвращает эллипс, который является наилучшим приближением к контуру. Это означает, что не все точки контура будут включены в эллипс, возвращаемый *cvFitEllipse2()*. Подгонка производится методом наименьших квадратов.

Результат подгонки возвращается в виде структуры *CvBox2D*, которая точно охватывает эллипс (рисунок 8-8).

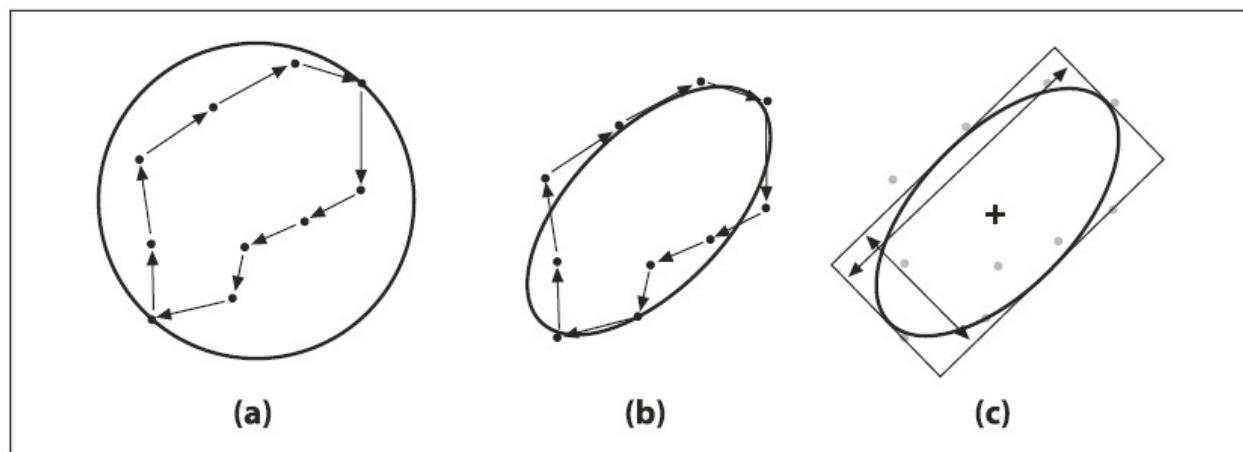


Рисунок 8-8. Контур из 10 точек с наименьшим ограничивающим кругом (а) и с наиболее подходящим эллипсом (б); рамка (с) используется для представления этого эллипса

Геометрия

Когда доходит дело до ограничительной рамки и других полигонов контура, зачастую бывает нужно выполнить такие простые геометрические проверки, как перекрытие полигонов или прямоугольников. OpenCV предлагает небольшой, но довольно такой удобный набор функций для такого рода геометрических проверок.

```

CvRect cvMaxRect(
    const CvRect* rect1
    ,const CvRect* rect2
);

void cvBoxPoints(
    CvBox2D      box
    ,CvPoint2D32f pt[4]
);

CvSeq* cvPointSeqFromMat(
    int          seq_kind
    ,const CvArr* mat
    ,CvContour*   contour_header
    ,CvSeqBlock*  block
);

double cvPointPolygonTest(
    const CvArr*  contour
    ,CvPoint2D32f pt
    ,int          measure_dist
);

```

Первая функция *cvMaxRect()* вычисляет новый прямоугольник на основе двух входных. Новый, меньших размеров, прямоугольник связывает оба входных прямоугольника.

Следующая функция *cvBoxPoints()* просто вычисляет точки в углах структуры *CvBox2D*. Данное вычисление можно произвести и самостоятельно, используя тригонометрию, однако это довольно таки утомительно.

Следующая функция *cvPointSeqFromMat()*, генерирует структуру последовательности из матрицы. Это бывает полезным, когда необходимо использовать функции для работы с контурами, которые не принимают матрицы в качестве аргументов. Первый аргумент используется для указания типа последовательности, которую необходимо получить. Переменная *seq_kind* может быть установлена в одно из следующих значений: 0 - простой набор точек; *CV_SEQ_KIND_CURVE* - последовательность должна представлять кривую; *CV_SEQ_KIND_CURVE | CV_SEQ_FLAG_CLOSED* - последовательность должна быть замкнутой кривой. Следующий аргумент *nx1* массив точек. Точки должны быть типа *CV_32SC2* или *CV_32FC2* (т.е. массив должен быть двухканальным и состоять из одного столбца). Следующие два аргумента - указатели на значения, которые будут вычисляться в функции, где *contour_header* - структура контура, которая должна быть создана перед вызовом функции, а заполнена во время выполнения функции. Для аргумента *block* идея та же. (Как правило, *block* это мало используемая переменная. Она существует, потому что никакой памяти при вызове *cvPointSeqFromMat()* не копируется; вместо этого, создается "виртуальный"

блок памяти, указывающий на предоставленную матрицу. Переменная *block* используется для создания ссылки на эту память для соответствующей внутренней последовательности или рассчитываемого контура).

Возвращаемым значением будет указатель *CvSeq**, который на самом деле указывает на саму структуру контура, переданную перед вызовом функции. Это удобно, потому что обычно необходим адрес последовательности во время вызова функций, работающих с последовательностями и требующие выполнять эти преобразования в первую очередь.

Последняя функция из набора *cvPointPolygonTest()* позволяет проверять попадание точки внутрь многоугольника (представленного последовательностью). В частности, если аргумент *measure_dist* имеет не нулевое значение, тогда функция возвращает расстояние до ближайшего края контура; это расстояние равно 0, если точка находится внутри контура, иначе это значение будет положительным, а точка будет находиться за пределами контура. Если аргумент *measure_dist* равен 0, то возвращаемым значением будет +1, -1 или 0 в зависимости от того, где находится точка: внутри, снаружи или на краю (или на вершине) соответственно. Контур может быть представлен последовательностью или *nx1* двухканальной матрицей точек.

[П]|[РС]|(РП) Сопоставление контуров

Теперь, имея довольно таки хорошее представление о том, что такое контур и как с ними работать, можно перейти к рассмотрению применения этих знаний на практике. Наиболее распространённой задачей является *сопоставление контуров* тем или иным способом. Может возникнуть потребность сравнивать два вычисленных контура между собой или сравнить вычисленный контур с каким-то абстрактным шаблоном. Собственно далее речь пойдет именно об этих двух случаях.

Моменты

Наиболее простым способом сравнения двух контуров является вычисление *моментов контура*. Грубо говоря, момент – это грубая характеристика контура, вычисляемая путем интегрирования (или суммирования) по всем пикселям контура. В общем, момент контура (p, q) определяется следующим образом:

$$m_{p,q} = \sum_{i=1}^n I(x_i, y_i) x_i^p y_i^q$$

где p порядок x , q порядок y , а термин *порядок* означает степень, в которую возводится компонента суммы. Суммирование ведется по всем пикселям границы контура (обозначено в уравнение как n). Если p и q равны 0, то момент m_{00} равен длине пикселей контура. (Математики-пуристы могут возразить, что m_{00} на самом деле не длина, а площадь контура. Но так как в рассмотрение берется только контур, а не весь многоугольник, длина и площадь, на самом деле, одно и то же в дискретном пространстве пикселей (по крайней мере, для соответствующего дистанционного критерия в данном пространстве пикселей). Так же существует функция для вычисления моментов *lplImage* изображения; в этом случае, m_{00} будет фактически равен площади ненулевых пикселей).

Следующая функция вычисляет моменты контура:

```
void cvContoursMoments(
    CvSeq*      contour
    , CvMoments* moments
)
```

Первый аргумент это контур, а второй это указатель на структуру, которую необходимо создать для хранения возвращаемых данных. Структура *CvMoments* выглядит следующим образом:

```

typedef struct CvMoments {
    // пространственные моменты
    double m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;

    // центральные моменты
    double mu20, mu11, mu02, mu30, mu21, mu12, mu03;

    // m00 != 0 ? 1/sqrt(m00) : 0
    double inv_sqrt_m00;
} CvMoments;

```

Функция `cvContoursMoments()` использует только элементы $m_{00}, m_{01}, \dots, m_{03}$; элементы с именами μ_{00}, \dots используются при работе с другой функцией.

При работе со структурой `CvMoment` можно использовать вспомогательную функцию, которая будет возвращать конкретный момент из структуры:

```

double cvGetSpatialMoment(
    CvMoments* moments
    , int      x_order
    , int      y_order
);

```

При первом вызове функция `cvContoursMoments()` вычисляет все моменты до третьего порядка (т.е. m_{30} и m_{03} будут вычислены после m_{21} и m_{12} , но перед m_{22}).

Подробнее о моментах

Только что описанное вычисление моментов дает некоторые элементарные характеристики контура, которые могут быть использованы для сравнения двух контуров. Тем не менее, моменты, получаемые из этих вычислений, в большинстве практических приложений не являются наилучшим способом для сравнения контуров. В частности, наиболее популярным подходом является использование *нормированных моментов* (так, объекты одинаковой формы, но разных размеров дают схожие значения).

OpenCV предоставляет функции для вычисления нормированных моментов, а также *Ни инвариантного момента*. Структура `CvMoments` может быть вычислена при помощи `cvMoments()` или `cvContourMoments()`. Более того, для данного случая `cvContourMoments()` является псевдонимом для `cvMoments()`.

Стоит отметить один трюк, связанный с использованием `cvDrawContours()`: вызов одной из функций для работы с моментами на результате применения функции рисования контура позволяет контролировать заполнение контура.

```

void cvMoments(
    const CvArr*   image
    ,CvMoments*   moments
    ,int          isBinary = 0
);

double cvGetCentralMoment(
    CvMoments* moments
    ,int        x_order
    ,int        y_order
);

double cvGetNormalizedCentralMoment(
    CvMoments* moments
    ,int        x_order
    ,int        y_order
);

void cvGetHuMoments(
    CvMoments*   moments
    ,CvHuMoments* HuMoments
);

```

Первая функция аналогична `cvContoursMoments()` за исключением того, что принимает изображение (вместо контура) и один дополнительный аргумент. Этот дополнительный аргумент, если он установлен в `CV_TRUE`, говорит `cvMoments()` рассматривать все пиксели либо как 1, либо как 0, где 1 присваивается любому пикселю отличному от нуля. При вызове этой функции все моменты вычисляются сразу, в том числе и центральные (см. следующий параграф).

Центральные моменты вычисляются почти так же, как и пространственные, за исключением того, что значения x и y имеют смещение на значение их среднего:

$$\mu_{p,q} = \sum_{i=0}^n I(x, y)(x - x_{\text{avg}})^p(y - y_{\text{avg}})^q$$

где $x_{\text{avg}} = m_{10}/m_{00}$ и $y_{\text{avg}} = m_{01}/m_{00}$.

Нормированные моменты вычисляются почти так же, как и центральные, за исключением того, что необходимо еще разделить на соответствующую степень m_{00} (под "соответствующей" подразумевается, что момент масштабируется за счет степени m_{00} в результате чего нормированный момент не зависит от масштаба объекта):

$$\eta_{p,q} = \frac{\mu_{p,q}}{m_{00}^{(p+q)/2+1}}$$

И в завершении, *Hu инвариантные моменты* являются линейными комбинациями центральных моментов. Идея заключается в том, чтобы комбинируя различными нормированными центральными моментами можно было бы создать инвариантные функции, представляющие различные аспекты изображения таким образом, чтобы они были инвариантны к масштабированию, вращению и отражению (для всех, кроме h_1)

Функция `cvGetHuMoments()` вычисляет *Hu* моменты на основе центральных моментов. Для полноты картины, ниже приведены фактические определения *Hu* моментов:

$$h_1 = \eta_{20} + \eta_{02}$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) \\ + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$

$$h_6 = (\eta_{20} - \eta_{02})((\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) \\ - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2)$$

Глядя на рисунок 8-9 и таблицу 8-1 можно понять, как ведут себя *Hu* моменты. Не трудно заметить, что моменты уменьшаются по мере продвижения к более высоким порядкам. Это не должно вызывать удивления, т.к. по определению *Hu* моменты высших порядков имеют большее разнообразие нормированных коэффициентов. Т.к. каждый из этих коэффициентов меньше 1, их произведение и увеличение их количества дает в результате меньшее число.



Рисунок 8-9. Изображение пяти простых символов; глядя на их *Hu* моменты можно интуитивно определить их поведение

Таблица 8-1. Значения *Hu* моментов для пяти простых символов показанных на рисунке 8-9

	h_1	h_2	h_3	h_4	h_5	h_6	h_7
A	2.837e-1	1.961e-3	1.484e-2	2.265e-4	-4.152e-7	1.003e-5	-7.941e-9
I	4.578e-1	1.820e-1	0.000	0.000	0.000	0.000	0.000
O	3.791e-1	2.623e-4	4.501e-7	5.858e-7	1.529e-13	7.775e-9	-2.591e-13
M	2.465e-1	4.775e-4	7.263e-5	2.617e-6	-3.607e-11	-5.718e-8	-7.218e-24
F	3.186e-1	2.914e-2	9.397e-3	8.221e-4	3.872e-8	2.019e-5	2.285e-6

Коэффициенты, вызывающие интерес: символ "I" симметричен при вращении на 180 градусов и отображении и значения моментов равны 0 для $h_3 - h_7$; для символа "O", имеющего аналогичную симметрию, все моменты равны 0.

Сопоставление при помощи Ну моментов

```
double cvMatchShapes(
    const void*    object1
    ,const void*   object2
    ,int           method
    ,double        parameter = 0
);
```

Потребность в сравнении двух объектов при помощи Ну моментов, так или иначе, может возникнуть. При этом есть множество определений "похожести". Чтобы сделать этот процесс несколько проще, в OpenCV есть функция `cvMatchShapes()`, которой необходимо просто передать два объекта и метод их сравнения.

Эти объекты могут быть серыми изображениями или контурами. Если будут переданы изображения, то функция сначала вычислит их моменты и только после этого приступит к сравнению. Аргумент `method` может иметь одно из значений, перечисленных в таблицы 8-2.

Таблица 8-2. Методы сравнения, используемые в `cvMatchShapes()`

Значение method	Возвращаемое значение
CV_CONTOURS_MATCH_I1	$I_1(A,B) = \sum_{i=1}^7 \left \frac{1}{m_i^A} - \frac{1}{m_i^B} \right $
CV_CONTOURS_MATCH_I2	$I_2(A,B) = \sum_{i=1}^7 \left m_i^A - m_i^B \right $
CV_CONTOURS_MATCH_I3	$I_3(A,B) = \sum_{i=1}^7 \left \frac{m_i^A - m_i^B}{m_i^A} \right $

В этой таблице m_i^A и m_i^B это:

$$m_i^A = \text{sign}(h_i^A) \cdot \log |h_i^A|$$

$$m_i^B = \text{sign}(h_i^B) \cdot \log |h_i^B|$$

где h_i^A и h_i^B это Ну моменты A и B соответственно.

Каждая из трех констант, перечисленная в таблице 8-2, имеет разное значение в плане того, как вычисляются метрики сравнения. В конечном счете, эта метрика определяет возвращаемое функцией значение. Последний аргумент *parameter* на момент написания книги не использовался, поэтому смело можно присвоить ему значение по умолчанию равное 0.

Иерархическое сравнение

Довольно таки часто будет возникать необходимость сравнения двух контуров, а также придумывание признаков подобия, которые бы принимали во внимание всю структуру сопоставляемых контуров. Методы, использующие сводную характеристику (такие как моменты) довольно таки быстры, однако они могут задействовать не так много информации.

Для более точного сравнения будет полезно сначала рассмотреть структуру известную как *дерево контура*. Дерево контура не следует путать с иерархическим представлением контура, возвращаемое такими функциями, как *cvFindContours()*. Вместо этого они являются иерархическим представлением формы какого-то одного контура.

Понять что такое дерево контура проще, если сначала понять, как оно устроено. Построение дерева контура начинается снизу (конечные узлы) и заканчивается сверху (корневой узел). Процесс начинается с поиска периметра треугольных выступов и углублений (точки контура не всегда коллинеарны со своими соседями). Каждый такой треугольник замещается линией, которая соединяет две не смежные точки на кривой,

таким образом, треугольник либо отсекается (треугольник D, рисунок 8-10), либо заполняется (треугольник C). Каждое такое замещение удаляет одну вершину из контура и добавляет одну вершину в дерево. Если такой треугольник имеет две стороны, соответствующие оригинальной части контура, то это лист дерева; если одна из сторон данного треугольника является стороной существующего треугольника, то существующий треугольник является родителем для данного треугольника. Повторение данного процесса в конечном итоге сводит фигуру до четырехугольника, который в последующем разделяется пополам; получившиеся в результате треугольники являются потомками корневой вершины.

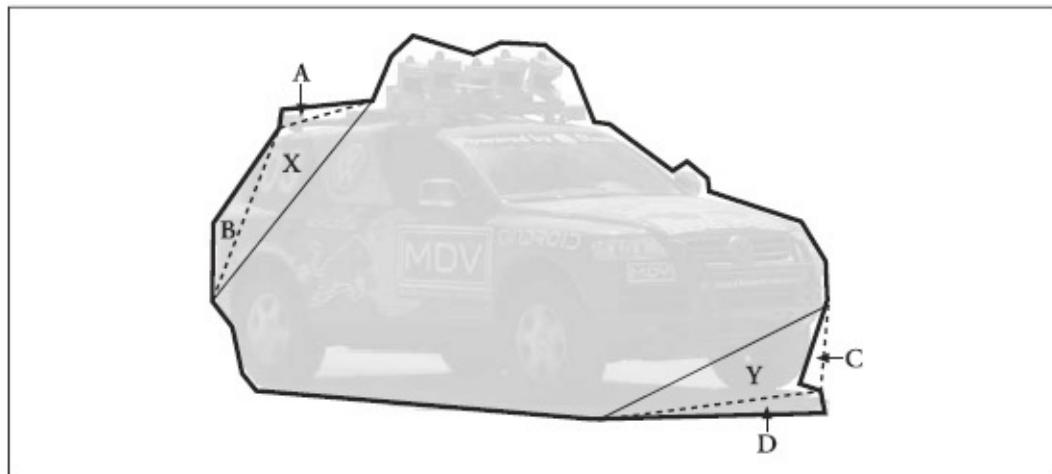


Рисунок 8-10. Построение дерева контура: в первом проходе вокруг автомобиля создаются конечные узлы A, B, C, и D; во втором проходе создаются X и Y (X является родителем для A и B, а Y является родителем для C и D)

Результирующее бинарное дерево (рисунок 8-11) в конечном счете, кодирует информацию о форме исходного контура. Каждый узел несет в себе информацию о треугольнике, с которым он связан (размер, а так же способ создания: отсечение или заполнение).

После того, как деревья построены, они могут быть использованы для эффективного сравнения двух контуров. Процесс сравнения начинается с попыток установки соответствия между узлами двух деревьев, а затем сравнения характеристик соответствующих узлов. Конечным результатом является степень сходства двух деревьев.

На практике не обязательно понимать все тонкости этого процесса. OpenCV предоставляет процедуры, позволяющие автоматически создавать деревья контуров из обычных CvContour объектов и конвертировать их обратно; так же имеется метод для сравнения двух деревьев. Но, к сожалению, данные деревья не очень надежные (так, незначительные изменения в контуре могут существенно изменить дерево). Кроме того, первоначальный треугольник (корень дерева) выбирается произвольно. В результате, чтобы получить лучшее представление, для начала необходимо

применить функцию *cvApproxPoly()*, а затем выровнить контур (выполняя циклический сдвиг), чтобы начальный треугольник стал значительно менее зависимым от вращений.

```
CvContourTree* cvCreateContourTree(
    const CvSeq* contour
    ,CvMemStorage* storage
    ,double threshold
);

CvSeq* cvContourFromContourTree(
    const CvContourTree* tree
    ,CvMemStorage* storage
    ,CvTermCriteria criteria
);

double cvMatchContourTrees(
    const CvContourTree* tree1
    ,const CvContourTree* tree2
    ,int method
    ,double threshold
);
```

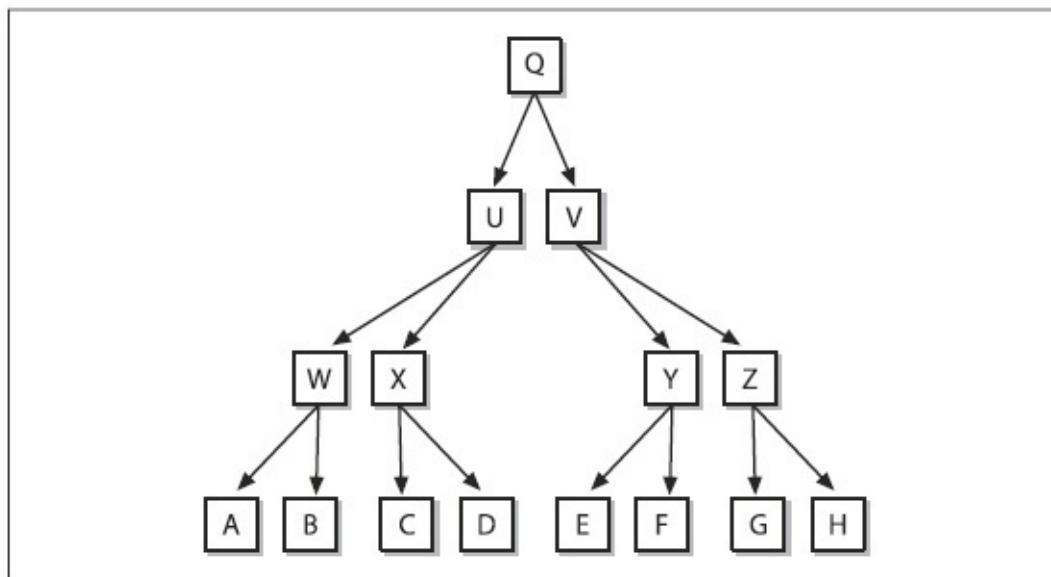


Рисунок 8-11. Бинарное представление двоичного дерева, которое может соответствовать контуру с рисунка 8-10

Этот код ссылается на структуру *CvTermCriteria*, о которой более подробно пойдет речь в главе 9. А пока достаточно будет знать, что эту структуру можно создать при помощи *cvTermCriteria()* с использованием следующих значений по умолчанию:

```
CvTermCriteria termcrit = cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 5, 1 );
```

Выпуклость контура и дефекты выпуклости

Ещё один полезный способ, позволяющий понять, что такое форма объекта или контур это вычислить каркас выпуклости для объекта, а затем вычислить *дефект выпуклости*. Формы многих сложных объектов хорошо характеризуются такими дефектами.

Рисунок 8-12 иллюстрирует концепцию дефекта выпуклости, используя изображение человеческой руки. Каркас выпуклости изображен тёмной линией вокруг руки, а дефекты, относящиеся к этому каркасу, помечены буквами от А до Н. Как можно было заметить, эти дефекты позволяют характеризовать не только саму руку, но так же и её состояние.

```
#define CV_CLOCKWISE 1
#define CV_COUNTER_CLOCKWISE 2

CvSeq* cvConvexHull2(
    const CvArr* input
    ,void* hull_storage = NULL
    ,int orientation = CV_CLOCKWISE
    ,int return_points = 0
);

int cvCheckContourConvexity(
    const CvArr* contour
);

CvSeq* cvConvexityDefects(
    const CvArr* contour
    ,const CvArr* convexhull
    ,CvMemStorage* storage = NULL
);
```

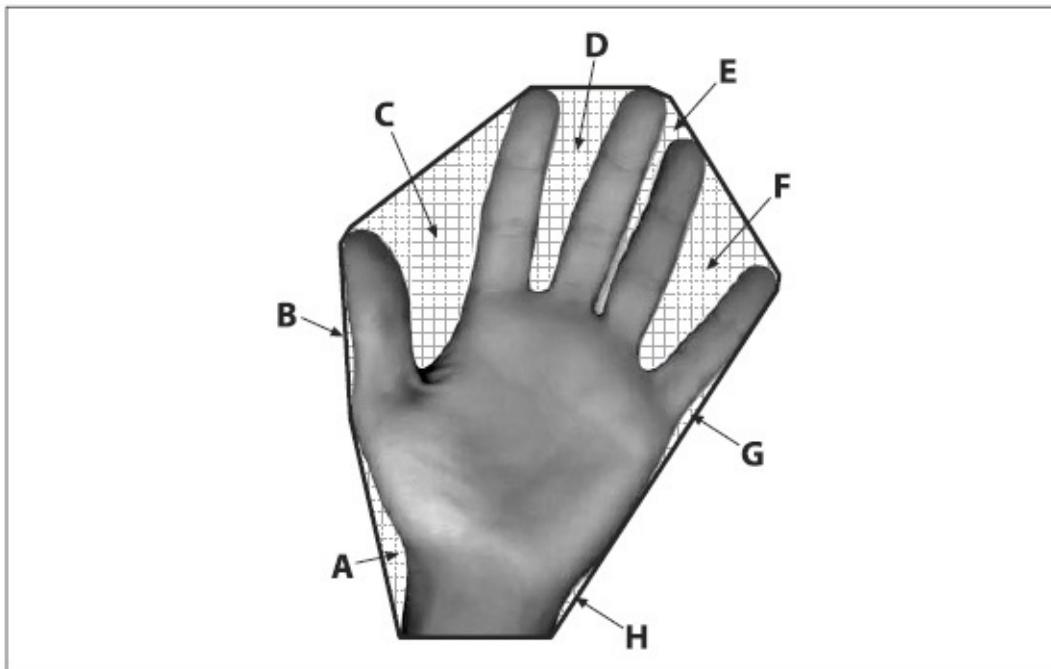


Рисунок 8-12. Дефекты выпуклости: черная линия это каркас выпуклости вокруг руки; сетчатые регионы (A-H) это дефекты выпуклости контура руки по отношению к выпуклой оболочке

В OpenCV есть три наиболее важных метода, которые занимаются сложными каркасами и выпуклостями. Первый просто вычисляет каркас контура, а второй позволяет проверить является ли контур выпуклым. Третий метод вычисляет дефекты выпуклости в контуре, для которого уже известен каркас выпуклости.

Функция `cvConvexHull2()` в качестве первого аргумента принимает массив точек. Этот массив обычно является матрицей из двух столбцов и n строк (т.е. $nx2$) или контуром. Точки должны быть 32-битными целыми (`CV_32SC1`) или вещественными (`CV_32FC1`) числами. Следующий аргумент уже привычное хранилище памяти, в котором можно выделить место под результат. Следующий аргумент может быть либо `CV_CLOCKWISE`, либо `CV_COUNTER_CLOCKWISE`, определяющий ориентацию точек, которые будут возвращены функцией. Последний аргумент `returnPoints` может быть 0 или 1. Если установлен в 0, то в возвращаемом массиве будут сохранены только индексы (при этом, если аргумент имеет тип `CvSeq*` или `CvContour*`, то результатом будут указатели на точки), которые ссылаются на записи в массиве, переданный в `cvConvexHull2()`.

В этом месте внимательный читатель может спросить: "Если аргумент `hull_storage` это хранилище памяти, то почему он имеет тип `void*`?" Это хороший вопрос. Так вот причина в том, что во многих случаях удобнее, если точки каркаса будут размещены в массиве, а не в последовательности. Принимая это во внимание, появляется возможность для передачи указателя на матрицу `CvMat*` через аргумент `hull_storage`. В этом случае матрица должна быть одномерной и иметь столько же записей, сколько

входных точек. На самом деле при вызове `cvConvexHull2()` заголовок матрицы будет меняться в соответствии с требуемым количеством столбцов. (Память, выделенная под данные матрицы, не перераспределяется в любом случае. Так как массивы являются С-массивами, то при удалении матриц память будет освобождаться верно).

Иногда даже при наличии контура, не известно является ли он выпуклым. В этом случае можно вызвать функцию `cvCheckContourConvexity()`. Тестирование данной функцией выполняется просто и быстро (на самом деле это занимает $O(N)$ времени, что немного быстрее, чем $O(N \log N)$ времени, необходимого для построения каркаса выпуклости), однако будут некорректные результаты, если контур содержит самопересечения.

Третья функция `cvConvexityDefects()` вычисляет и возвращает последовательность дефектов. Функция запрашивает контур, каркас выпуклости и хранилище памяти для хранения результата. Первые два аргумента имеют тип `CvArr*` и такую же форму, как аргумент `input` функции `cvConvexHull2()`.

```
typedef struct CvConvexityDefect {
    // точка контура, где начинается дефект
    CvPoint* start;

    // точка контура, где заканчивается дефект
    CvPoint* end;

    // наиболее удалённая точка внутри дефекта от выпуклого каркаса
    CvPoint* depth_point;

    // расстояние между наиболее удалённой точкой и выпуклым каркасом
    float depth;
} CvConvexityDefect;
```

Функция `cvConvexityDefects()` возвращает последовательность структур `CvConvexityDefect`, содержащих несколько простых параметров, с помощью которых можно характеризовать дефект. Поля `start` и `end` - это точки на каркасе, где дефект начинается и заканчивается. Поле `depth_point` - это самая дальняя точка от края каркаса, в этой точке дефект начинает отклоняться. Поле `depth` - это расстояние между самой дальней точкой и краем каркаса.

Попарные геометрические гистограммы

Ранее уже были кратко рассмотрены цепные коды Фримена. Как было тогда сказано цепные коды Фримена являются представлением многоугольника с точки зрения последовательности "ходов", где каждый ход имеет фиксированную длину и

определенное направление. Однако не было рассказано о том, как можно было бы использовать такое представление.

Цепные коды Фримена находят свое применение при решении различного рода задач, но самое популярное применение стоит рассмотреть более подробно, потому что в основе идеи лежит использование *поларных геометрических гистограмм* (PGH - pairwise geometrical histogram) (в OpenCV реализован метод Iivarinen, Peura, Särelä и Visa).

PGH на самом деле являются обобщением или расширением, так называемых *цепных кодов гистограмм* (CCH - chain code histogram). CCH – это гистограмма, созданная путем подсчета количества разновидностей кодов Фримена. Эта гистограмма обладает рядом хороших свойств. В частности, поворот объекта на 45 градусов циклически преобразует гистограмму (рисунок 8-13). Это обеспечивает независимость от вращений при распознавании объектов.

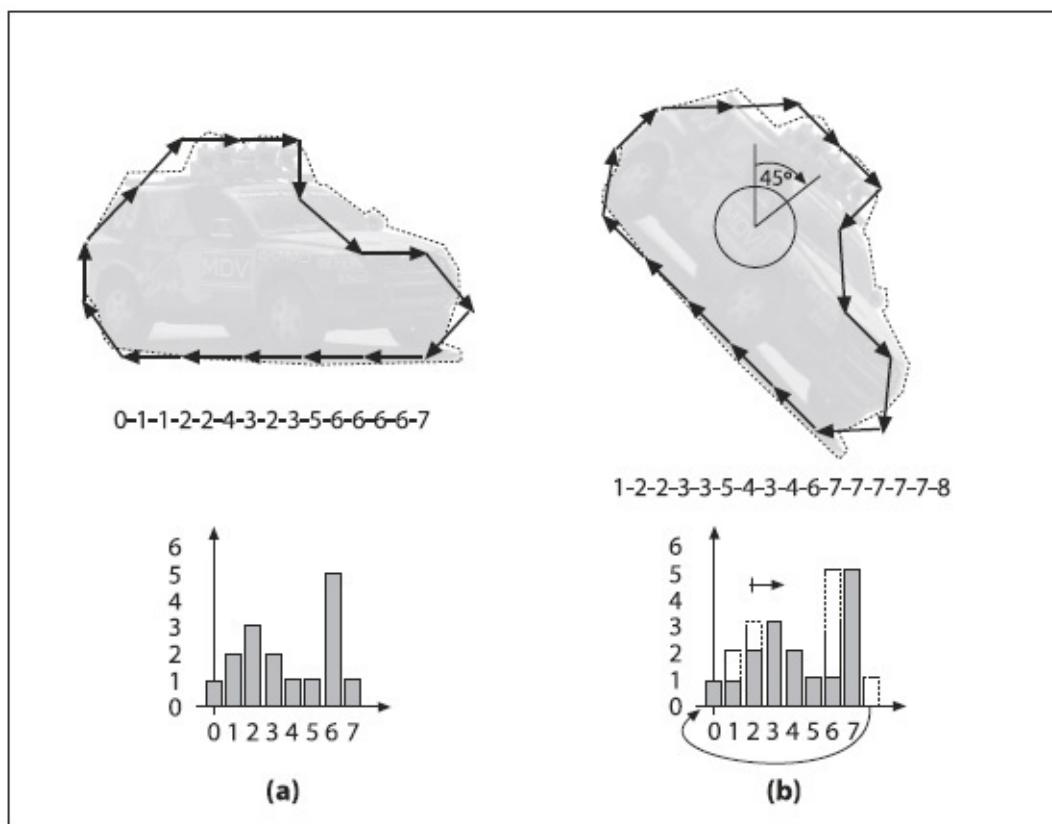


Рисунок 8-13. Представление контура цепными кодами Фримена (сверху) и связных с ними цепными кодами гистограмм (снизу); когда исходный контур (a) повернут на 45 градусов по часовой стрелке (b), результаты цепных кодов гистограмм такие же, за исключением того факта, что во втором случае имеет место смещение на одну единицу вправо

Построение PGH происходит следующим образом (рисунок 8-14). Каждый из углов многоугольника последовательно выбирается в качестве "угла основания". Затем каждый из остальных краёв рассматривается относительно базового края, и вычисляются 3 значения: d_{\min} , d_{\max} и θ . d_{\min} – это наименьшее расстояние между двумя краями, d_{\max} – наибольшее, а θ – угол между ними. PGH – это двумерная гистограмма, размеры которой угол и расстояние. В частности, для каждого парного ребра имеется соответствующий контейнер (d_{\min} , θ) и соответствующий контейнер (d_{\max} , θ). Для каждой такой пары краёв, эти два контейнера увеличиваются – так же, как и все контейнеры на участке между d_{\min} и d_{\max} .

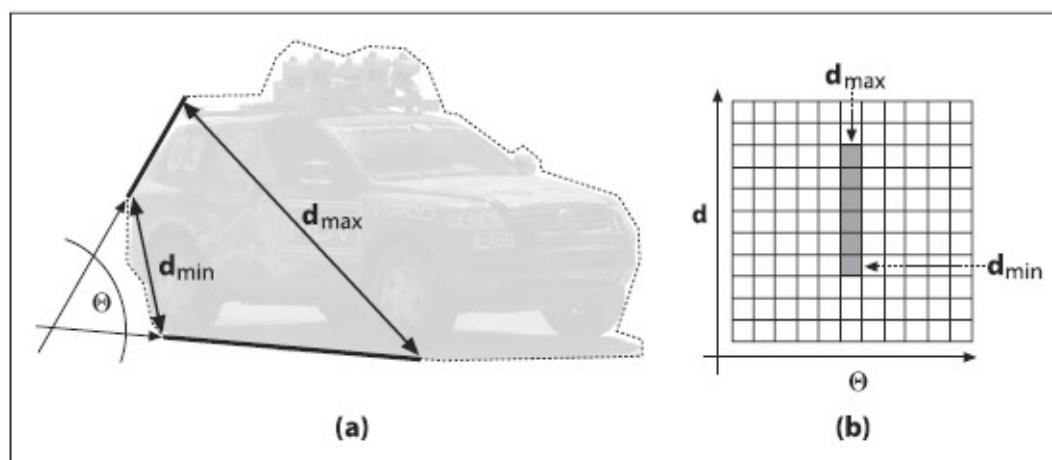


Рисунок 8-14. Парные геометрические гистограммы: каждые два края сегмента охватывающего многоугольника имеют угол, минимальное и максимальное расстояние (a); эти цифры кодируются в двумерную гистограмму (b), которая не зависит от вращений

И PGH и FCC полезны. Однако подход с использованием PGH обладает большей мощностью, поэтому он более полезен при решении сложных проблем, связанных с наличием большого количества форм, которые требуется распознать и/или которые обладают большой изменчивостью фонового шума. Функция для вычисления PGH выглядит следующим образом:

```
void cvCalcPGH(
    const CvSeq* contour
    , CvHistogram* hist
);
```

Аргумент *contour* содержит целочисленные координаты точек; аргумент *hist* может быть двумерным.

[П]|[РС]|(РП) Упражнения

1. Пренебрегая шумами на изображении при применении алгоритма IPAN будут ли возвращены те же "доминирующие точки" после изменения размеров объекта? После разворота объекта?
 - a. Причины такого ответа
 - b. А теперь попробуйте следующее. В PowerPoint или аналогичной программе нарисуйте объект "интересной" формы белым цветом на черном фоне. Поместите нарисованный объект на исходное изображение. Сохраните несколько вариантов полученного изображения, изменяя размеры и поворачивая объект. Затем загрузите весь набор изображений в программу, конвертируйте в оттенки серого, примените пороговое преобразование и найдите контур. Используя функцию `cvFindDominantPoints()`, найдите доминирующие точки на изображениях, в которых производились изменения объекта. Совпадают ли найденные точки?
2. Поиск экстремальных точек (т.е. двух точек максимально удаленных друг от друга) в замкнутом контуре из N точек может происходить путем сравнения расстояния между каждой из точек контура с любой другой точкой контура.
 - a. В чем сложность такого алгоритма?
 - b. Предложите идею по ускорению его работы?
3. Создайте очередь изображения круга, используя функцию `CvSeq`.
4. Чему равна максимальная длина замкнутого контура, который ограничен изображением 4×4 ? Чему равна площадь такого контура?
5. Используя PowerPoint или подобную программу, нарисуйте белый круг радиусом 20 на черном фоне (длина окружности $2 \pi \cdot 20 \approx 126.7$). Сохраните полученное изображение.
 - a. Загрузите изображение в программу, конвертируйте в оттенки серого, примените пороговое преобразование и найдите контур. Чему равна длина контура? Полученное значение равно (в пределах округления) рассчитанному значению?
 - b. Используя значение 126.7 в качестве базовой длины контура, примените `cvApproxPoly()` со следующими значениями параметров: 90, 66, 33, 10. Определите длину контура и отобразите результат.

6. Используя нарисованный круг из 5 упражнения, исследуйте следующие результаты функции `cvFindDominantPoints()`.
 - а. Переменные расстояния d_{\min} и d_{\max} .
 - б. Теперь измените переменную расстояния до соседей d_n и опишите произошедшие изменения.
 - в. Теперь измените переменную максимального угла θ_{\max} и опишите результат.
7. Поиск угла субпикселя. Создайте белый на черном фоне угол в PowerPoint (или подобной программе) так, чтобы координаты угла были целыми значениями. Сохраните полученное изображение и загрузите в программу.
 - а. Найдите и распечатайте точные координаты угла.
 - б. Измените исходное изображение: удалите кончик угла при помощи небольшого черного круга. Сохраните изображение и найдите координаты угла субпикселя. Это те же координаты? Почему да или почему нет?
8. Допустим необходимо построить детектор бутылки и создать "бутылочные" признаки. Например, имеется изображение с множеством бутылок на сцене, которые легко сегментировать и найти для них контур, однако, бутылки могут быть повернуты и иметь различные размеры. Можно нарисовать контуры и найти Ну моменты для получения инвариантного вектора. Уже не плохо, но что необходимо использовать далее – заполненный контур или просто линию контура? Поясните свой ответ.
9. Какое значение нужно установить для `isBinary` при использовании `cvMoments()` для извлечения моментов контуров бутылок из упражнения 8?
10. В данном упражнении воспользуйтесь буквами, используемые при обсуждении Ну моментов. Сгенерируйте несколько вариантов этих букв за счет их вращения, масштабирования и одновременного вращения, и масштабирования. Опишите результаты воздействия этих операций на Ну моменты.
11. Нарисуйте объект простой формы в PowerPoint (или другой аналогичной программе) и сохраните его в виде изображения. Затем сгенерируйте варианты данного изображения за счет вращения, масштабирования и одновременного вращения, и масштабирования объекта. Сравните полученные изображения при помощи `cvMatchContourTrees()` и `cvConvexityDefects()`. Какая из функций лучше сравнивает объекты? Почему?

Составные части и сегментация изображения

[П]||[РС]||(РП) Составные части и сегментация

Эта глава посвящена теме отделения объекта или его части от всего изображения. Причины для этого должны быть очевидны. Например, в камерах слежения для обеспечения безопасности в основном фиксируется одна и та же сцена, которая в действительности не вызывает интереса. Наибольший интерес представляет появление в кадре людей, транспортных средств или какого-то предмета, которые отсутствовали в сцене на протяжении длительного времени. Выделение этих событий позволяет отсекать моменты во времени, в которые ничего интересного не происходило.

Помимо выделения объекта на изображении, существует множество ситуаций, при которых возникает необходимость только в части объекта, например, при выделении лица или кисти руки человека. Помимо этого, может потребоваться предобработка изображения для преобразования его в осмыслиенный набор *супер пикселей*, которые сегментируют изображение на содержимое: конечности, волосы, лицо, туловище, листья деревьев, озеро, путь, газон и т.д. Использование супер пикселей экономит время при вычислениях; например, при запуске классификатора объектов изображения, потребуется найти только ограничительную рамку для каждого супер пикселя. Это позволяет отслеживать эти большие куски, а не каждую точку в отдельности.

Ранее уже было рассмотрено несколько алгоритмов сегментации при обсуждении обработки изображений в пятой главе. Функции, рассматриваемые в той главе, включали в себя рассмотрение таких процессов, как морфология, заливка, пороговое преобразование и пирамidalная сегментация. В данной главе будут рассмотрены другие алгоритмы, которые касаются поиска, заливки и выделения объекта или его части на изображении. Вначале будет рассмотрено отделение объекта переднего плана от изученной сцены фона. Рассматриваемые функции отделения объекта от фона не являются встроенными в OpenCV; скорее они являются примерами использования встроенных функций OpenCV для реализации более сложных алгоритмов.

[П]|[РС]|(РП) Вычитание фона

Из-за своей простоты и т.к. камера, как правило, зафиксирована, *вычитание фона*, вероятно, наиболее фундаментальная операция обработки изображения из видеопотока. Для выполнения операции вычитания фона для начала необходимо "изучить" модель фона. Однажды изученная модель фона сравнивается с текущим изображением с последующим исключением известных частей фона. Объекты, оставшиеся после исключения, предположительно будут являться новыми объектами переднего плана.

"Фон" является плохо определенным понятием, которое изменяется в зависимости от применения. Например, в случае рассмотрения шоссе, возможно, обычный транспортный поток следует считать фоном. Как правило, за фон принимаются любые статические или периодически движущие части сцены, которые остаются неизменными или периодическими в течение интересующего периода времени. Группа может иметь изменяющиеся во времени компоненты, такие, например, как деревья, которые развиваются от утра к вечеру, но неподвижны в полдень. Две распространенные, но существенно различающиеся категории окружения, которые могут встретиться, это сцены внутри и снаружи помещений. Возникает интерес в инструментах, которые помогли бы разобраться с обеими категориями окружения. В данной главе вначале будут рассмотрены недостатки типичных моделей фона, а затем будут рассмотрены модели сцен высших порядков. Далее будет представлен быстрый метод, который в основном хорош для статичных фоновых сцен внутри помещений, освещение которых практически не меняется. Затем будет рассмотрен метод "кодовых книг", который немного медленнее, однако может работать в сценах и внутри и снаружи помещений; этот метод подходит для периодических движений (таких, как раскачивающихся на ветру деревьев) и для медленного или периодически изменяющегося освещения. Этот метод так же устойчив при изучении фона, даже когда есть случайные движущиеся объекты переднего плана. Ранее эта тема уже была затронута во время обсуждения связанных компонентов (впервые в главе 5) в контексте обнаружения объекта переднего плана. И в конце главы, будет представлен сравнительный анализ быстрого метода исключения фона с методом "кодовой книги".

Слабые стороны исключения фона

Хотя методы моделирования фона, упомянутые в данной главе, работают достаточно хорошо для простых сцен, однако, они страдают от предположения, которое часто нарушается: что все пиксели независимы. Рассматриваемые методы обучают модель изменения пикселей без учета соседних пикселей. Для принятия окружающих

пикселей во внимание, необходимо изучить модель, состоящую из нескольких частей; простым примером такой модели является расширенная основная модель независимых пикселей путем элементарного включения чувствительных к яркости соседних пикселей. В этом случае используется яркость соседних пикселей, чтобы различать случаи, когда значение соседнего пикселя будет относительно ярким или тусклым. В связи с этим существует две модели для конкретного пикселя: одна для случая, когда соседние пиксели более яркие, а другая для случая, когда соседние пиксели более тусклые. В общем, имеется модель, которая принимает во внимание окружающий контекст. Однако это приводит к возрастанию используемой памяти в 2 раза и количеству операций вычислений, так как возникает потребность в значениях для случаев, когда окружающие пиксели либо ярче, либо более тусклые. Помимо этого, необходимо в два раза больше данных, чтобы заполнить эти две модели состояний. Можно обобщить идею "высокого" и "низкого" контекстов в многомерную гистограмму интенсивности конкретного и соседнего пикселей, а также сделать её ещё более сложной, выполнив это за несколько временных шагов. При этом стоит принимать во внимание, что более сложная в пространстве и времени модель требует ещё больше памяти, собираемых данных и вычислительных ресурсов.

Из-за дополнительных расходов использование более сложных моделей, как правило, стараются избегать. Для более эффективного распоряжения ресурсами, можно избавляться от ложноположительных пикселей, которые оказывают влияние в тех случаях, когда нарушается предположение о независимости пикселей. Избавление принимает форму операций обработки изображений (как правило, `cvErode()`, `cvDilate()` и `cvFloodFill()`), которые исключают ложноположительные пиксели. Ранее эти операции уже были рассмотрены (глава 5) в контексте поиска больших и компактных (компактные - это математический термин, который не имеет ничего общего с размером) связанных компонентов в данных с наличием шума. В данной главе связанные компоненты так же будут использованы, а на данный момент ограничимся подходом, который предполагает независимое изменение пикселей.

Моделирование сцены

Итак, каким же образом отделить фон от переднего плана? Например, если ведется наблюдение за стоянкой и на парковку въезжает автомобиль, то он будет являться новым объектом переднего плана. Но должен ли он оставаться объектом переднего плана навсегда? А как насчет перемещенного мусорного бака? Он будет отображаться на переднем плане в двух местах: туда, куда переместили и "дырой" в месте, откуда он был перемещен. Как объяснить эту разницу? И ещё, как долго мусорный бак (или "дыра") остается объектом переднего плана? Если смоделировать темную комнату, и кто-то вдруг включит свет, должна ли вся комната стать объектом переднего плана? Чтобы ответить на все эти вопросы, необходима высокоуровневая модель "сцены",

которая определяет несколько уровней между состояниями переднего плана и фона, а также временной метод медленной передачи неподвижных объектов переднего плана фону. К тому же требуется определять и создавать новую модель при глобальных изменениях в сцене.

В общем, модель сцены может содержать несколько слоев, от "нового переднего плана" до старого переднего плана и так вплоть до фона. Также должно быть реализовано детектирование движения таким образом, чтобы при перемещении объекта можно было идентифицировать "позитивную" часть (новое местоположение) и "негативную" часть (старое местоположение, "дыра").

Таким образом, новый объект переднего плана должен быть перемещен на уровень "новые объекты переднего плана" и отмечен как позитивный объект или как дыра. В районах, где нет объектов переднего плана, можно продолжать обновление модели фона. Если объект переднего плана не перемещался в течение заданного участка времени, то он будет перемещен на уровень "старые объекты переднего плана", где пиксельная статистика предварительно изучается до тех пор, пока изучаемая модель не присоединиться к модели фона.

Для отслеживания глобальных изменений, таких, как включение освещения в помещении, необходимо использовать глобальную разность кадров. Например, если одновременно изменениям подверглось большое количество пикселей, тогда можно классифицировать это скорее, как глобальное изменение, а не локальное, а затем переключиться на использование модели для новой ситуации.

Срез пикселей

Прежде, чем перейти к моделированию пиксельных изменений, необходимо получить представление о том, как изменяются пиксели изображения во времени. Рассмотрим случай, когда камера следит за деревом на улице, которое раскачивается на ветру. На рисунке 9-1 показано, как выглядят пиксели выбранного линейного сегмента на протяжении 60 кадров. Зададимся целью получения модели этих колебаний. Однако, прежде, сделаем небольшое отступление, чтобы обсудить способ получения этой линии, потому что в целом это полезный прием для генерации признаков и для отладки.

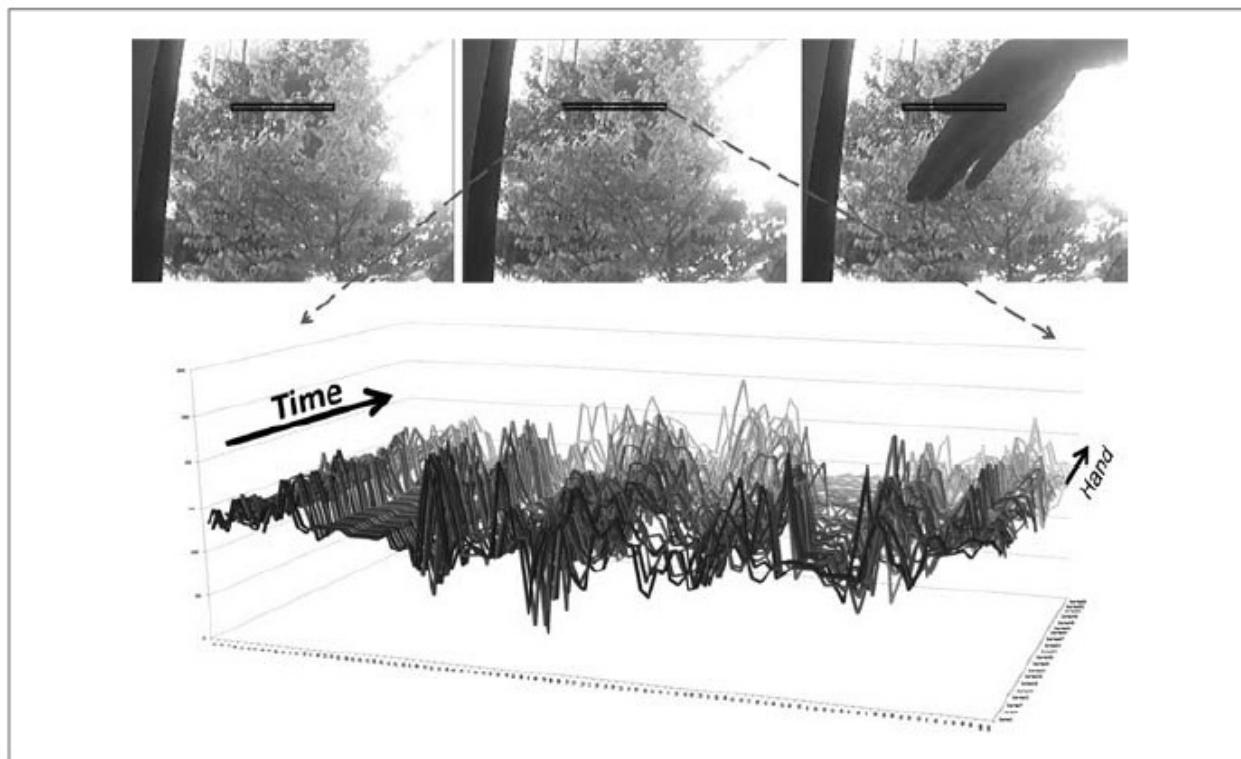


Рисунок 9-1. Колебание пикселей линии на протяжении 60 кадров из сцены раскачивающегося дерева: некоторые темные области (вверху слева) довольно таки стабильны, тогда как в области движущихся ветвей (вверху в центре) могут изменяться в широких пределах

В OpenCV имеются функции, которые позволяют с легкостью получить произвольную линию пикселей. Это функции `cvInitLineIterator()` и `CV_NEXT_LINE_POINT()`. Прототип функции `cvInitLineIterator()`:

```
int cvInitLineIterator(
    const CvArr*      image
    ,CvPoint          pt1
    ,CvPoint          pt2
    ,CvLineIterator*   line_iterator
    ,int              connectivity = 8
    ,int              left_to_right = 0
);
```

Исходное изображение *images* может иметь любой тип или количество каналов. Точки *pt1* и *pt2* являются концами линейного сегмента. Итератор *line_iterator* отвечает за перемещение между точками вдоль линии. В случае использования многоканальных изображений, каждый вызов `CV_NEXT_LINE_POINT()` перемещает *line_iterator* к следующему пикслю. Для получения доступа ко всем каналам необходимо использовать *line_iterator.ptr[0]*, *line_iterator.ptr[1]* и так далее. Связность *connectivity* может быть равна 4 (линия может совершать шаги вверх, вниз, влево, вправо) или 8 (линия может дополнительно делать шаги по диагоналям). Если *left_to_right* равен 0

(*false*), тогда *line_iterator* совершает шаги от *pt1* до *pt2*; иначе, шаги будут осуществляться от крайней левой точки к крайней правой. (Флаг *left_to_right* был введен из-за того, что дискретная линия, проведенная от *pt1* к *pt2*, не всегда соответствует линии, проведенной от *pt2* к *pt1*. Таким образом, установка этого флага дает пользователю получить точную растеризацию в независимости от последовательности *pt1*, *pt2*). Функция *cvInitLineIterator()* возвращает число точек, которые были пройдены для этой линии. Сопутствующий макрос *CV_NEXT_LINE_POINT(line_iterator)* перемещает итератор от одного пикселя к другому.

Прервемся от обсуждения и посмотрим на то, как этот метод может быть использован для извлечения некоторых данных из файла (пример 9-1). При этом переосмыслим рисунок 9-1 с точки зрения полученных данных из файла.

Пример 9-1. Чтение RGB значений всех пикселей одной строки файла и сохранение этих значений в трех отдельных файлах

```

// Сохранение на диск линейного сегмента из BGR пикселей от p1 до p2
//
CvCapture* capture = cvCreateFileCapture( argv[1] );
int max_buffer;
IplImage* rawImage;
int r[10000], g[10000], b[10000];
CvLineIterator iterator;

FILE *fptrb = fopen("blines.csv", "w"); // Создание файлов для сохранения
FILE *fptrg = fopen("glines.csv", "w"); // каждого из каналов в отдельности
FILE *fptrr = fopen("rlines.csv", "w");

// Главный цикл обработки
//
for(;;) {

    if( !cvGrabFrame( capture ) )
        break;

    rawImage = cvRetrieveFrame( capture );
    max_buffer = cvInitLineIterator(rawImage,pt1,pt2,&iterator,8,0);
    for(int j=0; j < max_buffer; j++){

        // Запись значений
        //
        fprintf(fptrb, "%d,", iterator.ptr[0]); // синий
        fprintf(fptra, "%d,", iterator.ptr[1]); // зеленый
        fprintf(fptrr, "%d,", iterator.ptr[2]); // красный

        iterator.ptr[2] = 255; // Маркировка этого образца красным

        CV_NEXT_LINE_POINT(iterator); // Переход к следующему пикселью
    }

    // Вывод данных по строкам
    //
    fprintf(fptrb, "\n"); fprintf(fptra, "\n"); fprintf(fptrr, "\n");
}

// Очистка
//
fclose(fptrb); fclose(fptra); fclose(fptrr);
cvReleaseCapture( &capture );

```

Получить линию выборки можно ещё проще, а именно:

```

int cvSampleLine(
    const CvArr*   image
    ,CvPoint       pt1
    ,CvPoint       pt2
    ,void*         buffer
    ,int           connectivity = 8
);

```

Эта функция просто обертка функции `cvInitLineIterator()` вместе с макросом `CV_NEXT_LINE_POINT(line_iterator)`. Она производит выборку от `pt1` до `pt2`; затем передает указатель `buffer` нужного типа и длиной

$N_{\text{channels}} \times \max(|pt2_x - pt1_x| + 1, |pt2_y - pt1_y| + 1)$. Так же как и линейный итератор, `cvSampleLine()` пошагово проходит по каждому каналу каждого пикселя многоканального изображения, прежде, чем переместиться к следующему пикслю. Функция возвращает число элементов `buffer`.

Теперь можно переходить к рассмотрению методов моделирования колебаний пикселей, рассмотренных на рисунке 9-1. По мере продвижения от простой к более сложной модели, будут представлены лишь те, которые работают в режиме реального времени и в рамках разумных ограничений по памяти.

Выявление отличительных признаков между кадрами

Самым простым способом вычитания фона является вычитание одного кадра из другого (возможно расположенного несколько позже) с последующим выделением любой разницы, которая "достаточно большая" на переднем плане. Этот процесс стремится поймать границы движущихся объектов. Для простоты рассмотрим три одноканальных изображения: `frameTime1`, `frameTime2` и `frameForeground`. Во `frameTime1` поместим предыдущий кадр в оттенках серого, а в `frameTime2` текущий кадр в оттенках серого. Затем, воспользовавшись `cvAbsDiff()`, вычислим (в абсолютных значениях) разницу между передними планами и поместим результат в `frameForeground`.

```

cvAbsDiff(
    frameTime1
    ,frameTime2
    ,frameForeground
);

```

Так как в значениях пикселя всегда присутствует шум и колебания, необходимо игнорировать (устанавливать в 0) малые различия (например, меньше 15) и помечать остальные как большие различия (устанавливать в 255).

```
cvThreshold(  
    frameForeground  
, frameForeground  
, 15  
, 255  
, CV_THRESH_BINARY  
);
```

Таким образом изображение *frameForeground* будет содержать кандидатов на объекты переднего плана, отмеченные пикселями со значениями равными 255 и фон, отмеченный пикселями со значениями 0. Теперь, как уже было сказано ранее, необходимо избавиться от мелких шумовых областей; для этого можно использовать функцию *cvErode()* или связанную компоненту. Для цветных изображений данный подход также может быть использован, но необходимо применять его к каждому каналу в отдельности с последующим соединением их обратно с помощью *cvOr()*. Этот метод является слишком простым для большинства приложений и позволяет отмечать лишь области движения. Для получения более эффективной модели фона, необходимо накапливать некоторую статистику о средних значениях и средних различиях между пикселями сцены. Забегая немного вперед, можно посмотреть примеры различий между кадрами на рисунке 9-5 и рисунке 9-6 в разделе "Быстрый тест".

Метод усреднения фона

Основу метода усреднения для создания модели фона составляют среднее значение и стандартное отклонение (или аналогичная, но более быстро вычисляемая, средняя разница) каждого пикселя.

Рассмотрим линию пикселей с рисунка 9-1. Вместо построения одной последовательности значений для каждого пикселя, можно представить изменения каждого пикселя на протяжении всего видеофайла с точки зрения среднего значения и средней разности (рисунок 9-2). В этом же видео объект переднего плана (который, по факту, является рукой) перемещается перед камерой. Этот объект на переднем плане не столь яркий, как небо или дерево на фоне. Яркость руки так же показана на рисунке.

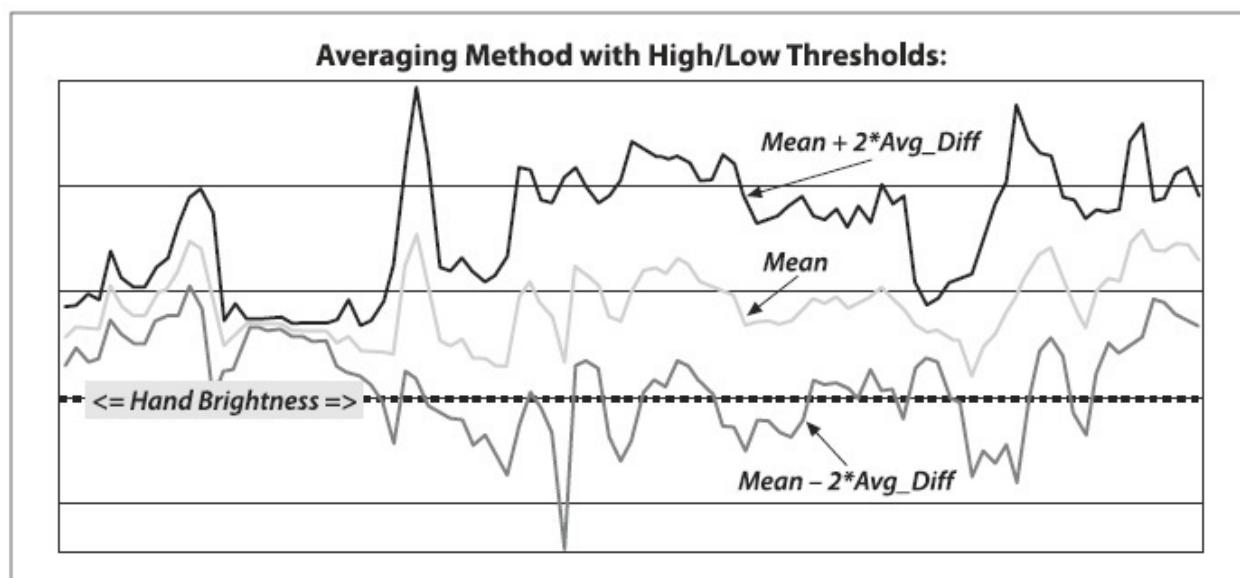


Рисунок 9-2. Данные рисунка 9-1 представлены в виде средних различий: объект (рука), который перемещается перед камерой несколько темнее, что, собственно, и отображает график

Метод усреднения использует четыре функции OpenCV: *cvAcc()* для накопления кадров в течении заданного времени; *cvAbsDiff()* для накопления изменений от кадра к кадру в течении заданного времени; *cvInRange()* для выделения сегментов переднего и заднего планов (после получения модели фона); *cvOr()* для объединения сегментов, полученных от разных цветовых каналов, в единую маску изображения. Так как этот пример содержит значительное количество кода, он будет разбит на смысловые части и каждая часть будет рассмотрена в отдельности.

Вначале создаются указатели на несколько черновых и хранящих статистику изображений, потребность в которых будет на протяжении всего жизненного цикла программы. Возможно, будет полезно отсортировать эти указатели, исходя из типа изображений, на которые они ссылаются.

```
// Глобальное хранилище
// Float, 3-channel images
//
IplImage *IavgF,*IdiffF, *IprevF, *IhiF, *IlowF;
IplImage *Iscratch, *Iscratch2;

// Float, 1-channel images
//
IplImage *Igray1,*Igray2, *Igray3;
IplImage *Ilow1, *Ilow2, *Ilow3;
IplImage *Ihi1, *Ihi2, *Ihi3;

// Byte, 1-channel image
//
IplImage *Imaskt;

// Подсчет количества изображений, которые были изучены
// для последующего усреднения
//
float Icount;
```

Затем создается единый вызов для выделения памяти под все необходимые промежуточные изображения. Для удобства будет передано одно изображение (из видео), которое может быть использовано в качестве эталона для калибровки промежуточных изображений.

```

// Это просто эталонное изображения для распределительных
// целей (используется для калибровки)
//

void AllocateImages( IplImage* I ) {
    CvSize sz = cvGetSize( I );

    IavgF   = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    IdiffF  = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    IprevF  = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    IhiF    = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    IlowF   = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    Ilow1   = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Ilow2   = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Ilow3   = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Ihi1    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Ihi2    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Ihi3    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );

    cvZero( IavgF );
    cvZero( IdiffF );
    cvZero( IprevF );
    cvZero( IhiF );
    cvZero( IlowF );

    Icount = 0.00001; // защита от деления на 0

    Iscratch   = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    Iscratch2  = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
    Igray1    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Igray2    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Igray3    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
    Imaskt    = cvCreateImage( sz, IPL_DEPTH_8U, 1 );

    cvZero( Iscratch );
    cvZero( Iscratch2 );
}

```

В следующем куске кода представлены процессы накопления фона изображения и накопления абсолютных значений межкадровых разностей изображения (процесс вычисления длится быстрее "прокси" (средняя разница не является математическим эквивалентом стандартному отклонению, но в данном контексте достаточно близко, чтобы давать результаты аналогичного качества) для изучения стандартного отклонения пикселей изображения). Эта операция выполняется, как правило, для 30-1000 кадров, иногда только один кадр от каждой секунды, а иногда и для всех кадров. Процедура вызывается для трехканального изображения глубиной 8 бит.

```

// Изучение статистики фона кадра
// I - это цветной образец фона, 3-канальный, 8и
//
void accumulateBackground( IplImage *I ) {
    static int first = 1;                      // не потокобезопасно
    cvCvtScale( I, Iscratch, 1, 0 );           // конвертация во float

    if( !first ) {
        cvAcc( Iscratch, IavgF );
        cvAbsDiff( Iscratch, IprevF, Iscratch2 );
        cvAcc( Iscratch2, IdiffF );
        Icount += 1.0;
    }

    first = 0;
    cvCopy( Iscratch, IprevF );
}

```

Сначала используется функция `cvCvtScale()` для преобразования необработанного фонового 8-битного трехканального изображения в трехканальное изображение типа float. Затем происходит накопление необработанного float-изображения в `IavgF`. Потом происходит вычисление межкадровой абсолютной разницы при помощи `cvAbsDiff()` и накопление её в изображении `IdiffF`. Каждый раз при накоплении этих изображений увеличивается глобальный счетчик изображений `Icount`, который будет использован позже для усреднения.

Единожды накопив достаточное количество кадров, можно конвертировать их в статистическую модель фона. В результате это позволит вычислить среднее значение и отклонение (абсолютное значение средней разницы) каждого пикселя.

```

void createModelsfromStats() {
    cvConvertScale( IavgF, IavgF, (double)(1.0/Icount) );
    cvConvertScale( IdiffF, IdiffF, (double)(1.0/Icount) );

    // Модификация разности, чтобы она всегда была не равна 0
    //
    cvAddS( IdiffF, cvScalar( 1.0, 1.0, 1.0 ), IdiffF );
    setHighThreshold( 7.0 );
    setLowThreshold( 6.0 );
}

```

В представленном куске кода функция `cvConvertScale()` вычисляет значения среднего и абсолютной разницы за счет деления на число накопленных изображений. В качестве меры предосторожности, предполагается, что средняя разница изображений,

по крайней мере, меньше 1; данный показатель необходимо будет изменять во время пороговых преобразований и избегать ситуаций, когда эти два порога могут стать равными.

Обе вспомогательные функции *setHighThreshold()* и *setLowThreshold()* задают порог, основываясь на абсолютном значении межкадровой средней разности. Вызов *setHighThreshold(7.0)* устанавливает порог таким, что при любом значении, которое превысит в 7 раз среднюю абсолютную разность для этого пикселя, указывает на то, что этот пиксель принадлежит переднему плану; аналогичным образом вызов *setLowThreshold(6.0)* устанавливает нижний порог. Значение, лежащее между этими порогами, указывает на принадлежность пикселя к фону.

```
void setHighThreshold( float scale ) {
    cvConvertScale( IdiffF, Iscratch, scale );
    cvAdd( Iscratch, IavgF, IhiF );
    cvSplit( IhiF, Ihi1, Ihi2, Ihi3, 0 );
}

void setLowThreshold( float scale ) {
    cvConvertScale( IdiffF, Iscratch, scale );
    cvSub( IavgF, Iscratch, IlowF );
    cvSplit( IlowF, Ilow1, Ilow2, Ilow3, 0 );
}
```

И вновь в *setLowThreshold()* и *setHighThreshold()* используется *cvConvertScale()* для перемножения значений перед сложением или вычитанием этих диапазонов по отношению к *IavgF*. Это действие устанавливает *IhiF* и *IlowF* диапазоны для каждого канала изображения при помощи *cvSplit()*.

Как только появляется модель фона в комплекте с верхним и нижним порогами можно использовать это для сегментации изображения на передний план (то, что не "указано" на фоновом изображении) и фон (все, что находится между верхним и нижним порогами фоновой модели). Сегментация выполняется при помощи следующего куска кода:

```

// Создание бинарного изображения: маска 0,255, где 255 указывает на передний план
// I - входное трехканальное 8-битное изображение
// Imask - одноканальное 8-битное изображение маски, которое должно быть создано
//
void backgroundDiff( IplImage *I, IplImage *Imask ) {
    cvCvtScale(I,Iscratch,1,0); // Конвертирование во float
    cvSplit( Iscratch, Igray1,Igray2,Igray3, 0 );

    // Channel 1
    //
    cvInRange(Igray1,Ilow1,Ihi1,Imask);

    // Channel 2
    //
    cvInRange(Igray2,Ilow2,Ihi2,Imaskt);
    cvOr(Imask,Imaskt,Imask);

    // Channel 3
    //
    cvInRange(Igray3,Ilow3,Ihi3,Imaskt);
    cvOr(Imask,Imaskt,Imask);

    // Инвертирование результата
    //
    cvSubRS(Imask, cvScalar(255), Imask);
}

```

Вначале эта функция преобразует исходное изображение *I* (изображение, которое необходимо сегментировать) в вещественное изображение с помощью функции *cvCvtScale()*. Затем выполняется конвертирование трехканального изображения в одноканальные при помощи *cvSplit()*. Потом эти одноканальные изображения проходят проверку на соответствие верхней и нижней амплитуде среднего фонового пикселя при помощи функции *cvInRange()*, которая устанавливает значения пикселей 8-битного изображения *Imask* в *max* (255), когда данное значение лежит в указанном диапазоне, иначе в 0. Используя логическую функцию OR для каждого канала, осуществляется перенос результатов сегментации на изображение *Imask*; в результате любые сильные различия в любом канале можно рассматривать как свидетельство принадлежности пикселя к объекту переднего плана. В заключении, происходит инвертирование *Imask* при помощи *cvSubRS()*, потому что передний фон должен содержать значения вне диапазона, а не в диапазоне. Изображение маски является результатом.

Так же необходимо освободить память, занимаемую изображениями по завершении использования фоновой модели:

```

void DeallocateImages() {
    cvReleaseImage( &IavgF );
    cvReleaseImage( &IdiffF );
    cvReleaseImage( &IprevF );
    cvReleaseImage( &IhiF );
    cvReleaseImage( &IlowF );
    cvReleaseImage( &Ilow1 );
    cvReleaseImage( &Ilow2 );
    cvReleaseImage( &Ilow3 );
    cvReleaseImage( &Ihi1 );
    cvReleaseImage( &Ihi2 );
    cvReleaseImage( &Ihi3 );
    cvReleaseImage( &Iscratch );
    cvReleaseImage( &Iscratch2 );
    cvReleaseImage( &Igray1 );
    cvReleaseImage( &Igray2 );
    cvReleaseImage( &Igray3 );
    cvReleaseImage( &Imaskt );
}

```

В результате был рассмотрен простой метод изучения фона сцены, и сегментация объектов переднего плана. Этот метод показывает хорошие результаты, когда сцена не содержит движущихся объектов фона (например, развивающихся занавесок или деревьев). Так же предполагается, что освещение остается постоянным (например, в помещении неподвижных сцен). Результаты работы метода представлены на рисунке 9-5.

Накопление среднего значения, дисперсии и ковариации

Только что рассмотренный метод усреднения при вычитании фона использовал одну аккумулирующую функцию *cvAcc()*. Она принадлежит к группе вспомогательных функций и служит для накопления (аккумулирования) сумм изображений, квадратов изображений, перемноженных изображений и усредненных изображений, из которых можно вычислить базовую статистику (среднее значение, дисперсию, ковариацию) сцены в целом или в частности. В этом разделе также будут рассмотрены и другие функции из этой группы.

Изображение во всех указанных функциях должно иметь одинаковые размеры по высоте и ширине. В каждой из функций, исходные изображения, именуемые *image*, *image1* и *image2*, могут быть одно- или трехканальными (8-бит на канал) или вещественным (32F) массивом изображений. Конечные изображения, именуемые *sum*, *sqsum* и *acc*, могут быть массивами одинарной точности (32F) или двойной (64F). На аккумулирующую функцию изображение *mask* (если имеется) накладывает ограничения на обработку только тех мест, где элементы маски отличны от нуля.

Определение среднего значения. Наиболее простой метод нахождения среднего заключается в сложении всего набора изображений с использованием `cvAcc()` и последующим делением на общее количество изображений.

```
void cvAcc(
    const Cvrr*    image
    ,CvArr*        sum
    ,const CvArr*   mask = NULL
);
```

Альтернативный метод заключается в использование скользящего среднего:

```
void cvRunningAvg(
    const CvArr*   image
    ,CvArr*        acc
    ,double         alpha
    ,const CvArr*   mask = NULL
);
```

Скользящее среднее может быть найдено по следующей формуле:

$$\text{acc}(x, y) = (1 - \alpha) \cdot \text{acc}(x, y) + \alpha \cdot \text{image}(x, y) \quad \text{if } \text{mask}(x, y) \neq 0$$

Для константного значения α , скользящее среднее не будет эквивалентно результатам суммирования с помощью `cvAcc()`. Чтобы в этом убедиться, просто просуммируем числа (2, 3 и 4) и установим $\alpha = 0.5$. Если просуммировать их при помощи `cvAcc()`, то сумма была бы равна 9, а среднее значение 3. Если же просуммировать их при помощи `cvRunningAverage()`, тогда первая сумма даст $0.5 \times 2 + 0.5 \times 3 = 2.5$, а последующее добавление третьей составляющей даст результат $0.5 \times 2.5 + 0.5 \times 4 = 3.25$. Причина, по которой второе число больше заключается в том, что последние вклады дают большие веса, чем более отдаленные во времени. Скользящее среднее зачастую называют *трекером*. Параметр α устанавливает время, на протяжении которого предыдущий кадр будет оказывать влияние.

Определение дисперсии. Возможность аккумулировать квадраты изображений позволяет быстро вычислять дисперсии отдельных пикселей.

```
void cvSquareAcc(
    const CvArr*   image
    ,CvArr*        sqsum
    ,const CvArr*   mask = NULL
);
```

Дисперсия конечной последовательности определяется по формуле:

$$\sigma^2 = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \bar{x})^2$$

где \bar{x} – среднее значение x для всех образцов N . Слабой стороной данной формулы является то, что для её вычисления требуется выполнять два прохода по изображению: один проход для вычисления \bar{x} , второй проход для вычисления σ^2 . Ниже представленная формула работает столь же хорошо, как и ранее представленная:

$$\sigma^2 = \left(\frac{1}{N} \sum_{i=0}^{N-1} x_i^2 \right) - \left(\frac{1}{N} \sum_{i=0}^{N-1} x_i \right)^2$$

Используя данную формулу, можно накопить за один проход и значения пикселей и их квадраты. Таким образом, дисперсия пикселя равна разности между средним арифметическим квадратов и квадратом среднего арифметического.

Определение ковариации. Увидеть, как меняется изображение с течением времени, можно путем выбора определенного лага (отставания), а затем умножить текущее изображение на изображение из прошлого, которое соответствует данному лагу (отставанию). Функция `cvMultiplyAcc()` будет выполнять попиксельное перемножение двух изображений, а затем добавлять результат "нарастающим итогом" в `acc`:

```
void cvMultiplyAcc(
    const CvArr* image1
    , const CvArr* image2
    , CvArr* acc
    , const CvArr* mask = NULL
);
```

Для ковариации есть формула, аналогичная формуле для дисперсии. Эта формула выполняет вычисления за один проход за счет алгебраических преобразований стандартной формулы:

$$\text{Cov}(x, y) = \left(\frac{1}{N} \sum_{i=0}^{N-1} (x_i y_i) \right) - \left(\frac{1}{N} \sum_{i=0}^{N-1} x_i \right) \left(\frac{1}{N} \sum_{j=0}^{N-1} y_j \right)$$

где x - изображение в момент времени t , y - в момент времени $t - d$ (d - отставание).

Можно использовать аккумулирующие функции, описанные здесь, чтобы создать несколько моделей фона, основанных на статистике. В иной литературе можно найти различные вариации базовой модели, используемой в приведенном примере. Как правило, при применении данной модели возникает необходимость в расширении

данной модели до более специализированной версии. К примеру, простым улучшением будет являться адаптивная бинаризация, способная приспосабливаться к изменению глобальных переменных.

Усовершенствованная модель фона

Многие фоновые сцены содержат движущиеся объекты, такие как раскачивающиеся деревья на ветру, включенные вентиляторы, развевающиеся занавески и т.д. Зачастую такие сцены также содержат переменное освещение из-за движущихся облаков или дверей и окон по-разному пропускающие свет.

Хороший способ справиться с этим заключается в применении временной модели для каждого пикселя или группы пикселей. Такой вид модели хорошо взаимодействует с временными колебаниями, однако, большим минусом является то, что данная модель потребляет большое количество памяти. Если использовать 2 секунды предыдущей информации с частотой 30 Hz, то потребуется 60 образцов для каждого пикселя. Затем результирующая модель каждого пикселя должна закодировать то, что она изучила 60 различных адаптивных весов. Зачастую возникает потребность в накоплении фоновой статистики значительно больше, чем 2 секунды; это означает, что такие методы нецелесообразно использовать на современном оборудовании.

Для максимально возможного увеличения производительности адаптивной фильтрации, необходимо позаимствовать метод сжатия видео и попытаться сформировать *кодовую книгу* для воспроизведения наиболее важных положений на фоне. Простой путь сделать это заключается в сравнении нового значения определенного пикселя с его предыдущим значением. Если значение довольно таки близко к предыдущему, то моделируется искажение цвета. Иначе должна создаваться новая группа из цветов, ассоциированная с этим пикселим. Результат может быть представлен как набор *blobs* (капель), перемещающихся в пространстве RGB, где каждый *blob*, представляющий отдельную часть, рассматривается как возможная составляющая фона.

На практике выбор пространства RGB не является оптимальным. Практически всегда лучше использовать цветовое пространство, у которого ось совпадают с яркостью, такое как YUV. (Выбор YUV является наиболее распространенным, хотя пространство HSV, где V – яркость, также является неплохим выбором.) Причина использования таких цветовых пространств заключается в том, что эмпирически большинство вариаций фона принадлежат оси яркости, а не цветовой оси.

Теперь рассмотрим, как моделировать *blobs*. Собственно, это можно сделать так же, как и раньше в простой модели. Например, можно выбрать модель *blob* как Гауссовые кластеры со средним или ковариацией. Оказывается, в простейшем случае, *blobs*

являются просто *boxes* с изученной степенью каждой из трех осей цветового пространства. Этот простейший метод с точки зрения требуемой памяти и вычислительных ресурсов для определения того, будет ли вновь наблюдаемый пиксель внутри любого из изученных *boxes*.

Для понимания того, что такое кодовая книга, воспользуемся простейшим примером (рисунок 9-3). Кодовая книга состоит из *boxes*, которые разрастаются по мере покрытия значений, наблюдаемые во времени. В верхней части рисунка 9-3 отображено колебание сигналов во времени. В нижней части рисунка отображено формирование *box* из новых значений с последующим разрастанием за счет поглощения соседних значений. Если значение слишком далеко, то формируется новый *box* и процесс повторяется.

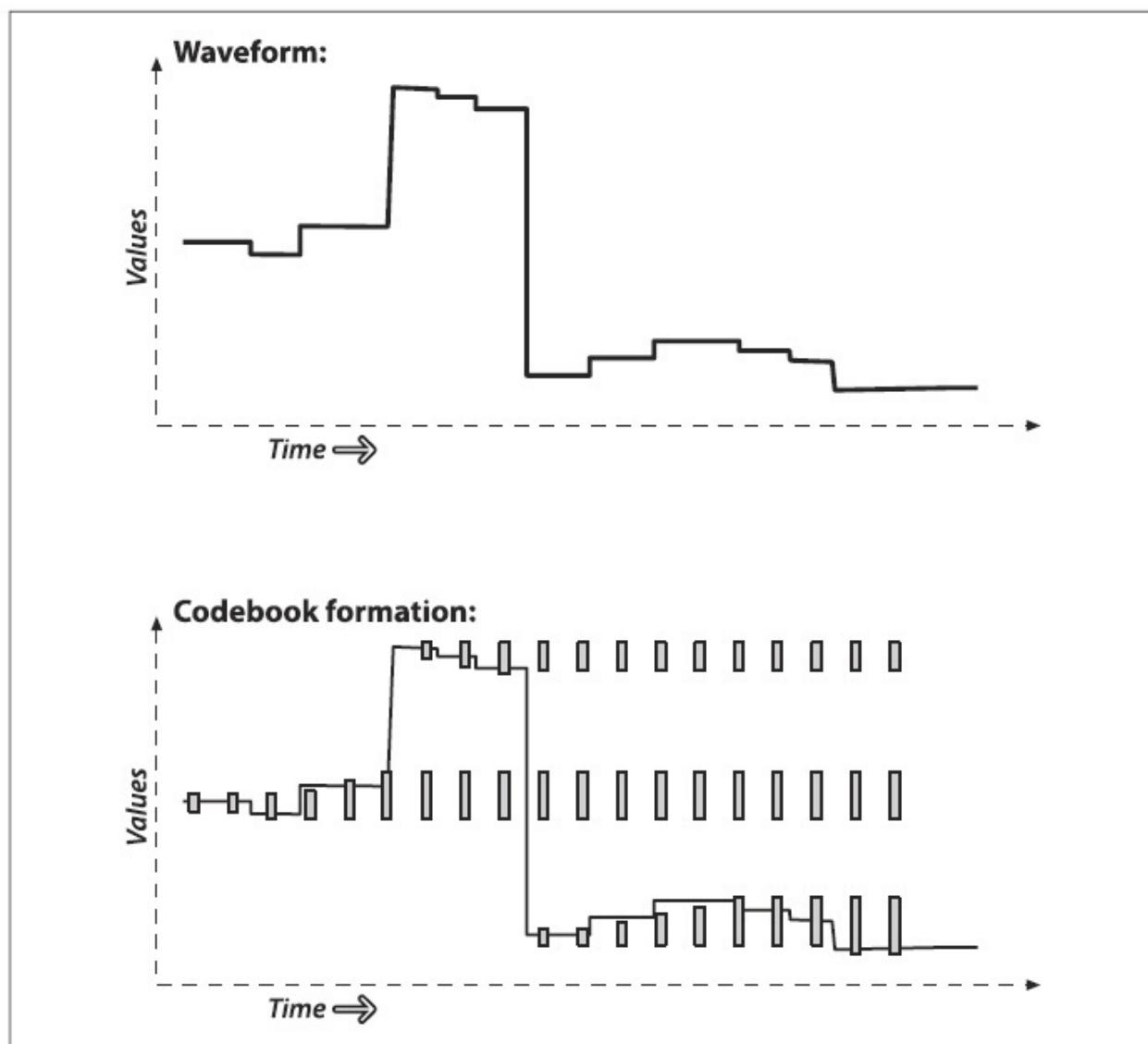


Рисунок 9-3. Кодовая книга — это просто *boxes*, разделяющие значения яркости: *box* формируется для поглощения нового значения и медленно разрастается за счет близлежащих значений; если значение лежит далеко, то формируется новый *box*

Для изучаемой модели фона будет рассмотрена кодовая книга, покрывающая три измерения: три канала каждого пикселя изображения. Рисунок 9-4 визуализирует (измерение интенсивности) кодовую книгу для шести различных пикселей, извлеченных из данных рисунка 9-1. Этот метод может взаимодействовать с пикселями, уровня которых резко изменяются (например, пиксели, относящиеся к листьям деревьев обдуваемые ветром или синему небу позади этих деревьев, которые могут быть представлены различными цветами). При помощи данного более точного метода моделирования можно обнаружить объекты переднего плана, которые имеют значения, лежащие между значениями пикселей. Сравнение, показанное на рисунке 9-2, доказывает, что метод усреднения не может выделить значения, соответствующие руке (показано прерывистой линией), из пиксельных колебаний. Забегая немного вперед, стоит отметить тот факт, что производительность кодовой книги в сравнении с методом усреднения (рисунок 9-7) выше.

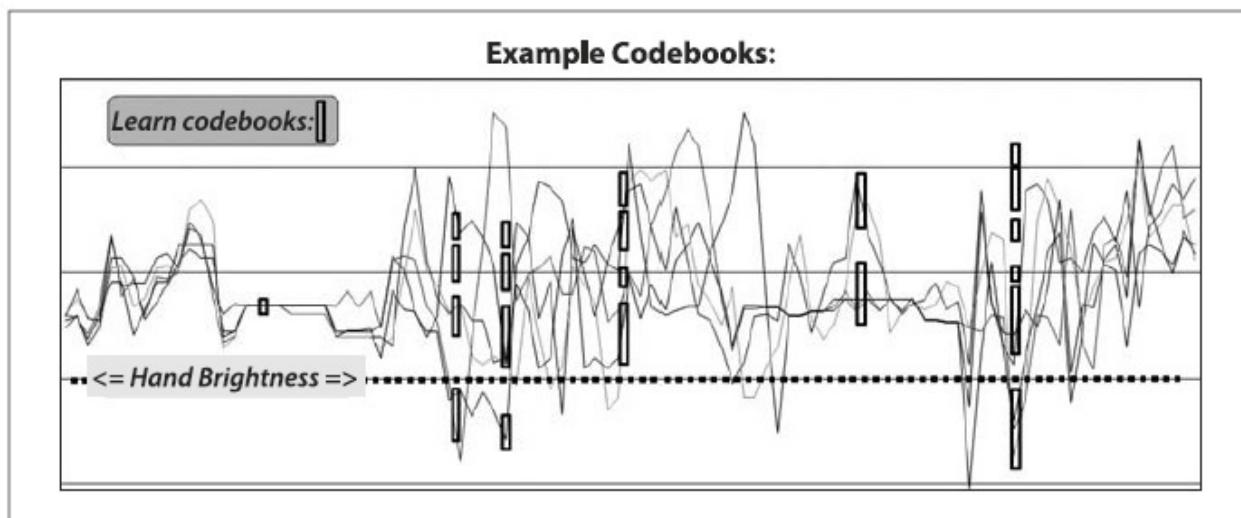


Рисунок 9-4. Часть колебаний яркости изученных записей кодовой книги для шести выбранных пикселей (показаны вертикальными *box*): *box* кодовой книги собирают пиксели, которые принимают одно из дискретных значений, в результате чего лучше моделируется модель дискретных распределений; это позволяет обнаружить руку, как объект переднего плана (показано на рисунке точечной линией), среднее значение которого лежит между значениями пикселей, принадлежащих фону. В данном случае кодовая книга имеет только одно измерение и может представлять только колебания яркости

В методе кодовой книги для изучения модели фона каждый *box* определяется двумя порогами (*max* и *min*) для каждой цветовой оси. Эти пороги границ *box* расширяются (*max* становится больше, *min* становится меньше), если новые фоновые образцы попадут внутрь изученных границ (*learnHigh* и *learnLow*) выше *max* или ниже *min* соответственно. Если новый фоновый образец попадает вне границ *box* и его изученных порогов, тогда будет создан новый *box*. В режиме *вычитания фона*

используются пороги *maxMod* и *minMod*, которые свидетельствуют о том, что если пиксель "достаточно близок" к *min* и *max* границе *box*, тогда можно считать, что он внутри *box*. Второй порог позволяет регулировать модели под конкретные условия.

Структуры

Теперь настало время для более детального разбора алгоритма кодовой книги. Для начала необходимо создать структуру кодовой книги, которая будет просто указывать на группу *boxes* пространства YUV:

```
typedef struct code_book {
    code_element **cb;
    int numEntries;
    int t; // количество обращений
} codeBook;
```

numEntries - количество записей в кодовой книге. Переменная *t* подсчитывает число точек, накопленных от начала или с последней операции очищения. Описание структуры элемента представлено ниже:

```
#define CHANNELS 3

typedef struct ce {
    uchar learnHigh[CHANNELS]; // Верхний порог обучения
    uchar learnLow[CHANNELS]; // Нижний порог обучения
    uchar max[CHANNELS]; // Верхняя граница box
    uchar min[CHANNELS]; // Нижняя граница box
    int t_last_update; // Позволяет убирать устаревшие записи
    int stale; // max negative run (продолжительный период неактивности)
} code_element;
```

Каждая запись кодовой книги расходует 4 байта на канал плюс 2 или *CHANNELSx4+4+4* байт (20 байт при использовании трех каналов). Можно установить *CHANNELS* в любое положительное целое значение меньшее или равное числу каналов цветного изображения, но, как правило, используется 1 ("Y" или только яркость) или 3 (YUV, HSV). В этой структуре для каждого канала *max* и *min* - это границы кодовой книги. Параметры *learnHigh* и *learnLow* - это пороговые значения, управляющие созданием новых записей, а именно: новая запись будет создана, если попадется новый пиксель не лежащий в диапазоне *min* - *learnLow* и *max* + *learnHigh* в каждом из каналов. Время последнего обновления *t_last_update* и *stale* используются для удаления редко используемых записей кодовой книги во время обучения. Теперь можно переходить к изучению функций, которые используют эту структуру для изучения динамически меняющегося фона.

Изучение фона

Предположим, имеется один объект `codeBook` с `code_elements` для каждого пикселя. Тогда для всего изображения, необходим массив `codeBook`, который по длине равен числу пикселей изображения. Для каждого пикселя вызывается функция `update_codebook()` ровным счетом столько раз, чтобы охватить соответствующие изменения в фоне. Процесс обучения может периодически обновляться, а функция `clear_stale_entries()` может быть использована для обучения модели фона в присутствии (небольшого числа) движущихся объектов. Это становится возможным за счет удаления редко используемых "устаревших" записей, созданных благодаря движущимся объектам. Реализация `update_codebook()` представлена далее:

```
///////////////////////////////
// int update_codebook(uchar *p, codeBook &c, unsigned cbBounds)
// Обновление записей codebook за счет указателя на новые данные
//
// p           Указатель на пиксель YUV
// c           Codebook для этого пикселя
// cbBounds    Изучаемые границы codebook (Rule of thumb: 10)
// numChannels Число каналов
//
// NOTES:
//      cvBounds должна иметь длину равную numChannels
//
// RETURN
//      codebook index
//

int update_codebook(
    uchar*        p
    ,codeBook&    c
    ,unsigned*    cbBounds
    ,int         numChannels
) {
    unsigned int high[3], low[3];
    for(n = 0; n < numChannels; n++) {
        high[n] = *(p+n) + *(cbBounds+n);

        if(high[n] > 255) {
            high[n] = 255;
        }

        low[n] = *(p+n) - *(cbBounds+n);

        if(low[n] < 0) {
            low[n] = 0;
        }
    }

    int matchChannel;
```

```

// Проверка соответствия существующему кодовому слову
//
for(int i = 0; i < c.numEntries; i++) {
    matchChannel = 0;
    for(n = 0; n < numChannels; n++) {
        // Found an entry for this channel
        if( (c.cb[i]->learnLow[n] <= *(p+n))
            && (*(p+n) <= c.cb[i]->learnHigh[n]))
        ) {
            matchChannel++;
        }
    }

    // Если запись найдена
    //
    if(matchChannel == numChannels) {
        c.cb[i]->t_last_update = c.t;

        // Настройка этого кодового слова для первого канала
        //
        for(n = 0; n < numChannels; n++) {
            if(c.cb[i]->max[n] < *(p+n)) {
                c.cb[i]->max[n] = *(p+n);
            } else if(c.cb[i]->min[n] > *(p+n)) {
                c.cb[i]->min[n] = *(p+n);
            }
        }

        break;
    }
}

// ... продолжение следует

```

Эта функция расширяет или добавляет запись в кодовую книгу, когда пиксель p выходит за границы существующего *box* кодовой книги. Если пиксель находится за пределами *cbBounds*, то создается новый *box* кодовой книги. В начале, функция задает значения уровней *high* и *low*, которые будут использованы позже. Затем происходит перебор каждой записи кодовой книги, чтобы проверить находится ли значение пикселя $*p$ внутри пределов исследуемого "box" кодовой книги. Если пиксель находится внутри исследуемой границы для всех каналов, тогда соответствующие уровни *max* и *min* корректируются для того, чтобы включить данный пиксель, а время последнего обновления устанавливается в соответствии с текущим временным отсчетом *c.t*. Затем *update_codebook()* подсчитывает статистику обращений к записям кодовой книги:

```
// ... продолжение

// Накладные расходы для отслеживания потенциально устаревших записей
//
for(int s = 0; s < c.numEntries; s++) {
    // Отслеживание устаревших записей
    //
    int negRun = c.t - c.cb[s]->t_last_update;
    if(c.cb[s]->stale < negRun) {
        c.cb[s]->stale = negRun;
    }
}

// ... продолжение следует
```

В результате переменная *stale* будет содержать наиболее *negative runtime* (т.е. наибольший промежуток времени, за который не было обращений к данным). Отслеживание неиспользуемых записей позволяет удалять кодовые книги, которые были созданы из шума или движущихся объектов и, следовательно, имеющие тенденцию оставаться не используемыми на протяжении всего времени. На следующем этапе обучения *update_codebook()* создает новую кодовую книгу, если это необходимо:

```

// ... продолжение

// Ввод нового кодового слова по необходимости
// если ни одно из существующих кодовых слов не найдено
//
if(i == c.numEntries) {
    code_element **foo = new code_element* [c.numEntries+1];

    for(int ii = 0; ii < c.numEntries; ii++) {
        foo[ii] = c.cb[ii];
    }

    foo[c.numEntries] = new code_element;

    if(c.numEntries) {
        delete [] c.cb;
    }

    c.cb = foo;

    for(n = 0; n < numChannels; n++) {
        c.cb[c.numEntries]->learnHigh[n] = high[n];
        c.cb[c.numEntries]->learnLow[n] = low[n];
        c.cb[c.numEntries]->max[n] = *(p+n);
        c.cb[c.numEntries]->min[n] = *(p+n);
    }

    c.cb[c.numEntries]->t_last_update = c.t;
    c.cb[c.numEntries]->stale = 0;
    c.numEntries += 1;
}

// ... продолжение следует

```

И наконец, *update_codebook()* неспешно корректирует (добавляя по 1) обучаемые границы *learnHigh* и *learnLow*, если пиксель был найден за пределами порогов, но все еще в пределах границ *high* и *low*:

```
// ... продолжение

// Несспешная корректировка обучаемых границ
//
for(n=0; n<numChannels; n++) {
    if(c.cb[i]->learnHigh[n] < high[n]) {
        c.cb[i]->learnHigh[n] += 1;
    }

    if(c.cb[i]->learnLow[n] > low[n]) {
        c.cb[i]->learnLow[n] -= 1;
    }
}

return(i);
}
```

Функция завершает свою работу возвращением индекса измененной кодовой книги. Для того чтобы вести обучение в присутствии движущихся фоновых объектов и для того чтобы не зависеть от шума необходимо удалять редко используемые во время обучения записи.

Изучение модели фона при наличии движущихся объектов переднего плана

Следующая функция *clear_stale_entries()* позволяет обучать модель в присутствии движущихся объектов.

```
///////////////////////////////
// int clear_stale_entries(codeBook &c)
// Во время обучения, по истечению некоторого времени, периодически
// происходит очищение от устаревших записей кодовой книги
//
// c      Codebook для очистки
//
// Return
//      количество удаленных записей
//
int clear_stale_entries(codeBook &c) {
    int staleThresh = c.t>>1;
    int *keep = new int [c.numEntries];
    int keepCnt = 0;

    // Просмотр устаревших записей кодовой книги
    //
    for(int i = 0; i < c.numEntries; i++){
        if(c.cb[i]->stale > staleThresh)
            keep[i] = 0; // Отметка удаления
        else {
            keep[i] = 1; // Отметка сохранения keepCnt += 1;
        }
    }

    // Сохранение только хороших
    //
    c.t = 0; // Сброс прошлых сложений
    code_element **foo = new code_element* [keepCnt]; int k=0;

    for(int ii=0; ii < c.numEntries; ii++){
        if(keep[ii]) {
            foo[k] = c.cb[ii];

            // Необходимо обновить записи до следующего clearStale
            //
            foo[k]->t_last_update = 0;
            k++;
        }
    }

    // Очистка
    //
    delete [] keep;
    delete [] c.cb;
    c.cb = foo;
    int numCleared = c.numEntries - keepCnt;
    c.numEntries = keepCnt;

    return(numCleared);
}
```

Функция начинает свою работу с определения переменной *staleThresh*, которая жестко хранит половину общего времени работы *c.t*. Это означает, что если за время обучения к записи *i* кодовой книги не происходило обращение в течение половины времени работы, то запись *i* помечается для удаления (*keep[i] = 0*). Длина вектора *keep[]* равна *c.numEntries*, чтобы существовала возможность отметить каждую запись кодовой книги. Переменная *keepCnt* хранит количество записей, которые необходимо оставить. После сохранения оставленных записей в кодовой книге, происходит создание указателя *foo* на вектор указателей *code_elements* длиной *keepCnt* с последующим копированием в него устаревших записей. В заключении удаляется старый указатель на вектор кодовой книги и заменяется новым, не устаревшим вектором.

Вычитание фона: поиск объектов переднего плана

На данный момент уже были рассмотрено создание кодовой книги фона и её чистка от устаревших элементов. Теперь необходимо рассмотреть функцию *background_diff()*, которая использует обученную модель для отделения пикселей переднего плана от фона:

```
///////////////////////////////
// uchar background_diff( uchar *p, codeBook &c,
// int minMod, int maxMod)
// Данная функция определяет, является ли пиксель частью кодовой книги
//
// p          Указатель на пиксель (YUV interleaved)
// c          Ссылка на codebook
// numChannels Число каналов
// maxMod     Добавление (возможно отрицательного) числа к уровню
//             max при определении, что новый пиксель является частью объекта
//             переднего плана
// minMod    Вычитание (возможно отрицательного) числа из уровня
//             min при определении, что новый пиксель является частью объекта
//             переднего плана
// NOTES:
//     длины minMod и maxMod должна быть numChannels,
//     т.е. 3 канала => minMod[3], maxMod[3]. При этом по одному min и
//     одному max порогу на канал.
//
// Return
//     0 => фон, 255 => объект переднего плана
//
uchar background_diff(
    uchar*      p
    ,codeBook&   c
    ,int        numChannels
    ,int*       minMod
    ,int*       maxMod
) {
```

```

int matchChannel;

// Проверка принадлежности к существующему кодовому слову
//
for(int i=0; i<c.numEntries; i++) {
    matchChannel = 0;
    for(int n=0; n<numChannels; n++) {
        if((c.cb[i]->min[n] - minMod[n] <= *(p+n)) &&
        (*(p+n) <= c.cb[i]->max[n] + maxMod[n])) {
            matchChannel++; // Количество найденных записей для данного канала
        } else {
            break;
        }
    }

    if(matchChannel == numChannels) {
        break; // Найдена запись, соответствующая всем каналам
    }
}

if(i >= c.numEntries) {
    return(255);
}

return(0);
}

```

Функция вычитания фона похожа на функцию обучения *update_codebook()*, за исключением того факта, что в данной функции рассматриваются *max* и *min* плюс смещение порога, *maxMod* и *minMod* для каждого *box* кодовой книги. Если пиксель в пределах *box* плюс *maxMod* для верхней границы и минус *minMod* для нижней для каждого канала, то происходит инкремент переменной *matchChannel*. Фиксация записи, соответствующая всем каналам, происходит, когда значение *matchChannel* равно числу каналов. Если пиксель в пределах изучаемого *box*, тогда возвращается значение 255 (соответствует объектам переднего плана), иначе 0 (фон).

Три функции: *update_codebook()*, *clear_stale_elements()* и *background_diff()* составляют основу метода отделения переднего плана от фона с помощью кодовой книги.

Использование модели фона при помощи кодовой книги

Чтобы использовать метод отделения фона при помощи кодовой книги, необходимо выполнить следующие действия:

1. Обучить модель фона в течение нескольких секунд или минут, используя *update_codebook()*.
2. Избавиться от устаревших записей при помощи *clear_stale_entries()*.
3. Настроить пороги *minMod* и *maxMod* для лучшего отделения переднего плана.

4. Поддерживать модель сцены на высоком уровне.
5. Использовать обученную модель для отделения переднего плана от фона при помощи `background_diff()`.
6. Периодически обновлять модель.
7. Удалять устаревшие записи кодовой книги при помощи `clear_stale_entries()`.

Еще несколько размышлений по поводу модели кодовой книги

В основном, метод кодового словаря хорошо работает с обширным набором условий и его относительно легко обучить и запустить. Этот метод не очень хорошо работает при изменчивом освещении, т.е. когда происходит смена утра, дня и вечера или если кто-то включает и выключает свет в помещении. Этот тип глобальной изменчивости может быть учтен с помощью нескольких различных моделей кодового словаря, по одной для каждого условия, с последующим контролированием активной модели.

Связанные компоненты для очистки объектов переднего плана

Прежде, чем перейти к сравнению метода усреднения и метода кодовой книги, необходимо рассмотреть пути очистки изображения, используя связанные компоненты. Данная форма анализа основывается на исходном шумовом изображении маски; при этом используется морфологическая операция *открытия* для сокращения небольших шумовых областей с последующим применением операции *закрытия* для восстановления областей, которые были удалены операцией открытия. После этого появляется возможность найти "достаточно большие" контуры сохранившихся сегментов и (необязательно) собрать статистику по всем таким сегментам. При этом появляется возможность в получении самого большого контура или всех контуров выше определенного порога. Далее будет представлено большинство функций, которые могут потребоваться при работе со связными компонентами:

- Аппроксимация уцелевших компонентов контура к полигонам или выпуклым оболочкам
- Установка, насколько большим должен быть контур, чтобы не быть удаленным
- Установка максимального числа возвращаемых контуров
- (Необязательно) Возвращение *bounding boxes* сохранившихся контуров
- (Необязательно) Возвращение центров сохранившихся контуров

Заголовок связной компоненты, которая реализует данные операции:

```

///////////
// void find_connected_components(IplImage *mask, int poly1_hull0,
//                                 float perimScale, int *num,
//                                 CvRect *bbs, CvPoint *centers)
// Эта функция очищает сегмент маски, полученный при вызове backgroundDiff,
// от объектов переднего плана
//
// mask Серая (глубиной 8-bit) "строка" изображения маски, которую
// необходимо очистить
//
// OPTIONAL PARAMETERS:
// poly1_hull0 Если установлен, то аппроксимация связной компоненты до
//                 полигона или выпуклой оболочки (θ)
//                 (DEFAULT: polygon)
// perimScale Len = image (width+height)/perimScale. Если длина контура
//                 len < this, тогда удалить данный контур
//                 (DEFAULT: 4)
// num Максимальное число возвращаемых прямоугольников и/или центров;
//      (DEFAULT: NULL)
// bbs Указатель на bounding box прямоугольных векторов длинною num
//      (DEFAULT SETTING: NULL)
// centers Указатель на вектор центров контура длинною num
//      (DEFAULT: NULL)
//
void find_connected_components(
    IplImage* mask
    ,int poly1_hull0 = 1
    ,float perimScale = 4
    ,int* num = NULL
    ,CvRect* bbs = NULL
    ,CvPoint* centers = NULL
);

```

Разбор тела данной функции представлен ниже. В начале, задается хранилище под связные компоненты контура. Затем применяются морфологические операции открытия и закрытия для удаления небольших шумовых пикселей, после чего происходит восстановление разрушенных областей, которые сохранились после операции открытия. Функция принимает два дополнительных параметра, которые были жестко объявлены при помощи `#define`. Значения по умолчанию дают хорошие результаты и их вряд ли придётся когда-либо менять. Эти дополнительные параметры управляют тем, насколько простой должна быть граница переднего плана (чем выше число, тем проще) и сколько раз необходимо выполнять морфологические операции; чем выше число, тем выше размытие при выполнении операции открытия перед расширением во время выполнения операции закрытия. (Стоит отметить, что значение `CVCLOSE_ITR` на самом деле зависит от разрешения. Для изображений с очень высоким разрешением, значение по умолчанию равное 1, скорее всего не даст хороших результатов). Большое размытие устраняет крупные регионы шума за счет

размытия границ у более крупных регионов. Параметры, используемые в приведенном коде, были подобраны опытным путем и дают довольно таки хорошие результаты, однако, никто не запрещает с ними поэкспериментировать.

```
// Для связанных компонент:  
// Approx.threshold - чем больше, тем проще граница  
  
#define CVCONTOUR_APPROX_LEVEL 2  
  
// Сколько итераций размытия и/или расширения должно быть  
  
#define CVCLOSE_ITR 1
```

Теперь можно перейти непосредственно к рассмотрению самого алгоритма. Первая часть подпрограммы выполняет морфологические операции открытия и закрытия:

```
void find_connected_components(  
    IplImage* mask  
    , int      poly1_hull0  
    , float    perimScale  
    , int*     num  
    , CvRect*   bbs  
    , CvPoint*  centers  
) {  
    static CvMemStorage* mem_storage = NULL;  
    static CvSeq*        contours     = NULL;  
  
    // Очистка строки маски  
    //  
    cvMorphologyEx( mask, mask, 0, 0, CV_MOP_OPEN, CVCLOSE_ITR );  
    cvMorphologyEx( mask, mask, 0, 0, CV_MOP_CLOSE, CVCLOSE_ITR );
```

Теперь, после удаления шумов из маски, можно найти все контуры:

```
// Поиск контуров только в больших регионах
//
if( mem_storage == NULL ) {
    mem_storage = cvCreateMemStorage(0);
} else {
    cvClearMemStorage(mem_storage);
}

CvContourScanner scanner = cvStartFindContours(
    mask
    ,mem_storage
    ,sizeof(CvContour)
    ,CV_RETR_EXTERNAL
    ,CV_CHAIN_APPROX_SIMPLE
);
```

Далее производится отсеивание контуров, которые слишком малы, и аппроксимация остальных до полигонов или выпуклых областей (чья сложность задается в *CVCONTOUR_APPROX_LEVEL*):

```

CvSeq* c;
int numCont = 0;

while( (c = cvFindNextContour( scanner )) != NULL ) {
    double len = cvContourPerimeter( c );

    // Вычисление периметра
    //
    double q = (mask->height + mask->width)/perimScale;

    // Отсеивание blob, если его периметр слишком мал
    //
    if( len < q ) {
        cvSubstituteContour( scanner, NULL );
    } else {
        // Сглаживание краев, если достаточно большие
        //
        CvSeq* c_new;
        if( poly1_hull0 ) {
            // Polygonal approximation
            //
            c_new = cvApproxPoly(
                c
                ,sizeof(CvContour)
                ,mem_storage
                ,CV_POLY_APPROX_DP
                ,CVCONTOUR_APPROX_LEVEL
                ,0
            );
        } else {
            // Convex Hull of the segmentation
            //
            c_new = cvConvexHull2(
                c
                ,mem_storage
                ,CV_CLOCKWISE
                ,1
            );
        }
        cvSubstituteContour( scanner, c_new );
        numCont++;
    }
}

contours = cvEndFindContours( &scanner );

```

В предыдущем коде, *CV_POLY_APPROX_DP* подразумевает использование алгоритма приближения *Douglas-Peucker*, а *CV_CLOCKWISE* по умолчанию подразумевает выпуклую оболочку контура. Все эти обработки в результате приводят к получению

списка контуров. Перед нанесением контуров обратно на маску, необходимо определить цвета:

```
// Некоторые вспомогательные переменные
//
const CvScalar CVX_WHITE = CV_RGB(0xff, 0xff, 0xff)
const CvScalar CVX_BLACK = CV_RGB(0x00, 0x00, 0x00)
```

Эти определения будут использованы в следующем коде, где впервые произойдет обнуление маски с последующим нанесением контуров на маску. Кроме того, будет выполнена проверка потребности в сборе статистики по контурам (*bounding boxes* и центры):

```
// PAINT THE FOUND REGIONS BACK INTO THE IMAGE
//
cvZero( mask );
IplImage *maskTemp;

// CALC CENTER OF MASS AND/OR BOUNDING RECTANGLES
//
if(num != NULL) {
    // Пользователь хочет собирать статистику
    //
    int N = *num, numFilled = 0, i=0;
    CvMoments moments;
    double M00, M01, M10;
    maskTemp = cvCloneImage(mask);

    for(i = 0, c = contours; c != NULL; c = c->h_next, i++ ) {
        if(i < N) {
            // Only process up to *num of them
            //
            cvDrawContours(
                maskTemp
                ,c
                ,CVX_WHITE
                ,CVX_WHITE
                ,-1
                ,CV_FILLED
                ,8
            );

            // Поиск центра для каждого контура
            //
            if(centers != NULL) {
                cvMoments( maskTemp, &moments, 1 );
                M00 = cvGetSpatialMoment( &moments, 0, 0 );
                M10 = cvGetSpatialMoment( &moments, 1, 0 );
                M01 = cvGetSpatialMoment( &moments, 0, 1 );
                centers[i].x = (int)(M10/M00);
            }
        }
    }
}
```

```
        centers[i].y = (int)(M01/M00);
    }

    // Ограничивающие прямоугольники вокруг blobs
    //
    if(bbs != NULL) {
        bbs[i] = cvBoundingRect(c);
    }

    cvZero(maskTemp);
    numFilled++;
}

// Рисование контуров на маске
//
cvDrawContours(
    mask
    ,c
    ,CVX_WHITE
    ,CVX_WHITE
    ,-1
    ,CV_FILLED
    ,8
);
}

*num = numFilled;
cvReleaseImage( &maskTemp );
}
```

Если пользователю не нужны *bounding boxes* и центры в конечных регионах маски, необходимо просто отбросить контуры, представляющие достаточно большие связанные компоненты фона.

```

// Иначе просто нарисовать обработанные контуры на маске
//
else {
    // Пользователю не нужна статистика, просто нарисовать контуры
    //
    for( c = contours; c != NULL; c = c->h_next ) {
        cvDrawContours(
            mask
            ,c
            ,CVX_WHITE
            ,CVX_BLACK
            ,-1
            ,CV_FILLED
            ,8
        );
    }
}

```

Это все, что касается процедуры по созданию очищенных от шума масок. Теперь можно провести небольшой сравнительный анализ представленных методов вычитания фона.

Быстрый тест

Итак, рассмотрим пример работы алгоритма в реальной обстановке. Будем придерживаться видео с деревом за окном. Вспомним (рисунок 9-1), что в определенный момент времени перед камерой проводят рукой. Найти руку относительно легко, например, при помощи метода кадровой разности (метод был рассмотрен ранее). Основная идея метода заключается в вычитании текущего кадра из предыдущего и применение порога к разнице.

Последовательные кадры в видео, как правило, почти одинаковые. Следовательно, можно предположить, что в результате вычитания можно получить не так много информации, как, если бы на переднем плане появился движущийся объект. (В контексте кадровой разности, объект идентифицируется как "передний план", в большей степени скоростной. Это разумно только в сценах, которые, как правило, статичны или в сценах, где объекты переднего плана находятся гораздо ближе к камере, чем фоновые объекты (и, соответственно, передвигаться быстрее в силу проективной геометрии камер)). Но что означает "не так много" в данном контексте? На самом деле это означает "просто шум". На практике основная проблема заключается в удалении шума при наличии объекта переднего плана.

Что бы лучше осознать, что это за шум, рассмотрим пару кадров из видео, без объектов переднего плана – только фон и шум. На изображении 9-5 показан типичный кадр из видео (сверху слева) и предыдущий кадр (сверху справа). На рисунке так же

представлен результат вычитания кадров со значением порога равным 15 (снизу слева). Можно отметить, что значительная часть шума принадлежит движущимся листьям дерева. Тем не менее, метод связанных компонент способен довольно таки хорошо удалить рассеянный шум (снизу справа). (Порог связанный компоненты был подобран таким образом, чтобы получить нулевой отклик в пустых кадрах. Остается лишь выяснить, останется ли объект переднего плана (рука) при таком значении порога. Рисунок 9-6 дает ответ на этот вопрос. И это не удивительно, потому что нет оснований ожидать пространственных корреляций, а сигнал характеризуется большим числом очень малых регионов.

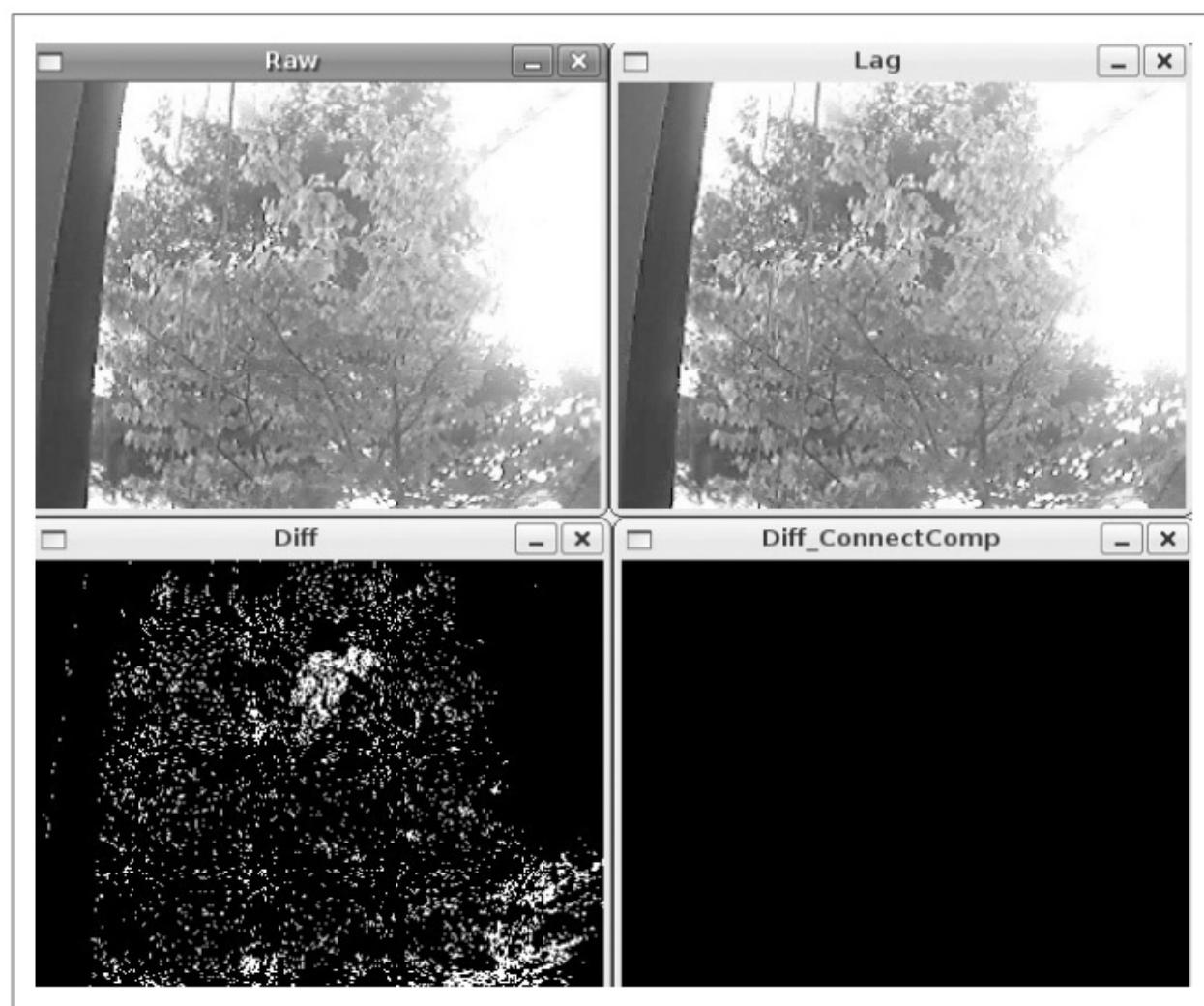


Рисунок 9-5. Межкадровая разность: раскачивающееся дерево на фоне в текущем (сверху слева) и предыдущем (сверху справа) кадрах; разность кадров (снизу слева) полностью очищенная (снизу справа) при помощи метода связанных компонент

Теперь рассмотрим ситуацию с объектом переднего плана (наша вездесущая рука) перемещающаяся перед камерой. На рисунке 9-6 показаны два кадра, схожие с кадрами с рисунка 9-5, за исключением того, что теперь на них присутствует перемещающаяся рука слева направо. Как и прежде показан текущий (сверху слева) и

предыдущий (сверху справа) кадры, разность кадров (снизу слева) очищенная (снизу справа) при помощи метода связанных компоненты. Результаты довольно-таки хорошие.

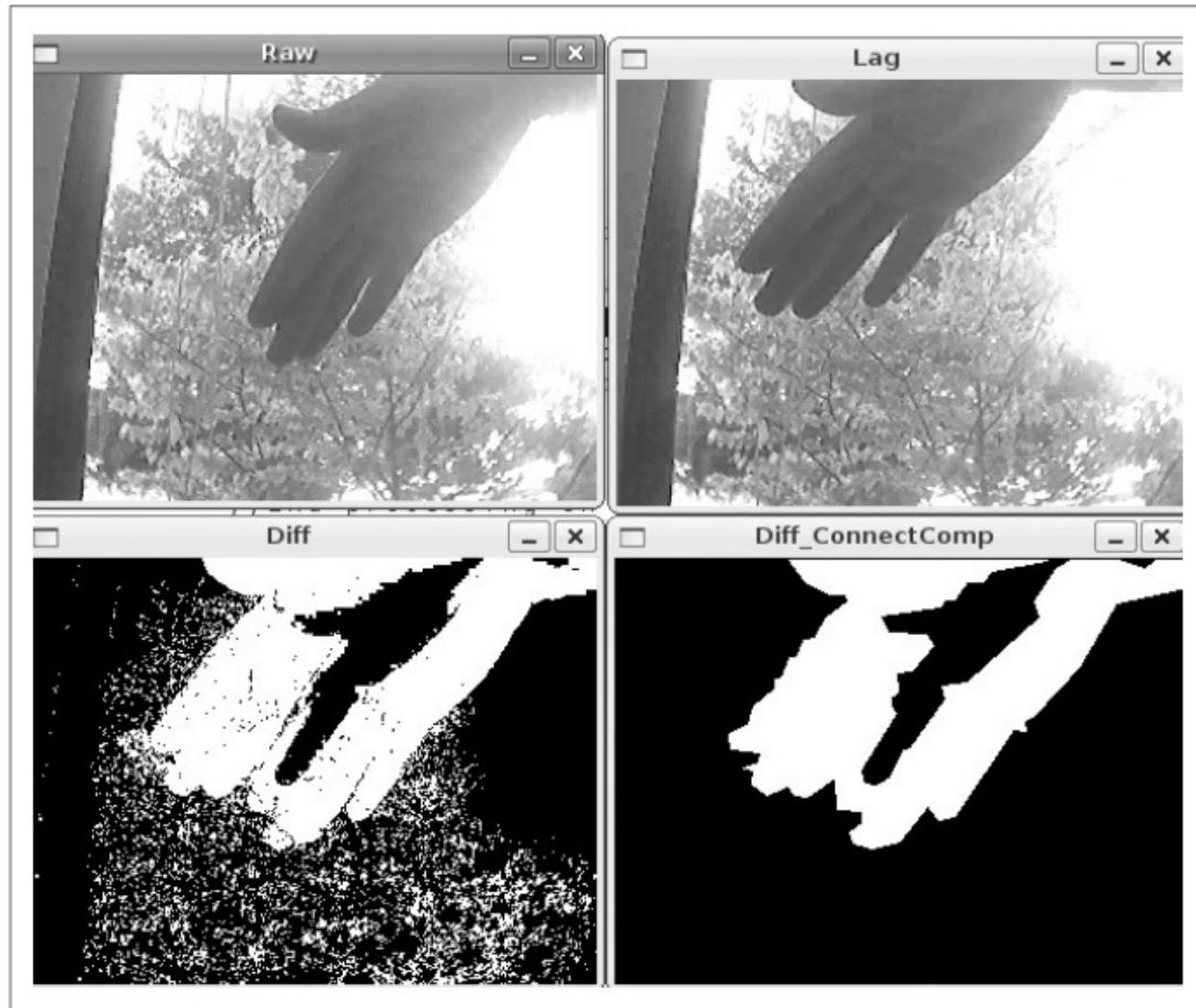


Рисунок 9-6. Использование метод межкадровой разности для обнаружения руки, которая перемещается слева направо и является объектом переднего плана (два верхних окна); изображение разности кадров (снизу слева) отображает "дыру" (где рука должна находиться) между её положением слева и передним краем справа и изображение, полученное методом связанный компоненты (снизу справа)

При всем этом стоит отметить один недостаток метода вычитания фона: он не отличает регионы, где объект перемещается ("дыра") и где находится в настоящий момент времени. Более того, в перекрывающихся регионах зачастую образуется разрыв, так как "тело минус тело" это 0 (или, по крайней мере, ниже порога).

В результате, метод связанных компонент — это хороший инструмент для удаления шума при вычитании фона. В качестве бонуса, были рассмотрены сильные и слабые стороны метода межкадровой разности.

Сравнение моделей фона

В этой главе были представлены два метода моделирования фона: метод усреднения и метод кодовой книги. Возможно, будет интересно знать, какой из методов лучше или, по крайней мере, какой проще в использовании. В таком случае лучше всего сравнить имеющиеся алгоритмы между собой.

Продолжим использовать видео с деревом. В дополнение к движущемуся дереву, на видео присутствуют блики от здания справа и от части внутренней стены слева. Это довольно-таки сложная модель фона.

На рисунке 9-7 приведено сравнение метода средней разности (сверху) и метода кодовой книги (снизу); необработанное изображение переднего плана (слева) очищенное при помощи метода связанный компоненты (справа). Как можно заметить, метод средней разности не затрагивает "небрежные" маски и разделяет руку на две составляющие. И это не удивительно; как было показано на рисунке 9-2 при использовании метода средней разности модель фона зачастую включает значения пикселей, связанных со значениями руки (показано пунктирной линией). Сравните это с рисунком 9-4, где метод кодовой книги может более точно смоделировать колебания листьев и веток и более точно отделить пиксели переднего плана (пунктирная линия) от пикселей заднего плана. Рисунок 9-7 подтверждает не только тот факт, что модель фона дает меньше шума, но также и то, что связанные компоненты могут генерировать достаточно точные контуры объекта.

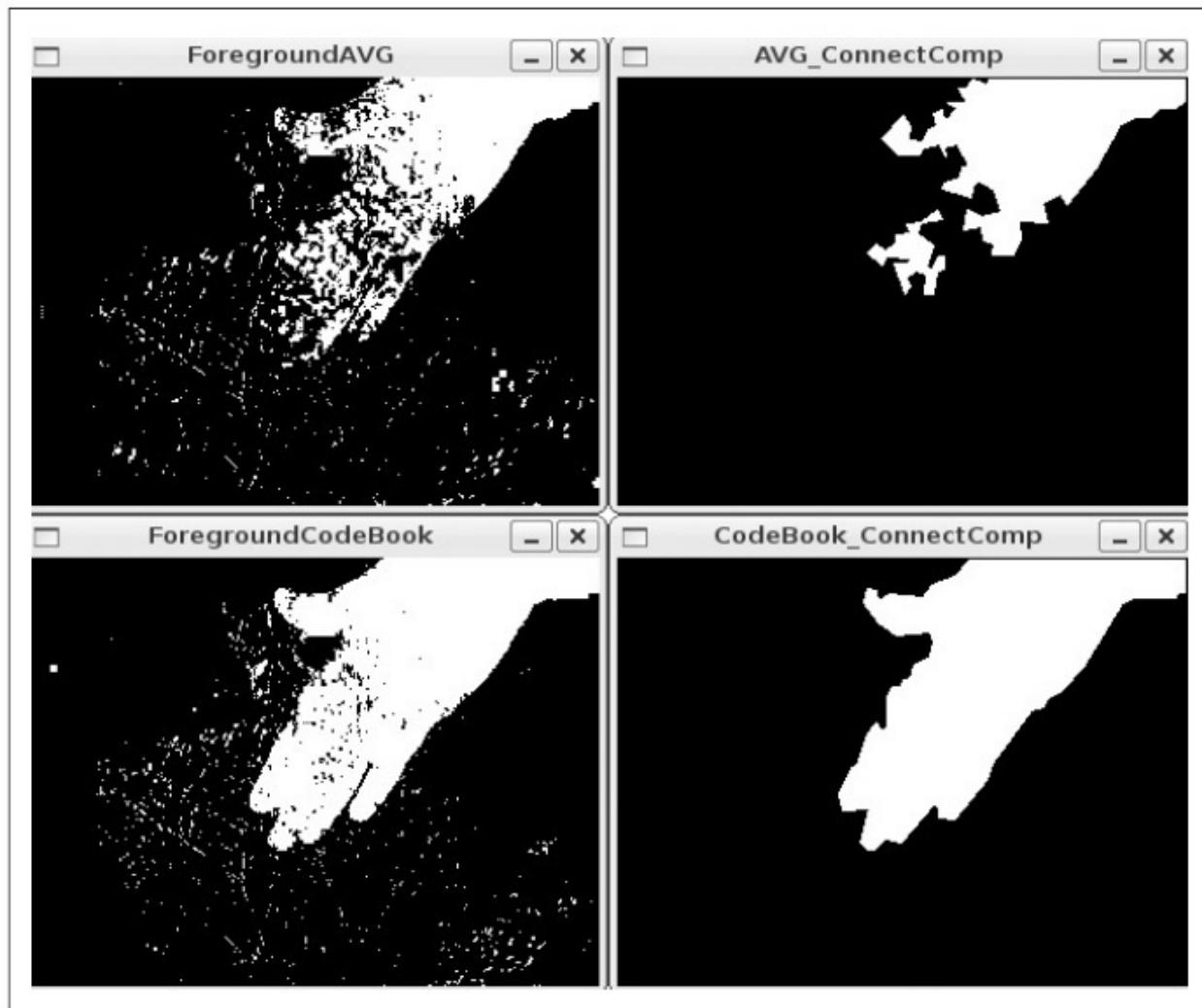


Рисунок 9-7. Метод усреднения (верхний ряд), очищенные связанные компоненты с исключением пальцев (сверху справа); метод кодовой книги (нижний ряд) дает более лучшую сегментацию и создает очищенную маску связанных компонентов (снизу справа)

[П]||[РС]||(РП) Алгоритм водораздела

Во многих практических задачах возникает необходимость в сегментации изображения, но не с целью выделения фона на изображении. В таком случае лучше всего воспользоваться *алгоритмом водораздела*. Этот алгоритм конвертирует линии на изображении в "горы", а однородные регионы во "впадины"; все это может помочь при сегментации объекта. Работает алгоритм следующим образом: в начале берутся градиенты яркости изображения; за счет этого формируются впадины или *водоёмы* (низшие точки), где нет текстур и горы или *хребты* (высокие хребты соответствуют краям), что соответствует доминирующими линиям на изображении. Затем последовательно заливаются водоёмы начиная с указанных пользователем (или алгоритмом) точек пока эти регионы не пересекутся. Регионы, которые объединяются по средствам отметок, генерируют сегменты, соотносящиеся друг с другом как изображения "заполнители". Таким образом регионы, связанные с точкой отметкой считаются "владельцами" этих отметок. В итоге изображение сегментируется в соответствии с помеченными регионами.

В результате алгоритм водораздела позволяет пользователю (или другому алгоритму!) помечать части объекта или фона как известные части объекта или фона.

Пользователь или алгоритм может просто нарисовать линию и "сказать" алгоритму водораздела: "группируй данные точки вместе". За этим следует сегментация изображения, позволяющая отметить регионы "владеющие" краями хребтов в градиентных изображениях, связанные с сегментами. Рисунок 9-8 поясняет данный процесс.

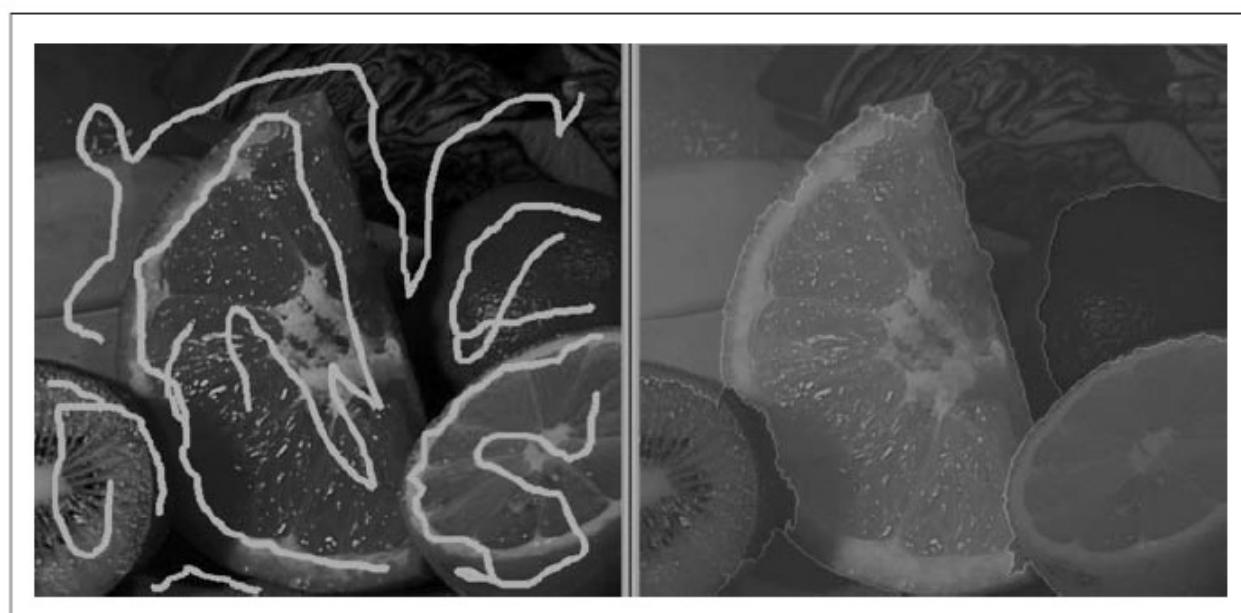


Рисунок 9-8. Алгоритм водораздела: после того, как пользователь отметил объекты, которые связаны друг с другом (слева), алгоритм сливает отмеченные области в сегменты (справа)

Функция, реализующая алгоритм водораздела, выглядит следующим образом:

```
void cvWatershed(  
    const CvArr*    image  
    ,CvArr*        markers  
);
```

image - это 8-битное цветное (трехканальное) изображение, *markers* - это одноканальное целочисленное (*IPL_DEPTH_32S*) изображение, оба изображения имеют одинаковый размер; значения *markers* равны 0 за исключением тех мест, где пользователь (или алгоритм) указал (положительными числами), что регионы должны быть вместе. Например, слева на рисунке 9-8 апельсин может быть отмечен "1", лимон "2", лайм "3", а фон сверху "4" и т.д. В результате будет получена сегментация наподобие той, что представлена на том же изображении справа.

[П]|[РС]|(РП) Восстановление изображения

Зачастую изображения повреждены шумом. Это может быть пыль или капли воды на линзах, царапины или испорченные части на старых фотографиях. *Inpainting* - это метод, который удаляет такого рода повреждения за счет копирования и смешивания цвета и текстур с границы поврежденных областей. На рисунке 9-9 показано как этот метод удаляет надпись с изображения.



Рисунок 9-9. *Inpainting*: изображение поврежденное текстом (слева) успешно восстановлено (справа)

Inpainting работает, если поврежденные области не слишком велики и текстур вокруг достаточно для их заполнения. На рисунке 9-10 показано что произойдет, если поврежденная область слишком велика.



Рисунок 9-10. Inpainting не может магически восстанавливать текстуры, которые полностью повреждены: центр апельсина полностью уничтожен (слева), эта область заполняется преимущественно оранжевой текстурой (справа)

Прототип функции *cvInpaint()*:

```
void cvInpaint(
    const CvArr*    src
    ,const CvArr*   mask
    ,CvArr*         dst
    ,double         inpaintRadius
    ,int            flags
);
```

src - это 8-битное одноканальное серое или трехканальное цветное изображение, которое должно быть восстановлено и *mask* - 8-битное одноканальное изображение того же размера, что и *src*, и в котором поврежденные области (например надпись на рисунке 9-9) помечены ненулевыми пикселями; остальные пиксели равны 0. Конечное изображение будет записано в *dst*, которое должно быть того же размера и с тем же количеством каналов, что и *src*. *inpaintRadius* это область вокруг каждого восстанавливаемого пикселя, от значений которой зависит цвет результирующего пикселя. Как показано на рисунке 9-10, внутренние пиксели достаточно большой восстанавливаемой области могут полностью позаимствовать цвет от других пикселей, лежащих возле границ. Почти всегда используют небольшой радиус, например 3, потому что большой радиус приводит к появлению заметного размытия. Аргумент *flags* позволяет задавать метод восстановления: *CV_INPAINT_NS* (метод Navier-Stokes) или *CV_INPAINT_TELEA* (метод A. Telea's)

[П]|[РС]|(РП) Mean-Shift сегментация

В главе 5 уже была рассмотрена функция `cvPyrSegmentation()`. Пирамидальная сегментация использует цветовое смешивание (по шкале зависимости от цветового сходства) для сегментации изображения. Данный подход основан на минимизации "полной энергии" изображения; энергия определяет *прочность связей*, которые в свою очередь определяют *схожесть цветов*. Данный раздел будет посвящен рассмотрению функции `cvPyrMeanShiftFiltering()`, которая реализует алгоритм, основанный на среднем сдвиге скопления цветов. Детали алгоритма среднего сдвига `cvMeanShift()` будут рассмотрены в главе 10, когда речь пойдет об алгоритме слежения за объектами. На данный момент достаточно будет знать, что алгоритм среднего сдвига ищет пики распределения цветного пространства (или другого признака) во времени. Сегментация методом среднего сдвига находит пики распределения цвета в пространстве. Общим у алгоритмов сегментации и слежения является то, что они оба полагаются на способность среднего сдвига находить режимы (пики) распределения.

Для множества многомерных точек (x, y , синий, зеленый, красный) алгоритм среднего сдвига может найти самую высокую плотность "сгустка" данных в данном пространстве при оконном сканировании над ним. Однако стоит обратить внимание на то, что пространственные переменные (x, y) могут сильно различаться в зависимости от величины цветового диапазона (синий, зеленый, красный). В данном случае необходимо иметь один радиус для пространственных переменных (*spatialRadius*) и ещё один для величины цвета (*colorRadius*). Средний сдвиг движущегося окна, все те точки которого сходятся в пике данных, становятся связанными или "принадлежащими" этому пику. Эта принадлежность, исходящая из самых плотных пиков, формирует сегментацию изображения. Сегментация на самом деле это градация пирамид (`cvPyrUp()`, `cvPyrDown()`), которая была описана в главе 5, поэтому цветовое скопление на верхних уровнях пирамид (съежившиеся изображения) имеют границы, уточненные пирамидами нижнего уровня. Вызов функции `cvPyrMeanShiftFiltering()` выглядит следующим образом:

```

void cvPyrMeanShiftFiltering(
    const CvArr* src
    ,CvArr* dst
    ,double spatialRadius
    ,double colorRadius
    ,int max_level = 1
    ,CvTermCriteria termcrit = cvTermCriteria(
        CV_TERMCRIT_ITER | CV_TERMCRIT_EPS
        ,5
        ,1
    )
);

```

У *cvPyrMeanShiftFiltering()* имеется исходное *src* и конечное *dst* изображения. Оба изображения должны быть 8-битными, трехканальными цветными изображениями одинакового размера. *spatialRadius* и *colorRadius* определяют, как алгоритм среднего сдвига оперирует средними цвета и пространства для формирования сегментации. Для 640x480 изображения, алгоритм показывает хорошие результаты для значений *spatialRadius* = 2 и *colorRadius* = 40. Аргумент *max_level* определяет количество уровней пирамид, которые будут использованы для сегментации. Для 640x480 изображения *max_level* равен 2 или 3.

Последний аргумент *CvTermCriteria* уже был рассмотрен в главе 8. Он используется во всех итеративных алгоритмах OpenCV. Значение по умолчанию дает довольно таки хорошие результаты и задается в случае, если параметр остался незаполненным. *cvTermCriteria* имеет следующий конструктор:

```

cvTermCriteria(
    int type // CV_TERMCRIT_ITER, CV_TERMCRIT_EPS
    ,int max_iter
    ,double epsilon
);

```

Как правило, данная функция используется для генерации структуры *CvTermCriteria*. Первый аргумент может быть равен *CV_TERMCRIT_ITER* или *CV_TERMCRIT_EPS*, что указывает алгоритму завершать работу после некоторого фиксированного числа итераций или когда метрика сходимости достигнет некоторого малого значения (соответственно). Следующие два аргумента это значения, при котором или которых алгоритм должен быть завершен. Можно использовать оба значения, т.к. можно написать *type* = *CV_TERMCRIT_ITER* | *CV_TERMCRIT_EPS*, в этом случае алгоритм завершиться при одном из условий. Аргумент *max_iter* задаёт максимальное число

итераций, если используется `CV_TERMCRIT_ITER`, тогда как `epsilon` устанавливает предел ошибки, если используется `CV_TERMCRIT_EPS`. Естественно точное значение `epsilon` зависит от алгоритма.

На рисунке 9-11 показан результат сегментации при использовании следующих значений:

```
cvPyrMeanShiftFiltering( src, dst, 20, 40, 2 );
```



Рисунок 9-11. Сегментация методом среднего сдвига при использовании функции `cvPyrMeanShiftFiltering()` с параметрами `max_level = 2`, `spatialRadius = 20`, и `colorRadius = 40`; в результате схожие области имеют близкие значения и могут рассматриваться как супер пиксели, что может значительно ускорить последующую обработку

[П] | [РС] | (РП) Триангуляция Delaunay, тесселяция Voronoi

Триангуляция Delaunay - это техника, изобретенная в 1934 для соединения точек в пространстве в треугольную группу таким образом, чтобы минимальный угол среди всех углов в триангуляции был максимальным. Это означает, что триангуляция Delaunay пытается избегать "тонких" треугольников при триангуляции точек. Для того чтобы понять суть триангуляции посмотрите на рисунок 9-12: любая окружность, описанная вокруг вершин любого треугольника не содержит других вершин. Это именуется *свойством описанной окружности* (часть с на рисунке).

Чтобы вычисления были эффективными, алгоритм Delaunay начинает свою работу с самого далекого внешнего ограничивающего треугольника. На рисунке 9-12(b) фиктивный внешний треугольник представлен пунктирными линиями, сходящимися в вершине. На рисунке 9-12(c) представлено несколько примеров описанных окружностей, при этом одна из них связывает две реальные вершины и одну из вершин фиктивного внешнего треугольника.

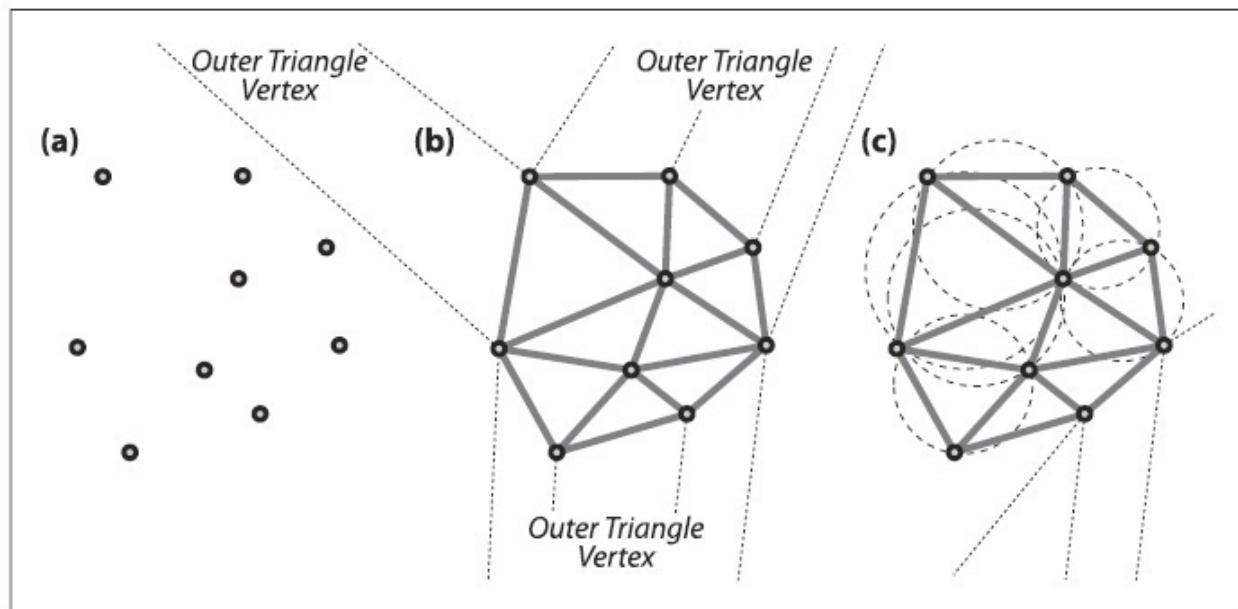


Рисунок 9-12. Триангуляция Delaunay: (а) множество точек; (б) Триангуляция Delaunay для набора точек, с пунктирными линиями внешнего ограничивающего треугольника; (в) пример описанных окружностей

На сегодняшний день существует множество алгоритмов для расчета триангуляции Delaunay; некоторые из них очень эффективные, но со сложными деталями реализации. Суть одного из наиболее простых алгоритмов в следующем:

1. Добавить внешний треугольник и начать с одной из его вершин (это гарантирует получение внешней отправной точки)
2. Добавить внутреннюю точку; затем "пройтись" по всем треугольникам с описанной окружностью содержащих эту точку и удалить эти триангуляции
3. Re-triangulate граф с включением новых точек в описанную окружность простым удалением триангуляций
4. Возвращаться к шагу 2 до тех пор, пока не останется точек для добавления

Сложность этого алгоритма составляет $O(n^2)$. Наиболее эффективные алгоритмы (в среднем) имеют сложность $O(n \log \log n)$.

Отлично – но для чего все это нужно? С одной стороны, нужно помнить, что этот алгоритм начинает свою работу с фиктивного внешнего треугольника, поэтому все настоящие внешние точки, на самом деле, соединены с двумя вершинами этого треугольника, а вот описанная окружность, проходящая через две реальные внешние точки и одну фиктивную внешнюю вершину, не содержит других внутренних вершин. Это означает, что компьютер сможет определить, какие реальные точки образуют внешний контур набора точек, просто просмотрев те точки, которые соединены с тремя внешними фиктивными вершинами. Другими словами, можно найти скелет набора точек сразу после того, как будет выполнена триангуляция Delaunay.

Так же можно определить, кто "владеет" пространством между точками, а именно, чьи координаты являются ближайшими соседями до вершин Delaunay. Таким образом, используя триангуляцию Delaunay, можно найти ближайшего соседа для новой точки. Такое разбиение на плоскости называется *тесселяция Voronoi*. Тесселяция является двойным образом триангуляции Delaunay, потому что линии Delaunay определяют расстояние между существующими точками, а линии Voronoi "знают", где они должны пересекаться с линиями Delaunay для сохранения равного расстояния между точками. Эти два метода для нахождения выпуклой оболочки и ближайшего соседа, являются основными операциями кластеризации и классификации точек и наборов точек.

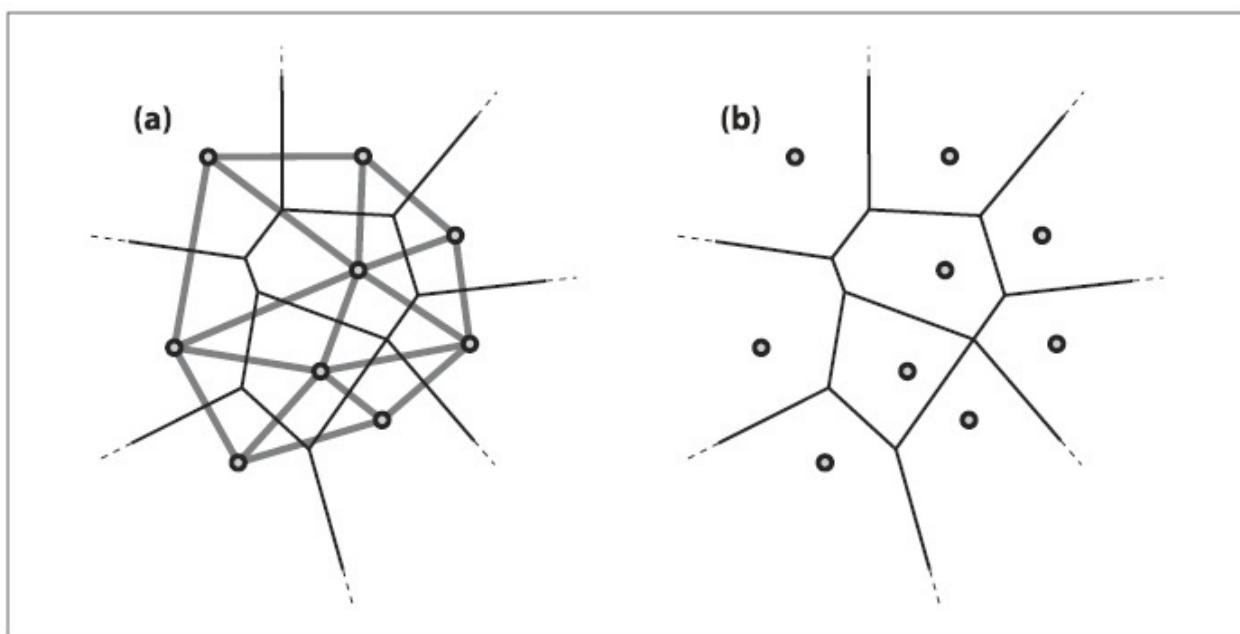


Рисунок 9-13. Тесселяция Voronoi, в результате которой все точки в пределах данной ячейки Voronoi являются ближайшими к своим точкам Delaunay, чем к любым другим точкам Delaunay: (а) триангуляция Delaunay отмечена жирными линиями, а тесселяции Voronoi тонкими линиями, (б) ячейки Voronoi вокруг каждой точки Delaunay

Все те, кто знаком с 3D графикой могут отметить, что триангуляция Delaunay зачастую используется как основа для представления 3D фигур. Если имеется 3D представление объекта, то можно создать 2D представление этого объекта с помощью проекции, а затем использовать триангуляцию Delaunay для анализа и идентификации объекта и/или для сравнения его с реальным объектом. Триангуляция Delaunay это связующее звено между компьютерным зрением и компьютерной графикой. Однако, триангуляция Delaunay в OpenCV выполняется только для двумерного измерения (в дальнейшем планируется это исправить, глава 14). Если бы существовала возможность триангулировать множество 3D точек – скажем для стереозрения (глава 11) – то можно было бы плавно переходить между 3D графикой и компьютерным зрением. Тем не менее, 2D триангуляция Delaunay часто используется в компьютерном зрении для того, чтобы зарегистрировать пространственное расположение особенностей на объекте или сцене для отслеживания движений, распознавания объектов или для сравнения проекций с двух разных камер (при выводе глубины стереозрения). На рисунке 9-14 показано, как можно отслеживать и распознавать объекты при помощи триангуляции Delaunay, где ключевые точки лица пространственно классифицированы в соответствии с их триангуляцией.

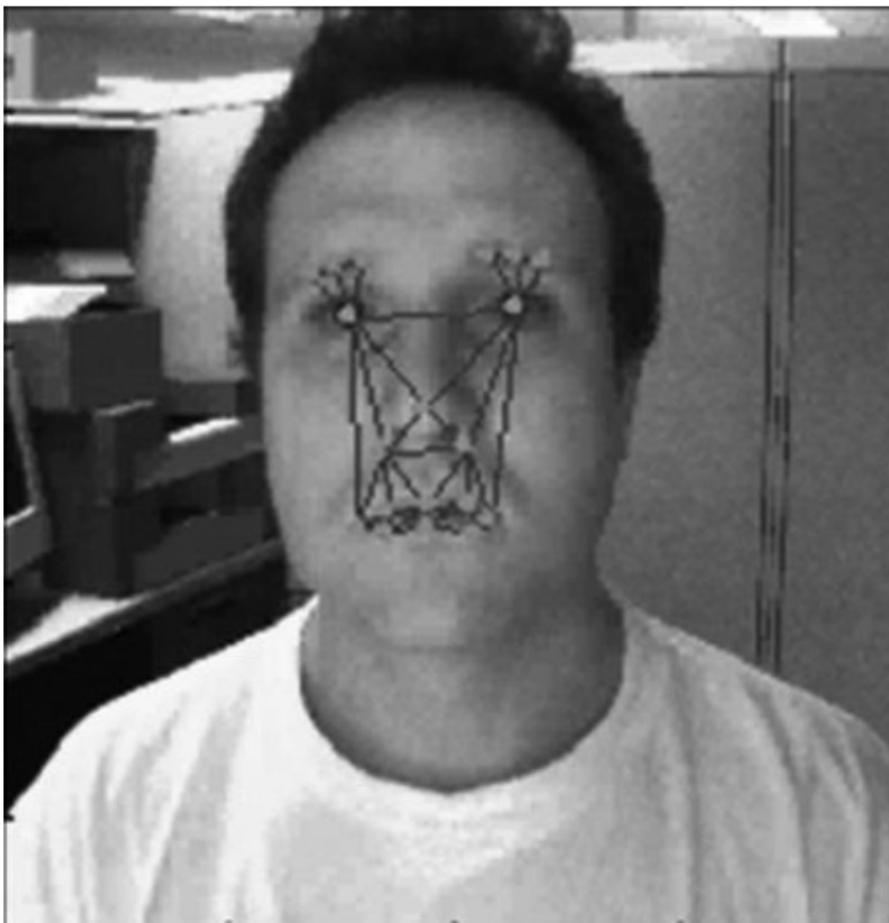


Рисунок 9-14. Точки Delaunay могут быть использованы для слежения за объектами; здесь лицо отслеживается с помощью точек, по которым можно также определить эмоции

Теперь, когда установлена потенциальная полезность триангуляции Delaunay, как собственно получить эту самую триангуляцию? Для решения этой задачи OpenCV поставляется вместе с примером .../opencv/samples/c/delaunay.c. В OpenCV триангуляцию Delaunay ссылается на понятие подраздел Delaunay, о чём собственно и пойдет речь в следующих разделах.

Построение триангуляции Delaunay и тесселяции Voronoi

Для начала необходимо выделить место под хранение результатов триангуляции и тесселяции. Так же понадобиться внешний ограничивающий прямоугольник (чтобы ускорить вычисления, алгоритм должен работать с фиктивным внешним треугольником, который расположен за пределами ограничивающего прямоугольника). Для выполнения поставленной задачи, предположим, что точки располагаются внутри изображения 600x600:

```
// Хранилище и структура для подраздела Delaunay
//
CvRect rect = { 0, 0, 600, 600 }; // Внешний ограничивающий прямоугольник

CvMemStorage* storage; // Хранилище
storage = cvCreateMemStorage(0); // Инициализация хранилища

CvSubdiv2D* subdiv; // Подраздел
subdiv = init_delaunay( storage, rect );
```

В коде используется функция *init_delaunay()*, которая является удобной "упаковкой" нескольких функций OpenCV:

```
// Инициализация удобной функции для подраздела Delaunay
//
CvSubdiv2D* init_delaunay( CvMemStorage* storage, CvRect rect ) {
    CvSubdiv2D* subdiv;

    subdiv = cvCreateSubdiv2D(
        CV_SEQ_KIND_SUBDIV2D
        ,sizeof(*subdiv)
        ,sizeof(CvSubdiv2DPoint)
        ,sizeof(CvQuadEdge2D)
        ,storage
    );

    cvInitSubdivDelaunay2D( subdiv, rect ); // Установка ограничительного прямоугольника

    return subdiv;
}
```

Далее представлена логика добавления точек. Эти точки должны быть типа float, 32F:

```
CvPoint2D32f fp; // Собственно сами точки
for( i = 0; i < as_many_points_as_you_want; i++ ) {
    // Собственно само добавление
    //
    fp = your_32f_point_list[i];
    cvSubdivDelaunay2DInsert( subdiv, fp );
}
```

Можно конвертировать целые точки в 32f точки с помощью удобного макроса *cvPoint2D32f(double x, double y)* или *cvPointTo32f(CvPoint point)*, расположенного в *cxtypes.h*. Теперь, когда можно добавлять точки в триангуляцию Делоне, появляется возможность создавать и удалять соответствующую тесселяцию Вороного:

```
cvCalcSubdivVoronoi2D( subdiv ); // Занести данные Voronoi в subdiv  
cvClearSubdivVoronoi2D( subdiv ); // Удалить данные Voronoi из subdiv
```

В обеих функциях *subdiv* имеет тип *CvSubdiv2D**. Теперь можно создавать триангуляции Delaunay из 2-х мерных наборов точек с последующим созданием и удалением тесселяций Voronoi. Однако, как получить нужные данные из этих структур? Это можно сделать, шагая от ребра к вершине или от ребра к ребру в *subdiv*; на рисунке 9-15 показаны основные манёвры, начиная с заданного ребра и его точки отсчета. Далее можно найти первые ребра или вершины двумя разными способами: (1) используя внешнюю точку для обнаружения ребра или вершин; или (2) шагая по последовательности вершин или рёбер.

Перемещение по подразделам Delaunay

Рисунок 9-15 сочетает в себе две структуры данных, которые будут использованы для перемещения по подразделам графа.

Структура *cvQuadEdge2D* содержит набор из двух точек *Delaunay* и двух точек *Voronoi*, и связанные с ними рёбра (предполагается, что точки *Voronoi* и рёбра были рассчитаны с помощью *cvCalcSubdivVoronoi2D()*); рисунок 9-16. Структура *CvSubdiv2DPoint* содержит ребро *Delaunay* с сопутствующей вершиной, как показано на рисунке 9-17. Структура из четырёх рёбер определена в соответствии с рисунком.

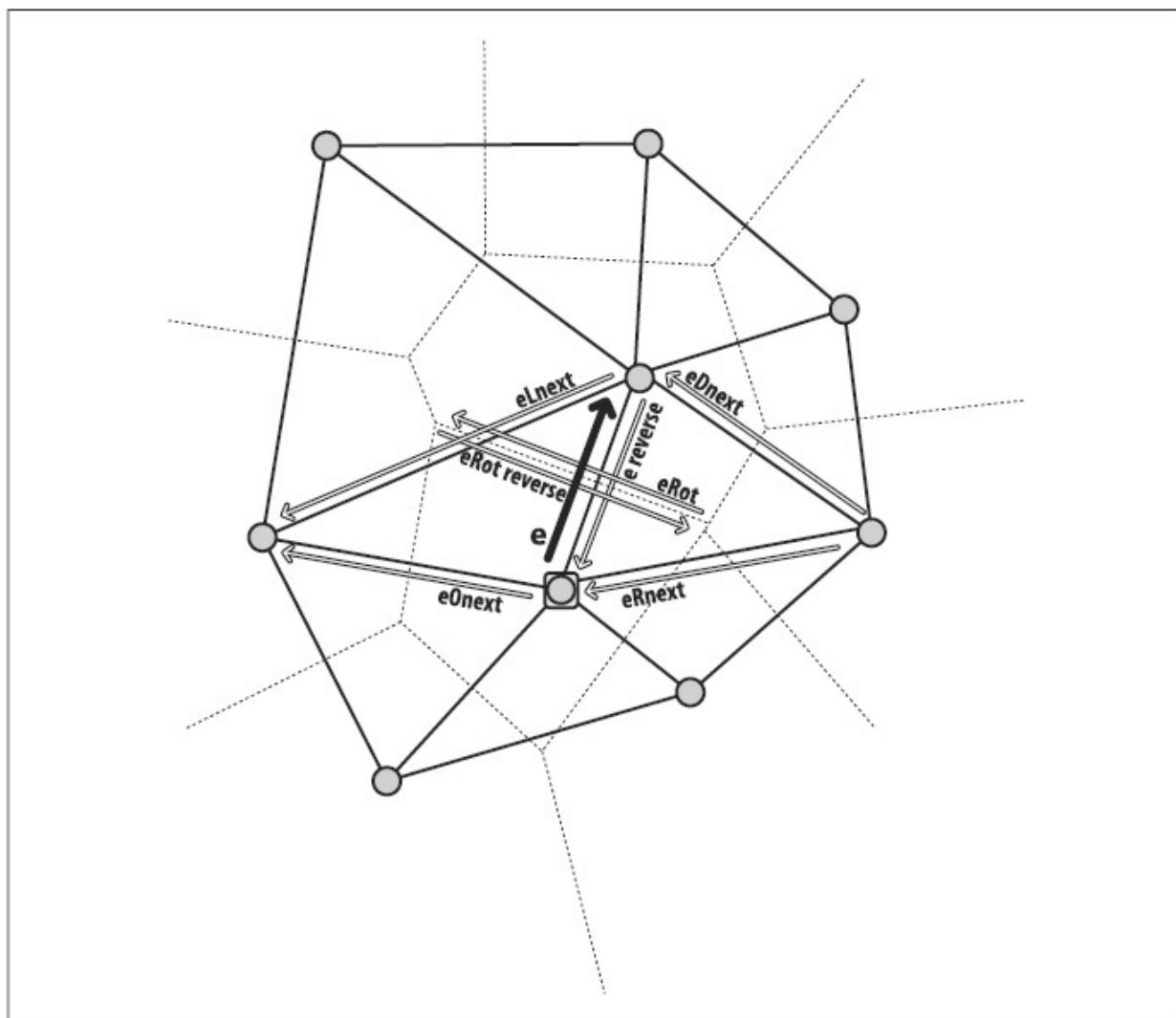


Рисунок 9-15. Ребра относительно заданного ребра, с меткой "е", и его вершина (отмечена квадратом)

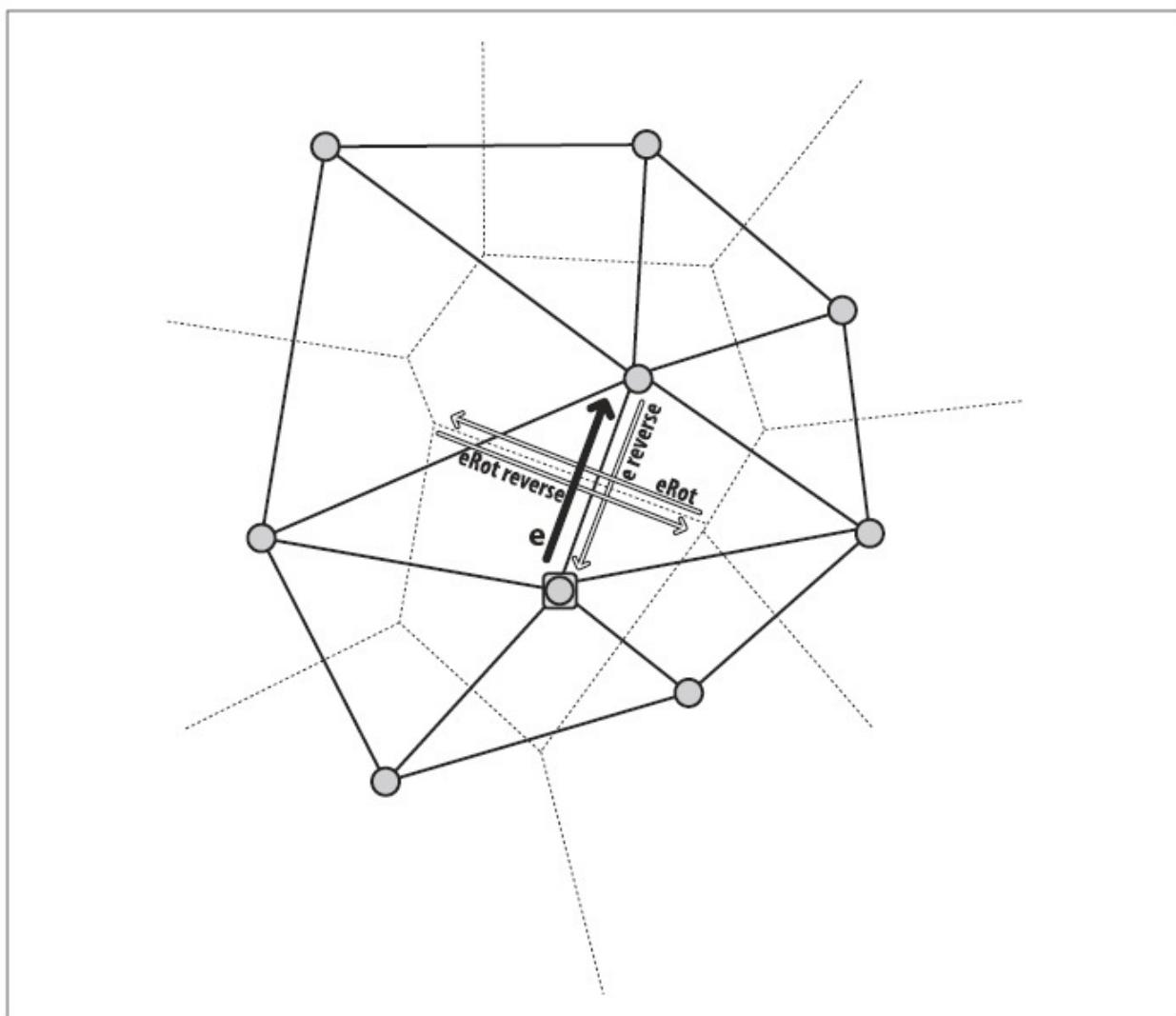


Рисунок 9-16. Quad edges, к которым можно получить доступ при помощи `cvSubdiv2DRotateEdge()`, включая ребро Delaunay и противоположное ему ребро, а также соответствующие рёбра и вершины Voronoi

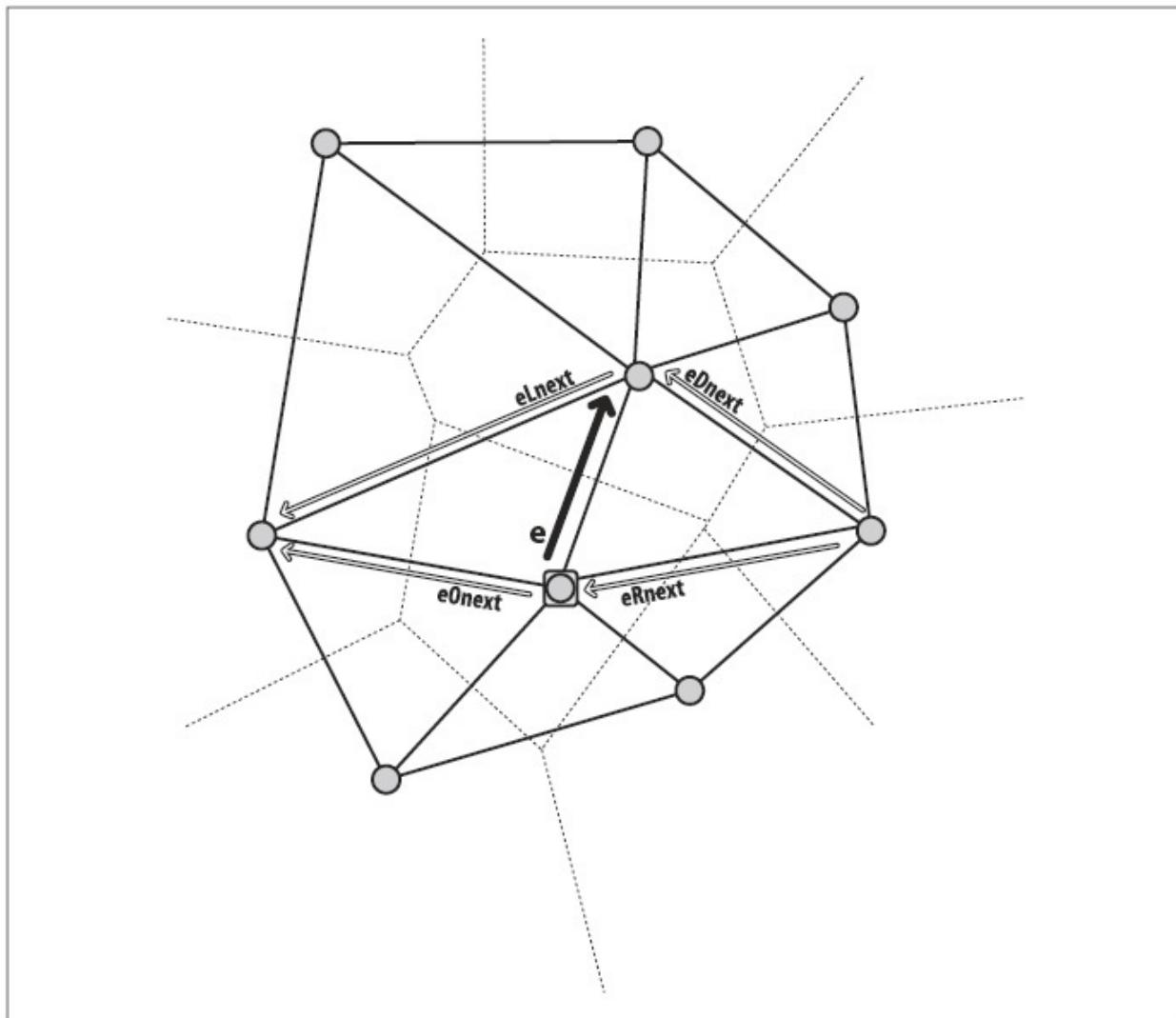


Рисунок 9-17. CvSubdiv2DPoint вершина и связанное с ней ребро e, вместе с другими связанными рёбрами, которые могут быть получены с помощью cvSubdiv2DGetEdge()

```

// Рёбра представляют собой длинные целые числа. Младшие два бита
// это их индекс (0..3), а старшие - указатель quad-edge
//
typedef long CvSubdiv2DEdge;

// Поля структуры quad-edge
//
#define CV_QUADEDGE2D_FIELDS()           /
    int flags;                         /
    struct CvSubdiv2DPoint* pt[4];     /
    CvSubdiv2DEdge next[4];

typedef struct CvQuadEdge2D {
    CV_QUADEDGE2D_FIELDS()
} CvQuadEdge2D;

```

Точки подраздела Delaunay и соответствующая структура ребра определяется следующим образом:

```
#define CV_SUBDIV2D_POINT_FIELDS() /
    int flags;                      /
    CvSubdiv2DEdge first;           /* Ребро "e" на рисунке */
    CvPoint2D32f pt;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint {
    CV_SUBDIV2D_POINT_FIELDS()
} CvSubdiv2DPoint;
```

С помощью этих структур, можно рассмотреть различные способы перемещения по вершинам и рёбрам.

Обход рёбер

Как показано на рисунке 9-16, перемещаться между четырьмя ребрами можно при помощи следующей функции:

```
CvSubdiv2DEdge cvSubdiv2DRotateEdge(
    CvSubdiv2DEdge edge
    ,int type
);
```

При наличии *edge* (ребра) получить следующее ребро можно используя аргумент *type*, который принимает одно из следующих значений:

- 0, исходное ребро (*e* на фигуре, если *e* исходное ребро)
- 1, поворотное ребро (*eRot*)
- 2, противоположное ребро (противоположное *e*)
- 3, противоположное поворотному ребру (противоположное *eRot*)

Ссылаясь на рисунок 9-17, так же возможно обойти граф Delaunay при помощи следующей функции:

```

CvSubdiv2DEdge cvSubdiv2DGetEdge(
    CvSubdiv2DEdge edge
    , CvNextEdgeType type
);

#define cvSubdiv2DNextEdge( edge ) \
    cvSubdiv2DGetEdge( \
        edge \
        , CV_NEXT_AROUND_ORG \
    )

```

Аргумент *type* может принимать следующие значения:

CV_NEXT_AROUND_ORG

Следующее от исходного ребро (*eOnext* на рисунке 9-17, если *e* исходное ребро)

CV_NEXT_AROUND_DST

Ребро следующей вершины (*eDnext*)

CV_PREV_AROUND_ORG

Предыдущее от исходного ребро (противоположное *eRnext*)

CV_PREV_AROUND_DST

Предыдущее от ребра назначения (противоположное *eLnext*)

CV_NEXT_AROUND_LEFT

Следующее от левой грани (*eLnext*)

CV_NEXT_AROUND_RIGHT

Следующее от правой грани (*eRnext*)

CV_PREV_AROUND_LEFT

Предыдущее от левой грани (противоположное *eOnext*)

CV_PREV_AROUND_RIGHT

Предыдущее от правой грани (противоположное *eDnext*)

Стоит отметить, что указанное ребро, связанное с вершиной, можно задействовать при использовании макроса *cvSubdiv2DNextEdge(edge)*, чтобы найти все остальные ребра этой вершины. Это полезно использовать в случае с выпуклой оболочкой, начинающейся из вершины внешнего (фиктивного) треугольника.

Важными видами обхода являются *CV_NEXT_AROUND_LEFT* и *CV_NEXT_AROUND_RIGHT*. Их можно использовать для обхода треугольника Delaunay, если находится на ребре Delaunay или для обхода ячеек Voronoi, если находится на ребре Voronoi.

Точки ребер

Помимо всего прочего, необходимо знать, как получать фактические точки из вершин Delaunay или Voronoi. Каждое ребро Delaunay или Voronoi имеет две точки, связанные с ним: *org* - исходная точка и *dst* - конечная точка. Эти точки легко можно получить при помощи следующих функций:

```
CvSubdiv2DPoint* cvSubdiv2DEdgeOrg( CvSubdiv2DEdge edge );
CvSubdiv2DPoint* cvSubdiv2DEdgeDst( CvSubdiv2DEdge edge );
```

Следующий метод можно использовать для преобразования *CvSubdiv2DPoint* в более привычный вид:

```
CvSubdiv2DPoint ptSub;                                // Вершина для преобразования
CvPoint2D32f    pt32f     = ptSub->pt;                // преобразование к типу 32f
CvPoint        pt       = cvPointFrom32f(pt32f);      // преобразование к целочисленному типу
```

На данный момент должно было сформироваться представление о структуре подраздела и о том, как совершать обход его вершин и ребер. Теперь можно вернуться к рассмотрению методов получения исходных ребер и вершин подразделов Delaunay/Voronoi.

Метод 1: использование внешних точек для нахождения ребер и вершин

Первый метод начинает свою работу с произвольной точки и поиска её местоположения в данном подразделе. Это необязательно должна быть точка, которая уже триангулирована; это может быть любая точка. Функция *cvSubdiv2DLocate()* занимает одно ребро и одну вершину (если необходимо) треугольника или грани Voronoi, в который эта точка упала.

```
CvSubdiv2DPointLocation cvSubdiv2DLocate(
    CvSubdiv2D*         subdiv
    , CvPoint2D32f      pt
    , CvSubdiv2DEdge*   edge
    , CvSubdiv2DPoint** vertex = NULL
);
```

Стоит обратить внимание на тот факт, что это не обязательно должны быть ближайшие *edge* (ребро) или *vertex* (вершина); достаточно будет и того, чтобы они были на треугольнике или грани. Возвращаемое значение данной функции сообщает о состоянии попадания точки.

CV_PTLOC_INSIDE

Точка попадает на некую грань; **edge* будет содержать одно из ребер грани

CV_PTLOC_ON_EDGE

Точка попадает на ребро; **edge* будет содержать это ребро

CV_PTLOC_VERTEX

Точка совпадает с одной из вершин подраздела; **vertex* будет содержать указатель на вершину

CV_PTLOC_OUTSIDE_RECT

Точка находится за пределами прямоугольника; функция вернет не заполненный указатель

CV_PTLOC_ERROR

Один из исходных аргументов является недействительным

Метод 2: последовательный обход точек и ребер

Удобен при использовании подраздела Delaunay, созданного из множества точек; его первые три точки и ребра образуют вершины и стороны фиктивного внешнего ограничивающего треугольника. Как результат, получение доступа к внешним точкам и ребрам, которые формируют выпуклую оболочку из фактических точек данных. После создания подраздела Delaunay (*subdiv*), необходимо вызвать *cvCalcSubdivVoronoi2D(subdiv)* для того, чтобы рассчитать связанную с ним тесселяцию Voronoi. Получить доступ к трем вершинам внешнего ограничивающего треугольника можно следующим образом:

```
CvSubdiv2DPoint* outer_vtx[3];

for( i = 0; i < 3; i++ ) {
    outer_vtx[i] = (CvSubdiv2DPoint*)cvGetSeqElem( (CvSeq*)subdiv, I );
}
```

Аналогичным образом можно получить три стороны внешнего ограничительного треугольника:

```
CvQuadEdge2D* outer_qedges[3];

for( i = 0; i < 3; i++ ) {
    outer_qedges[i] = (CvQuadEdge2D*)cvGetSeqElem( (CvSeq*)(my_subdiv->edges), I );
}
```

Теперь, зная, как построить граф и как по нему перемещаться, необходимо научиться определять местоположение точек – на внешнем краю или границе?

Определение ограничивающего треугольника, или краев выпуклой оболочки и её обход

Ограничивающий прямоугольник *rect* используется для инициализации триангуляции Delaunay при помощи функции *cvInitSubdivDelaunay2D(subdiv, rect)*. В таком случае, справедливы следующие утверждения:

1. Если имеется ребро, у которого и исходная и конечная точки находятся вне границ *rect* (прямоугольника), тогда ребро принадлежит фиктивному ограничивающему треугольнику подраздела.
2. Если одна из точек ребра находится внутри, а другая за пределами границ *rect* (прямоугольника), то точка границы находится в выпуклой оболочке множества; каждая точка выпуклой оболочки связана с двумя вершинами фиктивного внешнего ограничивающего треугольника и эти два ребра следуют друг за другом.

Для второго условия можно использовать *cvSubdiv2DNextEdge()*, чтобы перейти на первое ребро, у которого точка *dst* находится в пределах границ. После попадания на выпуклую оболочку, по ней можно перемещаться следующим образом:

1. Пока не произошел обход всей выпуклой оболочки, перейти к следующему ребру оболочки при помощи *cvSubdiv2DRotateEdge(CvSubdiv2DEdge edge, 0)*.
2. Последующие вызовы (дважды) *cvSubdiv2DNextEdge()* дают следующие ребра выпуклой оболочки. Вернуться к шагу 1.

На данный момент было рассмотрено, как инициализировать подразделы Delaunay и Voronoi, как искать исходные ребра, а также как совершать обход ребер и вершин графа. В следующем разделе будут представлены примеры реализации рассмотренного материала.

Примеры использования

Можно использовать *cvSubdiv2DLocate()* для обхода ребер треугольника Delaunay:

```

void locate_point(
    CvSubdiv2D*      subdiv
    ,CvPoint2D32f   fp
    ,IplImage*       img
    ,CvScalar        active_color
) {
    CvSubdiv2DEdge      e;
    CvSubdiv2DEdge      e0 = 0;
    CvSubdiv2DPoint*    p = 0;

    cvSubdiv2DLocate( subdiv, fp, &e0, &p );

    if( e0 ) {
        e = e0;

        // Всегда 3 ребра – это ведь триангуляция
        //
        do {
            // [Вставте свой код сюда]
            //

            // Сделать что-то с e ...
            //
            e = cvSubdiv2DGetEdge(e, CV_NEXT_AROUND_LEFT);
        }
        while( e != e0 );
    }
}

```

Найти ближайшие точки исходной точки можно при помощи

```

CvSubdiv2DPoint* cvFindNearestPoint2D(
    CvSubdiv2D*      subdiv
    ,CvPoint2D32f   pt
);

```

В отличие от `cvSubdiv2DLocate()`, `cvFindNearestPoint2D()` возвращает ближайшую точку вершины в подразделе Delaunay. Не гарантировано, что точка обязательно попадет на грань или треугольник.

Аналогичным образом можно обойти грань Voronoi при помощи

```

void draw_subdiv_facet(
    IplImage      *img
    ,CvSubdiv2DEdge edge
) {
    CvSubdiv2DEdge t = edge;
    int i, count = 0;
    CvPoint* buf = 0;

```

```

// Подсчет количества ребер на грани
//
do{
    count++;
    t = cvSubdiv2DGetEdge( t, CV_NEXT_AROUND_LEFT );
} while (t != edge );

// Сбор точек
//
buf = (CvPoint*)malloc( count * sizeof(buf[0]) );
t = edge;
for( i = 0; i < count; i++ ) {
    CvSubdiv2DPoint* pt = cvSubdiv2DEdge0rg( t );
    if( !pt ) {
        break;
    }
    buf[i] = cvPoint( cvRound(pt->pt.x), cvRound(pt->pt.y) );
    t = cvSubdiv2DGetEdge( t, CV_NEXT_AROUND_LEFT );
}

// Обход
//
if( i == count ){
    CvSubdiv2DPoint* pt = cvSubdiv2DEdgeDst(
        cvSubdiv2DRotateEdge( edge, 1 )
    );

    cvFillConvexPoly(
        img
        ,buf
        ,count
        ,CV_RGB(rand()&255,rand()&255,rand()&255)
        ,CV_AA
        ,0
    );
}

cvPolyLine(
    img
    ,&buf
    ,&count
    ,1
    ,1
    ,CV_RGB(0,0,0)
    ,1
    ,CV_AA
    ,0
);

draw_subdiv_point( img, pt->pt, CV_RGB(0,0,0) );
}

free( buf );

```

```
}
```

И в заключении, есть ещё один способ получения доступа к подразделу, используя *CvSeqReader* для перебора последовательности ребер. Перебор всех ребер Delaunay и Voronoi можно осуществить следующим образом:

```
void visit_edges( CvSubdiv2D* subdiv ) {
    CvSeqReader reader;                                // Последовательность reader
    int i, total = subdiv->edges->total;             // Количество ребер
    int elem_size = subdiv->edges->elem_size;         // Размер ребра

    cvStartReadSeq( (CvSeq*)(subdiv->edges), &reader, 0 );
    cvCalcSubdivVoronoi2D( subdiv ); // Проверка существования

    for( i = 0; i < total; i++ ) {
        CvQuadEdge2D* edge = (CvQuadEdge2D*)(reader.ptr);

        if( CV_IS_SET_ELEM( edge ) ) {
            // ЧТО-ТО СДЕЛАТЬ С РЕБРАМИ DELAUNAY И VORONOI
            //

            CvSubdiv2DEdge voronoi_edge = (CvSubdiv2DEdge)edge + 1;
            CvSubdiv2DEdge delaunay_edge = (CvSubdiv2DEdge)edge;

            // ... ИЛИ СОСРЕДОТОЧИТЬСЯ ТОЛЬКО НА VORONOI

            // ЛЕВОЕ
            //
            voronoi_edge = cvSubdiv2DRotateEdge( edge, 1 );

            // ПРАВОЕ
            //
            voronoi_edge = cvSubdiv2DRotateEdge( edge, 3 );
        }
        CV_NEXT_SEQ_ELEM( elem_size, reader );
    }
}
```

И в заключении: выполнив поиск трех вершин, можно найти площадь треугольника

```
double cvTriangleArea(
    CvPoint2D32f a
    ,CvPoint2D32f b
    ,CvPoint2D32f c
)
```

[П]|[РС]|(РП) Упражнения

1. Используя функцию `cvRunningAvg()`, повторно реализуйте метод усреднения фона. Для того, чтобы это сделать, изучите скользящее среднее значение пикселей на сцене, чтобы найти абсолютную разницу среднего и скользящего среднего (`cvAbsDiff()`) в качестве прокси стандартного отклонения изображения.
2. Тени часто являются проблемой при вычитании фона, т.к. их можно принять за объекты переднего плана. Воспользуйтесь методом вычитания фона усреднения или кодовой книги. Смоделируйте ситуацию, в которой человек перемещается на переднем плане. Тени будут "исходить" от нижней части объекта переднего плана.
 - a. На улице тенями являются более темные и синеватые участки; используйте данный факт, чтобы удалить тени.
 - b. В помещении тенями являются более темные участки; используйте данный факт, чтобы удалить тени.
3. Простые модели фона, представленные в данной главе, зачастую весьма чувствительны к их пороговым параметрам. В главе 10 будет показано, как отслеживать движения, что может быть использовано в качестве "реальной" проверки модели фона и его порогов. Данные знания так же можно использовать, чтобы распознать человека по "калибровочным движениям" на переднем плане камеры: ищется движущийся объект и настраиваются параметры до тех пор, пока на переднем плане объект не будет соответствовать границам движения. Так же можно использовать определенные шаблоны на самом калиброванном объекте (или фоне) для реальной проверки и настройки направляющего, чтобы знать, какая часть фона является замкнутой.
 - a. Измените код, чтобы включить режим авто калибровки. Изучите модель фона, а затем разместите цветной объект на сцене. Используйте цвет для поиска цветного объекта, а затем используйте этот объект, чтобы автоматически установить пороговые значения фона для сегментации объекта. При этом можно оставить объект на сцене для плавной перенастройки.
 - b. Используйте полученное решение для удаления теней из упражнения 2.
4. Используйте фоновую сегментацию для сегментации человеческой руки. Исследуйте влияние различных параметров и значений по умолчанию на процедуру `find_connected_components()`. Отобразите результаты различных установок следующих параметров:

- a. `poly1_hull0`
 - b. `perimScale`
 - c. `CVCONTOUR_APPROX_LEVEL`
 - d. `CVCLOSE_ITR`
5. В 2005 году на *DARPA Grand Challenge robot race*, авторы *Stanford team* использовали своего рода алгоритм цветной кластеризации для отделения дороги от не дороги. Цвета выбирались лазерной трапецией клочка дороги, взятой перед машиной. Другие цвета на сцене, близкие по цвету к данному клочку принимались за связанные компоненты и помечались как дорога. Взгляните на рисунок 9-18, где алгоритм водораздела был использован для сегментации дороги (отмечена трапецией), а все что не дорога отмечено перевернутой "U". Предположим, что имеется возможность автоматически генерировать эти метки. Что в таком случае может пойти не так в этом методе сегментации дороги?
- Подсказка: посмотрите на рисунок 9-8 и предположите за счет каких вещей можно расширить трапецию дороги.

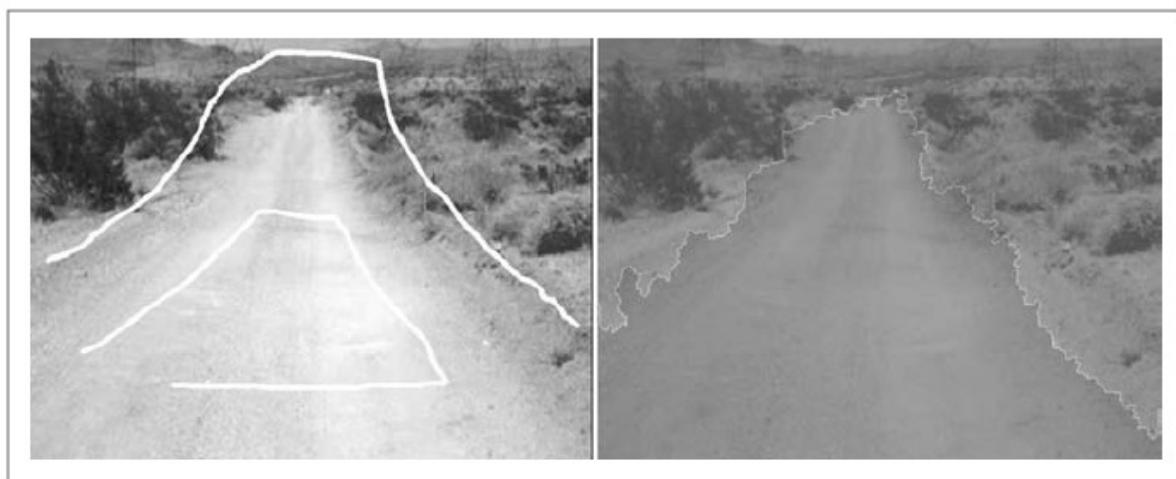


Рисунок 9-18. Использование алгоритма водораздела для идентификации дороги: размещение маркеров на оригинальном изображении (слева) и результат применения алгоритма - сегментированная дорога (справа)

6. Inpainting работает наиболее хорошо для восстановления письменного текста на наиболее текстурированных областях. Что будет, если письменный текст будет неразборчив? Попробуйте.
7. Это может быть немного медленно, однако попробуйте запустить фоновую сегментацию с предварительной предобработкой исходного видеопотока при помощи `cvPyrMeanShiftFiltering()`. Т.е. в начале на исходном потоке выполните

mean-shift сегментацию, а затем выполните изучение фона - а затем и тестирование переднего плана - при помощи метода кодовой книги.

- a. Отобразите результаты до выполнения *mean-shift* сегментации.
- b. Попробуйте систематически изменять *max_level*, *spatialRadius* и *colorRadius* *mean-shift* сегментации.
8. Насколько хорошо inpainting работает с письменным текстом, который обработан *mean-shift* сегментацией? Поэкспериментируйте с настройками и отобразите результаты.
9. Модифицируйте код *.../opencv/samples/delaunay.c* так, чтобы появилась возможность отслеживания двойного нажатия мыши в указанном месте (вместо существующего метода, где место выбирается случайным образом).
Поэкспериментируйте с результатами триангуляции.
10. Вновь модифицируйте *delaunay.c* так, чтобы можно было бы задействовать клавиатуру для рисования выпуклой оболочки, на основе которой бы строилось множество точек.
11. Можно ли говорить о триангуляции Delaunay, имея три точки на линии?
12. Является ли триангуляция на рисунке 9-19(а) триангуляцией Delaunay? Если да, то объясните почему. Если нет, то как необходимо изменить фигуру, чтобы это была триангуляция Delaunay?
13. Выполните триангуляцию Delaunay на множестве точек с рисунка 9-19(б) вручную.
Для выполнения этого задания не используйте внешний фиктивный внешний треугольник.

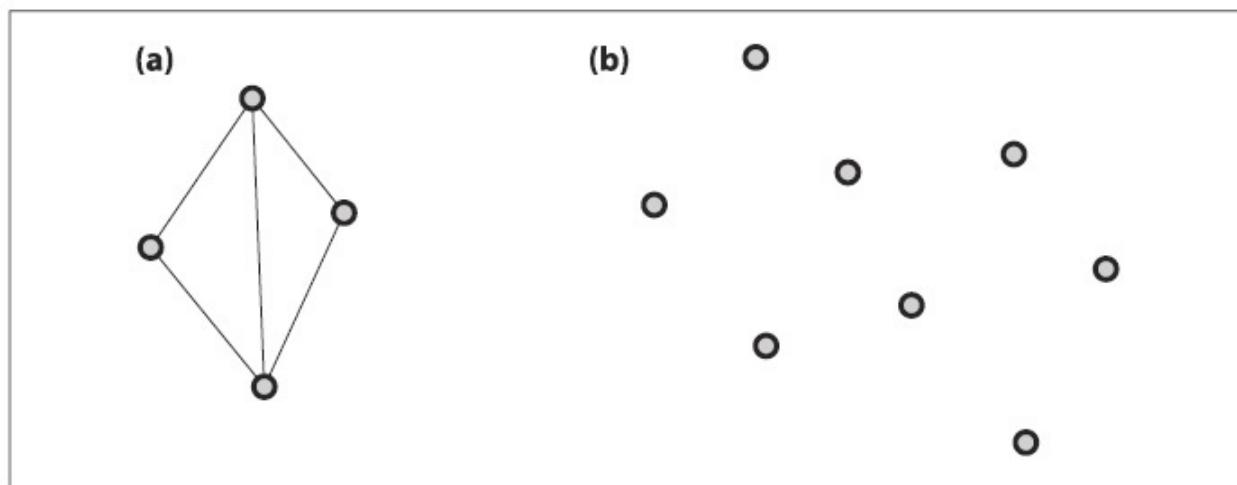


Рисунок 9-19. Упражнение 12 и 13

Слежение и движение

[П]|[РС]|(РП) Основы слежения

Имея дело с видео потоком в противоположность статическому изображению, зачастую возникает необходимость в отслеживании объекта или группы объектов, попадающих в поле зрения. В предыдущей главе уже было показано, как изолировать определенные формы, например, человека или автомобиль, на основе анализа кадра за кадром. Теперь можно перейти к рассмотрению решения задачи распознавания движений этих объектов, которая имеет две основные составляющие: идентификацию и моделирование.

Идентификация сводится к поиску объекта, который вызывает интерес, от кадра к кадру в видеопотоке. Такие методы, как моменты или цветные гистограммы из предыдущих глав помогут справиться с этой задачей. Отслеживание объектов, которые ещё не идентифицированы, является смежной проблемой. Отслеживание не идентифицированных объектов важно тогда, когда необходимо определить интересующий нас объект на основе движения не идентифицированных объектов или когда движение не отслеживаемого объекта является именно тем, что и интересно. Методы отслеживания неопознанных объектов, как правило, предполагают отслеживание визуально значимых ключевых точек, а не расширение объектов. OpenCV предоставляет два метода для решения данной задачи: *Lucas-Kanade* и *Horn-Schunck*, которые в свою очередь именуются так же как *пространственный* и *плотный* потоки соответственно.

Вторая составляющая, моделирование, помогает справиться с тем фактом, что ранее представленные методы на самом деле предоставляют зашумленное положение объекта. Многие математические методы по оценке траектории объекта были разработаны на основе измерения шума. Эти методы применимы к двух- и трехмерным моделям объектов и их расположениям.

[П]||[РС]||(РП) Поиск углов

Существует множество локальных особенностей, которые можно отслеживать. Для начала заострим внимание на разъяснении того, что такая особенность. Очевидно, что если выбрать точку на большой пустой стене, то не так уж и легко будет обнаружить её в следующем кадре видеопотока. Если все точки на стене будут одинаковыми или очень похожими, то существует очень малый шанс отследить конкретную точку в последующих кадрах. С другой стороны, выбирая уникальную точку, шанс найти её в последующих кадрах в разы возрастают. На практике должны выбираться уникальные (или почти уникальные) точки или особенности, чтобы максимизировать шансы распознать их в последующих кадрах (Рисунок 10-1).

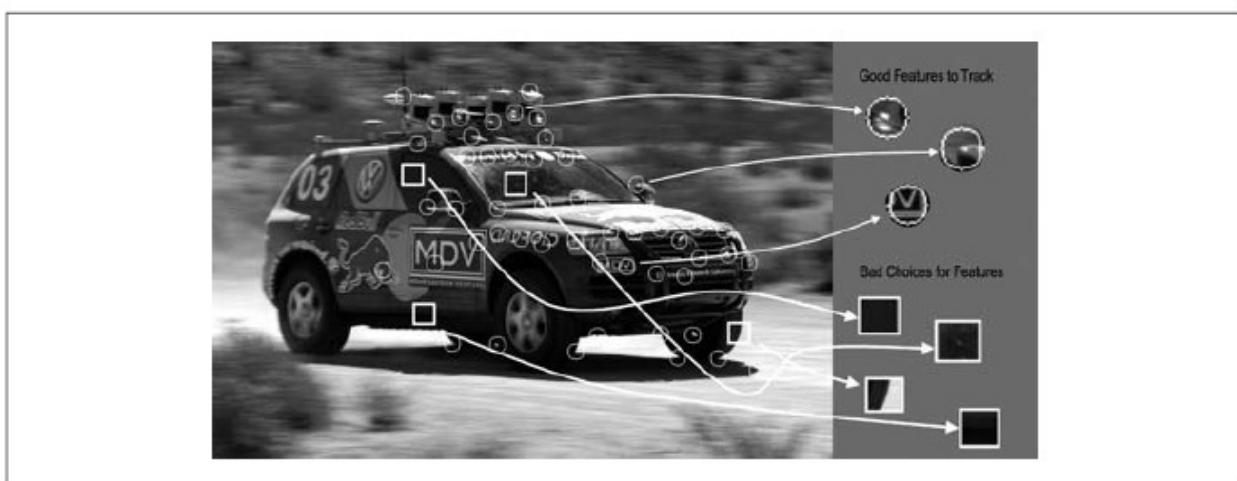


Рисунок 10-1. Точки в кругах - это хорошие точки для отслеживания, в то время как точки в прямоугольниках – это наихудший выбор

Теперь вновь вернемся к рассмотрению большой пустой стены с точками; можно попытаться найти точки, претерпевающие существенные изменения - например, те, которые имеют сильную производную. На самом деле этого не достаточно, но это только начало. Точка с сильной производной может быть связана с каким-то краем, при этом выглядеть так же, как и все другие точки вдоль этого края (раздел *Метод Lucas-Kanade*, рисунок 10-8).

Если сильная производная наблюдается в двух ортогональных направлениях, то можно надеяться на то, что точка является уникальной. По этой причине зачастую отслеживают так называемую угловую особенность. По идеи, углы – а не края – это точки, которые содержат достаточно информации, чтобы их можно было бы отслеживать от кадра к кадру.

Наиболее часто используемое определение угла было представлено *Harris*. Данное определение опирается на матрицу производных интенсивности второго порядка $(\partial^2 x, \partial^2 y, \partial x \partial y)$. Производные второго порядка, взятые во всех точках изображения, можно представить, как формирование новых "вторых производных изображений" или как объединение в новое изображение *Hessian*. Эта терминология возникла из матрицы *Hessian* вокруг точки, определенной в двух измерениях:

$$H(p) = \begin{bmatrix} \frac{\partial^2 I}{\partial x^2} & \frac{\partial^2 I}{\partial x \partial y} \\ \frac{\partial^2 I}{\partial y \partial x} & \frac{\partial^2 I}{\partial y^2} \end{bmatrix}_p$$

Для угла Harris рассматривается *автокорреляционная матрица* вторых производных изображений по малой области вокруг каждой точки. Такая матрица определяется следующим образом:

$$M(x, y) = \begin{bmatrix} \sum_{-K \leq i, j \leq K} w_{i,j} I_x^2(x+i, y+j) & \sum_{-K \leq i, j \leq K} w_{i,j} I_x(x+i, y+j) I_y(x+i, y+j) \\ \sum_{-K \leq i, j \leq K} w_{i,j} I_x(x+i, y+j) I_y(x+i, y+j) & \sum_{-K \leq i, j \leq K} w_{i,j} I_y^2(x+i, y+j) \end{bmatrix}$$

где $w_{i,j}$ – весовой вклад, который может быть постоянным, но зачастую используется для создания кругового окна или Гауссова взвешивания. Углы по определению Harris это места на изображении, где автокорреляционная матрица вторых производных имеет два больших собственных значения. В сущности, это означает, что текстура (или края) проходят, по крайней мере, два отдельных направления, центрируемых вокруг такой точки, которая соответствует реальному углу, имеющего два ребра, встречающихся в точке. Вторые производные полезны, т.к. они не зависят от однородного градиента. (Градиент получается из первых производных. Если первые производные являются однородными (константами), то вторая производная равна 0). Это определение также полезно в случае рассмотрения только собственных значений автокорреляционной матрицы, т.к. в таком случае рассматриваются значения, которые инвариантны к вращению, что очень важно, потому что отслеживаемые объекты могут вращаться и перемещаться. В добавок к этому, эти два собственных числа определяют не только является ли выбранная особенность хорошей для отслеживания, но и обеспечивают идентификационную подпись для точки.

В исходное определение Harris включено взятие детерминанта $H(p)$, с вычетом из него следа $H(p)$ (с некоторым весовым коэффициентом) и последующим сравнением полученной разницы с заданным порогом. Впоследствии, *Shi* и *Tomasi* обнаружили, что

хорошие углы можно получить, если наименьшее из собственных чисел больше минимального порога. Метод *Shi* и *Tomasi* является не только самодостаточным, но и во многих случаях дает более хорошие результаты, чем метод *Harris*.

Функция *cvGoodFeaturesToTrack()* реализует метод *Shi* и *Tomasi*. Эта функция вычисляет вторые производные (с помощью оператора Собеля), которые необходимы для вычисления собственных чисел. В результате функция возвращает список точек, которые соответствуют определению хороших точек для отслеживания.

```
void cvGoodFeaturesToTrack(
    const CvArr*    image
    ,CvArr*        eigImage
    ,CvArr*        tempImage
    ,CvPoint2D32f* corners
    ,int*          corner_count
    ,double         quality_level
    ,double         min_distance
    ,const CvArr*   mask      = NULL
    ,int            block_size = 3
    ,int            use_harris = 0
    ,double         k          = 0.4
);
```

image - это исходное 8 или 32 битное (т.е. *IPL_DEPTH_8U* или *IPL_DEPTH_32F*) одноканальное изображение. Следующие два аргумента – это одноканальные 32 битные изображения одинакового размера. И *tempImage* и *eigImage* используются алгоритмом в качестве отправной точки, однако, содержимое *eigImage* имеет смысл. В частности, каждая запись содержит минимальное собственное значение соответствующей точки исходного изображения. *corners* - это массив 32-битных точек (*CvPoint2D32f*), который содержит точки, полученные в результате выполнения алгоритма; память под массив должна быть выделена до вызова функции. При этом естественно объем выделяемой памяти ограничен. *corner_count* указывает на максимальное количество точек. В конечном итоге этот параметр на выходе будет содержать число точек, действительно найденных. Параметр *quality_level* указывает минимально возможное собственное значение для точки, которая будет помечена как угол. Фактически минимальное собственное значение, используемое для сокращения, является продуктом *quality_level* и самого маленького собственного значения, наблюдаемого в изображении. Следовательно, *quality_level* не должен превышать 1 (как правило, это значения равно 0.10 или 0.01). Как только выбраны кандидаты, дальнейший отбор осуществляется таким образом, чтобы точки в пределах небольшой области не были включены в результаты. В частности, *min_distance* гарантирует, что в решении не будут присутствовать две точки с указанным количеством пикселей.

Дополнительный параметр *mask* - это изображение, которое содержит 0 и 1, указывающие на то, какие точки должны рассматриваться как углы, а какие нет. Если данный параметр установлен в NULL, то маска не используется. Аргумент *block_size* задает размер области вокруг пикселя при вычислении автокорреляционной матрицы производных. Как оказалось, лучше всего производить вычисления производных с использованием небольшого окна, нежели производить вычисления лишь в одной точке (т.е. при *block_size* = 1). Если *use_harris* != 0, то определяются углы Harris, а не Shi-Tomasi. Если *use_harris* != 0, то значение *k* - это весовой коэффициент, используемый для установки относительного веса данного следа автокорреляционной матрицы Hessian, сравниваемого с детерминантом той же матрицы.

Результат вызова функции *cvGoodFeaturesToTrack()* - это массив местоположений пикселей, которые необходимо найти на другом подобном изображении. В контексте данной главы интерес вызывает поиск в последующих кадрах видео. Представленный метод может быть использован при попытке связать набор изображений одного и того же изображения, взятого с разных ракурсов. Данная проблема будет более подробно представлена при рассмотрении темы стереозрения.

[П]|[РС]|(РП) Углы субпикселя

Если изображения обрабатываются с целью извлечения геометрических измерений, то в отличие от выделения особенностей необходимо выполнить нечто большее, чем просто выделить пиксель. При работе с пикселями приходиться иметь дело с целочисленными координатами, хотя иногда возникает необходимость и в вещественных координатах (например, в пикселе (8.25, 117.16)).

Если допустить, что необходимо взглянуть на острый пик в значениях изображения, то окажется, что расположение пика почти никогда не будет точно в центре пикселя камеры. Чтобы это исправить, можно подобрать кривую (например, параболу) к значениям изображения, а затем использовать немного математики, чтобы найти местоположение пика между пикселями. Все методы обнаружения субпикселя используют данную хитрость. В общем, измерения изображения используются для отслеживания трехмерных перестроений, калибровки камеры, деформации частично перекрытых сцен для того, чтобы сшить их вместе в более естественный вид, и для поиска внешнего сигнала (например, точного местоположения здания на изображении со спутника).

Местоположение угла субпикселя является обобщенным измерением и используется для калибровки камер, или для отслеживания перестроений траектории камеры или трехмерной структуры отслеживаемого объекта. Теперь, зная как искать местоположение угла в целочисленной сетке пикселей, вот хитрость, которую необходимо использовать для уточнения местоположения субпикселя: все сводится к использованию математического факта - скалярное произведение между вектором и ортогональю вектора равно 0; данный факт имеет место для местоположения углов, показанных на рисунке 10-2.

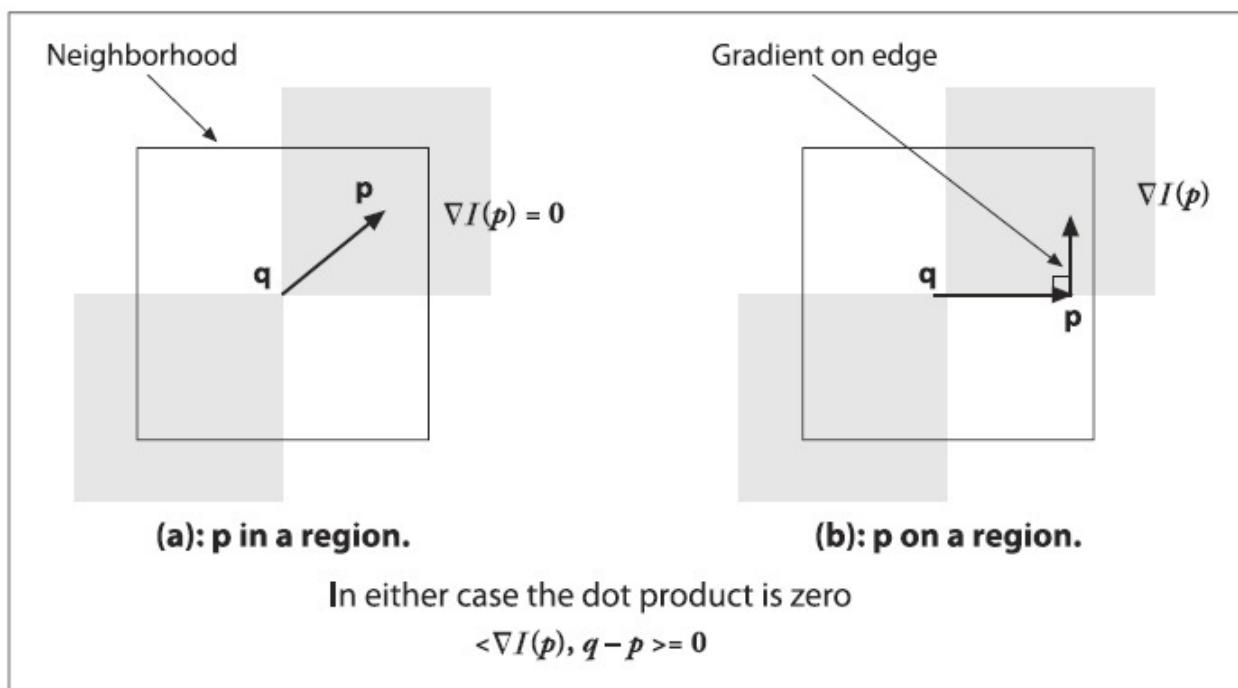


Рисунок 10-2. Поиск углов с точностью до субпикселей: (а) область изображения вокруг точки p является однородной и потому её градиент равен 0; (б) градиент края это ортогональный вектор $q-p$; в любом случае, скалярное произведение между градиентом p и вектором $q-p$ равно 0

На рисунке начальные координаты местоположения угла q находятся вблизи фактического местоположения угла субпикселя. Рассматривая вектор $q-p$ и принимая во внимание, что p расположено вблизи однородного или "ровного" региона, градиент равен 0. С другой стороны, если вектор $q-p$ совпадает с краем, то градиент p этого края ортогонален вектору $q-p$. В любом случае, скалярное произведение между градиентом p и вектором $q-p$ равно 0. Можно подобрать много таких пар градиента для соседей точки p и связанных с ней векторов $q-p$, задать скалярное произведение равное 0 и решить получившуюся систему уравнений; решение предоставит более точную информацию о местоположении субпикселя q и о его *corner* (угол).

Функция, которая осуществляет поиск угла, выглядит следующим образом:

```
void cvFindCornerSubPix(
    const CvArr*    image
    ,CvPoint2D32f*  corners
    ,int            count
    ,CvSize         win
    ,CvSize         zero_zone
    ,CvTermCriteria criteria
);
```

image - это исходное одноканальное 8-битное серое изображение. Структура *corners* содержит целочисленные местоположения пикселя, полученные от *cvGoodFeaturesToTrack()* и которые принято считать в качестве исходных местоположений углов; *count* содержит количество точек, участвующих в вычислениях.

Фактически при вычислении местоположения субпикселя используется система выражений скалярного произведения равные 0 (рисунок 10-2), где каждое уравнение возникает из рассмотрения одного пикселя в регионе вокруг *p*. Параметр *win* определяет размер окна, за счет которого эти уравнения будут сгенерированы. Это окно центрируется на исходном целочисленном местоположении угла и распространяется во всех направлениях на то количество пикселей, что указано в *win* (например, если *win.width = 4*, тогда область поиска будет равна $4 + 1 + 4 = 9$ пикселям). Эти уравнения образуют линейную систему уравнений, которая может быть решена путем инверсии одной автокорреляционной матрицы (это не та автокорреляционная матрица, которая уже была рассмотрена при обсуждении углов Harris). На практике эта матрица не всегда обратима вследствие малых собственных значений, возникающих от пикселей близких к *p*. Для решения данной проблемы необходимо просто отказаться от рассмотрения пикселей близких к *p*. Параметр *zero_zone* определяет окно (аналогичное *win*, но всегда меньшего размера), которое не учитывается в системе сдерживающих уравнений и соответственно в автокорреляционной матрице тоже. Если параметр *zero_zone* не требуется, то он должен быть установлен в *cvSize(-1, -1)*.

После нахождения нового местоположения для *q*, алгоритм приступает к перебору, используя данное значение в качестве отправной точки, до тех пор, пока не будет выполнено условие остановки, определенное пользователем. Условие может быть типа *CV_TERMCRIT_ITER* или типа *CV_TERMCRIT_EPS* (или оба) и, как правило, создано при помощи функции *cvTermCriteria()*. *CV_TERMCRIT_EPS* используется для указания точности значения субпикселя. Таким образом, если будет указано 0.10, то точность значения субпикселя составит одна десятая пикселя.

[П]|[РС]|(РП) Инвариантность особенностей

Со времен работы Harris и последующей работы Shi и Tomasi, было предложено множество других типов углов и смежных местных особенностей. Одним из наиболее широко используемых типов является функция SIFT ("scale-invariant feature transform" или "масштабно-инвариантная функция преобразования"). Такие особенности, как следует из названия, масштабно-инвариантны. Т.к. SIFT может определить доминирующую ориентацию градиента на данной локации и зафиксировать собственные локальные результаты градиента гистограммы по отношению к ориентации, то SIFT также инвариантна и к вращению. В результате, функция SIFT дает относительно хорошие результаты при малых аффинных преобразованиях. И хотя SIFT не реализована как часть библиотеки OpenCV (глава 14), она может быть реализована при помощи примитивов OpenCV. Собственно в оставшихся главах данная проблема больше не будет рассматриваться, т.к. большинство функций, представленных в литературе про компьютерное зрение, можно (хотя и не столь удобно) реализовать при помощи примитивов OpenCV (перспективы расширения функционала OpenCV будут представлены в главе 14).

[П]||[РС]||(РП) Оптические потоки

Ранее уже было сказано, что иногда может возникнуть необходимость в оценке движений между двумя кадрами (или последовательностью кадров) без каких-либо других предварительно полученных знаний о содержании этих кадров. Как правило, само движение является знаком того, что происходит что-то интересное. Оптический поток представлен на рисунке 10-3.

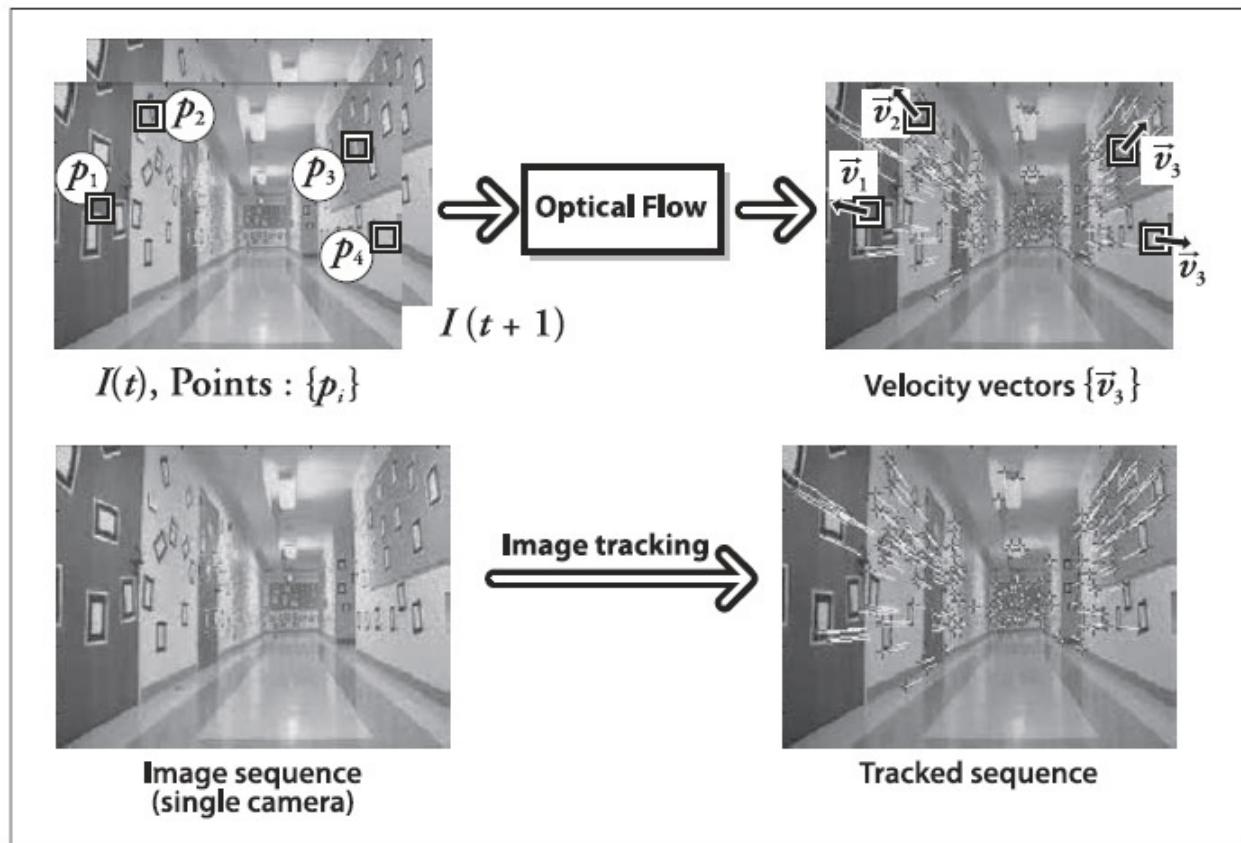


Рисунок 10-3. Оптический поток: целевые особенности (сверху слева) отслеживаются в течение долгого времени, а их движения преобразуются в вектор скорости (сверху справа); в нижней части рисунку представлен общий вид коридора (слева) и вектор потока (справа) того, как движется камера по коридору

Можно связать некую скорость каждого пикселя в кадре или, что эквивалентно, некоторое смещение, представляющее собой расстояние на которое сместится пиксель при сравнении предыдущего и текущего кадра. Такая конструкция обычно именуется *плотным оптическим потоком*, который связывает скорость с каждым пикселям изображения. Метод Horn-Schunck пытается вычислять только скорости полей. Еще один, казалось бы, простой метод – просто пытающийся сопоставить окна

вокруг каждого пикселя от кадра к кадру – так же реализован в OpenCV; этот метод известен как *метод блочного сопоставления*. Оба представленных метода будут более детально представлены в разделе "Методы плотного слежения".

На практике рассчитать плотный оптический поток не так то и легко. Рассмотрим пример движения белого листа бумаги. Многие белые пиксели из предыдущего кадра остаются белыми и в последующем кадре. При этом изменениям могут быть подвержены лишь края и то только те, что перпендикулярны направлению движения. Как результат, плотные методы должны иметь в наличии некоторые методы интерполяции между точками, которые легче отследить, для вычисления неоднозначных точек. Эти проблемы наиболее ярко проявляются в виде высоких вычислительных затратах плотного оптического потока.

Это порождает альтернативный вариант, *разряженный оптический поток*. Алгоритмы такого рода опираются на заранее заданное некоторым образом подмножество точек, которое подлежит отслеживанию. Если точки обладают заранее определенными подходящими свойствами, такими как ранее обсуждаемые "углы", то отслеживание будет относительно прочным и надежным. OpenCV может помочь при решении данного вопроса, путем предоставления процедур для выявления лучших особенностей для отслеживания. На практике вычислительная стоимость от использования разряженного потока намного меньше, чем от использования плотного потока, поэтому интерес к последнему лишь академический. (Black и Anandan создали методы плотного оптического потока, которые зачастую используются при создании фильмов, в которых упор делается на получение качественной картинки. Эти методы планируется включить в более поздние версии OpenCV, глава 14).

В следующих разделах будет представлено несколько различных методов отслеживания. Вначале будет представлена наиболее популярная разряженная техника слежения, оптический поток Lucas-Kanade (LK); этот метод также имеет вариант реализации для работы с пирамидой изображений, позволяющий отслеживать быстрые движения. Затем будут представлены две плотные техники слежения, метод Horn-Schunck и метод сопоставления блоков.

Метод Lucas-Kanade

Первоначально предполагалось (в 1981 году), что алгоритм Lucas-Kanade (LK) будет использоваться для создания плотных потоков. Но, т.к. метод легко применить к подмножеству точек исходного изображения, он стал более важен для создания разряженных потоков. Алгоритм LK может быть применен в контексте разряженного потока, т.к. опирается только на локальную информацию, которая является производной от некоторого небольшого окна, окружающее каждую, вызывающую интерес, точку. Это главное отличие от алгоритма Horn и Schunck, где используются

глобальные характеристики (подробнее об этом позже). Недостаток использования небольших локальных окон в методе Lukas-Kanade заключается в том, что большие сдвиги могут переместить точки за пределы локального окна, в результате чего невозможно будет осуществить поиск. Эта проблема привела к появлению "пирамидального" алгоритма LK, который начинает отслеживать от наивысшего (с низкой детализацией) к более низшему (с высокой детализацией) уровню пирамиды. Следование при помощи пирамид позволяет поймать большие перемещения в пределах локального окна.

Т.к. это важный и эффективный метод, то вначале будут представлены математические детали; читатели, которым данные детали не интересны, могут сразу перейти к описанию функций и разбору кода. Однако, рекомендуется все же изучить эти детали хотя бы для того, чтобы интуитивно понимать, что делать, если отслеживание работает плохо.

Принципы работы алгоритма Lucas-Kanade

Основная идея алгоритма LK заключена в трех предположениях.

1. *Постоянство яркости.* Пиксель объекта на изображении не изменяется внешне (по возможности) при перемещении от кадра к кадру. Для серого изображения (LK так же можно применять и для цветного изображения) это предположение означает, что яркость пикселя не изменяется при слежении от кадра к кадру.
2. *Временное постоянство или "малые перемещения".* Изменение движущейся поверхности патча изображения во времени происходит очень медленно. На практике это означает, что приращение времени достаточно велико относительно масштаба движения в изображении, то есть объект мало перемещаем от кадра к кадру.
3. *Пространственная когерентность.* Соседние точки на сцене, принадлежащие одной поверхности, имеют аналогичные движения и проектируются к соседним точкам на плоскости изображения.

Представленные предположения, проиллюстрированные на рисунке 10-4, дают эффективный алгоритм слежения. Первое требование, постоянство яркости, является требованием, которое свидетельствует о том, что пиксели в одном отслеживаемом патче должны выглядеть одинаково в течение долгого времени:

$$f(x, t) \equiv I(x(t), t) = I(x(t+dt), t+dt)$$

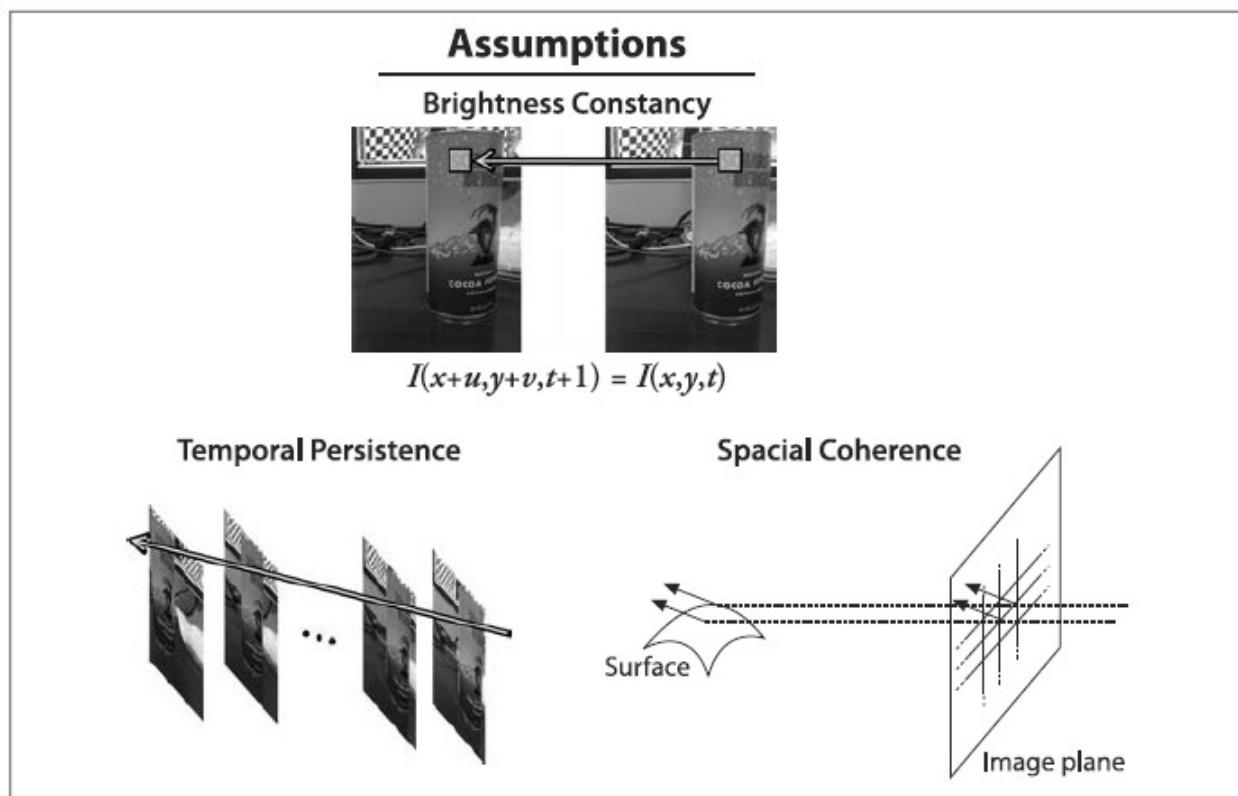


Рисунок 10-4. Предположения, составляющие основу оптического потока Lucas-Kanade: яркость патча, с отслеживаемым объектом в сцене, не меняется (сверху); перемещения медленны относительно частоты кадров (снизу слева); соседние точки остаются соседними (снизу справа)

Первое предположение достаточно простое и означает лишь то, что отслеживаемая интенсивность пикселя не изменяется в течение долгого времени:

$$\frac{\partial f(x)}{\partial t} = 0$$

Второе предположение, временное постоянство, по существу означает, что движения малы от кадра к кадру. Другими словами, можно рассматривать эти изменения как аппроксимацию производной интенсивности по времени (т.е. изменения между текущим и следующим кадрами малы). Для понимания данного предположения, рассмотрим случай с участием одной пространственной размерности.

В этом случае, все начинается с уравнения постоянства яркости с заменой яркости $f(x, t)$ (принимая во внимание неявную зависимость x от t) на $I(x(t), t)$ и последующим применением правила частичного дифференцирования. Все это даёт:

$$\underbrace{\frac{\partial I}{\partial x}}_{I_x} \left(\underbrace{\frac{\partial x}{\partial t}}_v \right) + \underbrace{\frac{\partial I}{\partial t}}_{I_t} \Big|_{x(t)} = 0$$

где I_x пространственная производная первого изображения, I_t производная между изображениями на определенном промежутке времени и v скорость, которую необходимо найти. В результате получаем простое уравнение скорости оптического потока в простом одномерном случае:

$$v = -\frac{I_t}{I_x}$$

Для начала необходимо разобраться с некоторыми интуитивными вещами одномерной задачи слежения. На рисунок 10-5 изображен "край" - состоящий из большого значения слева и малого значения справа - движущиеся вправо вдоль оси x . Цель заключается в том, чтобы определить скорость v с которой перемещается край, что собственно и изображено в верхней части рисунка 10-5. В нижней части рисунка показано, что измерение этой скорости — это просто "прирост движения", где прирост — это время, а движение — это наклон (пространственная производная). Отрицательный знак корректирует наклон относительно x .

Рисунок 10-5 показывает еще один аспект формулировки оптического потока: ранее изложенные предположения, вероятно, не совсем верны. Т.е. яркость изображения не очень стабильна, а временные шаги (которые устанавливаются при помощи камеры) зачастую не столь быстры относительно движения, как хотелось бы. В результате получаемая скорость не точна. Тем не менее, если быть "достаточно близкими", то можно справиться с проблемой итерационно. Итерация, показанная на рисунке 10-6, использует первоначальную (неточную) оценку скорости как отправную точку для последующих неоднократно повторяющихся итераций. Можно сохранить ту же пространственную производную по оси x , вычисленную в первом кадре, за счет предположения постоянства яркости — как результат, перемещение пикселей по оси x неизменно. Повторное использование уже вычисленной пространственной производной дает значительную вычислительную экономию. При этом производная по времени должна повторно вычисляться для каждой итерации и каждого кадра, однако, если быть достаточно близкими к началу, то сходимость (почти точная) достигается в пять итераций. Данный подход более известен как *метод Ньютона*. Если первая оценка не была достаточно близка, то метод Ньютона будет давать неверный ответ.

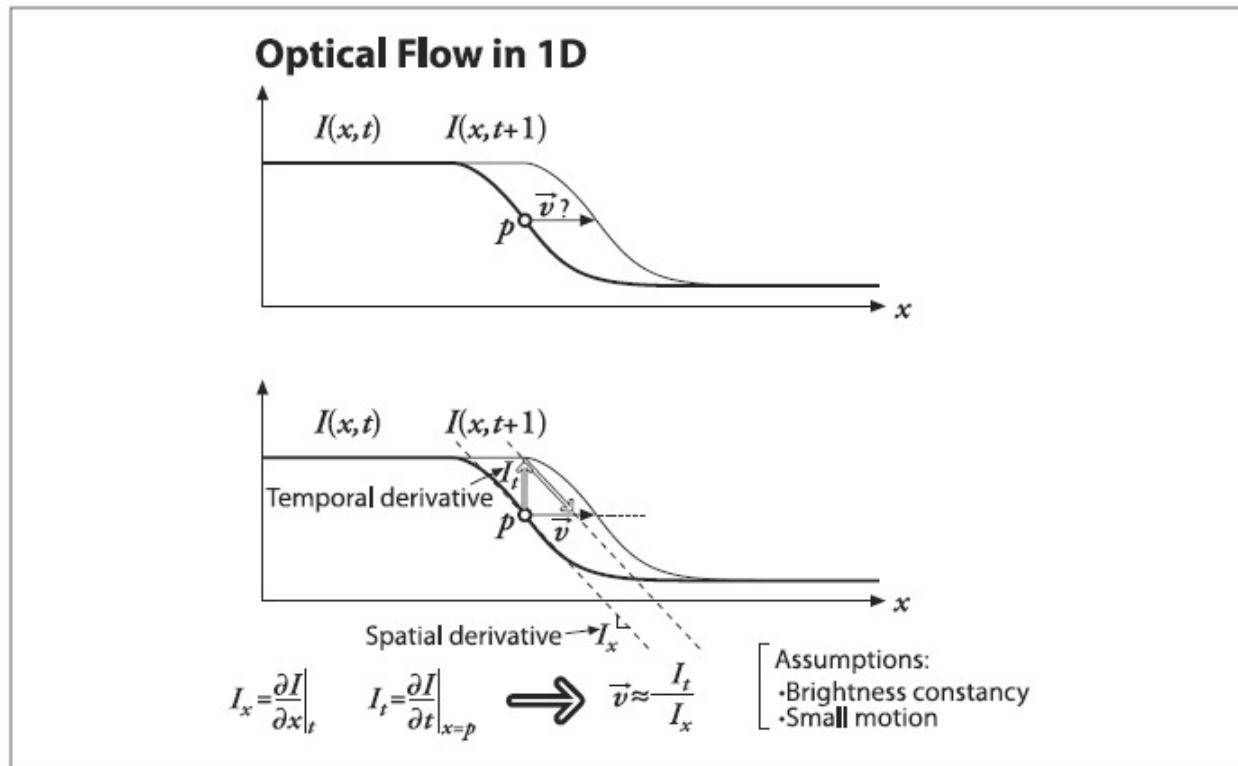


Рисунок 10-5. Оптический поток Lucas-Kanade для одного измерения: можно оценить скорость движущегося края (сверху) путем измерения отношения производной по времени к производной пространственной интенсивности

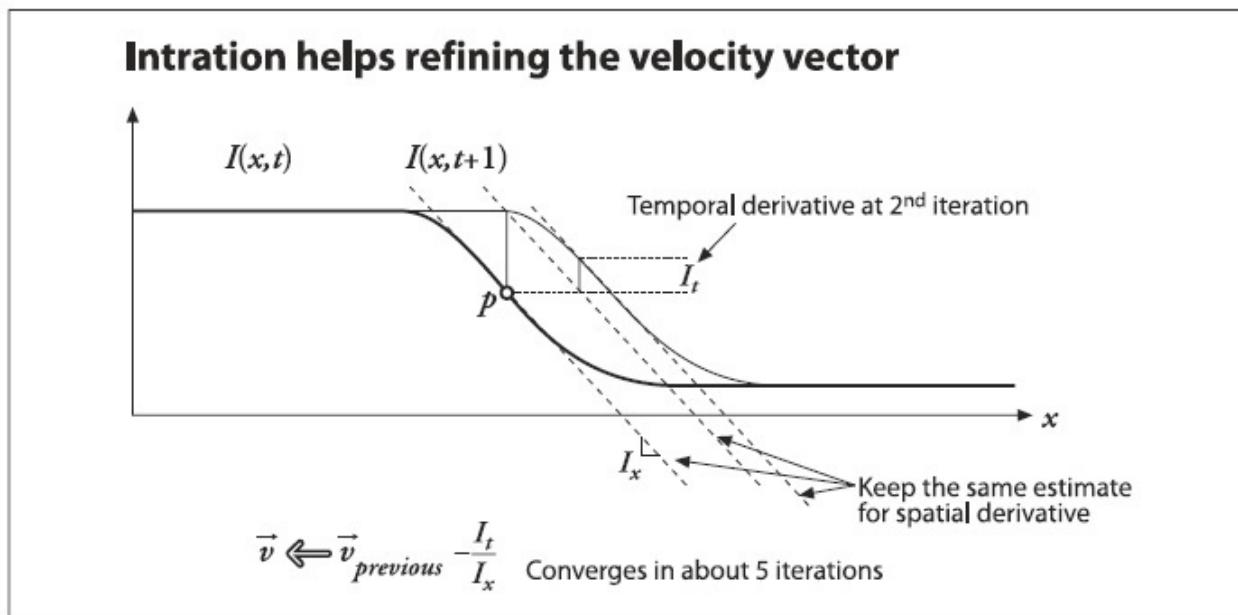


Рисунок 10-6. Итерационное уточнение решения оптического потока (метод Ньютона): используя два изображения и одну и ту же пространственную производную (наклон) для них, ищется производная по времени; сходимость к устойчивому решению, как правило, происходит по истечении нескольких итераций

Рассмотрев решение для одного измерения, можно перейти к рассмотрению случая с двумя измерениями. На первый взгляд может показаться, что все просто: всего то нужно добавить координату y . Небольшие изменения в обозначении и связывание компоненты y со скоростью v и компоненты x со скоростью u , получаем следующее уравнение:

$$I_x u + I_y v + I_t = 0$$

К сожалению, это уравнение с двумя неизвестными для любого пикселя. Это означает, что использование измерений на уровне одного пикселя является недостаточным условием и потому не позволяет получить уникальное решение двумерного движения точки. В таком случае, решение возможно получить только для движущейся компоненты, которая перпендикулярна или "нормальна" к линии, описанная уравнением потока. На рисунке 10-7 представлены математические и геометрические детали.

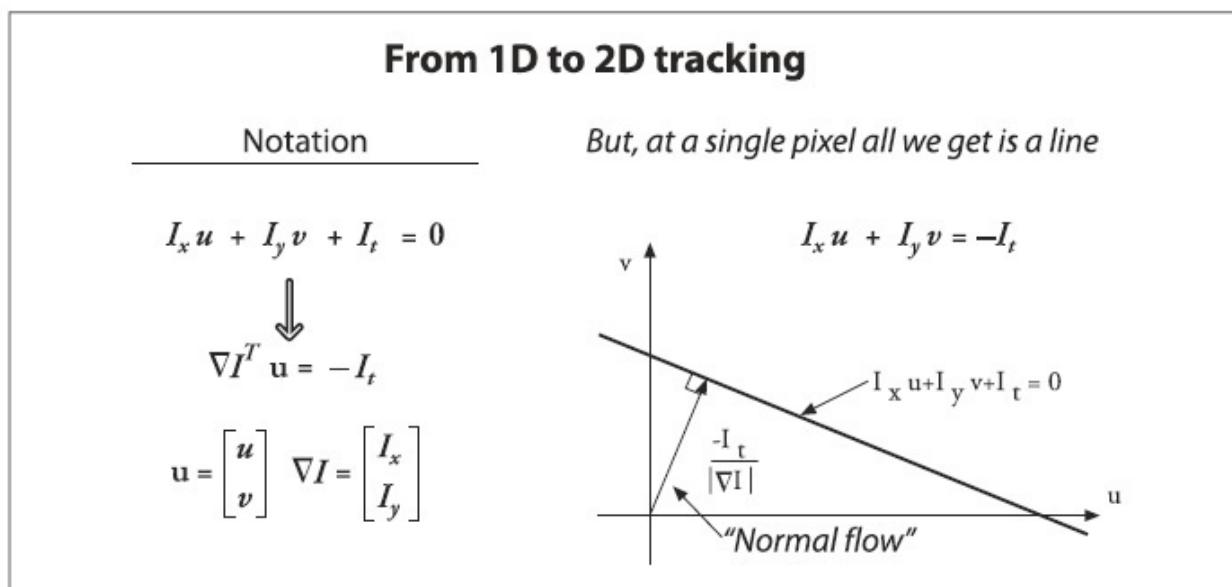


Рисунок 10-7. Двумерный оптический поток одного пикселя: оптический поток одного пикселя не определён и может привести к большему перемещению, которое перпендикулярно ("нормально") к линии, описанная уравнением потока

Полученные результаты нормального оптического потока – это последствия *проблемы отверстия*, возникающей от использования небольшого отверстия или окна, в котором производятся измерения движения. Чаще всего при слежении за движениями в малом отверстии можно увидеть только края, а не углы. Однако наличие только края является недостаточным условием для точного определения того, как (т.е. в каком направлении) перемещается весь объект, рисунок 10-8.

И все же, как обойти проблему, при которой по одному пикслю невозможно определить всё перемещение? Для решения проблемы необходимо обратиться к последнему предположению. Если локальный патч пикселей перемещается когерентно, то вполне легко можно рассчитать движение центрального пикселя при помощи окружающих пикселей, составив систему уравнений. Например, при использовании окна 5×5 (конечно окно может быть и 3×3 и 7×7 или еще каким-либо; если размер окна слишком большой, то это может привести к нарушению когерентности; если окно будет слишком маленьким, то вновь возникнет проблема отверстия) со значениями яркости (а можно и утроить для основного цветного оптического потока) текущего пикселя для вычисления его движения, то можно составить 25 уравнений следующего вида:

$$\underbrace{\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_{25}) & I_y(p_{25}) \end{bmatrix}}_A^{25 \times 2} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{d \times 1} = -\underbrace{\begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ \vdots \\ I_t(p_{25}) \end{bmatrix}}_b^{2 \times 1}$$

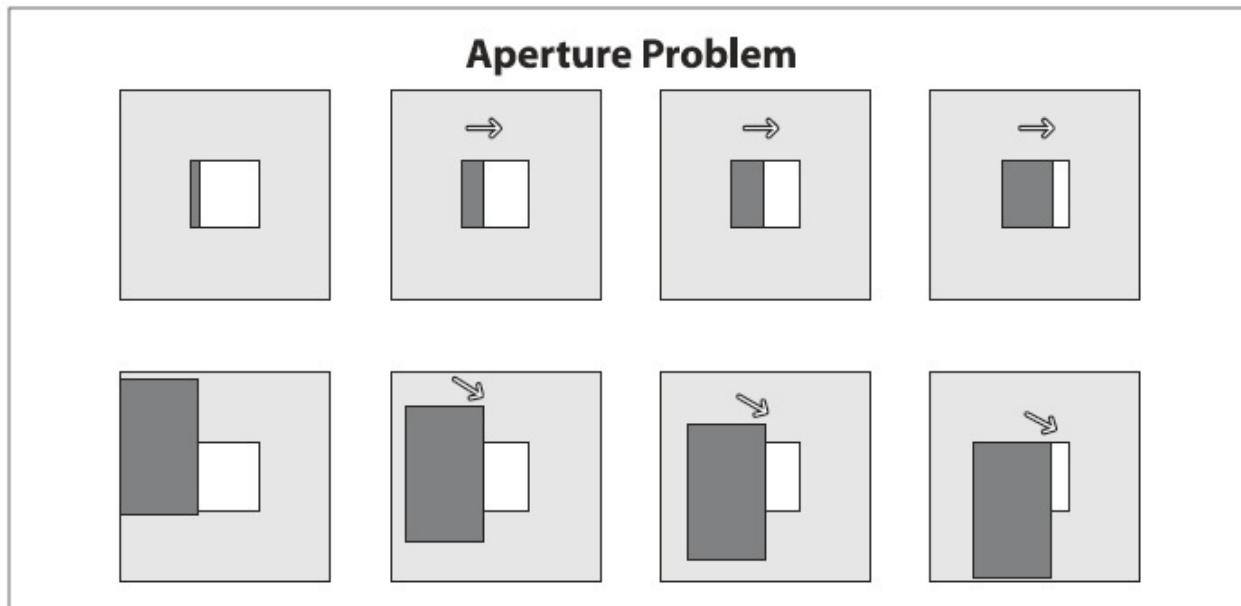


Рисунок 10-8. Проблема отверстия: через отверстие окна (верхний ряд) можно отследить движения края вправо, однако движения вниз отследить не возможно (нижний ряд)

Теперь для имеющейся ограниченной системы можно найти решение, если окно 5×5 будет содержать нечто большее, чем просто край. Для решения данной системы необходимо провести исследование минимума методом наименьших квадратов, решив $\min \|Ad - b\|^2$ в стандартной форме:

$$\underbrace{(A^T A)}_{2 \times 2} \underbrace{d}_{2 \times 1} = \underbrace{A^T b}_{2 \times 2}$$

Из этого соотношения можно получить компоненты движения v и u . Вот более подробное представление данного соотношения:

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

Решение данного уравнения выглядит следующим образом:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A)^{-1} A^T b$$

При каких условиях можно найти решение? – когда $(A^T A)$ обратимо. А $(A^T A)$ обратимо, когда имеет полный ранг матрицы (2), что осуществимо при наличии двух больших собственных векторов. Это справедливо для областей изображения, в которых есть текстуры, использующиеся как минимум в двух направлениях. В этом случае $(A^T A)$ будет иметь хорошие свойства тогда, когда окно слежения будет центрироваться в угловой области изображения. В данном случае вспоминается ранее обсуждаемый угловой детектор Harris. На самом деле, эти углы являются "хорошими особенностями для отслеживания" (см. ранее рассмотренные замечания, касающиеся `cvGoodFeaturesToTrack()`) именно потому, что $(A^T A)$ обладает двумя большими собственными векторами! Чуть позже будет показано, что все эти вычисления можно произвести при помощи функции `cvCalcOpticalFlowLK()`.

Читатели, которые понимают последствия предположения о малых и когерентных движениях могут быть обеспокоены тем фактом, что для большинства камер, работающих с частотой 30 Гц, наличие больших и некогерентных движений является обычным явлением. На самом деле оптический поток Lukas-Kanade сам по себе не очень хорошо работает именно по этой причине: стремление использовать большое окно для отслеживания большей составляющей движений слишком часто нарушает предположение о когерентности движения! Чтобы обойти эту проблему, можно сначала произвести слежение в больших пространственных масштабах, используя пирамиду изображения, а затем улучшать найденную начальную скорость движения, двигаясь вниз по уровням пирамиды пока не будут достигнуты необработанные пиксели изображения.

Следовательно, рекомендуемый метод вначале найдет решение для оптического потока в верхнем слое, а затем использует полученные оценки движения в качестве отправной точки в последующем нижележащем слое. Спуск по пирамиде осуществляется таким образом до тех пор, пока не будет достигнут последний слой. Таким образом, нарушения предположений сводятся к минимуму, и потому отслеживание движений осуществляется быстрее и продолжительнее. Эта более сложная функция известна как *пирамидальный оптический поток Lucas-Kanade*, рисунок 10-9. Функция `cvCalcOpticalFlowPyrLK()` реализует пирамидальный оптический поток Lucas-Kanade.

Не пирамидальный оптический поток Lucas-Kanade

Функция, реализующая алгоритм не пирамидального плотного оптического потока Lucas-Kanade:

```
void cvCalcOpticalFlowLK(
    const CvArr* imgA
    ,const CvArr* imgB
    ,CvSize      winSize
    ,CvArr*      velx
    ,CvArr*      vely
);
```

Результирующие массивы этой функции будут заполнены только теми пикселями, для которых возможно вычислить минимальную ошибку. Для пикселей, у которых не может быть вычислена эта ошибка (и таким образом смещение), связанная с ними скорость будет установлена в 0. В большинстве случаев, данная функция не используется. Следующий метод, основанный на пирамиде, в большинстве случаев дает намного лучшие результаты.

Пирамидальный оптический поток Lucas-Kanade

Функция `cvCalcOpticalFlowPyrLK()` используется для реализации алгоритма пирамидального оптического потока Lucas-Kanade. Как будет показано далее, данная функция использует "хорошие особенности для отслеживания", а также возвращает показания того, насколько хорошо происходит слежение.

```

void cvCalcOpticalFlowPyrLK(
    const CvArr* imgA
    ,const CvArr* imgB
    ,CvArr* pyrA
    ,CvArr* pyrB
    ,CvPoint2D32f* featuresA
    ,CvPoint2D32f* featuresB
    ,int count
    ,CvSize winSize
    ,int level
    ,char* status
    ,float* track_error
    ,CvTermCriteria criteria
    ,int flags
);

```

Эта функция имеет большой набор входных параметров. Первых два аргумента – это исходное и конечное изображения; оба одноканальные, 8-битные изображения. Следующие два аргумента это буферы, выделяемые под хранение пирамиды изображений. Размер этих буферов должен быть не менее $(img.width + 8) \times img.height / 3$ байт (размер выбран таким потому что в этих scratch пространства необходимо разместить не только изображение, но и пирамиду), по одному для каждого исходного изображения (*pyrA* и *pyrB*). (Если эти два указателя установлены в NULL, то выделение, использование и освобождение памяти будет происходить при вызове функции, что сказывается на производительности.) Массив *featuresA* содержит перемещающиеся точки, которые необходимо найти, *featuresB* – это аналогичный массив, в котором размещаются новые вычисленные локации точек массива *featuresA*; *count* – это количество точек в списке *featuresA*. Параметр *winSize* задает окно, используемое для вычисления локально когерентного перемещения. В связи с этим происходит построение пирамиды изображения для которой аргумент *level* задает глубину стека изображений. Если *level* установлен в 0, то пирамида не используется. Массив *status* имеет размерность *count*; в результате выполнения функции, этот массив будет содержать либо 1 (если соответствующая точка была найдена на втором изображении), либо 0 (если не найдена). Параметр *track_error* необязателен и может быть выключен путем установки в NULL. Если все же параметр *track_error* используется, то это массив чисел, по одному числу для каждой отслеживаемой точки и равное разнице между патчем отслеживаемой точки на первом изображении и патчем с места, куда данная точка переместилась на втором изображении. Можно использовать *track_error* для отбора точек, у которых локальные изменения патчей слишком велики.

Следующий аргумент *criteria* используется для установления критерия окончания вычислений. Данная структура используется во многих алгоритмах OpenCV, использующие итерационное решение:

```
cvTermCriteria(
    int type      // CV_TERMCRIT_ITER, CV_TERMCRIT_EPS или оба
    ,int max_iter
    ,double epsilon
);
```

Как правило, данной функции вполне достаточно, чтобы сгенерировать необходимое условие. Первый аргумент данной функции *CV_TERMCRIT_ITER* или *CV_TERMCRIT_EPS* сообщает алгоритму момент прекращения - либо после некоторого количества итераций, либо когда метрика сходимости достигнет некоторого малого значения, соответственно. Следующие два вспомогательных параметра функции задают непосредственно сами критерии останова. Использование *CV_TERMCRIT_ITER | CV_TERMCRIT_EPS* позволяет задействовать оба критерия останова (что, чаще всего и используется в реальном коде).

Последний аргумент *flags* позволяет контролировать внутреннюю организацию функции; он может быть установлен в любой из или во все (при помощи побитового OR) нижеперечисленные варианты:

CV_LKFLOW_PYR_A_READY

Пирамида изображения первого кадра, вычисляемая перед вызовом и сохраняемая в *pyrA*

CV_LKFLOW_PYR_B_READY

Пирамида изображения второго кадра, вычисляемая перед вызовом и сохраняемая в *pyrB*

CV_LKFLOW_INITIAL_GUESSES

Массив *B* уже содержит исходные предположительные координаты особенностей во время вызова функции

Этот флаг полезен при работе с последовательным видео. Пирамиды изображения несколько дороговаты на вычисления, поэтому следует избегать повторных вычислений, когда это возможно. Конечный кадр составляет пару исходному кадру и вычисляется следующим за исходным кадром. Если выделение памяти под буферы производится самим пользователем (в альтернативе, функция может выполнить данное действие сама), то пирамиды для каждого изображения будут располагаться именно в этих буферах после выполнения функции. Если вызвать данную функцию с

уже вычисленными значениями буферов, то перерасчет производиться не будет. Имея вычисленные передвижения точек в предыдущем кадре, можно рассчитывать на хорошую возможность задания правильных начальных локаций данных точек в следующем кадре.

В результате основной план довольно таки прост: необходимо предоставить изображения со списком точек, которые подлежат слежению *featuresA*, и вызвать функцию. После выполнения функции, необходимо проверить состояние массива для определения точек, которые удалось отследить, а затем проверить найденные новые локации данных точек *featuresB*.

Все это порождает вопрос, который до текущего момента не был рассмотрен: как решить, какие особенности являются хорошими для отслеживания? Ранее, при рассмотрении функции *cvGoodFeaturesToTrack()*, которая использует метод Shi и Tomasi, данная проблема решалась вполне надежным способом. В большинстве случаев, хороших результатов можно достичь за счет комбинированного использования *cvGoodFeaturesToTrack()* и *cvCalcOpticalFlowPyrLK()*. И разумеется никто не может запретить использование собственных критериев определения движения!

И в заключение будет представлен небольшой пример (Пример 10-1) использования *cvGoodFeaturesToTrack()* и *cvCalcOpticalFlowPyrLK()*; рисунок 10-10.

Пример 10-1. Пирамидальный оптический поток Lukas-Kanade

```
// Pyramid L-K optical flow example
//
#include <cv.h>
#include <cxcvcore.h>
#include <highgui.h>

const int MAX_CORNERS = 500;

int main(int argc, char** argv) {
    // Инициализация, загрузка двух изображений из файловой системы и
    // выделение памяти под изображения и структуры, необходимые для
    // получения результата
    //
    IplImage* imgA = cvLoadImage( "image0.jpg", CV_LOAD_IMAGE_GRAYSCALE );
    IplImage* imgB = cvLoadImage( "image1.jpg", CV_LOAD_IMAGE_GRAYSCALE );

    CvSize img_sz = cvGetSize( imgA );
    int win_size = 10;

    IplImage* imgC = cvLoadImage(
        "../Data/OpticalFlow1.jpg"
        ,CV_LOAD_IMAGE_UNCHANGED
    );
}
```

```

// Для начала необходимо отобрать особенности, которые подлежат
// отслеживанию
//
IplImage* eig_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );
IplImage* tmp_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );

int corner_count = MAX_CORNERS;
CvPoint2D32f* cornersA = new CvPoint2D32f[ MAX_CORNERS ];

cvGoodFeaturesToTrack(
    imgA
    ,eig_image
    ,tmp_image
    ,cornersA
    ,&corner_count
    ,0.01
    ,5.0
    ,0
    ,3
    ,0
    ,0.04
);

cvFindCornerSubPix(
    imgA
    ,cornersA
    ,corner_count
    ,cvSize(win_size, win_size)
    ,cvSize(-1, -1)
    ,cvTermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, 0.03)
);

// Применение алгоритма Lucas Kanade
//
char features_found[ MAX_CORNERS ];
float feature_errors[ MAX_CORNERS ];

CvSize pyr_sz = cvSize( imgA->width+8, imgB->height/3 );

IplImage* pyrA = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );
IplImage* pyrB = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );

CvPoint2D32f* cornersB = new CvPoint2D32f[ MAX_CORNERS ];

cvCalcOpticalFlowPyrLK(
    imgA
    ,imgB
    ,pyrA
    ,pyrB
    ,cornersA
    ,cornersB
    ,corner_count

```

```
,cvSize( win_size,win_size )
,5
,features_found
,feature_errors
, cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .3 )
,0
);

// Отрисовка полученного результата для наглядного представления
//
for( int i = 0; i < corner_count; i++ ) {
    if( features_found[i] == 0 || feature_errors[i] > 550 ) {
        printf("Error is %f/n", feature_errors[i]);
        continue;
    }

    printf("Got it/n");

    CvPoint p0 = cvPoint(
        cvRound( cornersA[i].x )
        ,cvRound( cornersA[i].y )
    );

    CvPoint p1 = cvPoint(
        cvRound( cornersB[i].x )
        ,cvRound( cornersB[i].y )
    );

    cvLine( imgC, p0, p1, CV_RGB(255,0,0), 2 );
}

cvNamedWindow("ImageA", 0);
cvNamedWindow("ImageB", 0);
cvNamedWindow("LKpyr_OpticalFlow", 0);

cvShowImage("ImageA", imgA);
cvShowImage("ImageB", imgB);
cvShowImage("LKpyr_OpticalFlow", imgC);

cvWaitKey(0);

return 0;
}
```



Рисунок 10-10. Разряженный пирамидальный оптический поток Lucas-Kanade: центральное изображение - это один из кадров видео, следуемый за изображением слева; на изображении справа проиллюстрированы вычисляемые движения - "хорошие особенности для отслеживания" (внизу, справа показан поток векторов на темном фоне для повышения смотрительности)

Методы плотного слежения

OpenCV содержит ещё два метода оптического потока, которые на данный момент используются крайне редко. Эти функции, как правило, гораздо медленнее Lukas-Kanade; кроме того, они не поддерживают сопоставление в пределах масштабирования пирамиды изображения и потому не могут отслеживать большие перемещения. По этим причинам в данном разделе данные методы будут рассмотрены кратко.

Метод Horn-Schunck

Метод Horn и Schunck был разработан в 1981 году. Это был один из первых методов, который использовал предположение о постоянстве яркости и на основе которого были выведены основные уравнения постоянства яркости. Решение этих уравнений, разработанное Horn и Schunck, основывалось на гипотезе о сглаживании ограничения скоростей v_x и v_y . Это ограничение было получено путем минимизации скорости упорядоченного Лапласиана оптического потока:

$$\frac{\partial}{\partial x} \frac{\partial v_x}{\partial x} - \frac{1}{\alpha} I_x (I_x v_x + I_y v_y + I_t) = 0$$

$$\frac{\partial}{\partial y} \frac{\partial v_y}{\partial y} - \frac{1}{\alpha} I_y (I_x v_x + I_y v_y + I_t) = 0$$

где α постоянный весовой коэффициент, более известный как *постоянная регуляризации*. Значение α влияет на сглаженность оптического потока (чем больше значение, тем выше сглаженность). Это относительно простое ограничение принудительного сглаживания и имеет эффект "наказания области", в которой поток изменяется по величине. Как и в Lucas-Kanade, метод Horn-Schunck полагается на итерационное решение дифференциальных уравнений.

```
void cvCalcOpticalFlowHS(
    const CvArr* imgA
    ,const CvArr* imgB
    ,int usePrevious
    ,CvArr* velx
    ,CvArr* vely
    ,double lambda
    ,CvTermCriteria criteria
);
```

где $imgA$ и $imgB$ должны быть 8-битными, одноканальными изображениями. Скорости по x и по y будут сохранены в параметры $velx$ и $vely$, соответственно, и должны быть 32-битными, вещественными, одноканальными изображениями. Параметр $usePrevious$ сообщает алгоритму об необходимости использования скоростей $velx$ и $vely$, вычисленных на предыдущем кадре, в качестве отправной точки для вычисления новых скоростей. Параметр $lambda$ это вес, связанный с *множителем Лагранжа*. Скорее всего в этом месте может возникнуть вопрос: "Что такое множитель Лагранжа?" Множитель Лагранжа появляется при попытке минимизации (одновременной) обоих уравнений: уравнения перемещения яркости и уравнений сглаживания; множитель - это относительный вес ошибок минимизации.

Метод блочного сопоставления

Скорее всего у любознательного читателя, дочитавшего до данного раздела, может возникнуть вопрос: "Почему столь большое внимание привлекает к себе оптический поток? Можно ведь просто сопоставить пиксели из одного кадра с пикселями другого кадра". Термин "блочное сопоставление" является обобщенным для целого класса подобных алгоритмов, в которых изображение разделяется на небольшие области, именуемые *блоками*. Блоки, как правило, имеют форму квадрата и содержат определенное количество пикселей. На практике довольно часто данные блоки могут

пересекаться. Алгоритмы блочного сопоставления разделяют текущее и предыдущее изображения на блоки с последующим вычислением перемещения этих блоков. Алгоритмы подобного рода играют важную роль во многих алгоритмах сжатия видео, а также в оптических потоках компьютерного зрения.

Т.к. алгоритм блочного сопоставления оперирует скоплениями пикселей, а не отдельными пикселями, то возвращаемые "изображения скорости", как правило, более низкого разрешения, чем исходные изображения. Но это не всегда так; все зависит от степени пересечения между блоками. Размер изображения, получаемого после выполнения алгоритма, задается следующей формулой:

$$W_{\text{result}} = \left\lfloor \frac{W_{\text{prev}} - W_{\text{block}} + W_{\text{shiftsize}}}{W_{\text{shiftsize}}} \right\rfloor_{\text{floor}}$$

$$H_{\text{result}} = \left\lfloor \frac{H_{\text{prev}} - H_{\text{block}} + H_{\text{shiftsize}}}{H_{\text{shiftsize}}} \right\rfloor_{\text{floor}}$$

Реализация данного метода в OpenCV использует поиск по спирали, который выстраивает свою работу на основе локации исходного блока (в предыдущем кадре) и последующим сравнением кандидатов в новые блоки с исходным. Это сравнение является абсолютной суммой разницы между пикселями (т.е. расстоянием L1). Если найдено достаточно хорошее совпадение, то поиск прекращается. Функция, реализующая данный метод, имеет следующий вид:

```
void cvCalcOpticalFlowBM(
    const CvArr* prev
    ,const CvArr* curr
    ,CvSize block_size
    ,CvSize shift_size
    ,CvSize max_range
    ,int use_previous
    ,CvArr* velx
    ,CvArr* vely
);
```

Параметры *prev* и *curr* это предыдущее и текущее изображения; оба должны быть 8-битными, одноканальными изображениями. *block_size* это размер блока, а *shift_size* размер шага между блоками (этот параметр контролирует - если это происходит - степень пересечения блоков). Параметр *max_range* это размер области вокруг блока, в которой будет производиться поиск соответствующего блока в следующем кадре. Если задан параметр *use_previous*, то *velx* и *vely* указывают на отправную точку, с которой начинается поиск блоков. (Если *use_previous* == 0, поиск блока будет производиться по

области на расстоянии *max_range* от локации исходного блока. Если *use_previous* != 0, то центр поиска будет соответствовать первым смещениям $\Delta x = \text{vel}_x(x, y)$ и $\Delta y = \text{vel}_y(x, y)$. Параметры *velx* и *vely* являются 32-битными одноканальными изображениями, которые хранят рассчитанные движения блоков. Как уже было сказано ранее, вычисления производятся на уровне блоков, поэтому вычисленные координаты будут соответствовать блокам, а не отдельным пикселям исходного изображения.

[П]|[РС]|(РП) Mean-Shift и Camshift слежение

В данном разделе будут рассмотрены два метода: *mean-shift* и *camshift* (где "camshift" означает "непрерывно адаптирующийся mean-shift"). Первый является обобщенным методом анализа данных (обсуждался в главе 9, в контексте сегментации). После введения в общую теорию метода, будет рассмотрено, как OpenCV применяет его для отслеживания на изображениях. Второй метод построен на первом методе и обеспечивает слежение за объектами, чей размер может изменяться в течение всей видео последовательности.

Mean-Shift

Алгоритм mean-shift – это надежный метод нахождения локальных экстремумов в плотном распределении набора данных. Это просто обработка непрерывного распределения; в данном контексте это по существу просто *алгоритм поиска восхождением к вершине*, применяемый к плотной гистограмме данных (слово "по существу" используется из-за наличия аспекта, зависящего от масштаба, а точнее: mean-shift эквивалентен применению свертки ядра mean-shift к непрерывному распределению с последующим применением алгоритма поиска восхождением к вершине). Однако, для дискретного набора данных это несколько менее тривиальная проблема.

Дескриптор "надежный" используется в своем формальном статистическом смысле: то есть, mean-shift игнорирует излишки данных. Это означает, что игнорируются точки данных, которые находятся далеко от максимумов. Это происходит за счет обработки точек только локального окна данных, в котором производится вся работа.

Алгоритм mean-shift работает следующим образом:

1. Выбирается окно поиска:
 - его первоначальное расположение;
 - его тип (равномерное, полиномиальное, экспоненциальное или гауссово);
 - его форма (симметричное или асимметричное, возможно повернутое, округлое или прямоугольное);
 - его размер (степень, при которой происходит сворачивание или отсечение).
2. Вычисляется центр масс окна (возможно взвешенный).
3. Совмещается центр окна с центром масс.

4. Возвращаться к шагу 2 до тех пор, пока окно не остановится (это происходит всегда). (Количество итераций обычно ограничено каким-то максимальным числом или каким-то эпсилон изменения сдвигаемого центра между итерациями; однако, количество, в конечном счете, гарантированно достигнет установленного значения).

Немного более формальный смысл алгоритма mean-shift: алгоритм связан с оценкой плотности ядра, где "ядро" — это функция, имеющая в основном местный фокус (например, распределение Гаусса). С довольно средневзвешенными и отсортированными по величине расположения точками ядер, можно представить распределение данных исключительно в терминах этих ядер. Mean-shift расходиться с оценкой плотности ядра, т.к. стремится оценить только градиент (направление изменений) распределения данных. Когда это изменение равное 0, то имеем стабильный (возможно локальный) максимум распределения. Около этого максимума могут быть и другие максимумы в том же или другом масштабе.

На рисунке 10-11 показаны уравнения, задействованные в алгоритме mean-shift. Эти уравнения могут быть упрощены, если ядро будет *прямоугольным*; благодаря данному факту сокращается векторное уравнение mean-shift, которое вычисляет центр масс распределения пикселей изображения:

$$x_c = \frac{M_{10}}{M_{00}}, \quad y_c = \frac{M_{01}}{M_{00}}$$

где нулевой момент вычисляется по следующей формуле:

$$M_{00} = \sum_x \sum_y I(x, y)$$

а первые моменты:

$$M_{10} = \sum_x \sum_y xI(x, y) \quad M_{01} = \sum_x \sum_y yI(x, y)$$

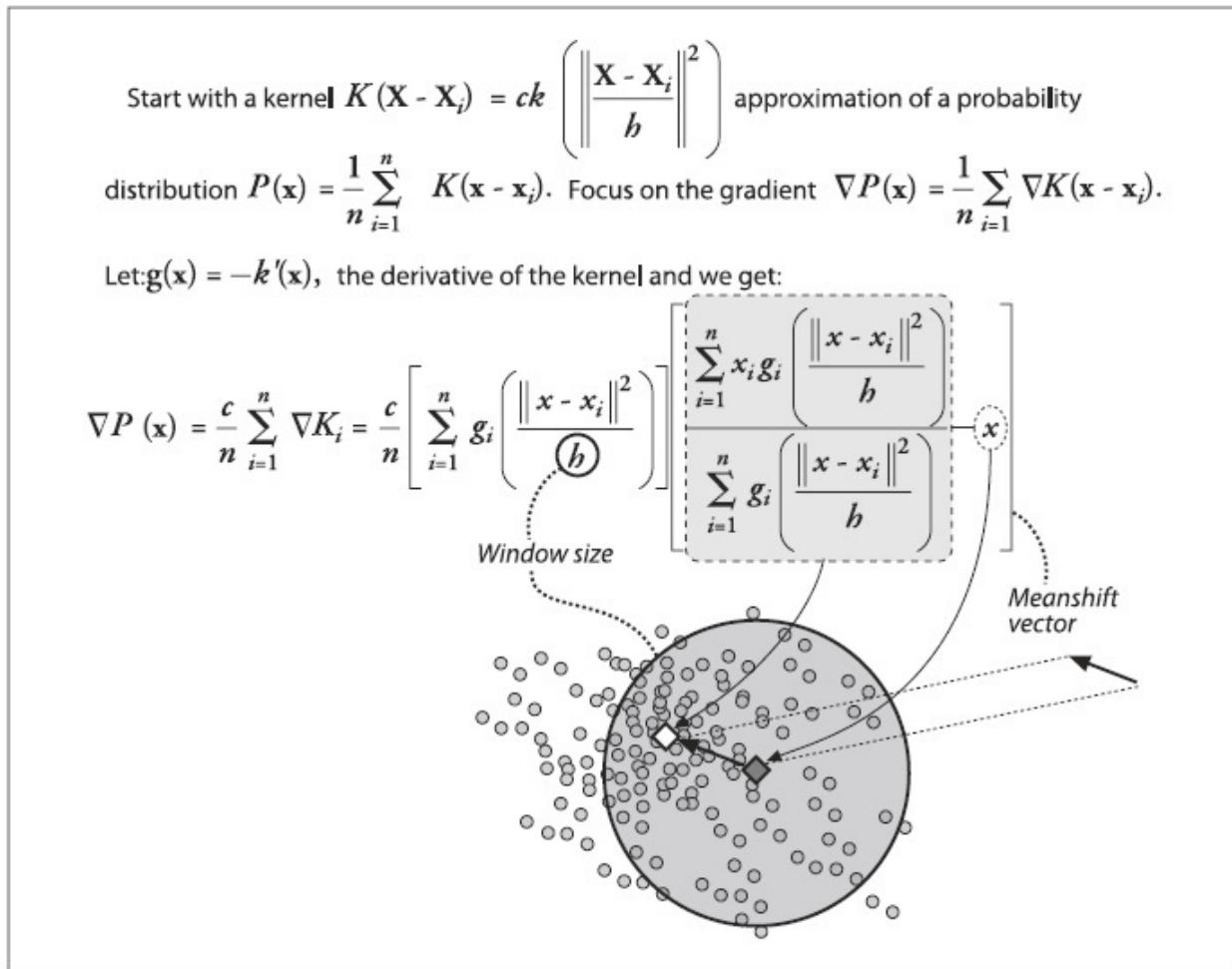


Рисунок 10-11. Уравнения mean-shift и их смысл

Вектор mean-shift показывает, как происходит центрирование окна mean-shift в соответствии с рассчитанным центром масс этого окна. Перемещение окна изменяет все то, что находится "под" ним, именно поэтому процесс центрирования итерационный. Такое центрирование всегда будет обеспечивать сходимость вектора mean-shift в 0 (т.е. когда центрирование более не возможно). Расположение сходимости является локальным максимумом распределения под окном. Различные размеры окна порождают различные максимумы, т.к. "максимум" зависит от масштаба.

На рисунке 10-12 представлен пример двумерного распределения данных и инициализация (в данном случае прямоугольного) окна. Стрелками показан процесс сходимости распределения в локальной моде (что соответствует максимуму). Найденный максимум статистически надежен в том смысле, что точки вне окна mean-shift не влияют на сходимость – алгоритм просто не "отвлекается" на далекие точки.

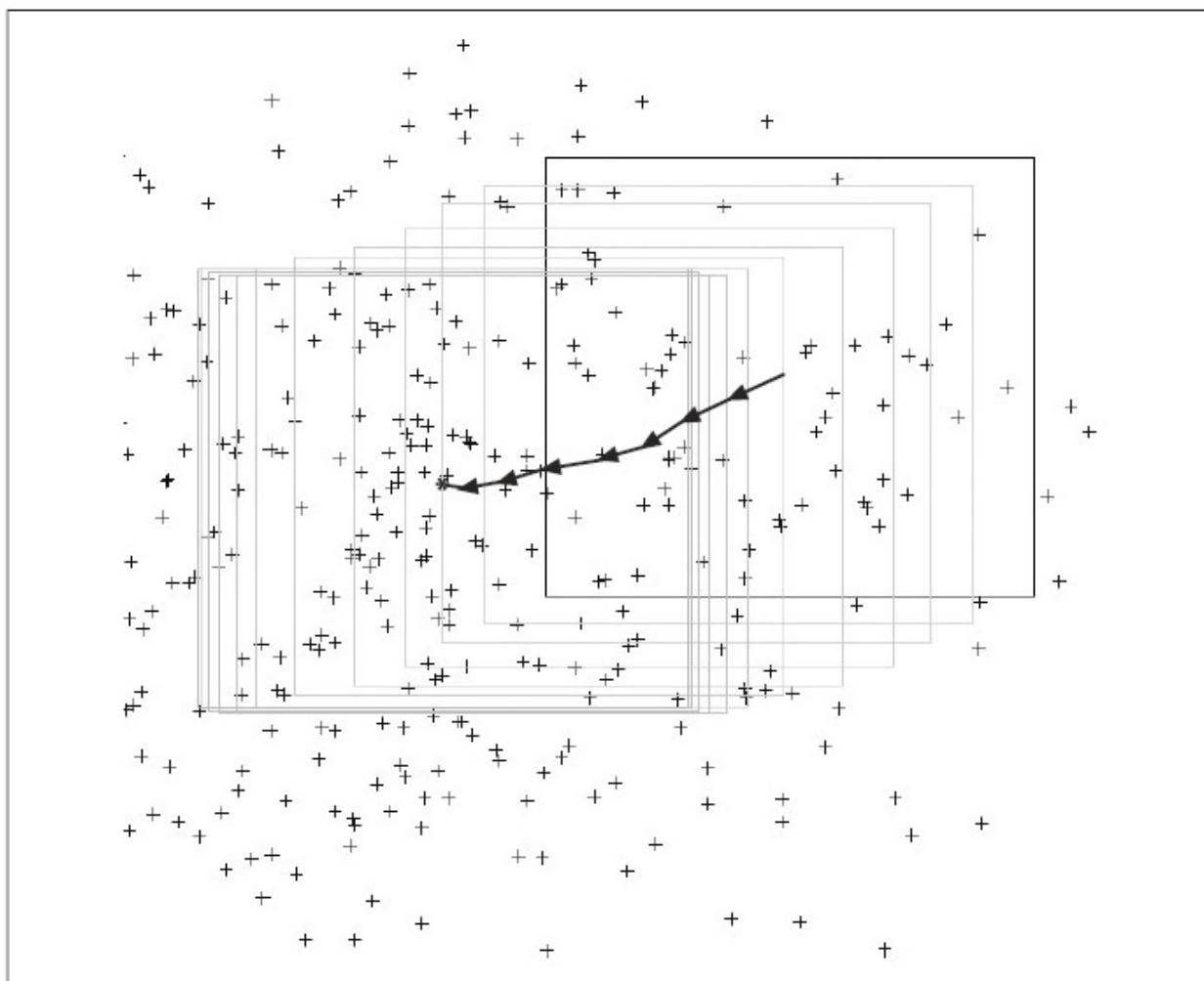


Рисунок 10-12. Алгоритм mean-shift в действии: исходное окно находится над двумерным массивом точек и последовательно центрирует его распределение данных до момента сходимости

В 1998 году стало ясно, что этот алгоритм может быть использован для отслеживания движущихся объектов в видеопотоке; с тех пор алгоритм был значительно расширен. Функция OpenCV, представляющая алгоритм mean-shift, реализована в контексте анализа изображения. Это, в частности, означает, что вместо некоторого произвольного набора точек (возможно, с какой-то произвольной размерностью), в OpenCV реализован mean-shift, ожидающий исходное изображение, представляющее плотное распределение для анализа. Это может быть представлено, как некое изображение двумерной гистограммы измерения плотности точек в некотором двумерном пространстве. Как оказалось, в рамках компьютерного зрения, это именно то, что необходимо делать большую часть времени: а именно, отслеживать движения кластера с интересными особенностями.

```

int cvMeanShift(
    const CvArr*     prob_image
    ,CvRect          window
    ,CvTermCriteria  criteria
    ,CvConnectedComp* comp
);

```

Параметр *prob_image*, представляющий из себя плотное распределение возможных локаций, может быть только одноканальным, но любого типа (*byte* или *float*) изображением. *window* задает начальное расположение и размер ядра окна. Критерий прекращения *criteria* был уже описан ранее и в основном содержит максимально доступное число итераций передвижения mean-shift и минимальные передвижения для которых рассчитывается расположение окна для выполнения условия сходимости. (Опять же, сходимость наступает всегда, однако, сам процесс может быть очень медленным вблизи локального максимума распределения, если распределение является "плоским"). Компонента *comp* содержит условие сходимости в *comp->rect* и сумму всех пикселей под окном в *comp->area*.

Функция *cvMeanShift()* является одним из вариантов реализации алгоритма mean-shift с прямоугольным окном, но она так же может быть использована для слежения. В этом случае, для начала необходимо составить распределение особенностей предоставленного объекта (например, цвет + текстура), затем запустить окно mean-shift над этим распределением и в заключении произвести вычисления в следующем кадре видеопотока с использованием составленного распределения. Начиная от текущего расположения окна, алгоритм mean-shift ищет новый максимум или моду распределения особенностей, которое (по-видимому) сосредоточено на объекте, производящий цвет и текстуру в первоначальном месте. Таким образом, окно mean-shift отслеживает движения объекта кадр за кадром.

Camshift

Отличие camshift от mean-shift заключается в том, что окно поиска приспосабливается к изменениям размера. Если имеется хорошо сегментированное распределение (например, особенностей лица), то данный алгоритм автоматически подстраивается под размеры лица, в зависимости от приближения или отдаления человека от камеры. Функция, реализующая данный алгоритм, выглядит следующим образом:

```
int cvCamShift(
    const CvArr*      prob_image
    ,CvRect          window
    ,CvTermCriteria   criteria
    ,CvConnectedComp* comp
    ,CvBox2D*         box = NULL
);
```

Первые четыре параметра аналогичны параметрам функции *cvMeanShift()*. Параметр *box*, если он задан, содержит недавно измененный размер окна, а также включает в себя ориентацию объекта, вычисленную при помощи моментов второго порядка. Приложения, занимающиеся слежением, будут использовать найденное измененное окно на предыдущем кадре в следующем кадре.

Многие люди думают, что mean-shift и camshift осуществляют слежение за счет цветных особенностей, но это не совсем так. Оба алгоритма могут использовать любого рода особенности, указанные в *prob_image*; следовательно, они являются очень легкими, прочными и эффективными.

[П]|[РС]|(РП) Шаблоны движения

Шаблоны движения были изобретены в MIT Media Lab Bobick и Davis и получили дальнейшее развитие благодаря одному из авторов. Это относительно новая работа сформировала основу реализации шаблонов движения в OpenCV.

Шаблоны движения являются эффективным способом отслеживания общих движений и особенно применимы при распознавании жестов. При использовании шаблонов движения необходимо наличие силуэта (или части силуэта) объекта. Силуэты объекта могут быть получены несколькими способами.

1. Простейший способ получения силуэтов объектов – это разумное использование стационарной камеры и последующего использования разностей между кадрами (как было описано в главе 9). Эти действия приводят к получению движущихся контуров объектов, которых вполне достаточно для производства работоспособных шаблонов движения.
2. Можно использовать цветные манипуляции. Например, при знании того, что фоном является все ярко зеленое, можно принять в качестве переднего плана все то, что не ярко зеленого цвета.
3. Другой способ (также обсуждаемый в главе 9) заключается в изучении фоновой модели, благодаря которой можно выделить новые объекты/людей на переднем плане как силуэты.
4. Можно использовать активный метод определения силуэта – например, создавая стену вблизи инфракрасного света при наличии инфракрасной камеры, смотрящей на эту стену. Любое попадание объекта в данное поле зрения будет отображаться как силуэт.
5. Можно использовать тепловизоры; любой горячий объект (например, лицо) можно будет рассматривать как объект переднего плана.
6. И в заключении, можно использовать методы сегментации для создания силуэтов (например, пирамидальную или mean-shift сегментацию), описанные в главе 9.

Пусть имеется хорошо сегментированный силуэт лица в виде белого прямоугольника, рисунок 10-13(А). Белый цвет используется для указания того, что все пиксели имеют значения типа float в самый последний момент времени. Движение прямоугольника свидетельствует о захвате нового силуэта и перекрытии им (новым) предыдущего в текущий момент времени; новый силуэт соответствует белому прямоугольнику на рисунке 10-13(В) и рисунке 10-14(С). Старые перемещения показаны на рисунке 10-13

в виде последовательности затемненных прямоугольников. Эти последовательно исчезающие силуэты фиксируются в истории, именуемая "изображением истории перемещений", как предыдущие перемещения.

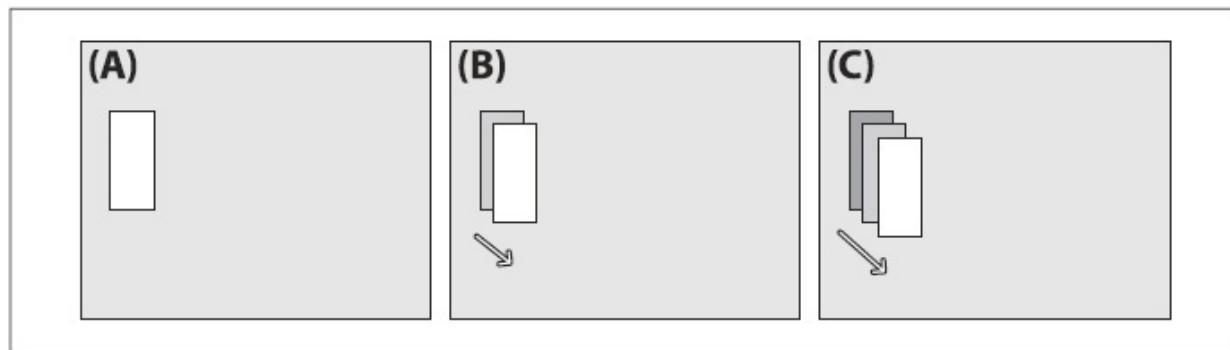


Рисунок 10-13. Диаграмма шаблона движения: (А) сегментированный объект в текущий момент времени (белый); (В) следующий временной шаг, объект переместился и был зафиксирован новый силуэт, старый переместился на задний план; (С) следующий временной шаг, объект вновь переместился, оставил старые сегменты позади в виде последовательности темных прямоугольников, тем самым сформировав изображение истории перемещений

Силуэты, временные отметки которых больше заданного значения, устанавливаются в 0, рисунок 10-14. Функция, производящая шаблоны движения, выглядит следующим образом:

```
void cvUpdateMotionHistory(
    const CvArr* silhouette
    , CvArr* mhi
    , double timestamp
    , double duration
);
```

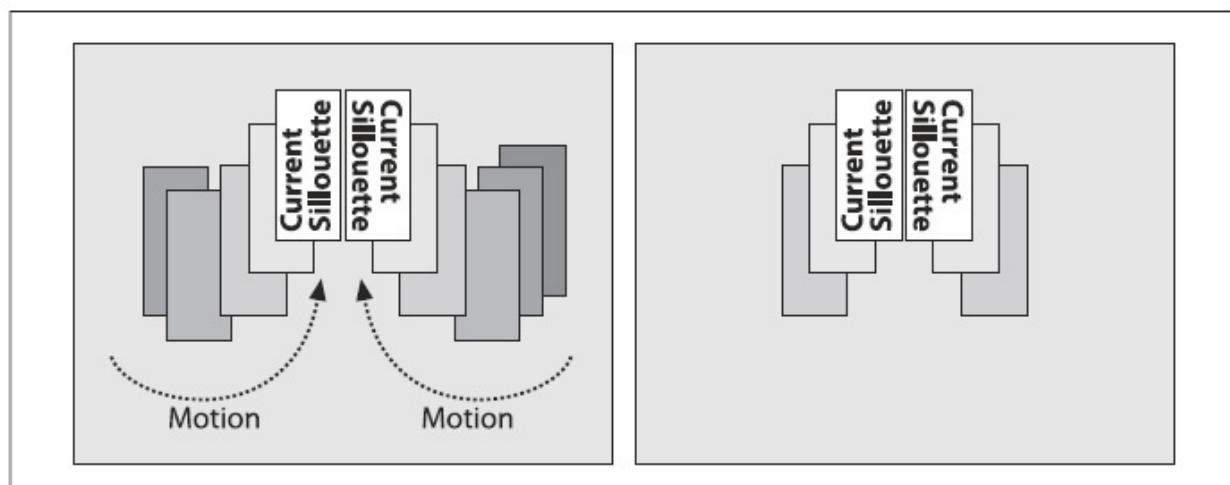


Рисунок 10-14. Шаблон движения силуэтов двух движущихся объектов (слева); силуэты, чьи временные отметки превышают заданное значение, устанавливаются в 0 (справа)

В `cvUpdateMotionHistory()` все массивы изображений содержат только одноканальные изображения. *silhouette* - это изображение, в котором ненулевые пиксели представляют собой недавно сегментированные силуэты объектов переднего плана. Изображение *mhi* это вещественное изображение, которое представляет шаблон движения. Параметр *timestamp* - это текущая отметка времени (как правило, в миллисекундах), а также продолжительность, на протяжении которой силуэт должен оставаться в *mhi*. Другими словами, любые пиксели *mhi*, у которых временная отметка старше (меньше), чем *timestamp* минус *duration*, устанавливаются в 0.

После получения коллекции силуэтов объекта, наложенных во времени, формируется представление о перемещениях в целом за счет взятия градиента *mhi* изображения. После взятия этих градиентов (например, при помощи функций Scharr или Sobel, описанных в главе 6), некоторые градиенты будут большими и неработоспособными. Градиенты будут неработоспособными, когда старые или неактивные части изображения *mhi* будут установлены в 0, в результате чего будут получаться большие градиенты вокруг внешних границ силуэтов, рисунок 10-15(А). Поэтому, зная продолжительность временного шага, с которым заносятся новые силуэты в *mhi* через `cvUpdateMotionHistory()`, можно узнать насколько большим должен быть градиент. Таким образом, можно использовать данную величину градиента для удаления градиентов, которые слишком велики, рисунок 10-15 (Б). И в заключение, за счет коллекции можно выявить общее направление перемещения, рисунок 10-15(С). Всё что показано на частях (А) и (Б) изображения выполняется при помощи функция `cvCalcMotionGradient()`:

```
void cvCalcMotionGradient(
    const CvArr*    mhi
    ,CvArr*         mask
    ,CvArr*         orientation
    ,double          delta1
    ,double          delta2
    ,int             aperture_size = 3
);
```

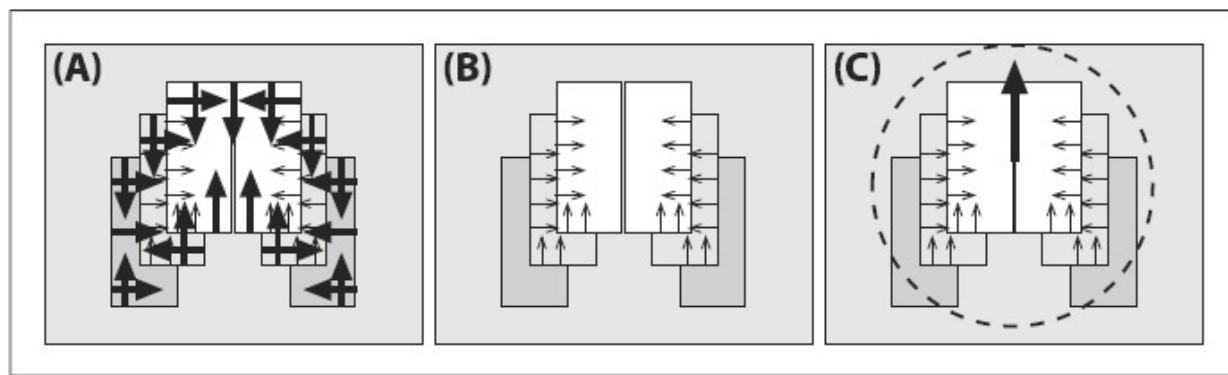


Рисунок 10-15. Градиенты перемещения *mhi* изображения: (A) величина и направление градиентов; (B) устранение больших градиентов; (C) поиск общего направления движения

В *cvCalcMotionGradient()* все массивы изображения являются одноканальными. Входное изображение *mhi* является вещественным изображением истории перемещения, а входные переменные *delta1* и *delta2* являются (соответственно) минимальным и максимальным значениями градиента. Ожидаемая величина градиента – это среднее значение временной отметки между каждым силуэтом при последовательном вызове *cvUpdateMotionHistory()*; хорошие результаты можно получить при *delta1* установленного в (среднее значение)/2, а *delta2* в 3×(среднее значение)/2. Переменная *aperture_size* задает размер оператора градиента по ширине и высоте. Это значение может быть установлено в -1 (градиент 3×3, фильтр *CV_SCHARR*), 3 (по умолчанию фильтр Sobel 3×3), 5 (фильтр Sobel 5×5) или 7 (фильтр 7×7). Выходной параметр *mask* – это одноканальное 8-битное изображение, в котором ненулевые элементы указывают на валидные градиенты, *orientation* – это вещественное изображение, содержащее угол направления градиента в каждой точке.

Функция *cvCalcGlobalOrientation()* определяет общее направление движения как векторную сумму действующих направлений градиента.

```
double cvCalcGlobalOrientation(
    const CvArr* orientation
    ,const CvArr* mask
    ,const CvArr* mhi
    ,double timestamp
    ,double duration
);
```

При использовании функции *cvCalcGlobalOrientation()*, в нее передаются вычисленные в *cvCalcMotionGradient()* *orientation* и *mask* вместе с *timestamp*, *duration* и конечным *mhi* из *cvUpdateMotionHistory()*; в результате будет возвращен суммарный вектор общего направления, рисунок 10-15(C). *timestamp* вместе с *duration* сообщают функции сколько движений необходимо рассмотреть из изображений *mhi* и *orientation*.

Вычислить общее направление передвижения можно за счет центра масс каждого силуэта *mhi*, однако, суммирование заранее вычисленных векторов передвижения дает результат гораздо быстрее.

Помимо этого, можно изолировать определенный регион изображения *mhi* и определить локальное перемещение этого региона, рисунок 10-16. На рисунке изображение *mhi* сканируется на текущих регионах силуэта. После того, как будет найден регион, помеченный последней отметкой времени, за счет его периметра ищется недавнее перемещение (последние силуэты) простым использованием пространства за пределами периметра данного региона. Когда такие движения будут найдены, нисходяще-пошаговое потоковое заполнение изолирует локальный регион передвижения, который "разливается" от текущего расположения интересующего объекта. После выполнения поиска, можно переходить к вычислению локального направления градиента перемещения в "разлитой" области, затем удалить этот регион и повторять процесс до тех пор, пока все регионы не будут найдены (рисунок 10-16).

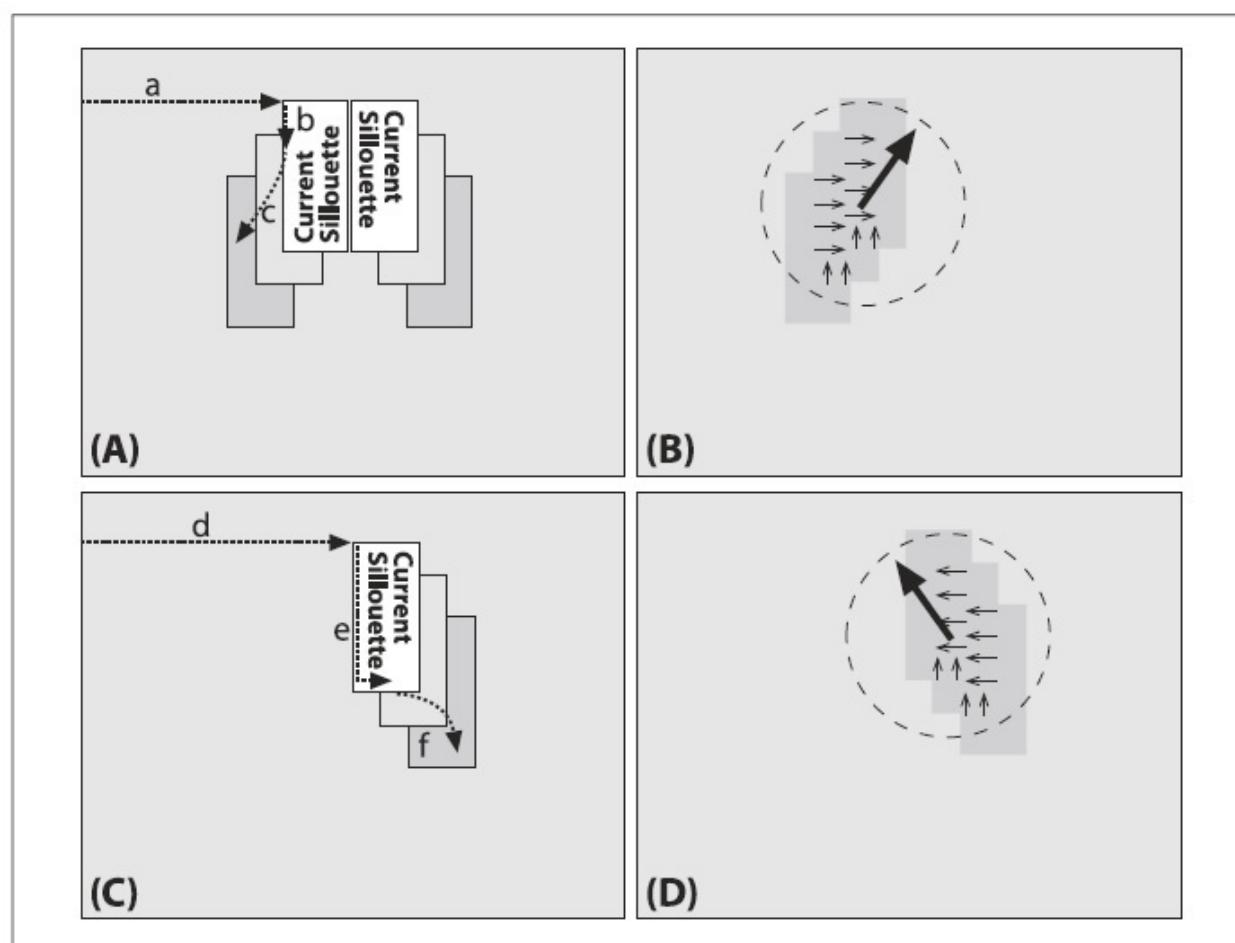


Рисунок 10-16. Сегментация регионов перемещения на изображении *mhi*: (A) сканирование изображения *mhi* для текущих силуэтов (a) и если найден результат, то выйти за периметр в поисках других недавних силуэтов (b); когда будет найден последний силуэт, выполнить нисходяще-пошаговое заполнение для

изоляции локального перемещения; (B) использовать градиенты, найденные в изолированных локальных регионах перемещения, для вычисления локального перемещения; (C) удалить ранее найденный регион и выполнить поиск для следующего региона силуэта (d), выполнить по нему сканирование (e) с последующим применением исходящего-пошагового потокового заполнения (f); (D) вычислить перемещение в рамках недавно изолированного региона и повторять процессы (A)-(C) до тех пор, пока не закончатся силуэты

`cvSegmentMotion()` - это функция, которая изолирует и вычисляет локальное перемещение:

```
CvSeq* cvSegmentMotion(
    const CvArr* mhi
    , CvArr* seg_mask
    , CvMemStorage* storage
    , double timestamp
    , double seg_thresh
);
```

Параметр `mhi` - это одноканальное вещественное изображение. Параметр `storage` - это структура типа `CvMemoryStorage` под которую заранее необходимо выделить память при помощи `cvCreateMemStorage()`. Другой входной параметр `timestamp` содержит значение, указывающее на последний силуэт из `mhi` в котором необходимо выделить сегмент локальных передвижений. И в завершении, необходимо передавать значение `seg_thresh`, указывающее на максимально возможный уровень спуска (от текущего момента времени и в сторону предыдущих передвижений). Этот параметр необходимо задавать во избежания ситуаций, при которых перекрывающиеся предыдущие силуэты несливались вместе с намного старыми передвижениями.

Как правило, лучше всего устанавливать `seg_thresh` в 1.5 от средней разницы между отметками времени силуэтов. Эта функция возвращает последовательность `CvSeq` структур `CvConnectedComp`, по одной для каждого отдельно найденного движения, очерченными локально перемещающимися регионами; в дополнение к этому, функция также возвращает `seg_mask` - одноканальное, вещественное изображение, в котором на каждом регионе выделяются передвижения за счет отметок с ненулевыми значениями (нулевое значение указывает на отсутствие передвижения). Для вычисления локальных передвижений, по одному за раз, необходимо вызывать `cvCalcGlobalOrientation()` с использованием соответствующей маски региона, которая выбирается из соответствующей структуры `CvConnectedComp` или обособленной части `seg_mask`; например, так:

```
cvCmpS(  
    seg_mask  
    // , [value_wanted_in_seg_mask]  
    // , [your_destination_mask]  
    , CV_CMP_EQ  
)
```

Теперь, с учетом всего рассмотренного материала данного раздела главы, должен быть понятен пример *motempl.c*, который поставляется с OpenCV в разделе *.../opencv/samples/c/*. Функция *update_mhi()* извлекает шаблоны согласно пороговым различиям между кадрами с последующей передачей полученных силуэтов в *cvUpdateMotionHistory()*:

```
...  
cvAbsDiff( buf[idx1], buf[idx2], silh );  
cvThreshold( silh, silh, diff_threshold, 1, CV_THRESH_BINARY );  
cvUpdateMotionHistory( silh, mhi, timestamp, MHI_DURATION );  
...
```

В результате градиенты размещаются в изображение *mhi*, а маска валидных градиентов получается за счет использования *cvCalcMotionGradient()*. Затем под *CvMemStorage* выделяется память (если уже выделена, то только очищается) и в результате локальные передвижения размещаются в структурах *CvConnectedComp* последовательности *seq* типа *CvSeq*:

```
...
cvCalcMotionGradient(
    mhi
    ,mask
    ,orient
    ,MAX_TIME_DELTA
    ,MIN_TIME_DELTA
    ,3
);

if( !storage )
    storage = cvCreateMemStorage(0);
else
    cvClearMemStorage(storage);

seq = cvSegmentMotion(
    mhi
    ,segmask
    ,storage
    ,timestamp
    ,MAX_TIME_DELTA
);

```

Цикл "for" работает до выполнения условия достижения значения `seq->total` структуры `CvConnectedComp`, извлекая при этом ограничительные прямоугольники для каждого перемещения. Итерационный процесс начинается с -1, что соответствует особому случаю для нахождения общего перемещения на изображении. Для сегментов локальных перемещений, сначала отбрасываются малые сегментированные участки, а затем рассчитывается ориентация для оставшихся при помощи `cvCalcGlobalOrientation()`. Вместо использования точных масок, эта функция ограничивает расчеты перемещения за счет использования ROIs, которые ограничивают локальные перемещения. Любые малые участки отбрасываются. В заключении, функция рисует перемещения. Пример человека, хлопающего в ладони, показан на рисунке 10-17 в виде четырех последовательных кадров (полный пример кода можно найти в `.../opencv/samples/c/motempl.c`). В той же последовательности, также представлена поза "Y" дескрипторами (Ни моментами) из главы 8, однако, данный пример отсутствует в директории примеров `samples`.

```
for( i = -1; i < seq->total; i++ ) {
    // case of the whole image
    if( i < 0 ) {
        // ...[does the whole image]...
    // i-th motion component
    } else {
        comp_rect = ((CvConnectedComp*)cvGetSeqElem( seq, i ))->rect;
        // [reject very small components]...
    }

    ...[set component ROI regions]...
    angle = cvCalcGlobalOrientation( orient, mask, mhi, timestamp, MHI_DURATION );
    ...[find regions of valid motion]...
    ...[reset ROI regions]...
    ...[skip small valid motion regions]...
    ...[draw the motions]...
}
```

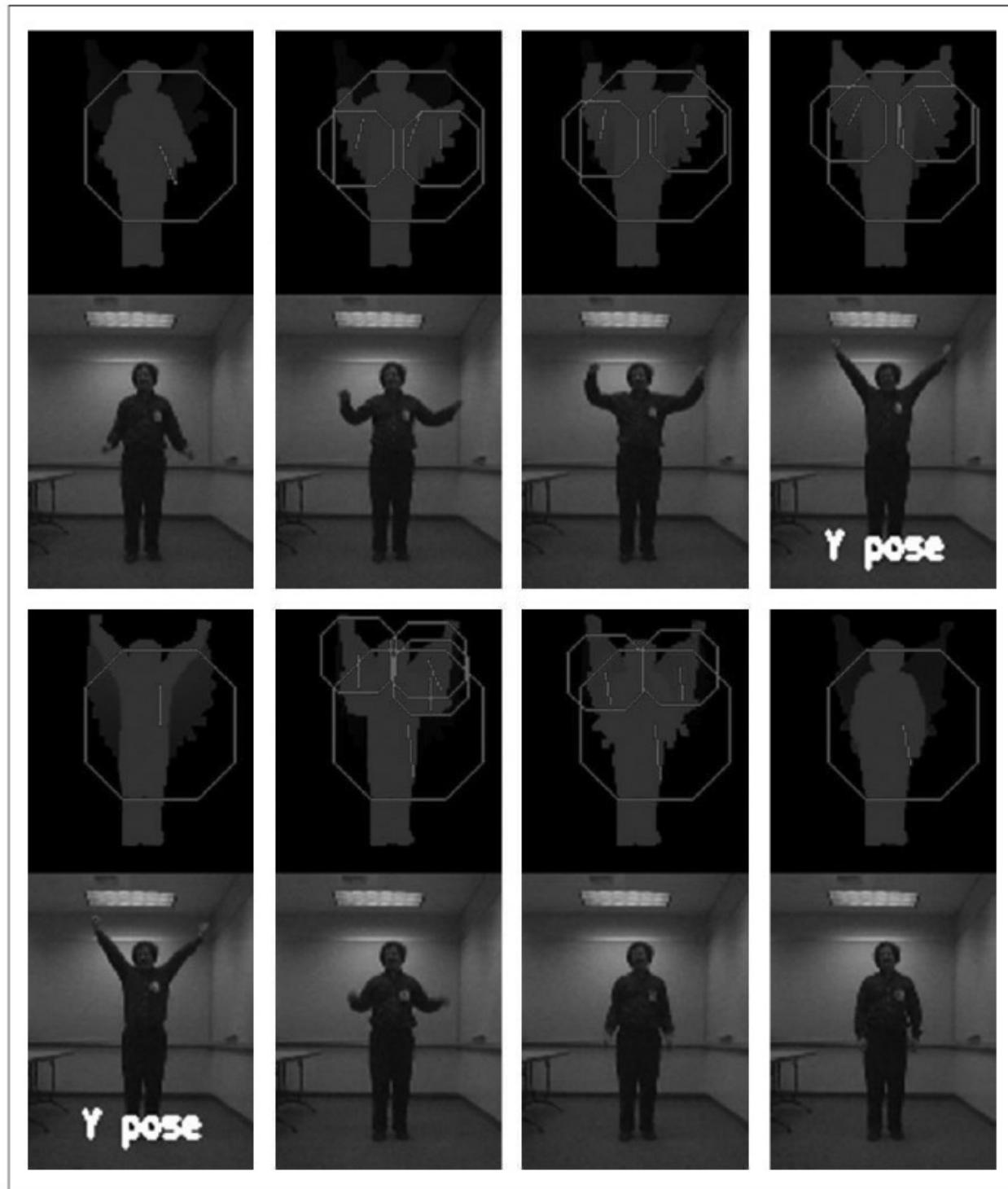


Рисунок 10-17. Результаты работы функции шаблонов движения: определение движений по горизонтали и сверху вниз, совершаемые человеком, а также общие направления перемещения (отмеченные большим восьмиугольником) и локальные перемещения (отмеченные небольшими восьмиугольниками); поза "Y" может быть распознана при помощи дескрипторов (Ну моментов)

[П]|[РС]|(РП) Оценочная функция

Пусть, в качестве примера, отслеживаются перемещения человека, гуляющего вдоль камеры. В каждом кадре происходит фиксация местоположения этого человека. Это может быть сделано любым, ранее описанным, способом, но в каждом отдельном случае ищется оценка положения человека в каждом кадре. Скорее всего, эта оценка будет не совсем точной. Причин для этого множество. Это может быть связано и с неточностью датчика, и из-за использования приближений на ранних этапах обработки, и с вопросами, связанными с сокрытиями или тенями, или с возможной сменой формы из-за того, что человек во время перемещений размахивает руками и ногами. Независимо от источника, ожидается, что изменения будут меняться несколько беспорядочно, а о "действительных" значениях может идти речь лишь при наличии идеализированного датчика. Все эти неточности можно представить, как простое добавление шума к процессу отслеживания.

Требуется иметь возможность оценки передвижений этого человека за счет максимально возможного использования сделанных измерений. Таким образом, совокупный эффект от всех измерений позволит обнаруживать часть перемещений человека, не зависящих от шума. Ключевым дополнительным ингредиентом является модель перемещения человека. Например, можно смоделировать перемещение человека со следующим утверждением: "Человек попадает в кадр с одной стороны и перемещается вдоль кадра с постоянной скоростью". За счет такой модели можно определить не только местоположение человека, но и какие параметры модели поддерживают данное наблюдение.

Данная задача разбивается на два этапа (рисунок 10-18). Первый этап, как правило, именуемый этапом предсказаний, использует информацию, полученную в прошлом, для улучшения модели с местоположением человека (или предмета). Второй этап, этап коррекции, производит измерение с последующим его сопоставлением с предсказанными ранее измерениями.

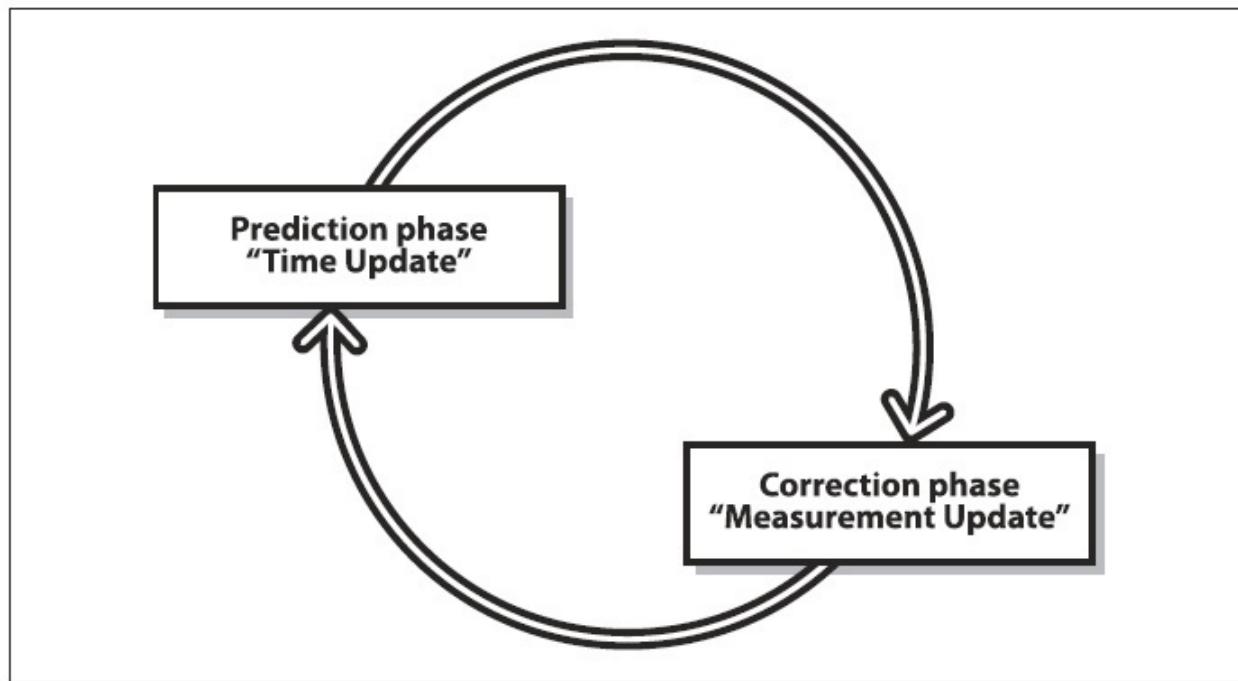


Рисунок 10-18. Двухэтапный цикл оценки: предсказание на основе предыдущих данных с последующим применением новых измерений

Принцип действия задачи двухэтапной оценки соответствует оценке наиболее популярного метода, использующего *фильтр Kalman*. В дополнение к этому методу, есть ещё один наиболее важный метод, *алгоритм условно плотного распространения*, реализующий широкий класс методов, известных как *фильтры частиц*. Основное различие между фильтром Kalman и алгоритмом условно плотного распространения сводится к описанию плотности вероятного состояния. Данное различие будет более подробно рассмотрено в следующих разделах.

Фильтр Kalman

С момента первых упоминаний (1960 г.) о фильтре Kalman его значение заметно возросло. Основная идея фильтра Kalman заключается в том, что при строгом, но разумно, наборе предположений можно – при учете истории изменений системы – построить модель состояний системы, которая максимизирует апостериорную вероятность предыдущих изменений. Более подробную информацию можно найти в работах Welsh и Bishop. Кроме того, можно максимизировать апостериорную (это академический жаргон, означающий "взгляд в прошлое"; таким образом, когда говорят, что такое то распределение "максимизирует апостериорную вероятность", то это можно трактовать как "что на самом деле произошло") вероятность без ведения длинной истории предыдущих изменений. Вместо этого происходит многократное обновление модели состояний системы и сохранение этой модели для последующей итерации. Это значительно упрощает вычислительную часть этого метода.

Перед тем, как переходить к деталям, необходимо уделить немного времени обсуждению предположений. Имеется три важных предположения, необходимые для теоретического истолкования фильтра Kalman: (1) моделируемая система линейна, (2) шум – это измерения, сделанные вне "белого", и (3) этот шум имеет Gaussian природу. Первое предположение означает, что состояние системы в момент времени k может быть смоделировано как некая матрица, помноженная на состояние в момент времени $k - 1$. Остальные предположения означают, что шум не коррелируется во времени, а его амплитуда может быть точно смоделирована только при помощи среднего значения и ковариации (т.е. шум полностью можно описать при помощи первых и вторых моментов). Эти предположения могут показаться ограниченными, однако, на самом деле они охватывают множество случаев (единственное, что стоит учесть, так это то, что если начальное состояние (например) имеет шансы 50×50 , то потребуется использовать что-то более сложное, чем только фильтр Kalman).

Что же означает "максимизация апостериорной вероятности предыдущих изменений"? Это означает, что новая модель строится после проведения измерений – с учетом неопределенности предыдущей модели и нового значения – модели, обладающей лучшей вероятностью быть правильной. Это в свою очередь означает, что фильтр Kalman, с учетом трех предположений, является лучшим способом объединения данных из различных источников или из одного, но с данными полученными в разные моменты времени.

При наличии уже знакомой информации происходит получение новой информации с последующим изменением уже знакомой информации, основываясь на надежности старой и новой информации в соответствии с взвешенным сочетанием старого и нового.

Далее будет более подробно рассмотрен принцип работы фильтра на примере одномерного движения.

Немного математики Kalman

Так в чем же суть фильтра Kalman? – сплав информации. Например, необходимо узнать, где некая точка находится на линии (одномерный сценарий). В результате наличия шума имеется два ненадежных (в Гауссовом смысле) места: \bar{x}_1 и \bar{x}_2 . Гауссова неопределенность имеет значения \bar{x}_1 и \bar{x}_2 в совокупности со стандартными отклонениями σ_1 и σ_2 . Стандартные отклонения – это выражения неопределенности относительно того, насколько хороши измерения. Распределение вероятности в зависимости от локаций именуется Гауссовым распределением:

$$p_i(x) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp\left(-\frac{(x - \bar{x}_i)^2}{2\sigma_i^2}\right) \quad (i=1, 2)$$

При наличии двух таких измерений, каждое с Гауссовым распределением вероятности, можно ожидать, что плотность вероятности при некотором значении x и учете обоих измерений будет пропорциональна $p(x) = p_1(x)p_2(x)$. Оказывается, этот результат также является распределением Гаусса, для которого можно вычислить среднее значение и стандартное отклонение следующим образом: при условии, что

$$p_{12}(x) = \exp\left(-\frac{(x - \bar{x}_1)^2}{2\sigma_1^2}\right) \exp\left(-\frac{(x - \bar{x}_2)^2}{2\sigma_2^2}\right) = \exp\left(-\frac{(x - \bar{x}_1)^2}{2\sigma_1^2} - \frac{(x - \bar{x}_2)^2}{2\sigma_2^2}\right)$$

а также при условии, что распределение Гаусса имеет максимум в среднем значение, можно найти это среднее значение просто вычислив производную $p(x)$ по x .

Производная функции в точке максимума равна 0:

$$\frac{dp_{12}}{dx}\Bigg|_{\bar{x}_{12}} = -\left[\frac{\bar{x}_{12} - \bar{x}_1}{\sigma_1^2} + \frac{\bar{x}_{12} - \bar{x}_2}{\sigma_2^2}\right] \cdot p_{12}(\bar{x}_{12}) = 0$$

Т.к. функция распределения вероятности $p(x)$ никогда не равна 0, то выражение в скобках должно быть равно 0. Решение данного уравнения для x дает очень важное соотношение:

$$\bar{x}_{12} = \left(\frac{\sigma_2^2}{\sigma_1^2 + \sigma_2^2} \right) x_1 + \left(\frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2} \right) x_2$$

Это новое среднее значение \bar{x}_{12} является просто взвешенной суммой двух измерений средних, где взвешивание определяется относительной неопределенности двух измерений. При этом, например, если неопределенность σ_2 второго измерения особенно велика, то новое среднее по существу будет таким же, как среднее x_1 для ранее определенного измерения.

После подстановления нового среднего значения \bar{x}_{12} в выражение $p_{12}(x)$ и существенных преобразований неопределенность σ_{12}^2 можно определить следующим образом:

$$\sigma_{12}^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

Но что означает полученная величина? На самом деле многое. А именно: при выполнении нового измерения с новыми значениями среднего и неопределенности, можно объединить эти значения с уже имеющимися значениями среднего и неопределенности для получения нового состояния, которое характеризуется ещё более новыми средним значением и неопределенностью.

Свойство, что два Гауссова измерения в сочетании эквивалентны одному Гауссову измерению (с вычисленными средним и неопределенностью), является наиболее важным. Это означает, что при наличии измерения M можно объединить первые два измерения, третье объединить с объединением первых двух, четвертое объединить с объединением первых трех и т.д. Это именно то, что происходит при сложении в компьютерном зрении; после получения одной меры следует получение следующей, а за ней следующей и т.д.

Приняв измерения (x_i, σ_i) за временные шаги, можно вычислить текущее состояние оценки $(\hat{x}_i, \hat{\sigma}_i)$ следующим образом. На первом шаге имеется только одно измерение $\hat{x}_1 = x_1$ и его неопределенность $\hat{\sigma}_1^2 = \sigma_1^2$. Подставляя эти значения в оптимальное уравнение оценки получается следующее уравнение:

$$\hat{x}_2 = \frac{\sigma_2^2}{\hat{\sigma}_1^2 + \sigma_2^2} x_1 + \frac{\sigma_1^2}{\hat{\sigma}_1^2 + \sigma_2^2} x_2$$

Преобразование данной формулы дает следующую полезную формулу:

$$\hat{x}_2 = \hat{x}_1 + \frac{\hat{\sigma}_1^2}{\hat{\sigma}_1^2 + \sigma_2^2} (x_2 - \hat{x}_1)$$

Прежде, чем беспокоиться о полезности этой формулы, необходимо также вычислить аналогичное уравнение для $\hat{\sigma}_2^2$. Во первых после подстановки $\hat{\sigma}_1^2 = \sigma_1^2$ получается следующая формула:

$$\hat{\sigma}_2^2 = \frac{\sigma_2^2 \hat{\sigma}_1^2}{\hat{\sigma}_1^2 + \sigma_2^2}$$

В результате преобразований, схожих с теми, что были сделаны для \hat{x}_2 , получается итерационное уравнение для оценки дисперсии нового измерения:

$$\hat{\sigma}_2^2 = \left(1 - \frac{\hat{\sigma}_1^2}{\hat{\sigma}_1^2 + \sigma_2^2}\right) \hat{\sigma}_1^2$$

В таком виде данные уравнения позволяют точно отделить "старую" информацию (то, что было известно ранее как новое измерение) от "новой" информации (последнее измерение). Новая информация $(x_2 - \hat{x}_1)$, получаемая на втором шаге, именуется новейшей. Коэффициент оптимального итерационного обновления выглядит следующим образом:

$$K = \frac{\hat{\sigma}_1^2}{\hat{\sigma}_1^2 + \sigma_2^2}$$

Этот коэффициент более известен как коэффициент *обновления прироста*. После применения данного коэффициента, ранее рассмотренные формулы преобразуются в удобные рекурсивные формулы:

$$\hat{x}_2 = \hat{x}_1 + K(x_2 - \hat{x}_1)$$

$$\hat{\sigma}_2^2 = (1 - K)\hat{\sigma}_1^2$$

В литературе, когда речь заходит о фильтре Kalman в рамках генеральной выборки измерение второго шага обозначается K , а первого $k - 1$.

Динамика системы

В случае с одномерным примером был рассмотрен объект, расположенный в некоторой точке x , с рядом последовательных измерений этой точки. В данном случае не было рассмотрено перемещение объекта между измерениями. Данный случай именуется *фазой прогнозирования*. Во время фазы прогнозирования используются ранее полученные знания для предсказания того, где будет находиться система прежде чем будет интегрировано новое измерение.

На практике фаза прогнозирования выполняется непосредственно после получения нового измерения, но перед тем, как новое измерение будет включено в систему оценки состояния. Например, имеется ситуация, когда измеряется положение автомобиля в момент времени t , а потом в момент времени $t + dt$. Если автомобиль имеет некоторую скорость v , то второе измерение не так просто добавить в ряд измерений. Для начала необходимо отмотать модель на состояние со знанием о том, что было известно на момент времени t таким образом, чтобы имелось состояние системы не только на момент времени t , но и на момент $t + dt$, за мгновение до включения новой информации. Таким образом, новую информацию, полученную в момент времени $t + dt$, сливают не со старой моделью системы, а со старой моделью системы за пределами момента времени $t + dt$. Этот процесс изображен в виде цикла на рисунке 10-18. В контексте фильтра Kalman существует три вида движения, которые будут рассмотрены далее.

Первый вид движения - динамичное движение. Это движение соотносится с состоянием системы на последний момент времени. Если измерить систему с положением x , скоростью v и моментом времени t , то система будет находиться в момент времени $t + dt$ в положении $x + v * dt$, возможно с той же скоростью.

Второй вид движения - контролируемое движение. Это движение проявляется в результате некоторого внешнего влияния на систему по какой-то известной причине. Как следует из названия, наиболее популярный пример такого движения является оценочная функция состояния системы, над которой имеется контроль и известна

причина приведения данной системы в движение. Наиболее практический пример использования данного движения проявляется в робототехнике, когда, например, необходимо придать ускорение или направить вперед робот. Очевидно, что в случае робота с положением x , скоростью v и моментом времени t , в момент времени $t + dt$ он минует положение $x + v \cdot dt^*$ на чуть-чуть подальше, т.к. роботу было придано ускорение.

Третий вид движения - *случайное движение*. Даже в простом одномерном случае возникают самопроизвольные по какой-то причине движения, которые необходимо включать в этап прогнозирования. Воздействие такого случайного движения сводиться к увеличению дисперсионной оценки состояния с течением времени. Случайное движение включает в себя любые не известные или подконтрольные движения. Как и все остальное в рамках фильтра Kalman, существует предположение, что случайное движение — это либо гауссиана (т.е. своего рода случайные блуждания), либо что-то столь же эффективное, как гауссиана.

Таким образом для добавления динамики к имитационной модели, необходимо вначале выполнить этап "обновление" до включения нового измерения. Этот этап включает в себя в первую очередь применение каких-либо знаний о движении объекта в соответствии с его предыдущим состоянием при действовании любой дополнительной информации, полученной в результате собственных действий или действий, совершенных в другой системе, а также представление о случайных событиях, которые могут изменить состояние системы во время последнего измерения. После применения всех этих факторов можно совершить добавление нового измерения.

На практике динамичное движение наиболее важно, когда "состояние" системы сложнее имитационной модели. Зачастую, при движении объекта, "состояние" состоит из нескольких компонентов: положения и скорости. В этом случае, состояние развивается в соответствии со скоростью. Процесс обработки многокомпонентной системы будет обсуждаться в следующем разделе.

Уравнения Kalman

Теперь настало время обобщить все эти уравнения движения. Более обобщенное обсуждение покажет, как управлять любой моделью, которая является линейной функцией F состояния объекта. Такая модель может содержать сочетание первых и вторых производных, например, предыдущего движения. Также будет показано, как получить разрешение на управление входом u_k модели. И в заключение будут показаны более реалистичные наблюдения за моделью z , в которой можно измерить лишь некоторые из переменных состояния модели и в которой измерения могут быть лишь косвенно связаны с переменными состояния.

Для начала необходимо рассмотреть, какое влияние K (коэффициент прироста из предыдущего раздела) оказывает на оценку. Если неопределенность нового измерения очень большая, то новое измерение по существу не оказывает влияния, а уравнения сводятся к обобщенному результату, точно к такому же, как в момент времени $k - 1$. С другой стороны, если начальное измерение имеет большую дисперсию, а последующее новое измерение будет более точным, то "доверие" будет вызывать более новое измерение. Когда оба измерения имеют равную уверенность (дисперсию), новое ожидаемое значение будет находиться между ними.

На рисунке 10-19 показано, как развивается неопределенность с течением времени при сборе новых наблюдений.

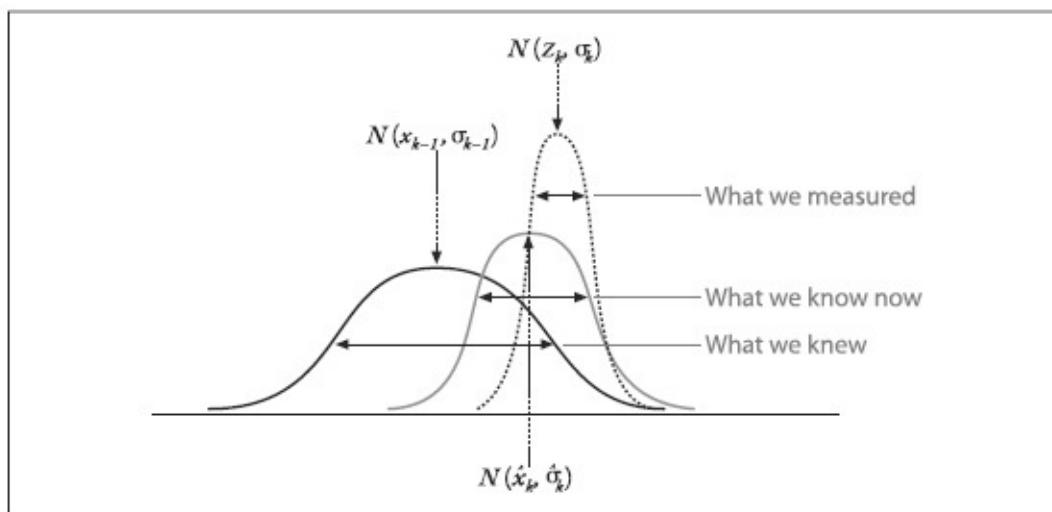


Рисунок 10-19. Сочетание предыдущих знаний $N(x_{k-1}, \sigma_{k-1})$ с текущим измерением $N(z_k, \sigma_k)$; как результат – новая оценка $N(\hat{x}_k, \hat{\sigma}_k)$

Идея обновления чувствительна к неопределенности и может быть обобщена на основе множества переменных состояния. Простейшим примером этого может быть процесс отслеживания на видео объекта, передвигающегося в двух или трех мерном измерении. В общем, состояние может содержать дополнительные элементы, такие как скорость отслеживаемого объекта. Обобщенным описанием состояния в момент времени k будет следующая функция состояния в момент времени $k - 1$:

$$x_k = Fx_{k-1} + Bu_k + w_k$$

где x_k n -мерный вектор состояния компонентов, F – это матрица $n \times n$, которую иногда называют *матрицей переноса*. u_k – это новый вектор. За счет него осуществляется внешний контроль над системой и состоит он из c -мерного вектора *управления входами*; B – это $n \times c$ матрица, которая связывает эти управляемые входы с переменными состояния. (Внимательный читатель или тот, кто уже что-то знает о фильтрах Kalman, может отметить еще одно важное допущение: существует линейная зависимость (за счет перемножения матриц) между управлениями u_k и изменениями

сстояния. На практике, данное предположение в первую очередь может сломать приложение.) Переменная w_k это случайная величина (именуемая *шумом обработки*), ассоциированная со случайными событиями или силами, которые непосредственно влияют на реальное состояние системы. Предполагается, что компонент w_k имеет гауссово распределение $N(0, Q_k)$ для некой ковариационной матрицы Q_k размера $n \times n$ (со временем Q можно изменять, однако, не рекомендуется делать это слишком часто).

В общем, выполняемые измерения z_k могут быть, а могут и не быть, непосредственными измерениями переменной состояния x_k . Обобщить данную ситуацию можно за счет ввода m -мерного вектора измерения z_k :

$$z_k = H_k x_k + v_k$$

где H_k – это матрица $m \times n$, v_k – это величина погрешности, которая является гауссовым распределением $N(0, R_k)$ для некой ковариационной матрицы R_k размера $m \times m$.

Теперь давайте рассмотрим конкретную реальную ситуацию с автомобилем на парковке. Состояние автомобиля можно свести к двум переменным положения x и y и скорости v_x и v_y . Эти четыре переменные составляют вектор состояния x_k . Это означает, что правильная форма F следующая:

$$x_k = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}_k, \quad F = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Тем не менее, при использовании камеры для выполнения измерения состояния автомобиля, скорее всего, достаточно измерить только переменные положения:

$$z_k = \begin{bmatrix} z_x \\ z_y \end{bmatrix}_k$$

Это означает, что структура H выглядит следующим образом:

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

В этом случае точно не возможно поверить в то, что скорость автомобиля постоянна, именно поэтому и происходит присвоение Q_k , чтобы отразить это. Выбор падает на R_k , основываясь на оценке того, насколько точно было определено положение автомобиля при помощи (например) методов анализа изображения в видео потоке.

Теперь осталось только разместить полученные выражения в обобщенных уравнениях обновления. Основная идея заключается в следующем. Во-первых, необходимо вычислить априорную оценку состояния \bar{x}_k . В литературе это относительно распространенный (но не универсальный) шаг, использование минуса в виде верхнего индекса сообщает, что вычисляемая оценка производится "непосредственно во время перед получением нового измерения". Данная априорная оценка определяется следующим образом:

$$\bar{x}_k = Fx_{k-1} + Bu_{k-1} + w_k$$

P_k^- используется для обозначения ковариационной ошибки, априорная оценка которой в момент времени k достигается за счет значения в момент времени $k - 1$:

$$P_k^- = FP_{k-1}F^T + Q_{k-1}$$

Это уравнение формирует основу по части прогноза оценки, а также сообщает "чего ожидать", основываясь на том, что уже было увидено. В следствии этого такое состояние (без дифференцирования) зачастую именуют *усилением Kalman* или *коэффициентом смещивания*, который сообщает о том, какой вес у новой информации относительно того, что уже известно:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$$

Хотя это уравнение и выглядит устрашающее, в действительности не все уж так и плохо. Данное уравнение можно с легкостью понять, рассмотрев различные простые случаи. В случае одномерного примера, в котором измерялась непосредственно одна переменная позиции, H_k - это матрица 1×1 , содержащая только 1! Таким образом, если ошибка измерения σ_{k+1}^2 , то R_k также матрица 1×1 , содержащая это значение. Точно также P_k просто дисперсия σ_k^2 . Так что большое уравнение сводится к следующему:

$$K = \frac{\sigma_k^2}{\sigma_k^2 + \sigma_{k+1}^2}$$

Прирост, впервые показанный в предыдущем разделе, позволяет оптимально рассчитать значение обновления для x_k и P_k на момент доступности нового измерения:

$$x_k = \hat{x}_k^- + K_k(z_k^- - H_k \hat{x}_k^-)$$

$$P_k = (I - K_k H_k) P_k^-$$

Собственно, что и требовалось доказать: на первый взгляд устрашающее уравнение оказалось при обсуждении простого одномерного случая не таким уж и устрашающим! Оптимальные веса и приrostы получаются тем же методом, что и в одномерном случае, только необходимо минимизировать неопределенность положения состояния x , установив в 0 частные производные по x перед решением.

Можно показать связь с простым одномерным случаем за счет установки $F = I$ (где I единичная матрица), $B = 0$ и $Q = 0$. Сходство с одномерным фильтром дифференцирования проявляется за счет выполнения следующих замен (в общих уравнениях): $\hat{x}_k \leftarrow \hat{x}_2$, $\hat{x}_k^- \leftarrow \hat{x}_1$, $K_k \leftarrow K$, $z_k \leftarrow x_2$, $H_k \leftarrow 1$, $P_k \leftarrow \hat{\sigma}_2^2$, $I \leftarrow 1$, $P_k^- \leftarrow \hat{\sigma}_1^2$ и $R_k \leftarrow \hat{\sigma}_2^2$.

OpenCV и фильтр Kalman

На основе всего ранее представленного может сложиться мнение о том, что для выполнения подобного рода задач OpenCV и не нужно или, наоборот, без OpenCV будет невозможно справиться. К счастью, OpenCV поддается любой интерпретации. OpenCV предоставляет четыре функции, которые непосредственно связаны с работой фильтра Kalman:

```
cvCreateKalman(
    int      nDynamParams
, int      nMeasureParams
, int      nControlParams
);

cvReleaseKalman(
    CvKalman** kalman
);
```

Первая функция создает и возвращает указатель на структуру типа *CvKalman*, вторая функция удаляет эту структуру.

```

typedef struct CvKalman {
    int MP; // размерность вектора измерений
    int DP; // размерность вектора состояния
    int CP; // размерность вектора управления
    CvMat* state_pre; // экстраполяция (предсказание) вектора состояния:
                      //  $x_k = F x_{k-1} + B u_k$ 
    CvMat* state_post; // корреляция вектора состояния:
                      //  $x_k = x'_k + K_k (z_k' - H x'_k)$ 
    CvMat* transition_matrix; // стохастической матрица состояния
                              // F
    CvMat* control_matrix; // матрица управления
                            // B
                            // (не использовать, если нет управления)
    CvMat* measurement_matrix; // матрица наблюдений
                                // H
    CvMat* process_noise_cov; // ковариационная матрица случайного процесса
                              // Q
    CvMat* measurement_noise_cov; // ковариационная матрица шума измерений
                                  // R
    CvMat* error_cov_pre; // ковариационная матрица экстраполированного
                          // вектора состояния:
                          //  $(P_{k'} = F P_{k-1} F^T) + Q$ 
    CvMat* gain; // Оптимальная по Kalman матрица коэффициентов
                  // усиления:
                  //  $K_k = P_{k'} H^T (H P_{k'} H^T + R)^{-1}$ 
    CvMat* error_cov_post; // ковариационная матрица оценки вектора
                           // состояния системы
                           //  $P_k = (I - K_k H) P_{k'}$ 
    CvMat* temp1; // временные матрицы
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
} CvKalman;

```

Следующие две функции реализуют сам фильтр Kalman. После заполнения структуры данными, можно вычислить вектор состояния на следующем временном шаге за счет вызова `cvKalmanCorrect()` с последующей интеграцией новых измерений за счет вызова `cvKalmanPredict()`. Результат вызова `cvKalmanCorrect()` размещается в `state_post`, а результат вызова `cvKalmanPredict()` в `state_pre`.

```

cvKalmanPredict(
    CvKalman*      kalman
    ,const CvMat*   control = NULL
);

cvKalmanCorrect(
    CvKalman*      kalman
    ,CvMat*        measured
);

```

Пример использования фильтра Kalman

Например, имеется автомобиль, движущийся по кругу гоночной трассы. Автомобиль движется в основном с постоянной скоростью, хотя присутствуют и некоторые отклонения (т.е. случайные процессы). Для измерения положения автомобиля будут задействованы алгоритмы компьютерного зрения. Это порождает некоторые (несвязанные) шумы (т.е. измерения шума).

Таким образом, модель довольно таки проста: автомобиль имеет положение и угловую скорость в любой момент времени. Вместе эти два коэффициента формируют двумерный вектор состояния x_k . Измерение только положения автомобиля формируют одномерный "вектор" z_k .

Далее представленная программа (Пример 10-2) рассматривает перемещение автомобиля, движущегося по кругу (красный), а также отображает сделанные измерения (желтый) и положения, предсказанные фильтром Kalman (белый).

Программа начинается со строк подключения заголовочных файлов библиотеки. Далее определяется макрос, который будет полезен в момент преобразования угловых координат положения автомобиля в декартовые для отображения.

Пример 10-2. Пример использования фильтра Kalman

```

// Использование фильтра Kalman для моделирования частицы с круговой траекторией
//
#include "cv.h"
#include "highgui.h"
#include "cvx_defs.h"

#define phi2xy(mat)                                /
cvPoint( cvRound(img->width/2 + img->width/3*cos(mat->data.f1[0])), /
cvRound( img->height/2 - img->width/3*sin(mat->data.f1[0])) )

int main(int argc, char** argv) {
    // Инициализация, создание объекта фильтр Kalman, окна,
    // генератора случайных чисел и т.д.
    //
    cvNamedWindow( "Kalman", 1 );
}

... продолжение далее

```

Далее создается генератор случайных чисел, изображение для рисования и структура фильтра Kalman. Стоит обратить внимание на то, что фильтру Kalman необходимо сообщить размерность переменных состояния (2) и переменных измерения (1).

```

... продолжение ранее

CvRandState rng;
cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
CvKalman* kalman = cvCreateKalman( 2, 1, 0 );

... продолжение далее
`
```

Далее создаются матрица (на самом деле вектор, однако в OpenCV все именуется матрицами) состояния x_k , матрица нормального случайного процесса w_k , измерение z_k и наиболее важная стохастическая матрица состояния F . Матрица состояния инициализируется случайными числами близкими к нулю.

Стochasticная матрица состояния очень важна, т.к. она связывает состояние системы в момент времени k с состоянием в момент времени $k+1$. В представленном случае данная матрица имеет размерность 2×2 (т.к. вектор состояния является двумерным). x_k представляет угловое положение (ϕ) и угловую скорость (ω) автомобиля. В этом случае матрица состояния содержит $\begin{bmatrix} 1, dt \\ 0, 1 \end{bmatrix}$. Следовательно, после умножения на F состояние (ϕ, ω) меняется на $(\phi + \omega \times dt, \omega)$, то есть угловая скорость остается неизменной, а угловое положение увеличивается на величину равную перемножению

угловой скорости и временного шага. В данном примере для удобства используется $dt = 1.0$, но на практике необходимо использовать что-то типа времени между последовательными видеокадрами.

```
... продолжение ранее

// состояние (phi, delta_phi) - угол и угловая скорость,
// инициализация случайными значениями
//
CvMat* x_k = cvCreateMat( 2, 1, CV_32FC1 );
cvRandSetRange( &rng, 0, 0.1, 0 );
rng.disttype = CV_RAND_NORMAL;
cvRand( &rng, x_k );

// матрица нормального случайного процесса
//
CvMat* w_k = cvCreateMat( 2, 1, CV_32FC1 );

// измерения, только один параметр для угла
//
CvMat* z_k = cvCreateMat( 1, 1, CV_32FC1 );
cvZero( z_k );

// стохастическая матрица состояния 'F' связывает параметры модели
// в момент времени k и в момент времени k + 1
//
const float F[] = { 1, 1, 0, 1 };
memcpy( kalman->transition_matrix->data.f1, F, sizeof(F) );

... продолжение далее
```

Помимо этого, фильтр Kalman имеет и другие внутренние параметры, которые также должны быть инициализированы. В частности, матрица наблюдений H размера 1×2 должна быть инициализирована $[1, 0]$. Ковариационная матрица случайного процесса и ковариационная матрица шума измерений должны быть установлены в разумно интересные значения (тут можно поэкспериментировать), ковариационная матрица экстраполированного вектора состояния должна быть инициализирована единичной матрицей (это гарантирует значимость первой итерации; в последствии это значение будет перезаписано).

Таким же образом инициализируется корреляционный вектор состояния (гипотетически на шаге, предшествующий первому!) случайным значением, т.к. на этот момент времени информация отсутствует.

```

... продолжение ранее

// инициализация других параметров фильтра Kalman
//
cvSetIdentity( kalman->measurement_matrix,      cvRealScalar(1) );
cvSetIdentity( kalman->process_noise_cov,        cvRealScalar(1e-5) );
cvSetIdentity( kalman->measurement_noise_cov,    cvRealScalar(1e-1) );
cvSetIdentity( kalman->error_cov_post,           cvRealScalar(1) );

// случайный выбор начального состояния
//
cvRand( &rng, kalman->state_post );
while( 1 ) {

... продолжение далее

```

Теперь все готово для проверки работы динамики фильтра. В начале, необходимо отправить запрос к фильтру для предсказания и обдумывания того, что будет получено на этом шаге (т.е. перед отдачей какой-либо новой информации); все это обозначено y_k . Затем происходит генерация нового значения z_k (измерения) для этой итерации. По определению это значение является "реальным" значением x_k помноженное на матрицу наблюдений H с добавлением случайного измерения шума. Стоит отметить, что только в этом игрушечном приложении генерируется z_k от x_k ; в действительности, генерируемое значение будет браться в зависимости от окружающей среды или датчиков. В рассматриваемом примере значение генерируется на основе "реальной" модели данных с добавлением случайного шума; в результате можно будет увидеть эффект от использования фильтра Kalman.

```

... продолжение ранее

// предсказанное положение точки
//
const CvMat* y_k = cvKalmanPredict( kalman, 0 );

// генерация измерения (z_k)
//
cvRandSetRange(
    &rng
    ,0
    ,sqrt(kalman->measurement_noise_cov->data.fl[0])
    ,0
);
cvRand( &rng, z_k );
cvMatMulAdd( kalman->measurement_matrix, x_k, z_k, z_k );

... продолжение далее

```

Далее рисуются три точки, соответствующие наблюдениям, синтезированные ранее и расположенные согласно предсказаниям фильтра Kalman и основному состоянию (которое случайным образом становится известно в момент моделирования).

```
... продолжение ранее

// рисование точек (eg convert to planar coordinates and draw)
//
cvZero( img );
cvCircle( img, phi2xy(z_k), 4, CVX_YELLOW );      // наблюдаемое состояние
cvCircle( img, phi2xy(y_k), 4, CVX_WHITE, 2 );    // "предсказанное" состояние
cvCircle( img, phi2xy(x_k), 4, CVX_RED );         // реальное состояние
cvShowImage( "Kalman", img );

... продолжение далее
```

Теперь можно перейти к обработке следующей итерации. Для начала необходимо вновь вызвать фильтр Kalman и информировать его о новейших изменениях. Далее генерируется матрица случайного процесса. Затем задействуется стохастическая матрица состояния F , чтобы передвинуть x_k на один временной шаг вперед с последующим добавлением сгенерированной матрицы случайного процесса; теперь все готово для совершения очередного круга.

```

... продолжение ранее

    // регулирование состояния фильтра Kalman
    //
    cvKalmanCorrect( kalman, z_k );

    // применение стохастической матрицы состояния 'F' и
    // матрицы случайного процесса w_k .
    //
    cvRandSetRange(
        &rng
        , 0
        , sqrt( kalman->process_noise_cov->data.fl[0] )
        , 0
    );
    cvRand( &rng, w_k );
    cvMatMulAdd( kalman->transition_matrix, x_k, w_k, x_k );

    // выход, если нажата клавиша 'Esc'
    if( cvWaitKey( 100 ) == 27 ) {
        break;
    }
}

return 0;
}

```

Собственно, как оказалось, реализовать фильтр Kalman не так уж и сложно; половина написанного кода служит для создания некоторой дополнительной информации. В любом случае, в заключении необходимо все обобщить, чтобы быть уверенными в том, что все это имеет смысл.

Все начинается с создания матриц для представления состояния системы и измерений, которые будут сделаны. Далее определяется стохастическая матрица состояния и ковариационная матрица шума измерений, а затем инициализируется ковариационная матрица случайного процесса и другие параметры фильтра.

После инициализации вектора состояния случайными значениями, вызывается фильтр Kalman для получения первого предсказания. После того, как предсказание (которое не очень важно на первой итерации) будет прочитано, оно выводиться на экран. Параллельно синтезируется новое наблюдение и тоже выводиться на экран для сравнения с предсказанным значением. Далее фильтр получает новую информацию в виде нового измерения, которое интегрируется во внутреннюю модель. И в заключении, синтезируется новое "реальное" состояние модели и цикл начинается по новой.

Запустив код, можно увидеть маленький красный шар, вращающийся по кругу. Маленький желтый шар, представляющий шум, через который фильтр Kalman "пытается смотреть", время от времени появляется и исчезает около красного шара. Белый шар, быстро сходящийся в небольшом пространстве вокруг красного шара, показывает, что фильтр Kalman дает обоснованную оценку движения частицы (машины) в рамках модели.

Единственное, что не было рассмотрено в данном примере, так это использование управляемых входов. Например, если бы был рассмотрен радиоуправляемый автомобиль и имелись некоторые знания о том, каким образом человек управляет данным автомобилем, то можно было бы включить данную информацию в модель. В этом случае скоростью можно было бы управлять. Помимо ранее перечисленных параметров, потребовалось бы задать матрицу B (*kalman->control_matrix*), а также позаботиться о наличии второго аргумента функции *cvKalmanPredict()* - векторе управления u .

Немного о расширенном фильтре Kalman

Динамика линейной системы, состоящая только из основных параметров, довольно-таки ограничена. Однако, как будет показано далее, при рассмотрении динамики нелинейной системы, фильтр Kalman будет полезен.

Термин "линейный" означает, что отдельно взятые шаги в определении фильтра Kalman могут быть представлены матрицами. Но когда эти шаги не могут быть представлены матрицами? На самом деле для этого есть множество возможностей. Например, мера управления является суммой, а педаль газа машины является дисперсией: тогда связь между скоростью автомобиля и педалью газа не является линейной. Другой более распространенной проблемой является то, что величина силы автомобиля более естественно выражается в декартовых координатах, в то время как движение автомобиля более естественно выражается в полярных координатах. Данная проблема могла бы возникнуть, если бы вместо машины была бы лодка, движущаяся по кругу в строго определенном направлении в едином потоке воды.

Во всех этих случаях, использование только фильтра Kalman будет недостаточно. Одним из способов справиться с этими нелинейностями (или по крайней мере попытка их обработать) заключается в линеаризации соответствующих процессов (например, обновление F или управление входным откликом B). Таким образом, потребуется вычислять новые значения F и B на каждом временном шаге на основе состояния x . Эти значения лишь приблизительно реальные обновления, а функции управления работают в непосредственной близости от конкретного значения x ; на практике этого зачастую бывает вполне достаточно. Данные дополнения формируют *расширенный фильтр Kalman*.

В OpenCV нет специально зарезервированных функций под расширенный фильтр Kalman, но они в действительности и не нужны. Все что необходимо сделать, так это повторно вычислять и сбрасывать значения *kalman->update_matrix* и *kalman->control_matrix* после каждого обновления. Благодаря этому элегантному расширению фильтра Kalman, которое именуется *сигма-точечным фильтром Kalman*, появляется возможность охватить нелинейные системы.

[П]|[РС]|(РП) Алгоритм условно плотного распространения

Модели фильтра Kalman могут иметь только одно предположение. Из-за того, что базовая модель распределения вероятности является унимодальной гауссианой, нельзя представить несколько предположений одновременно при использовании фильтра Kalman. Несколько более расширенный метод, известный как *алгоритм condensation (Conditional Density Propagation* - условно плотное распространение), основанный на широком классе оценок именуемых *фильтром particles*, позволит преодолеть данную проблему.

Для осознания назначения алгоритма условно плотного распространения, необходимо рассмотреть предположение, что объект движется с постоянной скоростью. Любые измеренные данные, по существу, должны быть интегрированы в модель, как если бы эта модель поддерживала данное предположение. Теперь пусть имеется объект, двигающийся по траектории с препятствиями. Для этого случая не известно, что объект делает; он может двигаться с постоянной скоростью, может остановиться и/или двигаться в противоположном направлении. Фильтр Kalman не может быть представлен этими несколькими вероятностями, просто расширив неопределенность, связанную с (гауссиана) распределением положений объекта. Фильтр Kalmana – это обязательно гауссиана, поэтому он не может быть представлен мультимодальными распределениями.

Как и в случае с фильтром Kalman имеется две функции для создания и уничтожения структуры данных, используемой в фильтре *condensation*. Разница между этими функциями заключается в том, что в случае функции создания *cvCreateConDensation()* имеется дополнительный параметр. Введенное значение для этого параметра задает число предположений, которые фильтр будет поддерживать в любой момент времени. Это число должно быть достаточно большим (50 или 100; возможно и больше для более сложных случаев), т.к. набор отдельных предположений занимает место в параметризованном распределении вероятности Гаусса фильтра Kalman (Рисунок 10-20).

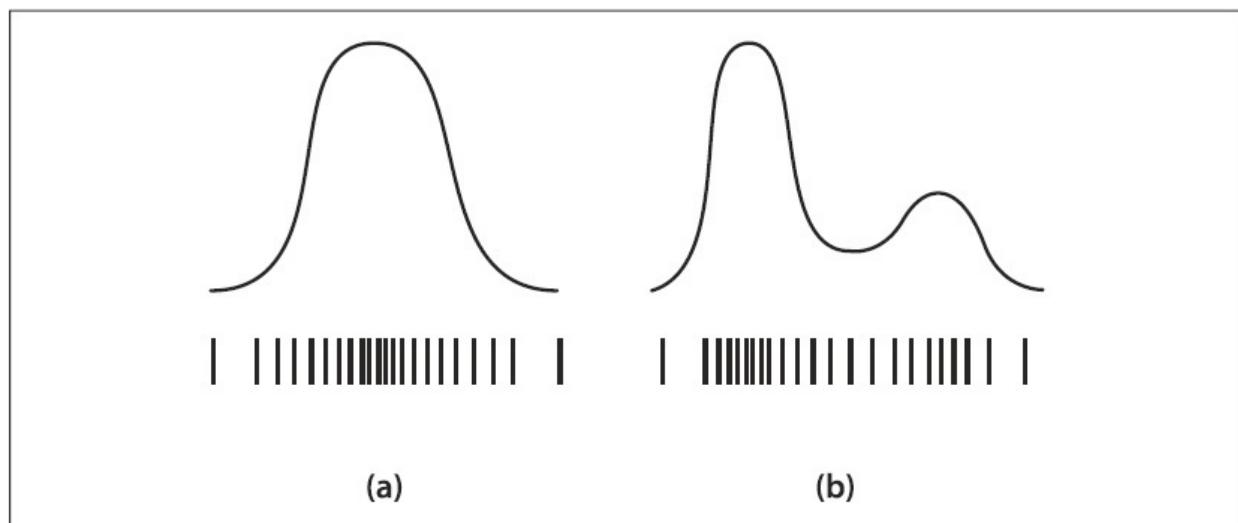


Рисунок 10-20. Распределение, которое может (а) и не может (б) быть представлено в виде непрерывного параметризованного распределения Гаусса со значениями среднего и неопределенности; оба распределения в качестве альтернативы могут быть представлены набором *particles*, плотность которых приблизительно соответствует представленному распределению

```
CvConDensation* cvCreateConDensation(  
    int      dynam_params  
    , int    measure_params  
    , int    sample_count  
);  
  
void cvReleaseConDensation(  
    CvConDensation** condens  
);
```

Структура данных *CvConDensation* имеет следующие внутренние элементы:

```
typedef struct CvConDensation {  
    int          MP;           // Размерность вектора измерений  
    int          DP;           // Размерность вектора состояния  
    float*       DynamMatr;    // Матрица линейной динамики системы  
    float*       State;        // Вектор состояния  
    int          SamplesNum;   // Число образцов  
    float**     flSamples;    // массив векторов-образцов  
    float**     flNewSamples; // временный массив векторов-образцов  
    float*       flConfidence; // Confidence для каждого образца  
    float*       flCumulative; // Cumulative confidence  
    float*       Temp;         // Временный вектор  
    float*       RandomSample; // RandomVector для обновления образца  
    CvRandState* RandS;      // массив структур, генерируемый случайными векторами  
} CvConDensation;
```

После выделения памяти под структуру данных фильтра *condensation* необходимо инициализировать эту структуру. Это делается за счет использования функции *cvConDensInitSampleSet()*. При создании структуры *CvConDensation* указывается количество *particles*, плюс для каждого *particle* размерность. Инициализация всех этих *particle* может быть довольно таки хлопотным занятием. К счастью для всего этого есть функция *cvConDensInitSampleSet()*; нужно только указать диапазоны для каждой размерности.

```
void cvConDensInitSampleSet(
    CvConDensation*    condens
    , CvMat*            lower_bound
    , CvMat*            upper_bound
);
```

Данная функция запрашивает две ранее проинициализированные структуры *CvMat*. Обе переменные являются векторами (т.е. матрица с одним столбцом) и имеют записи, количество которых равно размерности системы состояния. Эти вектора используются для установки диапазонов, которые в последующем будут задействованы при инициализации векторов образцов в структуре *CvConDensation*.

Ниже представленный код создает две матрицы размером *Dim* и инициализирует их значениями -1 и +1, соответственно. При вызове *cvConDensInitSampleSet()* первоначальные образцы инициализируются случайными числами, каждое из которых находится в пределах от -1 до +1. Таким образом, если *Dim* = 3, то фильтр будет проинициализирован *particles*, равномерно распределенными внутри куба с центром в нуле и со сторонами длиною 2.

```
CvMat LB = cvMat(Dim, 1, CV_MAT32F, NULL);
CvMat UB = cvMat(Dim, 1, CV_MAT32F, NULL);
cvAlloc(&LB);
cvAlloc(&UB);
ConDens = cvCreateConDensation(Dim, Dim, SamplesNum);

for( int i = 0; i < Dim; i++ ) {
    LB.data.fl[i] = -1.0f;
    UB.data.fl[i] = 1.0f;
}

cvConDensInitSampleSet(ConDens, &LB, &UB);
```

И в заключении следующая функция позволяет обновить состояние фильтра *condensation*:

```
void cvConDensUpdateByTime( CvConDensation* condens );
```

Эта функция делает немного больше, чем может показаться на первый взгляд. В частности необходимо обновлять *confidences* всех *particles* в свете того, что новая информация становится доступной после предыдущего обновления. К сожалению, в OpenCV удобной функции для этого не предусмотрено. Причина в том, что отношение между новым *confidence* для *particle* и новой информацией зависит от контекста. Пример обновления, который просто применяет обновление *confidence* для каждой *particle*, представлен далее:

```
// Обновление confidences всех particles в фильтре
// на основе нового измерения M[], где M имеет размерность
// particles в фильтре
//
void CondProbDens( CvConDensation* CD, float* M ) {
    for( int i=0; i<CD->SamplesNum; i++ ) {
        float p = 1.0f;
        for( int j=0; j<CD->DP; j++ ) {
            p *= (float) exp( -0.05*(M[j] - CD->flSamples[i][j])*(M[j] - CD->flSamples[i]
        }
        CD->flConfidence[i] = Prob;
    }
}
```

После обновления *confidences* можно вызывать *cvCondensUpdateByTime()* для обновления *particles*. Термин "обновление" означает *передискретизацию*, которая должна сообщить, что новый набор *particles* будет сгенерирован в соответствии с рассчитанными *confidences*. После обновления все *confidences* вновь будут установлены в *1.0f*, однако, распределение *particles* теперь будет включать ранее модифицированные *confidences* непосредственно в плотности *particles* на следующей итерации.

[П]|[РС]|(РП) Упражнения

Примеры реализации многих обсуждаемых в данной главе алгоритмов можно найти в `.../opencv/samples/c/`:

- `lkdemo.c` (оптический поток)
 - `camshiftdemo.c` (mean-shift отслеживание цветных регионов)
 - `motempl.c` (шаблоны движения)
 - `kalman.c` (фильтр Kalman)
1. Используйте матрицу ковариации *Hessian* в функции `cvGoodFeaturesToTrack()` для выполнения вычислений по некоторой квадратной области, заданной при помощи *block_size*, изображения.
 - a. Что будет происходить при увеличении размера блока? Будем получать больше или меньше "хороших особенностей"? Почему?
 - b. Покопайтесь в коде `lkdemo.c`, найдите `cvGoodFeaturesToTrack()` и поиграйтесь с *block_size*, чтобы увидеть разницу.
 2. Обратитесь к рисунку 10-2 и функции, которая реализует поиск угла субпикселя `cvFindCornerSubPix()`.
 - a. Что произойдет, если (рисунок 10-2) шахматная доска будет сплетена таким образом, что прямые черные-белые линии сформируют кривые, которые пересекаются в точке? Будет ли работать поиск угла субпикселя? Ответ поясните.
 - b. Если расширить размеры окна вокруг плетеной шахматной доски, то угол будет точкой (после расширения параметров *win* и *zero_size*), а вот после выполнения поиска угла субпикселя, угол станет более или менее точным? Ответ поясните.
 3. Оптический поток
 - a. Опишите объект, который будет лучше отслеживаться при помощи метода блочного сопоставление, а не метода *Lucas-Kanade*.
 - b. Опишите объект, который будет лучше отслеживаться при помощи метода *Lucas-Kanade*, а не метода блочного сопоставления.
 4. Скомпилируйте `lkdemo.c`. Подключите к программе камеру (или используйте ранее захваченную последовательность текстурированного движущегося объекта). Запустив программу, обратите внимание на то, что "r" автоматически

инициализирует слежение, "с" очищает слежение, а клики мыши добавляют/удаляют новую/старую точку. Запустите *lkdemo.c* и инициализируйте отслеживание точки, нажав "г". Понаблюдайте за последствиями произведенных действий.

a. Теперь откройте код и закомментируйте вызов функции *cvFindCornerSubPix()*. Навредит ли это результату? Каким образом?

b. Еще раз откройте код и передайте в функцию *cvGoodFeaturesToTrack()* сетку точек вокруг объекта при помощи ROI. Опишите что происходит с точками и почему.

Подсказка: часть происходящего является следствием проблем с диафрагмой - при фиксированном размере окна и линии, нельзя сказать, как перемещается линия.

5. Измените *lkdemo.c*, создав программу, которая выполняла бы простую стабилизацию изображения умеренно движущихся камер. Отобразите результаты стабилизации в центре наибольшего окна одной из камеры (так, чтобы кадр мог "зависнуть", пока первые точки остаются стабильными).
6. Скомпилируйте и запустите *camshiftdemo.c*, используя веб-камеру или цветное видео с передвигающимся цветным объектом. Используйте мышь, чтобы нарисовать прямоугольник вокруг движущегося объекта; соответствующая функция начнет отслеживать выбранный объект.
 - a. В *camshiftdemo.c* замените *cvCamShift()* на *cvMeanShift()*. Опишите ситуации при которых один трекер работает лучше другого.
 - b. Напишите функцию, которая бы передавала сетку точек в *cvMeanShift()*. Запустите оба трекера одновременно.
 - c. Как нужно использовать эти два трекера одновременно, чтобы отслеживание было более надежным? Ответ поясните и/или проведите эксперимент.
7. Скомпилируйте и запустите шаблон движения из кода *motempl.c* с использованием веб камеры или заранее сохраненного видеофайла.
 - a. Модифицируйте *motempl.c* так, чтобы можно было выполнить простое распознавание жестов.
 - b. При условии, что камера движется, объясните, как задействовать код из упражнения 5 для стабилизации этих движений, дав шаблону движения возможность для работы.
8. Объясните, как можно отслеживать круговые (не линейные) движения за счет использования линейного состояния модели (не расширенного) фильтра Kalman.

9. Используйте модель движения с утверждением, что текущее состояние зависит от положения и скорости предыдущего состояния. Объедините *lkdemo.c* (используя только несколько кликов-точек) с фильтром Kalman для отслеживания лучших точек *Lucas-Kanade*. Покажите неопределенность вокруг каждой точки. В каком месте отслеживание будет провальным?
10. Фильтр Kalman зависит от линейной динамики и от Марковской независимости (т.е. предполагается, что текущее состояние зависит только от недавнего прошлого состояния, а не от всех предыдущих состояний). Допустим имеется объект для отслеживания, движения которого связаны с его предыдущим положением и скоростью, при этом ошибочно добавляется элемент динамики только для состояния, зависимого от предыдущего положения - другими словами, предыдущий элемент скорости забывается.
 - a. Сохраняются ли предположения Kalman? Если да, то почему; если нет, то какие предположения нарушаются.
 - b. При каких условиях фильтр Kalman может продолжить свою работу, если были позабыты некоторое элементы динамики?
11. Используйте веб-камеру или видео, где человек размахивает двумя яркими цветными объектами, по одному в каждой руке. Используйте *condensation* для отслеживания обеих рук.

[П]|[РС]|(РП) Модели камер и их калибровка

Зрение начинается с обнаружения света окружающего мира. Свет в свою очередь начинается с лучей, исходящих от некоторого источника (например, лампочки или солнца), которые затем перемещаются сквозь пространство до момента столкновения с каким-либо предметом. После попадания света на объект, большая часть света поглощается, а оставшаяся часть воспринимается как цвет света. Отраженный свет, пробившийся к глазам (или камере) собирается сетчаткой (или фотоприёмником). Геометрия этого механизма – в частности перемещение луча от объекта, через призму глаза или камеры, к сетчатке или фотоприёмнику – имеет важное практическое значение в компьютерном зрении.

Простой, но чрезвычайно полезной моделью того, как это происходит, является модель *стенопа* (вариант камеры обскуры). Стеноп представляет собой мнимую стену с крошечным отверстием в центре, которая блокирует все лучи за исключением тех, что проходят через крошечное отверстие в центре. Данная глава начинается с рассмотрения стенопа для получения представления об основах геометрии проектирующих лучей. К сожалению, в действительности стеноп не очень хороший способ изготовления изображений, т.к. не позволяет собирать достаточное количество света. Именно поэтому глаза и камеры используют линзы для сбора большего количества света. Недостатком данного подхода является то, что сбор большего количества света за счет линзы заставляет не только выйти за рамки простой геометрии камеры обскуры, но и внести искажения от объектива.

В данной главе будет показано, как при помощи *калибровки камеры* можно исправить (математически) основные отклонения от простой модели обскуры, накладываемые от использования линз. Калибровка камеры также важна для связи измерений камеры с измерениями реального трехмерного мира. Это важно, т.к. сцены не только трехмерны; они также являются физическим пространством с физическими элементами. Таким образом, связь между естественными единицами камеры (пикселями) и элементами физического мира (например, метрами) является важной составляющей при любой попытке восстановить трехмерную сцену.

Результатом калибровки камеры является модель геометрии камеры и модель *искажения объектива*. Эти две информативные модели определяют *внутренние параметры камеры*. В данной главе эти модели будут использованы для коррекции искажений камеры; в главе 12 данные модели будут использованы для толкования физических сцен.

В начале главы речь пойдет о моделях камер и причинах искажения объектива. Затем будут исследованы *преобразования гомографии* и математические инструменты для охвата основ поведения камеры и её различных искажений и исправлений. Также будет уделено некоторое время обсуждению того, как именно преобразование, характеризующее определенную камеру, можно рассчитать математически. После рассмотрения теоретической части будут рассмотрены функции OpenCV, которые выполняют большую часть работы.

Абсолютно все в этой главе посвящено построению достаточной теории для того, чтобы осознать, что происходит внутри (и то, что получается на выходе) функции `cvCalibrateCamera2()`. Если необходимо знать лишь то, как использовать OpenCV для выполнения калибровки камеры, то можно сразу перейти к разделу "Функция калибровки".

[П]||[РС]||(РП) Модель камеры

В простой модели, модели камеры обскуры, свет представляет из себя, начинающийся от сцены или удаленного объекта, единичный луч, поступающий из какой-либо конкретной точки. С точки зрения физики, эта точка затем "проецируется" на поверхность изображения. В результате, изображение на этой *плоскости изображения* (также известной как *плоскость проекции*) всегда находится в фокусе, а размер изображения относительно удаленного объекта задается одним параметром камеры: *фокусным расстоянием*. Для идеализированной камеры обскуры расстояние от диафрагмы до экрана является фокусным расстоянием. Это продемонстрировано на рисунке 11-1, где f - фокусное расстояние камеры, Z - расстояние от камеры до объекта, X - длина объекта и x - изображение объекта на плоскости изображения. На рисунке можно отметить, согласно подобию треугольников, факт того, что $-x/f = X/Z$ или

$$-x = f \frac{X}{Z}$$

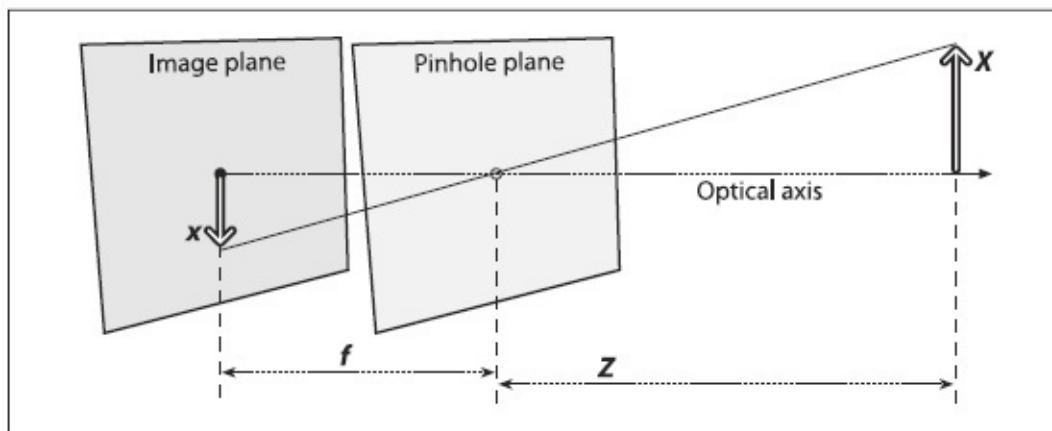


Рисунок 11-1. Модель камеры обскура: диафрагма пропускает только лишь те световые лучи, которые пересекают определенную точку в пространстве; затем эти лучи формируют изображение за счет "проекции" на плоскости изображения

Далее будет выполнено преобразование модели камеры обскура в эквивалентную модель для упрощения математических действий. Согласно рисунку 11-2 для этого необходимо произвести перестановку плоскости с отверстием и плоскости изображения. Основное отличие в результате произведенного действия заключается в том, что объект окажется на лицевой стороне. Точка на плоскости камеры обскура становится *центром проекции*. В этом случае, каждый луч оставляет точку на удаленном объекте и направляется к центру проекции. Точка пересечения плоскости изображения и оптической оси называется *главной точкой*. На этой новой фронтальной плоскости изображения (рисунок 11-2), которая эквивалентна старой

проективной плоскости или плоскости изображения, изображение удаленного объекта имеет точно такие же размеры, как и плоскость изображения на рисунке 11-1.

Изображение формируется при пересечении этих лучей с плоскостью изображения, что происходит точно на расстоянии f от центра проекции. Это делает отношение $x/f = X/Z$, полученное из подобия треугольников, более очевидным, чем раньше. Знак минус отсутствует, т.к. изображение объекта больше не перевернутое.

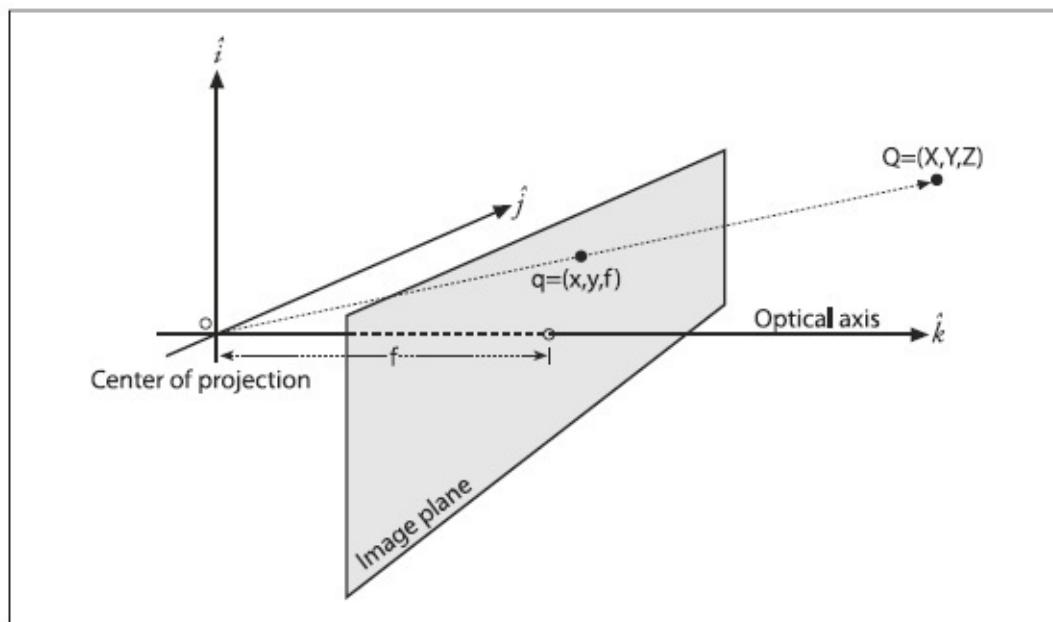


Рисунок 11-2. Точка $Q = (X, Y, Z)$ проецируется на плоскость изображения лучом, проходящим через центр проекции в результирующую точку на изображении $q = (x, y, z)$; плоскость изображения в действительности это просто проекционный экран "установленный" перед плоскостью камеры обскура

Можно подумать, что основная точка эквивалентна центру фотоприёмника, однако это будет означать, что какой-то парень с пинцетом и тюбиком клея сможет установить фотоприемник в камере с микро точностью. На самом деле, центр чипа, как правило, располагается не на оптической оси. В результате, для дальнейших объяснений будут введены два новых параметра, c_x и c_y для моделирования возможных перемещений (вдалеке от оптической оси) центра координат на проекционном экране. В результате относительно простая модель, в которой точка Q материального мира с координатами (X, Y, Z) проецируется на экран в какой-то заранее заданный пиксель $(x_{\text{screen}}, y_{\text{screen}})$ в соответствии со следующим уравнением (термин "экран" употребляется для напоминания о том, что координаты вычисляются в системе координат экрана (т.е. фотоприёмника); разница между $(x_{\text{screen}}, y_{\text{screen}})$ в уравнении и (x, y) на рисунке 11-2 отображена в точке c_x и c_y):

$$x_{\text{screen}} = f_x \left(\frac{X}{Z} \right) + c_x, \quad y_{\text{screen}} = f_y \left(\frac{Y}{Z} \right) + c_y$$

Стоит обратить внимание на то, что были введены два разных фокусных расстояния; причина этого в том, что одиночные пиксели на типичном недорогом фотоприемнике образуют прямоугольник, а не квадрат. Фокусное расстояние f_x (например) на самом деле является результатом материального фокусного расстояния объектива, а s_x размером отдельного элемента фотоприёмника (это имеет смысл, т.к. s_x измеряется в *пиксели/миллиметр*, а F в *миллиметрах* (это всего лишь удобная физическая величина; ничего не мешает воспользоваться, например, "метром" или "микроном"; в любом случае s_x преобразует материальные единицы в пиксели), что означает - f_x в необходимых единицах пикселей). Тоже самое справедливо и для f_y и s_y . Однако, необходимо иметь ввиду, что s_x и s_y не могут быть измерены в результате процесса калибровки камеры, и ни одна материальная фокусная длина F не может быть непосредственно измерена. Только комбинации $f_x = Fs_x$ и $f_y = Fs_y$ могут быть получены без разбора камеры и непосредственного измерения её компонентов.

Основы геометрии проецирования

Соотношение отображения точки Q_i с координатами (X_i, Y_i, Z_i) в материальном мире к точкам с координатами (x_i, y_i) на проекционном экране называется *проекционным преобразованием*. В работе с такими преобразованиями удобнее всего использовать так называемые *однородные координаты*. Однородные координаты, связанные с точкой из проекционного пространства размерности n , как правило, выражаются в $(n + 1)$ -мерный вектор (например, x, y, z становится x, y, z, w) с дополнительным ограничением, что любые две точки, значения которых пропорциональны, эквивалентны. Для рассматриваемого случая, плоскость изображения является пространством проекции и имеет два измерения, поэтому точка будет представлена на этой плоскости в виде трехмерных векторов $q = (q_1, q_2, q_3)$. Как уже было сказано ранее, все точки, имеющие пропорциональные значения в пространстве проекции, эквивалентны, поэтому можно восстановить действительные координаты пикселя за счет деления на q_3 . Это в свою очередь позволит организовать параметры, определяющие камеру (т.е. f_x, f_y, c_x, c_y), в одну матрицу 3×3 , которая именуется *матрицей встроенных параметров камеры* (OpenCV требует, чтобы встроенные параметры камеры происходили от Heikkila и Silven). Точки проекции в материальном мире для камеры обобщаются следующим образом:

$$q = MQ, \text{ где } q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

В результате перемножения можно увидеть, что $w = Z$, т.к. точка q имеет однородные координаты, вследствие чего необходимо разделить на w (или Z) для получения предыдущего определения. (Знак минус пропал, т.к. теперь рассматривается неперевернутое изображение на плоскости проекции в передней части камеры обскура вместо перевернутого изображения на плоскости проекции за камерой обскура.)

В связи с вводом понятия однородных координат, целесообразно будет рассмотреть функцию OpenCV `cvConvertPointsHomogenous()` (да, именно *Homogenous*, в наименование функции присутствует ошибка), удобную для преобразования в и из однородных координат; помимо этого, данная функция выполняет ещё ряд полезных вещей.

```
void cvConvertPointsHomogenous(
    const CvMat*   src
    ,CvMat*        dst
);
```

На первый взгляд может показаться, что у данной функции слишком простой набор аргументов; но это не так – на самом деле данная функция выполняет массу полезных вещей. Исходный массив src может быть $M_{src} \times N$ или $N \times M_{src}$ (для $M_{src} = 2, 3$ или 4); он также может быть $1 \times N$ или $N \times 1$ в массиве с $M_{src} = 2, 3$ или 4 каналами (N может быть любым числом; по существу это точки для преобразования, которые заполняют матрицу src). Конечный массив dst может быть любого типа с дополнительным ограничением по размерности M_{dst} – она должна быть равна M_{src} , $M_{src} - 1$ или $M_{src} + 1$.

Когда размерность исходного M_{src} и конечного M_{dst} равны, происходит просто копирование (и, если необходимо, транспонирование). Если $M_{src} > M_{dst}$, то элементы dst вычисляются за счет деления всех кроме последних элементов соответствующего вектора src на последний элемент этого вектора (т.е. предполагается, что src содержит однородные координаты). Если $M_{src} < M_{dst}$, то происходит копирование точек, только каждый вектор массива dst на конце будет содержать 1 (т.е. вектора из src расширяются до однородных координат).

Одним словом, при работе с данной функцией стоит принимать во внимание, что могут быть случаи (при $N < 5$), когда исходная и конечная размерность неодинакова. В этом случае функция вернет ошибку. Столкнувшись с данной ситуацией, необходимо дополнить матрицы фиктивными значениями. Кроме того, пользователь может передавать многоканальные матрицы $N \times 1$ и $1 \times N$, с числом каналов M_{src} (M_{dst}). Функция `cvReshape()` может быть использована для преобразования одноканальной матрицы к многоканальной без копирования данных.

Случай с идеальной камерой обскура - это полезная модель для некоторой трехмерной геометрии зрения. При этом стоит помнить, что через камеру обскура проходит очень малое количество света; таким образом, на практике изображение будет формироваться крайне медленно из-за ожидания получения достаточного количества света. Для ускорения процесса формирования изображения в камере необходимо собирать свет более широкой области и изгиба (т.е. фокуса), т.е. в точке проекции, где сходится свет. Для достижения этой цели необходимо использовать объектив. Объектив может фокусировать большее количество света в точке, что соответственно ускоряет визуализацию, однако, это приводит также и к появлению искажений.

Искажения объектива

В теории, можно определить объектив, который воспроизводит без искажений. На практике, однако, объективы не идеальны. Главная причина кроется в процессе производства объективов; гораздо легче сделать более "сферический", чем более математически идеальный "параболический" объектив. Так же технически сложно точно совместить объектив и фотоприёмник. В данном разделе будут рассмотрены два основных типа искажения объектива и их моделирование. *Радиальные искажения* возникают в результате формы объектива, в то время как *тангенциальные искажения* возникают как результат сборки камеры в целом.

Радиальные искажения. Линзы реальных камер часто искажают расположение пикселей вблизи краев фотоприёмника. Это выпуклое явление появляется в результате эффекта "бочка" или *рыбий глаз* (хороший пример данного эффекта представлен в верхней части рисунка 11-12). Рисунок 11-3 подталкивает к пониманию возникновения радиальных искажений. У некоторых объективов лучи, далекие от центра, более изогнуты, чем те, которые ближе к центру. Типичный недорогой объектив обладает именно таким эффектом. Искажение "бочка" особенно заметно в недорогих веб-камерах и менее очевидно в высококачественных камерах, где за счет использования причудливой системы линз минимизируются радиальные искажения.

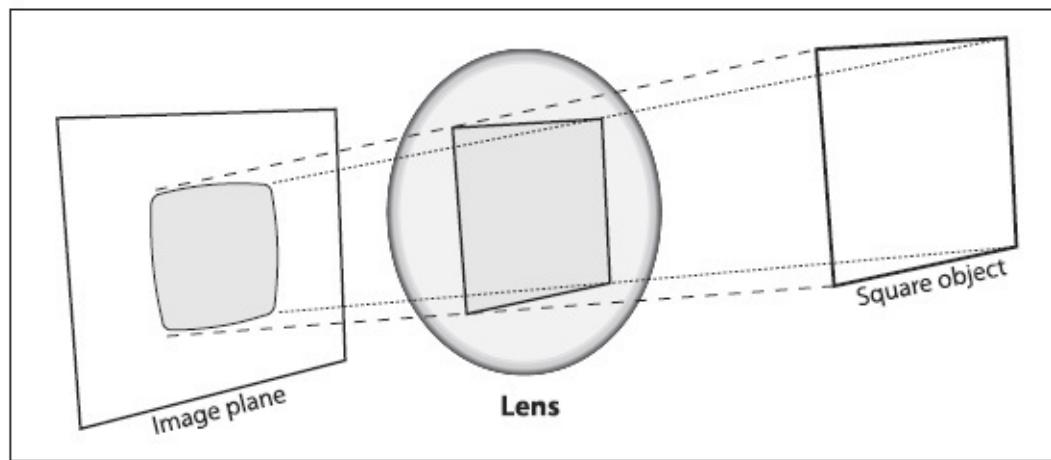


Рисунок 11-3. Радиальные искажения: лучи, отдаленные от центра, искажены в большей степени, чем лучи, проходящие ближе к центру; таким образом, стороны квадрата скругляются на плоскости изображения (это также известно, как эффект "бочка")

Для радиальных искажений характерно, что искажение в (оптическом) центре фотоприёмника 0 и увеличивается по мере продвижения к окраине. На практике данное искажение не велико и может быть охарактеризовано несколькими первыми членами разложения в ряд Тейлора вокруг $r = 0$. (Ряд Тейлора – это математический метод для представления (потенциально) сложной функции в виде полинома подобного значения приближенной функции, по меньшей мере, в малой окрестности некоторой определенной точки (чем больше элементов будет задействовано, тем более точный результат будет получен)). В рассматриваемом случае возникла необходимость расширения искажений до полинома в окрестности $r = 0$. Данный полином имеет следующий общий вид: $f(r) = a_0 + a_1r + a_2r^2 + \dots$, а для рассматриваемого случая при $f(r) = 0$ и $r = 0$ следует, что $a_0 = 0$. Кроме того, функция должна быть симметричной в r , поэтому только коэффициенты четных степеней r должны быть отличны от нуля. Из всего этого следует, что для описания радиальных искажений необходимо использовать коэффициенты r^2 , r^4 и (иногда) r^6). Для дешевых веб-камер обычно используются первые два члена; первый условно называется k_1 , а второй k_2 . Для сильно искаженных веб-камер, также необходимо задействовать и третий член k_3 . В общем, радиально расположенные точки будут масштабированы в соответствии со следующими уравнениями:

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

где (x, y) – это оригинальное положение (на фотоприёмнике) искаженной точки, а $(x_{\text{corrected}}, y_{\text{corrected}})$ – это новое положение после коррекции. На рисунке 11-4 показано смещение прямоугольной сетки, которое вызвано радиальными искажениями.

Внешние точки на внешней стороне прямоугольной сетки все больше смещаются внутрь при увеличении радиального расстояния от оптического центра.

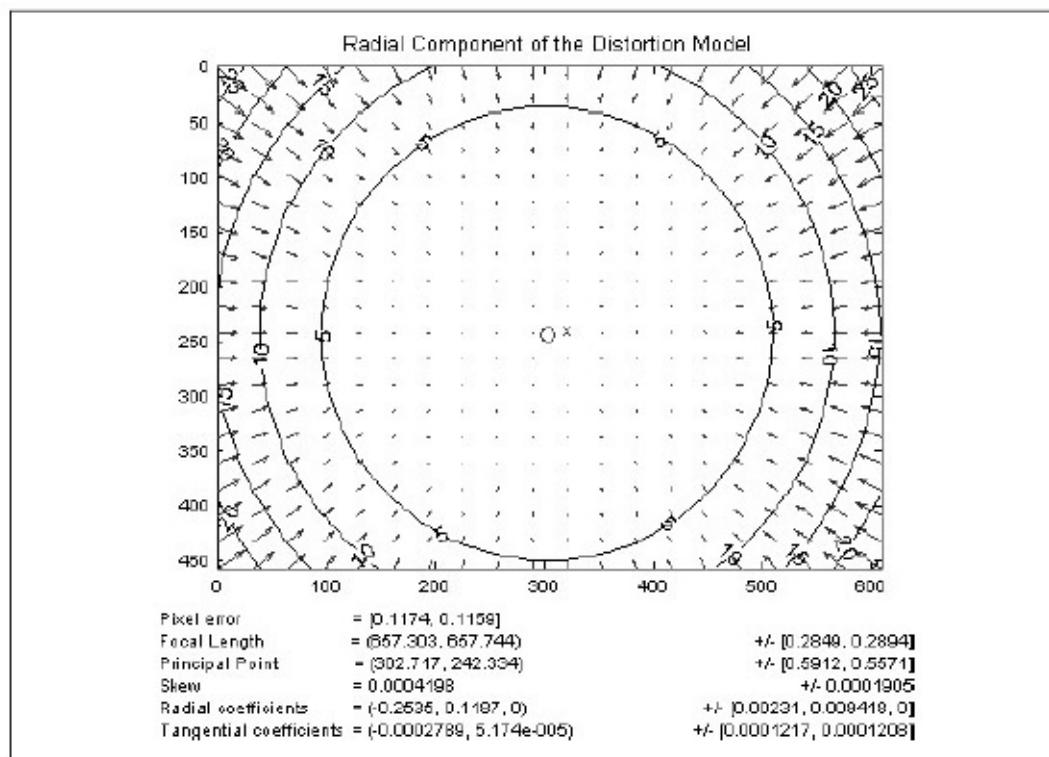


Рисунок 11-4. Сетка радиальных искажений для конкретного объектива камеры: стрелками показано, где точки на внешней прямоугольной сетке смещаются в радиально искаженном изображении (фото предоставлено Jean-Yves Bouguet)

Тангенциальное искажение. Эти искажения возникают в результате производственных дефектов, возникающих от не точно параллельно установленных линз к плоскости изображения, рисунок 11-5.

Тангенциальное искажение минимально характеризуется двумя дополнительными параметрами p_1 и p_2 :

$$x_{\text{corrected}} = x + [2p_1y + p_2(r^2 + 2x^2)]$$

$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2x]$$

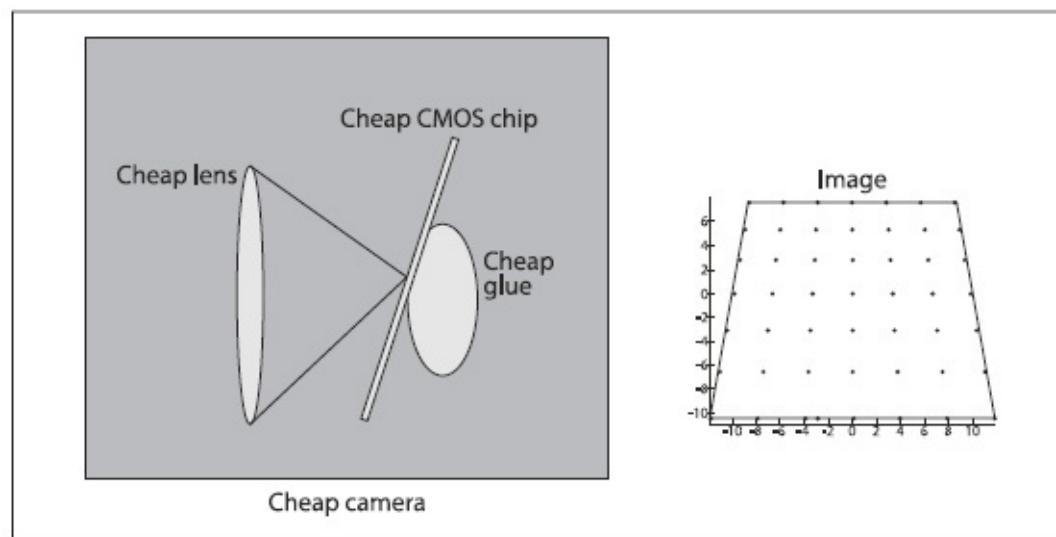


Рисунок 11-5. Тангенциальные искажения появляются, когда объектив не полностью параллелен плоскости изображения; в дешевых камерах это может произойти, когда фотоприёмник клеится к задней поверхности камеры (фото предоставлено Sebastian Thrun)

Таким образом, в общей сложности в работе с данными искажениями будут необходимы только пять коэффициентов. В работе с OpenCV все пять коэффициентов образуют *вектор искажений*; этот вектор просто матрица 5×1 , содержащая k_1 , k_2 , p_1 , p_2 и k_3 (именно в таком порядке). На рисунке 11-6 показано влияние тангенциального искажения на фронтально внешнюю прямоугольную сетку точек. Точки смещаются эллиптически согласно функции положения и радиуса.

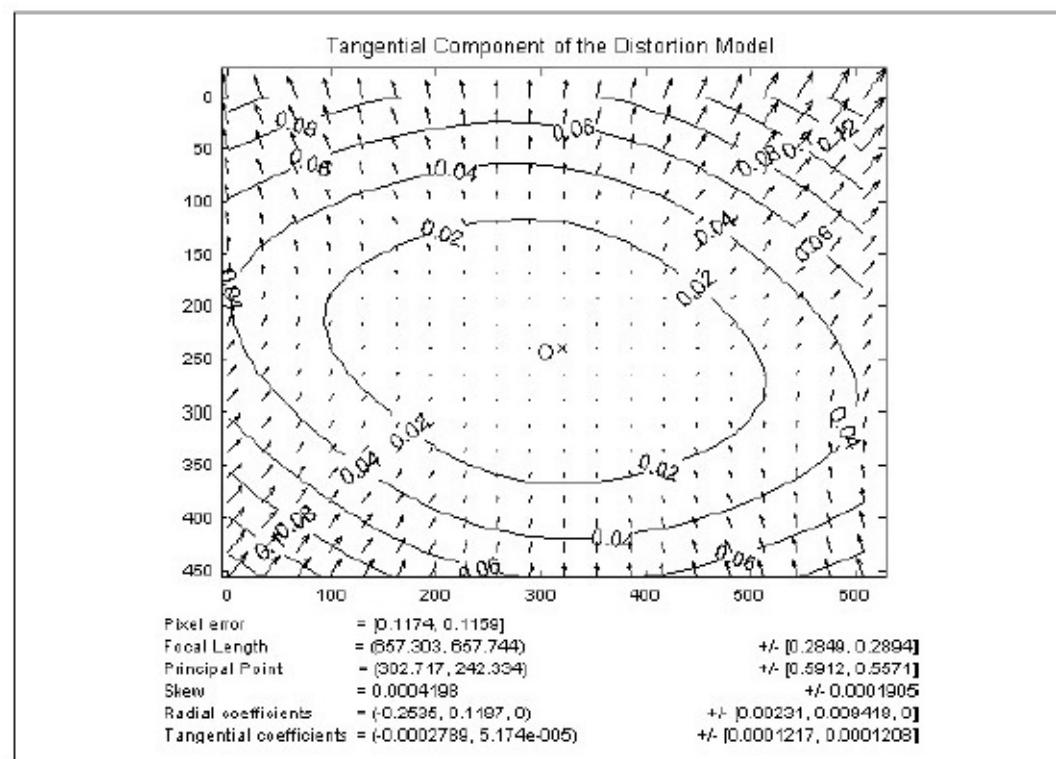


Рисунок 11-6. Сетка тангенциальных искажений для конкретного объектива камеры: стрелками показано, где точки на внешней прямоугольной сетке смещаются по касательной к искаженному изображению (фото предоставлено Jean-Yves Bouguet)

Существует ещё множество других видов искажений, которые возникают в системах визуализации, но они, как правило, имеют малый эффект по сравнению с радиальным и тангенциальным искажениями. Связь с этим данные (другие) искажения в дальнейшем рассматриваться не будут.

[П]||[РС]||(РП) Калибровка

Теперь, после получения некоторого представления о том, как описать внутренние свойства и искажения камеры математически, можно перейти к рассмотрению того, как использовать OpenCV для вычисления внутренней матрицы и вектора искажений. (Онлайн руководство по использованию калибровки камеры можно найти на сайте [Jean-Yves Bouguet](#))

OpenCV предоставляет несколько алгоритмов для вычисления внутренних параметров. Калибровка выполняется при помощи функции `cvCalibrateCamera2()`. В данной функции метод калибровки предоставленной камеры заключается в формировании структуры, содержащей множество индивидуальных и идентифицируемых точек. При рассмотрении данной структуры под разными углами можно в последующем вычислить (относительное) положение и ориентацию камеры во время каждого получаемого кадра, а также внутренние параметры камеры (рисунок 11-9, раздел "Шахматная доска"). Для получения набора представлений, необходимо поворачивать и смещать объект, поэтому вначале необходимо рассмотреть эти процессы более подробно.

Матрица поворота и вектор смещения

Для каждого кадра, содержащего определенный объект, существует возможность описать позу данного объекта по отношению к системе координат камеры в условиях вращения и смещения, рисунок 11-7.

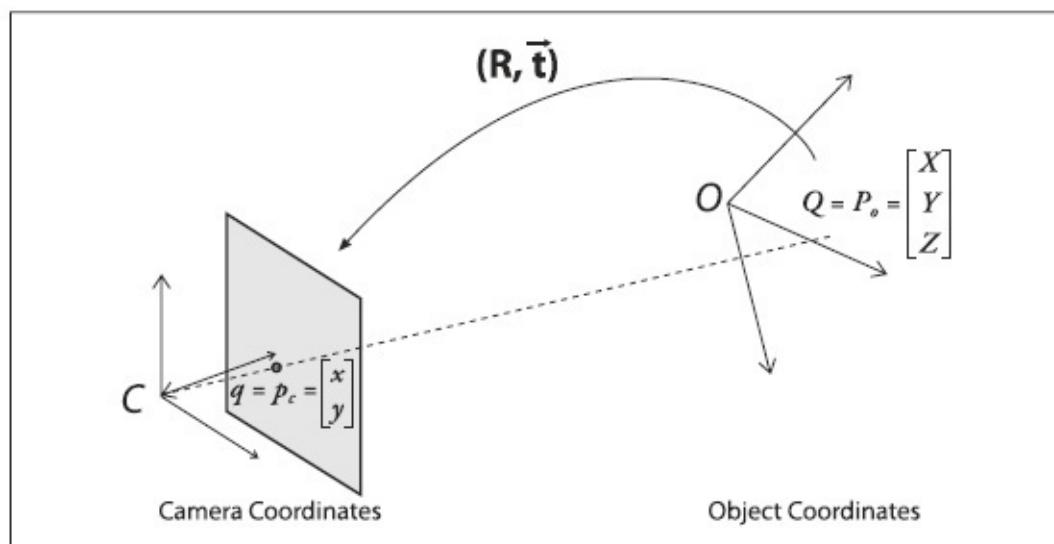


Рисунок 11-7. Преобразование объекта в систему координат камеры: точка P объекта рассматривается как точка p на плоскости изображения; точка p связана с точкой P за счет матрицы вращения R и вектора смещения t к P

В общем, вращение в любом n-мерном измерении может быть описано как перемножение координат вектора квадратной матрицы соответствующего размера. В конечном счете, вращение эквивалентно введению нового расположения точки в другой системе координат. Поворот системы координат на угол θ эквивалентен вращению в противоположном направлении целевой точки вокруг исходной системы координат на тот же угол θ . Двумерное вращение можно представить, как перемножение матриц, представленных на рисунке 11-8. Трехмерное вращение можно разложить на двумерное вращение вокруг каждой оси при условии, что ось вращения остается неизменной. При последовательном вращении (в представленном описании вращения сначала выполняется вращение вокруг оси z, затем вокруг нового положения оси y и, наконец, вокруг нового положения оси x) вокруг x-, y- и z-осей на углы ψ , ϕ и θ соответственно, общую матрицу поворота R можно получить в результате перемножения трех матриц $R_x(\psi)$, $R_y(\phi)$ и $R_z(\theta)$, где:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix}$$

$$R_y(\phi) = \begin{bmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

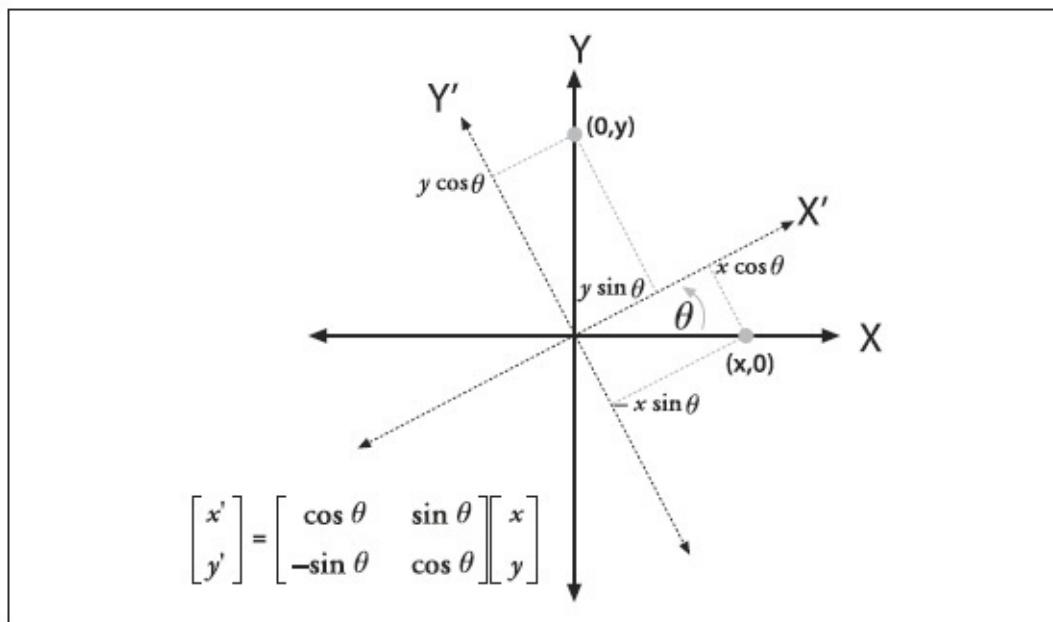


Рисунок 11-8. Поворот точек на θ (для данного случая, вокруг оси Z) это тоже самое, что и противоположное вращение оси координат на θ ; за счет простой тригонометрии можно увидеть, как вращение меняет координаты точек

Общая матрица вращения R имеет свойство, что обратная ей матрица является транспонированной матрицей; следовательно $R^T R = R R^T = I$, где I это единичная матрица (элементы по диагонали равны 1, а все остальные элементы равны 0).

Вектор смещения представляет собой переход от одной системы координат к другой как результат смещения в новое положение; другими словами, вектор смещения – это просто смещение относительно первоначальной системы координат. Таким образом, при переходе от системы координат с центрированием на объекте к центрированию на камере, соответствующий вектор смещения будет выглядеть следующим образом $T = \text{origin}_{\text{object}} - \text{origin}_{\text{camera}}$. В результате (рисунок 11-7) точка в системе координат объекта с координатами P_o имеет координаты P_c в системе координат камеры:

$$P_c = R(P_o - T)$$

Объединение данного уравнения с выше представленными исправлениями внутренних параметров камеры формирует базовую систему уравнений, которую необходимо передавать OpenCV для калибровки камеры.

Как уже было сказано ранее, для описания трехмерного вращения достаточно указать 6 параметров: три параметра положения и три параметра углов вращения. В добавок к этому в OpenCV матрица встроенных параметров камеры имеет ещё четыре параметра (f_x , f_y , c_x и c_y), что в общей сложности дает десять параметров, которые должны быть получены для каждого представления в отдельности (при этом внутренние параметры камеры остаются неизменными). В случае двумерного

вращения задействуются восемь параметров. Шесть параметров для описания вращения и смещения между представлениями и два для матрицы внутренних параметров камеры. И ещё, по крайней мере, в двух представлениях необходимо найти все геометрические параметры.

Далее будут более подробно рассмотрены эти параметры и накладываемые на них ограничения, но для начала необходимо уделить немного времени рассмотрению калибровочного объекта. Калибровочным объектом в OpenCV является плоская сетка с чередующимися черными и белыми квадратами, которую обычно называют "шахматной доской" (хотя и не обязательно иметь восемь квадратов или даже равное количество квадратов в каждом из направлений).

Шахматная доска

В принципе, любой достаточно характерный объект может быть использован в качестве калибровочного объекта, однако, практичней всего использовать такой шаблон, как шахматная доска. В литературе некоторые методы калибровки полагаются на трехмерные объекты (например, коробка накрытая ориентиром), однако, плоская модель шахматной доски гораздо легче в обращении; трудно сделать (а так же хранить и распространять) точные калибровочные 3D объекты. В результате, OpenCV работает с несколькими представлениями плоского калибровочного объекта (шахматная доска), а не с одним специально сконструированным 3D объектом. В дальнейшем будет использован шаблон, состоящий из сменяющихся черных и белых квадратов (рисунок 11-9), который гарантирует, что нет никакого смещения к той или иной стороне измерения. Кроме того, результатирующие углы сетки можно передать в функцию локализации субпикселей, обсуждаемую в главе 10.

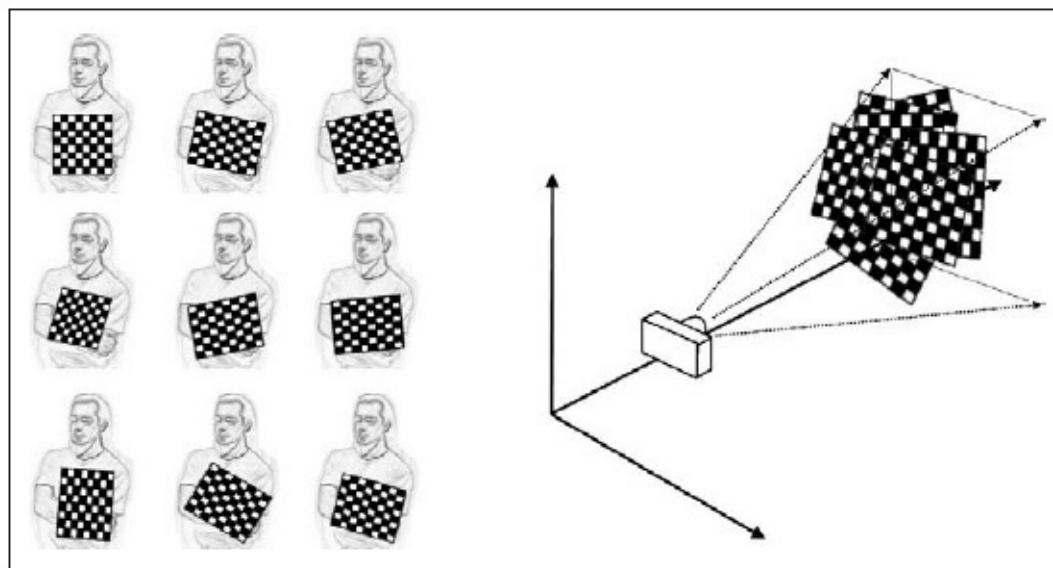


Рисунок 11-9. Изображения шахматной доски, получаемые в различных ориентациях (слева), формируют достаточное количество информации для понимания положения этих изображений в глобальной системе координат (относительно камеры), а также получения встроенных параметров камеры

Изображение шахматной доски (или изображение человека, держащего шахматную доску) можно использовать в функции OpenCV *cvFindChessboardCorners()* для поиска углов шахматной доски:

```
int cvFindChessboardCorners(
    const void*      image
, CvSize          pattern_size
, CvPoint2D32f*   corners
, int*            corner_count = NULL
, int             flags        = CV_CALIB_CB_ADAPTIVE_THRESH
);
```

На вход данной функции в качестве первого аргумента передается одно изображение шахматной доски. Это должно быть 8-битное одноканальное в градациях серого изображение. Второй аргумент *pattern_size* указывает на количество углов в каждой строке и столбце доски. Иными словами это счетчик числа углов в пределах доски; таким образом, для стандартной игровой шахматной доски это значение будет равно *CvSize(7, 7)*. (На практике, чаще всего бывает удобнее использовать ассиметричную шахматную доску четно-нечетного размера, например (5, 6). Ассиметричная доска имеет только одну ось симметрии, поэтому ориентация доски может быть определена однозначно). Следующий аргумент *corners* является указателем на массив, в который можно будет записать расположение углов. Этот массив должен быть инициализирован перед вызовом функции и, конечно же, должен обладать достаточным размером для размещения всех углов доски (для стандартной игровой шахматной доски это значение равно 49). Элементы массива это положения углов в пиксельных координатах. Аргумент *corner_count* не обязателен; если *corner_count != NULL*, то это указатель на *integer* для указания числа углов, которые необходимо записать. Если функция успешно нашла все углы (на самом деле условие успешного выполнения немного строже: должны быть найдены не только все углы, но и они же должны быть упорядочены в строках и столбцах в соответствии с ожиданиями), то возвращаемое значение будет не нулевым числом. Если функция завершилась с ошибкой, то будет возвращен 0. Последний аргумент *flags* может быть использован для реализации одного или нескольких шагов фильтрации для улучшения процесса поиска углов. Флаги могут быть объединены при помощи логического OR.

CV_CALIB_CB_ADAPTIVE_THRESH

По умолчанию в функции применяется пороговые преобразования на основе средней яркости, но если установлен этот флаг, то будут использованы адаптивные пороговые преобразования

CV_CALIB_CB_NORMALIZE_IMAGE

За счет этого флага изображение нормализуется при помощи *cvEqualizeHist()* перед применением пороговых преобразований

CV_CALIB_CB_FILTER_QUADS

После выполнения пороговых преобразований, алгоритм попытается найти четырехугольники как результат перспективного представления черных квадратов шахматной доски. Это приближенный результат, т.к. предполагается, что ребра четырехугольника прямые, но это не совсем верно, потому что имеют место радиальные искажения. Если этот флаг установлен, то к четырехугольникам применяются множество дополнительных ограничений для отбраса ложных четырехугольников.

Субпиксельные углы

Углы, возвращаемые *cvFindChessboardCorners()* являются приблизительными. На практике это означает, что положения точны только в пределах устройства обработки изображения, т.е. с точностью до пикселя. Функция разделения должна быть использована для вычисления точного расположения углов (после получения приблизительного положения и исходного изображения) с точностью до субпикселя. Для этого необходимо использовать функцию *cvFindCornerSubPix()*, которая уже была рассмотрена в главе 10. Использование данной функции в данном контексте не должно вызывать удивления, т.к. углы шахматной доски это всего на всего частный случай более общего случая углов Harris; просто углы шахматной доски проще найти и отследить. Пренебрежение субпиксельным уточнением может привести в существенным ошибкам в калибровке.

Отрисовка углов шахматной доски

В частности, при отладке зачастую желательно нарисовать найденные углы шахматной доски на изображение (обычно на том, по которому производился поиск); таким образом, можно сравнить прогнозируемые и наблюдаемые углы. Для выполнения данной задачи OpenCV предоставляет удобную функцию *cvDrawChessboardCorners()*, которая соответственно и рисует углы, найденные *cvFindChessboardCorners()* на предоставленном изображении. Если в результате не все углы будут найдены, то имеющиеся углы будут нарисованы в виде маленьких

красных кругов. Если найдены все углы, то они будут окрашены в разные цвета (на каждую строку по одному цвету) и соединены линиями, представляющие порядок определения углов.

```
void cvDrawChessboardCorners(  
    CvArr*      image  
    , CvSize     pattern_size  
    , CvPoint2D32f* corners  
    , int        count  
    , int        pattern_was_found  
) ;
```

Первый аргумент *cvDrawChessboardCorners()* - это изображение, на которое будут наноситься углы. Т.к. углы будут представлены в виде цветных кругов, то это должно быть 8-битное цветное изображение и, в большинстве случаев, это будет копия исходного изображения, которое передавалось в *cvFindChessboardCorners()* (при этом необходимо предварительно выполнить преобразование в трехканальное изображение). Следующие два аргумента *pattern_size* и *corners* имеют точно такой же смысл, как и соответствующие аргументы функции *cvFindChessboardCorners()*. Аргумент *count* это целое число и равно числу углов. Аргумент *pattern_was_found* указывает на успех в поиске углов и может быть установлен при помощи *cvFindChessboardCorners()*. На рисунке 11-10 представлен результат применения *cvDrawChessboardCorners()* к изображению шахматной доски.

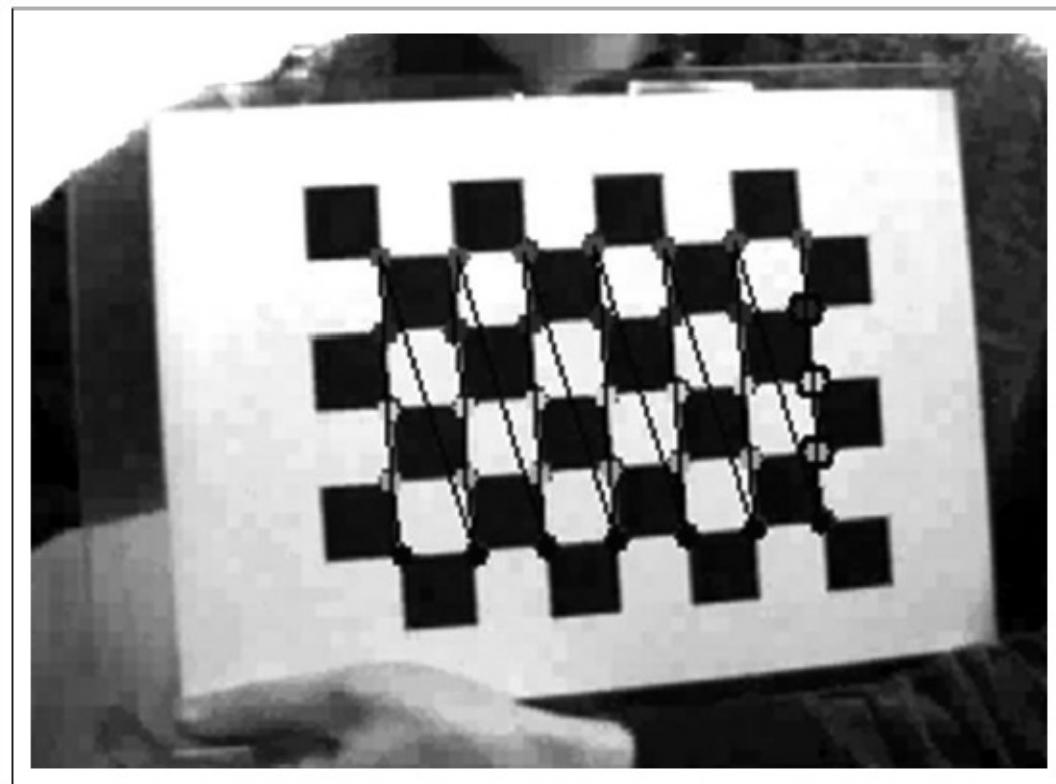


Рисунок 11-10. Результат применения `cvDrawChessboardCorners()`: после того, как углы были найдены при помощи `cvFindChessboardCorners()`, можно нарисовать эти углы (мелкие круги по углам) и порядок определения этих углов (показано линиями между углами)

Теперь можно перейти к рассмотрению плоского объекта. Точки на плоскости подвергаются перспективному преобразованию после прохождения через отверстие камеры обскуры или через объектив. Параметры данного преобразования содержаться в матрице гомографии размера 3×3 , которая в следующем разделе будет рассмотрена более подробно.

Гомография

В компьютерном зрении *плоская гомография* определяется как проективное отображение из одной плоскости в другую. Таким образом, отображение точек на двумерную плоскую поверхность фотоприёмника камеры является примером плоской гомографии. Данный пример можно показать в виде перемножения матриц при условии, что будут использованы однородные координаты для отображения точек Q и q на фотоприёмнике. Если определить:

$$\tilde{Q} = [X \ Y \ Z \ 1]^T$$

$$\tilde{q} = [x \ y \ 1]^T$$

тогда действие гомографии можно выразить следующим образом:

$$\tilde{q} = sH\tilde{Q}$$

Введённый параметр s - это произвольный масштабный коэффициент (предназначенный для того, чтобы явно показать, что гомография определяется только этим коэффициентом). Условно этот коэффициент вынесен из H и в дальнейшем при рассмотрении материала данное правило будет выполняться.

За счет небольшого применения геометрии и матричной алгебры можно найти решение для этой матрицы преобразования. Наиболее важным моментом является то, что H состоит из двух частей: материальных преобразований, которые по существу определяют положение объекта на рассматриваемой плоскости; и проекции, которая выводит матрицу встроенных параметров камеры (рисунок 11-11).

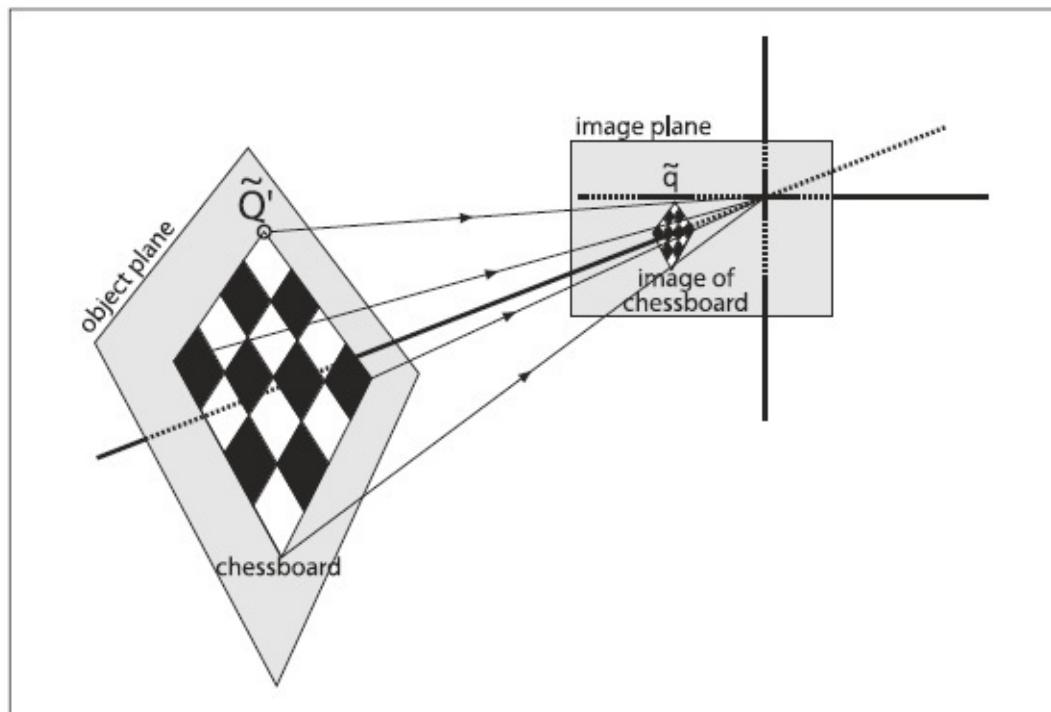


Рисунок 11-11. Представление плоского объекта, построенное гомографией: отображение - от плоскости объекта до плоскости изображения - одновременно охватывает относительное расположение этих двух плоскостей, а также матрицу проекции камеры

По части материальных преобразований - это суммарное действие некоторого вращения R и некоторого сдвига t , которые относятся к плоскости, на которой рассматривается плоскость изображения. Т.к. используются однородные координаты, то можно выполнить объединение в пределах одной матрицы следующим образом (W - это матрица 3×4 , первые три колонки которой включают девять элементов R , а последняя колонка содержит три компонента вектора t):

$$W = [R \ t]$$

И в заключении перемножив матрицу M действий камеры (выраженную в проекционных координатах) и $W\tilde{Q}$, получим следующее:

$$\tilde{q} = sMW\tilde{Q}, \text{ где } M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Казалось бы, что все это значит. Оказывается, на практике интерес вызывает не координата \tilde{Q} , которая определена для всех пространств, а координата \tilde{Q}' , которая определена только для рассматриваемой плоскости. Это возможно только при небольшом упрощении.

Без ограничения общности, необходимо выбрать определенную плоскость объекта так, чтобы $Z = 0$. Это необходимо, т.к. при разбиении матрицы вращения на три 3×1 столбца (т.е. $R = [r_1 \ r_2 \ r_3]$), один из этих столбцов не потребуется. В частности:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Матрицу гомографии H , которая отображает точки объекта на плоскости фотоприёмника, можно описать выражением $H = sM[r_1 \ r_2 \ t]$, тогда:

$$\tilde{q} = sH\tilde{Q}'$$

При этом стоит отметить тот факт, что теперь матрица H имеет размерность 3×3 .

OpenCV использует предыдущее уравнение для расчета матрицы гомографии. Библиотека использует несколько изображений одного объекта для вычисления преобразований и вращений для каждого представления, а также встроенных параметров камеры (которые одинаковы для всех представлений). Как уже было сказано ранее, вращение описывается тремя углами, а перемещение тремя смещениями; соответственно имеется шесть неизвестных для каждого представления. Это нормально, т.к. известно, что плоский объект (например, шахматная доска) дает восемь уравнений - т.е. отображение квадрата в четырехугольник можно описать четырьмя (x, y) точками. Каждый новый кадр дает восемь уравнений за счет шести новых внешних неизвестных, так что при наличии достаточного количества изображений имеется возможность вычислить любое количество собственных неизвестных (более подробно об этом чуть позже).

Матрица гомографии H связывает положения точек плоскости исходного изображения с точками плоскости конечного изображения (как правило, плоскости фотоприёмника) следующими уравнениями:

$$P_{dst} = H P_{src}, \quad P_{src} = H^{-1} P_{dst}$$

$$P_{dst} = \begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix}, \quad P_{src} = \begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix}$$

Стоит обратить внимание на тот факт, что существует возможность вычислить H , ничего не зная о встроенных параметрах камеры. На самом деле, вычисление множества гомографий из нескольких представлений - это метод, который использует OpenCV для вычисления внутренних параметров камеры.

OpenCV предоставляет удобную функцию `cvFindHomography()`, которая принимает список соответствий и возвращает матрицу гомографии, которая наилучшим образом описывает эти соответствия. Необходимо как минимум четыре точки, чтобы найти H , однако, всегда имеется возможность предоставить гораздо большее количество точек (при условии рассмотрения шахматной доски размера большего, чем 3×3). Использование большего количества точек гораздо выгоднее, т.к. всегда имеется шум и иные несоответствия, влияние которых необходимо сводить к минимуму.

```
void cvFindHomography(
    const CvMat*   src_points
    , const CvMat*   dst_points
    , CvMat*        homography
);
```

Входные массивы `src_points` и `dst_points` могут быть матрицами $N \times 2$ или $N \times 3$. В первом случае координаты пикселей, во втором однородные координаты. Последний аргумент `homography` это матрица 3×3 , заполненная при помощи данной функции таким образом, чтобы минимизировать ошибку обратной проекции. Т.к. матрица гомографии имеет только восемь свободных параметров, то лучше всего нормализовать $H_{33} = 1$. Масштабирование гомографии может быть применено к девятому параметру гомографии, но обычно масштабирование выполняется за счет умножения всей матрицы гомографии на коэффициент масштабирования.

Калибровка камеры

Теперь всё готово для калибровки камеры и получения встроенных параметров и параметров искажений. Данный раздел будет посвящен вычислению этих параметров за счет использования `cvCalibrateCamera2()`, а также тому, как исправлять искажения на изображениях, получаемых с калибровочной камеры. Вначале будет немного более подробно рассказано о том, как много представлений шахматной доски необходимо предоставить для вычисления внутренних параметров камеры и искажений. Затем будет высокоуровневый обзор того, как OpenCV на самом деле решает данную систему уравнений. И в заключении будет представлен код, который позволяет с лёгкостью справиться с данной задачей.

Сколько углов шахматной доски для скольких параметров?

Теперь необходимо рассмотреть ранее представленные неизвестные. Т.е. параметры, которые необходимо найти путем калибровки. В случае OpenCV имеется четыре внутренних параметра камеры (f_x , f_y , c_x , c_y) и пять параметров искажений: три для радиальных (k_1 , k_2 , k_3) и два для тангенциальных (p_1 , p_2). Внутренние параметры

камеры напрямую связаны с трехмерной геометрией (и, следовательно, с внешними параметрами) шахматной доски в пространстве; параметры искажений связаны с двухмерной геометрией набора искаженных точек, в результате имеем ограничение на эти два класса параметров по отдельности. Три угловые точки известной модели дают шесть наборов информации – в принципе, это все, что необходимо для вычисления пяти параметров искажения (хотя на практике используется гораздо большее количество углов). Таким образом, одного представления шахматной доски вполне достаточно для вычисления параметров искажения. Это же представление шахматной доски можно использовать для вычисления внутренних параметров, которые вычисляются после получения внешних параметров. В случае с *внешними* параметрами необходимо знать положение шахматной доски. Для этого требуется три параметра вращения (ψ, ϕ, θ) и три параметра перемещения (T_x, T_y, T_z), в общей сложности шесть на одно представление шахматной доски, т.к. в каждом изображении шахматной доски присутствуют перемещения. Итого, четыре внутренних и шесть внешних параметра дают десять параметров, которые необходимо вычислять для каждого представления.

Теперь пусть имеется N углов и K изображений шахматной доски (в различных позициях). Как много представлений и углов необходимо иметь для преодоления ограничений всех ранее представленных параметров?

- К изображений шахматной доски обеспечивает $2NK$ ограничений (коэффициент 2 используется в связи с тем, что каждая точка изображения имеет две координаты x и y)
- Не принимая во внимание параметры искажения, имеется 4 внутренних параметра и $6K$ внешних параметра (т.к. необходимо найти 6 параметров положения шахматной доски для каждого представления K)
- Для решения также необходимо, чтобы $2NK \geq 6K + 4$ (или, что эквивалентно $(N - 3)K \geq 2$)

Может показаться, что для $N = 5$ необходимо только $K = 1$ изображений. Однако это не так, K (количество представлений) должно быть больше 1. Причина, по которой $K > 1$ заключается в том, что используемая для калибровки шахматная доска должна соответствовать матрице гомографии для каждого K представления. Как уже было сказано ранее гомография может дать не более восьми параметров от четырех пар (x, y) . Это всё из-за того, что необходимо иметь лишь четыре точки для выражения всего того, что может дать плоское перспективное представление: растянуть квадрат в четырех различных направлениях, превратив его тем самым в абсолютно любой четырехугольник (перспективные преобразования, глава 6). Таким образом, независимо от количества найденных углов на плоскости, ценными являются только

четыре из них. Согласно уравнению одного представления шахматной доски, предоставляющее только четыре информативных угла или $(4 - 3)K > 1$, получается, что $K > 1$. Это означает, что два представления шахматной доски 3×3 (считая только внутренние углы) являются минимальным набором для решения проблемы калибровки. Принимая во внимание шум и вычислительную устойчивость, как правило, необходимо собирать больше изображений шахматной доски большего размера. На практике для получения наилучшего результата необходимо как минимум десять изображений 7×8 или большего размера шахматной доски (и это при условии, что будет предоставлен "богатый" набор представлений шахматной доски).

Что под капотом?

Этот раздел предназначен для тех, кто хочет более детально во всем разобраться; остальные могут благополучно пропустить этот раздел и сразу перейти к разбору функции калибровки. Итак, возникает вопрос: как вся эта математика используется для калибровки? При наличии множества способов найти параметры камеры, в OpenCV выбран один, который хорошо отрабатывает на плоских объектах. Алгоритм, используемый OpenCV для нахождения фокусного расстояния и смещения, основан на методе Zhang, а метод, основанный на методе Brown, используется для поиска параметров искажения.

Для начала пусть при поиске других параметров калибровки не существует искажений камеры. Для каждого представления шахматной доски вычисляется гомография H , как уже было показано ранее. В результате H будет состоять из столбцов-векторов, т.е. $H = [h_1, h_2, h_3]$, где каждая h это вектор 3×1 . Тогда, в силу ранее представленного материала по гомографии, можно установить, что H равна перемножению матрицы внутренних параметров M , комбинации первых двух столбцов матрицы вращения r_1, r_2 и вектора смещения t , и масштабного коэффициента s :

$$H = [h_1 \ h_2 \ h_3] = sM[r_1 \ r_2 \ t]$$

Разбор данного уравнения выглядит следующим образом:

$$h_1 = sMr_1 \text{ или } r_1 = \lambda M^{-1}h_1$$

$$h_2 = sMr_2 \text{ или } r_2 = \lambda M^{-1}h_2$$

$$h_3 = sMt \text{ или } t = \lambda M^{-1}h_3$$

где $\lambda = 1/s$.

Т.к. вектора вращения ортогональны относительно друг друга по своей конструкции и поскольку извлекается масштабный коэффициент, следовательно r_1 и r_2 ортонормированы. Ортонормированность подразумевает две вещи: скалярное

произведение векторов вращения равно 0 и величины векторов равны. Согласно первому утверждению:

$$\mathbf{r}_1^T \mathbf{r}_2 = 0$$

Для любых векторов a и b справедливо $(ab)^T = b^T a^T$, поэтому заменяя на \mathbf{r}_1 и \mathbf{r}_2 получаем первое ограничение:

$$\mathbf{h}_1^T M^{-T} M^{-1} \mathbf{h}_2 = 0$$

где A^{-T} более короткая запись $(A^{-1})^T$. Согласно второму утверждению:

$$\|\mathbf{r}_1\| = \|\mathbf{r}_2\| \text{ или } \mathbf{r}_1^T \mathbf{r}_1 = \mathbf{r}_2^T \mathbf{r}_2$$

Подставляя \mathbf{r}_1 и \mathbf{r}_2 получаем второе ограничение:

$$\mathbf{h}_1^T M^{-T} M^{-1} \mathbf{h}_1 = \mathbf{h}_2^T M^{-T} M^{-1} \mathbf{h}_2$$

Для упрощения $B = M^{-T} M^{-1}$. Тогда:

$$B = M^{-T} M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$$

Помимо этого, матрица В может иметь более общую замкнутую форму:

$$B = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix}$$

Используя матрицу В, получаем более общую форму двух ограничений $\mathbf{h}_i^T B \mathbf{h}_j$. Т.к. В симметрична, то может быть записана в виде скалярного произведения шестимерных векторов. Скомпоновав необходимые элементы В в новый вектор b, получим:

$$\mathbf{h}_i^T B \mathbf{h}_j = v_{ij}^T b = \begin{bmatrix} h_{ii} h_{j1} \\ h_{ii} h_{j2} + h_{i2} h_{j1} \\ h_{i2} h_{j2} \\ h_{i3} h_{j1} + h_{i1} h_{j3} \\ h_{i3} h_{j2} + h_{i2} h_{j3} \\ h_{i3} h_{j3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}$$

Используя v_{ij}^T , два ограничения могут записаны как:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} b = 0$$

Если собрать К изображений шахматной доски вместе, то можно скомпоновать К уравнений вместе:

$$Vb = 0$$

где V - это матрица 2Kx6. Если K ≥ 2, то уравнение может быть решено для $b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$. Внутренние параметры камеры в таком случае берутся непосредственно из замкнутой формы матрицы B:

$$\begin{aligned} f_x &= \sqrt{\lambda/B_{11}} \\ f_y &= \sqrt{\lambda B_{11}/(B_{11}B_{22} - B_{12}^2)} \\ c_x &= -B_{13}f_x^2/\lambda \\ c_y &= (B_{12}B_{13} - B_{11}B_{23})/(B_{11}B_{22} - B_{12}^2) \end{aligned}$$

где

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23}))/B_{11}$$

Внешние параметры (вращение и перемещение) вычисляются из уравнений, полученных из условий гомографии:

$$\begin{aligned} r_1 &= \lambda M^{-1} h_1 \\ r_2 &= \lambda M^{-1} h_2 \\ r_3 &= r_1 \times r_2 \\ t &= \lambda M^{-1} h_3 \end{aligned}$$

Здесь масштабный коэффициент определен исходя из условия ортонормальности $\lambda = 1/\|M^{-1}h_1\|$.

При этом необходимо соблюдать осторожность, т.к. при использовании реальных данных и, размещении r-векторов вместе ($R = [r_1 \ r_2 \ r_3]$), нет никакой гарантии того, что матрица вращения будет точной, т.е. будет ли выполняться условие $R^T R = R R^T = I$.

Для того чтобы обойти данную проблему, необходимо использовать сингулярное разложение (SVD) матрицы R. Как уже было сказано в главе 3, SVD - это метод разложения матрицы на две ортонормированные матрицы U и V, и на прямоугольную диагональную матрицу D. Это позволяет выполнить преобразование R в $R = UDV^T$. Т.к. R - это ортонормированная матрица, то матрица D должна быть единичной матрицей I, поэтому $R = UDV^T$. Таким образом, можно "принудительно" преобразовать

вычисленную матрицу R в другую матрицу вращения, применив сингулярное разложение к R, тем самым преобразовав её D в единичную матрицу; последующее умножение вновь на SVD дает новую матрицу, соответствующую матрице вращения R'.

Несмотря на всю проделанную работу, все ещё не были рассмотрены искажения линзы. Использование внутренних параметров камеры, найденных ранее - совместно с параметрами искажения, установленных в 0 - для начального приближения, дает толчок к началу решения большой системы уравнений.

Точки, "воспринимаемые" на изображении, действительно имеют неверное положение вследствие искажений. Пусть (x_p, y_p) будет соответствовать положению точки при условии, что отверстие камеры идеально, и пусть (x_d, y_d) это искаженное положение той же точки; тогда:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x X^w / Z^w + c_x \\ f_y X^w / Z^w + c_y \end{bmatrix}$$

При этом использование результатов калибровки без учета искажений, уравнение примет следующий вид:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2 (r^2 + 2x_d^2) \\ p_1 (r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix}$$

Для нахождения параметров искажения необходимо собрать список таких уравнений и решить их, в результате чего будут получены приблизительные внутренние и внешние параметры. Для выполнения всех этих сложных вычислений в OpenCV есть функция `cvCalibrateCamera2()`. (Как будет показано в главе 12, данная функция используется внутри функции стерео-калибровки. При стерео-калибровке, одновременно калибруются и соединяются вместе через матрицу вращения и вектор сдвига сразу две камеры).

Функция калибровки

После получения углов от нескольких изображений можно вызывать функцию `cvCalibrateCamera2()`. Эта функция производит математические вычисления и предоставляет необходимую информацию. В частности, в результате будет получена **матрица внутренних параметров камеры, коэффициенты искажения, вектор вращения и вектор перемещения**. Первые два измерения представляют внутренние параметры камеры, а последние два внешние (сообщают, где объект располагается и какая у него ориентация). Коэффициенты искажения (k_1, k_2, p_1, p_2 и k_3) (Третий компонент радиальных искажений идет последним, т.к. был добавлен в OpenCV после для получения наилучшей поправки для сильно искажающих линз типа "рыбий глаз" и должен быть использован только для таких случаев. В последующем будет показано,

что этот коэффициент может быть установлен в 0 при первой инициализации за счет флага `CV_CALIB_FIX_K3` - это коэффициенты радиальных и тангенциальных уравнений искажения, которые уже были рассмотрены ранее; за счет этих коэффициентов существует возможность исправить искажения. Матрица внутренних параметров камеры вызывает, скорее всего, наибольший интерес, т.к. именно она позволяет перейти от трехмерных координат к двумерным координатам изображения. Помимо этого, данную матрицу можно использовать для выполнения обратной операции, однако, для рассматриваемого случая можно вычислить только трехмерную линию, которой должна соответствовать точка данного изображения. В последующем данный момент будет рассмотрен более подробно.

Теперь пришло время рассмотреть непосредственно саму функцию калибровки камеры:

```
void cvCalibrateCamera2(
    CvMat* object_points
    ,CvMat* image_points
    ,int* point_counts
    ,CvSize image_size
    ,CvMat* intrinsic_matrix
    ,CvMat* distortion_coeffs
    ,CvMat* rotation_vectors = NULL
    ,CvMat* translation_vectors = NULL
    ,int flags = 0
);
```

Для получения верного результата функция `cvCalibrateCamera2()` вызывается с большим набором параметров. Почти все из них уже были рассмотрены ранее, и на данный момент уже должно было сложиться некоторое представление о них.

Первый аргумент `object_points` - это матрица $N \times 3$, содержащая материальные координаты каждой К точки каждого изображения М объекта (т.е. $N = K \times M$). Эти точки располагаются в системе координат объекта. Этот аргумент немного сложней, чем кажется, т.к. описание точек объекта будет неявно определяться материальными единицами и структурой системы координат. В случае с шахматной доской, например, можно определить координаты таким образом, чтобы все точки имели для z значение 0, а для x и y измерялись в сантиметрах. Если будут выбраны дюймы, то все вычисленные параметры (неявно) тоже будут в дюймах. Аналогичным образом, если установить для всех координат x (а не z) значение 0, то расположение шахматной доски относительно камеры будет в значительной степени вдоль оси x, а не z. Квадрат это одна единица, поэтому, если квадрат имеет длину стороны 90 мм, то мир камеры, объект и координаты единиц камеры будут иметь размерность 90/мм. В принципе, можно использовать любые другие объекты, нежели только шахматную доску, поэтому,

на самом деле, не нужно, чтобы все точки объекта лежали на плоскости, однако, это, как правило, самый простой способ калибровки камеры. В самом простом случае, необходимо просто определить каждый квадрат шахматной доски, который бы являлся "единицей" измерений, а координаты углов шахматной доски были бы целыми значениями углов строк и столбцов. S_{width} - количество квадратов по ширине, а S_{height} - количество квадратов по высоте:

$$(0,0), (0,1), (0,2), \dots, (1,0), (2,0), \dots, (1,1), \dots, (S_{\text{width}} - 1, S_{\text{height}} - 1)$$

Второй аргумент *image_points* - это матрица $N \times 2$, содержащая пиксельные координаты всех точек, находящихся в *object_points*. Если выполняется калибровка с использованием шахматной доски, то этот аргумент состоит просто из возвращаемых значений *M* при вызове *cvFindChessboardCorners()*, но уже перестроенных в несколько ином формате.

Аргумент *point_counts* указывает на число точек в каждом изображении и это матрица $M \times 1$. *image_size* - это размер, в пикселях, изображений из которых извлекаются точки (например, это изображения шахматной доски).

Следующие два аргумента *intrinsic_matrix* и *distortion_coeffs* представляют собой внутренние параметры камеры. Эти аргументы одновременно являются и выходными (их заполнение является основной причиной калибровки) и входными. Использование данных аргументов в качестве входных аргументов оказывает влияние на вычисляемый результат. Какая из этих матриц будет использована в качестве входного параметра, зависит от выбранного флага (будет показано чуть позже). Как уже было сказано ранее матрица внутренних параметров камеры полностью определяет идеальное поведение камеры, в то время как коэффициенты искажения характеризуют большую часть неидеального поведения камеры. Матрица внутренних параметров камеры всегда имеет размерность 3×3 , а коэффициентов искажения всегда пять, поэтому аргумент *distortion_coeffs* должен быть указателем на матрицу 5×1 (порядок записи k_1, k_2, p_1, p_2 и k_3).

В то время как предыдущие два аргумента включают в себя информацию о внутренних параметрах камеры, следующие два включают в себя информацию о внешних параметрах, т.е. информацию о положение объекта калибровки (например, шахматной доски) относительно камеры для каждого изображения. Положения объектов определяются вращением и перемещением. Вращения *rotation_vectors* определяются трехкомпонентными векторами *M*, расположенные в матрице $M \times 3$ (где *M* количество изображений). При этом стоит понимать, что это не матрица 3×3 , о которой ранее шла речь; в данном случае каждый вектор представляет собой ось в трехмерном пространстве системы камеры, вокруг которой вращается шахматная доска и где длина или величина вектора кодируется на угол поворота против часовой

стрелки. Каждый из векторов вращения может быть преобразован в матрицу вращения 3×3 при помощи функции `cvRodrigues2()`. Перемещения `translation_vectors` аналогичным образом располагаются в матрице $M \times 3$, в системе координат камеры. Как уже было сказано ранее, единицей системы координат камеры в точности те, что предполагались для шахматной доски. То есть, если квадрат шахматной доски 1 дюйм на 1 дюйм, то единицы в дюймах.

Определение параметров за счет оптимизации может показаться в некотором роде искусством. Иногда, попытка определить все параметры за раз может привести к неточным или противоречивым результатам, если первоначальная стартовая позиция в пространстве параметров далека от фактического решения. Таким образом, лучше использовать решение "подкрадывания" к хорошему показателю начальной позиции в несколько этапов. По этой причине зачастую некоторые параметры фиксируются, пока ищутся другие параметры, а затем вычисления производятся для фиксированных параметров, а не фиксированные становятся фиксированными и т.д. И в заключении, когда параметры становятся довольно таки близкими к фактическим, эти самые приближенные значения задействуются в решении в качестве отправной точки. OpenCV позволяет контролировать данный процесс за счет флагов. Аргумент `flags` предназначен для более точного управления процессом выполнения калибровки. Значения флага могут быть объединены вместе за счет использования логической операции OR.

`CV_CALIB_USE_INTRINSIC_GUESS`

Обычно матрица внутренних параметров вычисляется при помощи `cvCalibrateCamera2()` без дополнительной информации. В частности, начальные значения параметров c_x и c_y (центра изображения) берутся непосредственно из аргумента `image_size`. Если флаг установлен в данное значение, то предполагается, что `intrinsic_matrix` содержит действительные значения, которые будут использованы в качестве начального приближения для дальнейшей оптимизации в `cvCalibrateCamera2()`.

`CV_CALIB_FIX_PRINCIPAL_POINT`

Этот флаг может быть использован с и без `CV_CALIB_USE_INTRINSIC_GUESS`. Если без, то точка устанавливается, как правило, в центре изображения; если с, то точка устанавливается, как правило, на предполагаемое начальное значение `intrinsic_matrix`.

`CV_CALIB_FIX_ASPECT_RATIO`

Если установлен данный флаг, то процедура оптимизации одновременно изменяет только f_x и f_y и фиксирует их соотношение какой-либо величиной, установленной в `intrinsic_matrix` при вызове функции калибровки. (Если флаг

CV_CALIB_USE_INTRINSIC_GUESS не установлен, то значения f_x и f_y в *intrinsic_matrix* могут быть любыми произвольными значениями и только их соотношение будет считаться релевантным.)

CV_CALIB_FIX_FOCAL_LENGTH

Этот флаг задействует функцию оптимизации за счет использования f_x и f_y , переданных в *intrinsic_matrix*.

CV_CALIB_FIX_K1, *CV_CALIB_FIX_K2* и *CV_CALIB_FIX_K3*

Фиксирование параметров радиального искажения k_1 , k_2 и k_3 . Радиальные параметры могут быть установлены за счет любого совместного сочетания этих флагов. В общем, последний параметр должен быть зафиксирован в 0, только если не используется объектив типа "рыбий глаз".

CV_CALIB_ZERO_TANGENT_DIST

Этот флаг очень важен в случае калибровки высококачественных камер, которые в результате соблюдения точности при изготовлении имеют очень малые тангенциальные искажения. Попытки подгона параметров, близких к 0, могут привести к зашумлению ложным значениям и проблемам вычислительной устойчивости. Установка данного флага приводит к тому, что параметры тангенциального искажения P_1 и P_2 устанавливаются в 0.

Вычисление только внешних параметров

В некоторых случаях внутренние параметры уже имеются, следовательно, остаётся вычислить только положение рассматриваемого объекта (-ов). Этот сценарий несомненно отличается от обычной калибровки камеры и потому знания о выполнении данной задачи являются чрезвычайно полезными.

```
void cvFindExtrinsicCameraParams2(
    const CvMat* object_points
    ,const CvMat* image_points
    ,const CvMat* intrinsic_matrix
    ,const CvMat* distortion_coeffs
    ,CvMat* rotation_vector
    ,CvMat* translation_vector
);
```

Аргументы *cvFindExtrinsicCameraParams2()* идентичны соответствующим аргументам *cvCalibrateCamera2()* за исключением того, что матрица внутренних параметров и коэффициенты искажения подставляются, а не вычисляются. Функция возвращает вектор вращения *rotation_vector* 1×3 или 3×1 , который представляет трехмерную ось,

вокруг которой вращают шахматную доску или точки, а значение или длина вектора представляет из себя поворот на определенный угол против часовой стрелки. Этот вектор вращения может быть преобразован в матрицу вращения 3×3 при помощи `cvRodrigues2()`. Вектор перемещения является смещением координат камеры в первоначальное положение шахматной доски.

[П] | [РС] | (РП) Удаление искажений

Как уже было сказано ранее при калибровки камеры зачастую возникает необходимость в выполнении двух вещей. Во-первых, в корректировке эффектов искажения, во-вторых в построении трехмерного представления получаемых изображений. В данном разделе будет рассмотрена корректировка эффектов искажения, в то время, как решение второй задачи будет рассмотрено в главе 12.

OpenCV предоставляет готовый к использованию алгоритм удаления искажений, который принимает необработанное изображение и коэффициенты искажений от функции `cvCalibrateCamera2()` и выполняет исправления данного изображения (рисунок 11-12). Для того, чтобы воспользоваться данным алгоритмом необходимо воспользоваться функцией `cvUndistort2()`, которая выполняет все необходимое для одного кадра, или парой функций `cvInitUndistortMap()` и `cvRemap()`, которые позволяют обрабатывать некоторые вещи в видео немного более эффективно или ситуации, в которых имеется множество изображений от одной и той же камеры. (При этом стоит понимать разницу между исправлением искажений объектива и исправлением изображений относительно друг друга).



Рисунок 11-12. Изображение с камеры до удаления искажений (слева) и после удаления искажений (справа)

Основу метода составляет *карта искажений*, которая в последующем используется для исправления изображения. Функция `cvInitUndistortMap()` вычисляет карту искажений, а `cvRemap()` используется для применения карты искажений к произвольному изображению. Функция `cvUndistort2()` использует эти две функции при каждом вызове. Т.к. процесс вычисления карты искажений является трудоемкой операцией, то крайне не эффективно использовать `cvUndistort2()`, если карта

искажений остается не изменой. И, если имеется список двумерных точек, то необходимо использовать функцию *cvUndistortPoints()* для преобразования их первоначальных координат к неискаженным координатам.

```
// Удаление искажений с изображений
void cvInitUndistortMap(
    const CvMat*    intrinsic_matrix
    ,const CvMat*    distortion_coeffs
    ,cvArr*          mapx
    ,cvArr*          mapy
);

void cvUndistort2(
    const CvArr*    src
    ,CvArr*          dst
    ,const cvMat*    intrinsic_matrix
    ,const cvMat*    distortion_coeffs
);

// Удаление искажений двумерных точек
void cvUndistortPoints(
    const CvMat*    src
    ,CvMat*          dst
    ,const CvMat*    intrinsic_matrix
    ,const CvMat*    distortion_coeffs
    ,const CvMat*    R = 0
    ,const CvMat*    Mr = 0
);
```

Функция *cvInitUndistortMap()* вычисляет карту искажений, которая касается каждой точки изображения в месте, где данная точка находится на карте. Первые два аргумента - это матрица внутренних параметров камеры и коэффициенты искажения, получаемые от *cvCalibrateCamera2()*. В результате, карта искажений представляет из себя два отдельных 32-разрядных, одноканальных массива: первый предназначен для x-значений, а второй для y-значений. В данном случае может возникнуть вопрос, почему бы в таком случае не использовать один двухканальный массив. И причина вот в чём: результаты *cvUnitUndistortMap()* должны быть переданы непосредственно в *cvRemap()*.

Функция *cvUndistort2()* исполняет все эти действия в один проход. Она принимает начальное (искаженное) изображение, а также матрицу внутренних параметров и коэффициенты искажения камеры, а затем выводит неискаженное изображение того же размера. Как уже было сказано ранее, функция *cvUndistortPoints()* использует список координат двумерных точек исходного изображения и вычисляет связанные с ними неискаженные координаты. Так же эта функция имеет два дополнительных параметра, которые используются в случае со стерео зрением, о чём более подробно

будет рассказано в главе 12. Дополнительные параметры: матрица вращения между двумя камерами R и матрица внутренних параметров камеры Mr . Матрица Mr может иметь размеры 3×3 или 3×4 и формируется из первых трех или четырех столбцов возвращаемых `cvStereoRectify()` матриц $P1$ и $P2$ (для левой и правой камеры, глава 12). Эти параметры по умолчанию имеют значение NULL, что функция интерпретирует как единичная матрица.

[П]|[РС]|(РП) Пример реализации калибровки

OK, пришло время отразить всю полученную информацию в одном небольшом примере. В данном разделе будет представлена программа, которая выполняет следующие задачи: она ищет шахматные доски тех размеров, что указал пользователь, подцепляет цельные изображения (т.е. те, на которых можно найти все углы шахматной доски) в количестве, которое запросил пользователь и вычисляет встроенные параметры камеры и коэффициенты искажения. В итоге, программа входит в режим отображения, в котором можно просмотреть неискаженную версию изображения с камеры (рисунок 11-1). При использовании данного алгоритма необходимо существенно изменять положение шахматной доски между успешными захватами. В противном случае, матрицы точек, используемые при вычислении параметров калибровки, могут образовывать поврежденную (недостаточного ранга) матрицу и в конечном итоге, либо будет получено плохое решение, либо решение и вовсе не будет найдено.

Пример 11-1. Чтение ширины и высоты шахматной доски, чтение и формирование коллекции запрашиваемого количества представлений и калибровка камеры

```
// calib.cpp
// Пример использования:
// calib board_w board_h number_of_views
//
// Нажатие 'p' - пауза/воспроизведение, ESC - выход
//
#include <cv.h>
#include <highgui.h>
#include <stdio.h>
#include <stdlib.h>

int      n_boards = 0;    // Исходный список
const int board_dt = 20;  // Ожидание 20 кадров на одно
                         // представление шахматной доски
int      board_w;
int      board_h;

int main(int argc, char* argv[]) {
    if(argc != 4) {
        printf("ERROR: Wrong number of input parameters\n");
        return -1;
    }

    board_w    = atoi(argv[1]);
    board_h    = atoi(argv[2]);
    n_boards   = atoi(argv[3]);
}
```

```

int      board_n     = board_w * board_h;
CvSize   board_sz    = cvSize( board_w, board_h );
CvCapture* capture    = cvCreateCameraCapture( 0 );
assert( capture );

cvNamedWindow( "Calibration" );

// Выделение памяти под хранилища
//
CvMat* image_points     = cvCreateMat( n_boards*board_n, 2, CV_32FC1 );
CvMat* object_points    = cvCreateMat( n_boards*board_n, 3, CV_32FC1 );
CvMat* point_counts     = cvCreateMat( n_boards, 1, CV_32SC1 );
CvMat* intrinsic_matrix = cvCreateMat( 3, 3, CV_32FC1 );
CvMat* distortion_coeffs = cvCreateMat( 5, 1, CV_32FC1 );

CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
int corner_count;
int successes = 0;
int step, frame = 0;
IplImage *image = cvQueryFrame( capture );
IplImage *gray_image = cvCreateImage(cvGetSize(image), 8, 1); // subpixel

// Захват углов представления в цикле пока не будет достигнуто n_boards
// успешно захваченных представлений (т.е. тех, у которых найдены
// все углы шахматной доски)
//
while(successes < n_boards) {
    // Пропуск board_dt кадров, предоставленных пользователем,
    // передвигающий шахматную доску
    //
    if(frame++ % board_dt == 0) {
        // Поиск углов шахматной доски
        //
        int found = cvFindChessboardCorners(
            image
            ,board_sz
            ,corners
            ,&corner_count
            ,CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS
        );

        // Субпиксельная точность для полученных углов
        //
        cvCvtColor( image, gray_image, CV_BGR2GRAY );
        cvFindCornerSubPix(
            gray_image
            ,corners
            ,corner_count
            ,cvSize(11,11)
            ,cvSize(-1,-1)
            ,cvTermCriteria( CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1 )
        );
    }
}

```

```

    // Отрисовка углов
    //
    cvDrawChessboardCorners(
        image
        ,board_sz
        ,corners
        ,corner_count
        ,found
    );
    cvShowImage( "Calibration", image );

    // При получении хорошего представления доски, добавить в коллекцию
    //
    if( corner_count == board_n ) {
        step = successes*board_n;
        for( int i = step, j = 0; j < board_n; ++i, ++j ) {
            CV_MAT_ELEM( *image_points, float, i, 0 ) = corners[j].x;
            CV_MAT_ELEM( *image_points, float, i, 1 ) = corners[j].y;
            CV_MAT_ELEM( *object_points, float, i, 0 ) = j/board_w;
            CV_MAT_ELEM( *object_points, float, i, 1 ) = j%board_w;
            CV_MAT_ELEM( *object_points, float, i, 2 ) = 0.0f;
        }
        CV_MAT_ELEM( *point_counts, int, successes, 0 ) = board_n;
        successes++;
    }
} // окончание пропуска board_dt между захватами доски

// Обработка нажатия клавиш паузы/воспроизведения и выхода
//
int c = cvWaitKey(15);

if( c == 'p' ){
    c = 0;
    while( c != 'p' && c != 27 ){
        c = cvWaitKey( 250 );
    }
}

if( c == 27 ) {
    return 0;
}

image = cvQueryFrame( capture ); // Получение следующего изображения
} // Окончание цикла захвата представлений

// Выделение памяти под матрицы соразмерных с найденным количеством
// шахматных досок
//
CvMat* object_points2 = cvCreateMat( successes*board_n, 3, CV_32FC1);
CvMat* image_points2 = cvCreateMat( successes*board_n, 2, CV_32FC1);
CvMat* point_counts2 = cvCreateMat( successes, 1, CV_32SC1);

// Перемещение точек в матрицы правильного размера

```

```

// Детали реализации представлены двумя циклами. Общая
// форма записи:
// image_points->rows = object_points->rows = \
// successes*board_n; point_counts->rows = successes;
//
for( int i = 0; i < successes*board_n; ++i ) {
    CV_MAT_ELEM( *image_points2, float, i, 0 ) =
        CV_MAT_ELEM( *image_points, float, i, 0 );
    CV_MAT_ELEM( *image_points2, float,i,1 ) =
        CV_MAT_ELEM( *image_points, float, i, 1 );
    CV_MAT_ELEM( *object_points2, float, i, 0 ) =
        CV_MAT_ELEM( *object_points, float, i, 0 );
    CV_MAT_ELEM( *object_points2, float, i, 1 ) =
        CV_MAT_ELEM( *object_points, float, i, 1 );
    CV_MAT_ELEM( *object_points2, float, i, 2 ) =
        CV_MAT_ELEM( *object_points, float, i, 2 );
}

// Все те же номера
//
for( int i = 0; i < successes; ++i ) {
    CV_MAT_ELEM( *point_counts2, int, i, 0 ) =
        CV_MAT_ELEM( *point_counts, int, i, 0 );
}

cvReleaseMat( &object_points );
cvReleaseMat( &image_points );
cvReleaseMat( &point_counts );

// Теперь определены все углы шахматной доски для данных точек.
// Матрица внутренних параметров должна быть проинициализирована
// таким образом, что расстояние между двумя фокусными расстояниями
// имело соотношение, равное 1.0
//
CV_MAT_ELEM( *intrinsic_matrix, float, 0, 0 ) = 1.0f;
CV_MAT_ELEM( *intrinsic_matrix, float, 1, 1 ) = 1.0f;

// Калибровка камера
//
cvCalibrateCamera2(
    object_points2
    ,image_points2
    ,point_counts2
    ,cvGetSize( image )
    ,intrinsic_matrix
    ,distortion_coeffs
    ,NULL
    ,NULL
    ,0 // CV_CALIB_FIX_ASPECT_RATIO
);

// Сохранение внутренних параметров и коэффициентов
// искажения

```

```

//  

cvSave( "Intrinsics.xml", intrinsic_matrix );  

cvSave( "Distortion.xml", distortion_coeffs );  

// Пример загрузки сохраненных параметров  

//  

CvMat *intrinsic = (CvMat*)cvLoad( "Intrinsics.xml" );  

CvMat *distortion = (CvMat*)cvLoad( "Distortion.xml" );  

// Построение карты искажений, которая будет использована для  

// для всей последовательности кадров  

//  

IplImage* mapx = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );  

IplImage* mapy = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );  

cvInitUndistortMap(  

    intrinsic  

    ,distortion  

    ,mapx  

    ,mapy  

);  

// При запуске камеры, сразу отображается необработанное изображение  

//  

cvNamedWindow( "Undistort" );  

while( image ) {  

    IplImage *t = cvCloneImage(image);  

    // Отображение необработанного изображения  

//  

    cvShowImage( "Calibration", image );  

    // Удаление искажений с изображения  

//  

    cvRemap( t, image, mapx, mapy );  

    cvReleaseImage( &t );  

    // Отображение откорректированного изображения  

//  

    cvShowImage( "Undistort", image );  

    // Обработка клавиш паузы/воспроизведения и выхода  

//  

    int c = cvWaitKey( 15 );  

    if( c == 'p' ) {  

        c = 0;  

        while( c != 'p' && c != 27 ) {  

            c = cvWaitKey( 250 );
        }
    }
}

if( c == 27 ) {

```

```
        break;
    }

    image = cvQueryFrame( capture );
}

return 0;
}
```

[П] | [РС] | (РП) Преобразования Rodrigues

При работе с трехмерным пространством, наиболее часто встречается задача представления матрицы вращения 3x3 в этом пространстве. Это представление, как правило, наиболее удобно, т.к. перемножение вектора и данной матрицы эквивалентно повороту вектора. Недостатком является то, что может быть затруднительно интуитивно понять, как вращать матрицу 3x3. В альтернативе несколько проще визуализировать (в данном контексте "проще" не только человеку; вращение в трехмерном пространстве представляется только тремя компонентами, поэтому более эффективно обрабатывать три компонента представления Rodrigues, чем девять компонент матрицы вращения 3x3) представление для вращения в виде вектора, для которого вся работа сводится к оперированию одним углом. В этом случае стандартная практика использовать только один вектор для управления кодированием направления оси, вокруг которой будут производиться вращения, и его размер для кодирования количества вращений против часовой стрелки. Это легко выполнимо, т.к. направление может быть одинаково хорошо представлено вектором любого масштаба; следовательно, можно выбрать величину вектора равную величине поворота. Связи между этими двумя представлениями, матрицей и вектором, охвачены преобразованием Rodrigues. Пусть r это трехмерный вектор $r = [r_x \ r_y \ r_z]$; этот вектор неявно определяет θ , величину поворота длины (или величины) r . В результате можно преобразовать представление оси величин в матрицу вращения R следующим образом:

$$R = \cos(\theta) \cdot I + (1 - \cos(\theta)) \cdot rr^T + \sin(\theta) \cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix}$$

Так же возможно произвести обратную операцию (перейти от матрицы вращения к представлению оси величин) следующим образом:

$$\sin(\theta) \cdot \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix} = \frac{(R - R^T)}{2}$$

Таким образом, все сводится к ситуации, в которой имеется одно представление (представление матрицы), которое наиболее удобно использовать для расчетов, и ещё одно представление (представление Rodrigues), которое немного проще для восприятия мозгом. OpenCV предоставляет функцию для преобразования из любого представления в любое другое представление.

```
void cvRodrigues2(  
    const CvMat*   src  
    , CvMat*       dst  
    , CvMat*       jacobian = NULL  
) ;
```

Например, имеется вектор r и соответствующее необходимое представление матрицы вращения R ; src будет соответствовать вектору r размера 3×1 , а dst матрице вращения R . При этом значения src и dst можно поменять местами. В любом случае, $cvRodrigues2()$ предоставит верный результат. Последний аргумент необязателен. Если $jacobian \neq NULL$, то это должен быть указатель на матрицу размера 3×9 или 9×3 , которая будет заполнена частными производными компонентов конечного массива по отношению к компонентам исходного массива. Выходной аргумент $jacobian$ в основном используется для внутренних алгоритмов оптимизации в $cvFindExtrinsicCameraParameters2()$ и $cvCalibrateCamera2()$; в основном использование данного параметра ограничивается преобразованием выходов $cvFindExtrinsicCameraParameters2()$ и $cvCalibrateCamera2()$ из векторов Rodrigues ось-угол размера 1×3 или 3×1 в матрицы вращения. Для этого, необходимо просто оставить $jacobian = NULL$.

[П]|[РС]|(РП) Упражнения

1. Используйте рисунок 11-2 для получения уравнения
$$x = f_x \cdot (X/Z) + c_x \text{ and } y = f_y \cdot (Y/Z) + c_y$$
за счет подобия треугольников со смещенным центром.
2. Влияет ли ошибочная оценка правильного положения цента (c_x, c_y) на оценку других параметров, таких как фокус?

Подсказка: Обратите внимание на уравнение $q = MQ$.
3. Нарисуйте изображение квадрата:
 - a. Под радиальными искажениями.
 - b. Под тангенциальными искажениями.
 - c. Под обоими искажениями.
4. Обратитесь к рисунку 11-13. Для перспективных представлений, объясните следующее:
 - a. Откуда начинается "линия в бесконечность"?
 - b. Почему параллельные точки на плоскости объекта сходятся в точке на плоскости изображения?
 - c. Пусть объект и изображение плоскости перпендикулярны друг к другу. На плоскости объекта, начиная от точки P_1 , передвиньтесь на 10 единиц непосредственно от плоскости изображения в точку P_2 . Чему соответствует расстояние передвижения на плоскости изображения?
5. На рисунке 11-3 представлено наружно выпуклый эффект радиального искажения "бочка", что особенно заметно в левой части рисунка 11-12. Могут ли какие-либо объективы порождать внутренне выпуклый эффект? Как это возможно?
6. Используя дешевую веб-камеру или камеру мобильного телефона, создайте примеры радиального и тангенциального искажений в изображениях концентрических квадратов или шахматных досок.
7. Откалибруйте камеру из упражнения 6. Отобразите изображение с камеры до и после калибровки.

8. Поэкспериментируйте с вычислительной устойчивостью и шумом, собрав множество изображений шахматной доски и выполнив "хорошую" калибровку для них. Затем подумайте над тем, как изменить параметры калибровки для сокращения количества изображений шахматной доски. Отобразите результаты: параметры камеры в зависимости от количества изображений шахматной доски.

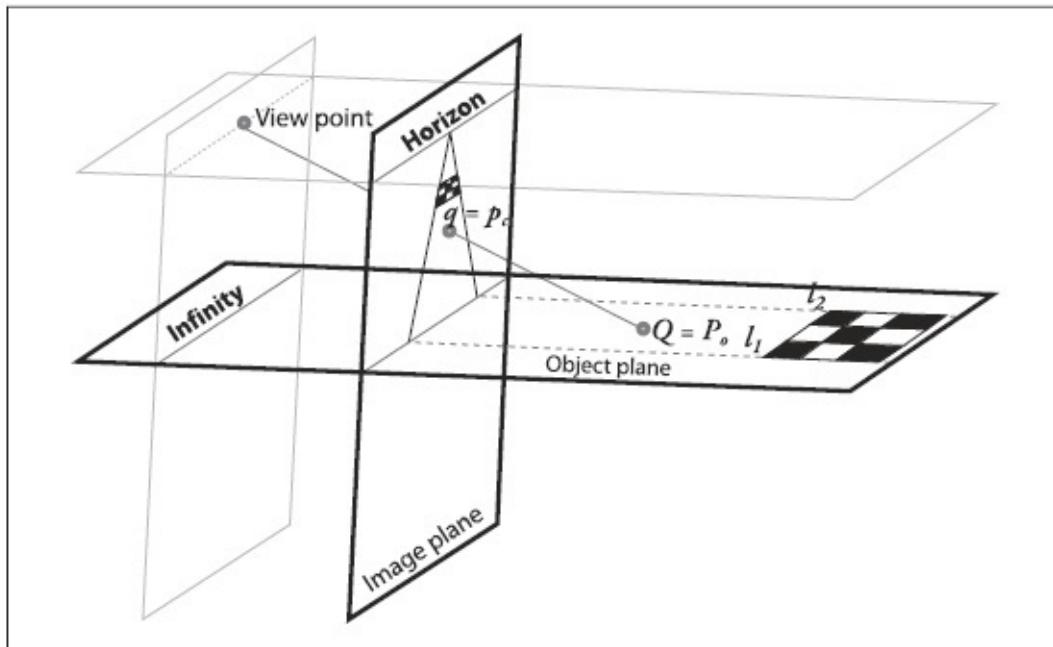


Рисунок 11-13. Диаграмма гомографии, показывающая пересечение плоскости объекта с плоскостью изображения и точку, представляющую центр проекции

9. Как изменяются параметры калибровки из упражнения 8 при использовании (например) 10 изображений шахматной доски размера 3x5, 4x6 и 5x7? Отобразите результаты.
10. Высококачественные камеры, как правило, имеют систему линз, которые физически исправляют искажения на изображениях. Что может произойти, если все-таки использовать multiterm (!)много элементную(!) модель искажений для такой камеры?
- Подсказка: это состояние известно, как переобучение.
11. *Трюк с трехмерным джойстиком.* Откалибруйте камеру. Используйте видео, где машут шахматной доской, и используйте `cvFindExtrinsicCameraParams2()` в качестве трехмерного джойстика. При этом помните, что `cvFindExtrinsicCameraParams2()` предоставляет поворот как 3x1 или 1x3 вектор вращения вокруг оси, где величина вектора представляет из себя вращение против часовой стрелки на угол совместно с перемещением трехмерного вектора.

- a. Выходная ось шахматной доски и угол вращения вместе в реальном времени - все это составляющие вращения доски вокруг оси. Поработайте со случаями, когда шахматная доска вне поля зрения.
- b. Используйте `cvRodrigues2()` для преобразования результата функции `cvFindExtrinsicCameraParams2()` из матрицы вращения 3×3 в вектор перемещения. Используйте полученный результат, чтобы оживить простую трехмерную фигуру самолета вновь отрисованную на изображении в реальном времени после перемещения доски в новое представление.

[П]|[РС]|(РП) Проекция и 3D Vision

Данная глава будет посвящена перемещениям в рамках трехмерного зрения, в первую очередь проекциям, а уже потом стереокамерам для восприятия глубины. Для выполнения всего этого будут необходимы знания, полученные в главе 11: матрица *внутренних параметров* камеры M , *коэффициенты искажения*, матрица вращения R , вектор перемещения T и в особенности *матрица гомографии* H .

Вначале будут рассмотрены проекция в трехмерном мире при использовании откалиброванной камеры и аффинные и проекционные преобразования (о которых уже шла речь в главе 6); затем будет рассмотрен пример получения обобщенного представления основной плоскости. Также будет рассмотрен алгоритм POSIT, который позволяет найти трехмерное представление (положение и поворот) известное как трехмерный объект на изображении.

Далее будет рассмотрена трехмерная геометрия множества изображений. В общем, нет надежного способа выполнить калибровку камеры или извлечь трехмерную информацию без множества изображений. Наиболее очевидный случай, в котором используется несколько изображений – это конструирование трехмерной сцены, *стерео зрение*. Особенностью стерео зрения является то, что два (или более) изображения, снятых в одно и то же время с отдельных камер, сопоставляются по соответствующим особенностям и анализируются различия между ними для получения информации о глубине. В остальных случаях, множество изображений используются для конструирования движений. В данном случае вполне достаточно иметь только одну камеру, при этом множество изображений, снятых в разное время и в разных местах. В первом случае наивысший интерес вызывают *эффекты несоответствия* (триангуляция) в качестве вычисления расстояния. В последнем вычисляется так называемая *фундаментальная матрица* (связывающая два различных представления вместе) в качестве источника понимания сцены.

[П]||[РС]||(РП) Проекции

После выполнения калибровки камеры (глава 11), появляется возможность однозначно спроектировать точки материального мира на изображение. Это означает, что засчёт координат положения точек материального мира, связанные с камерой, можно вычислить, где должна появиться на фотоприёмнике, в пикселях, внешняя трехмерная точка. В OpenCV данное преобразование может выполнить функция *cvProjectPoints2()*:

```
void cvProjectPoints2(
    const CvMat* object_points
    ,const CvMat* rotation_vector
    ,const CvMat* translation_vector
    ,const CvMat* intrinsic_matrix
    ,const CvMat* distortion_coeffs
    ,CvMat* image_points
    ,CvMat* dpdrot = NULL
    ,CvMat* dpdt = NULL
    ,CvMat* dpdf = NULL
    ,CvMat* dpdc = NULL
    ,CvMat* dpddist = NULL
    ,double aspectRatio = 0
);
```

На первый взгляд количество аргументов может быть немного пугающим, но на самом деле данная функция проста в использовании. Функция была разработана, чтобы приспособиться (очень обобщенно) к обстоятельствам, при которых проекционные точки должны расположиться на некотором твердом теле. В этом случае, естественно представить точки не просто как список положений в системе координат камеры, но и как список положения объекта в центре системы координат собственного тела; тогда можно добавить вращение и перемещение, указав отношение между координатами объекта и системой координат камеры. На самом деле, *cvProjectPoints2()* используется внутри *cvCalibrateCamera2()*. Все дополнительные аргументы предназначены, прежде всего, для *cvCalibrateCamera2()*, но опытные пользователи могут приспособить их и под свои нужды.

Первый аргумент *object_points* это список проецируемых точек, представляющий из себя матрицу Nx3 и содержащий положения точки. Положения этих точек могут быть предоставлены в системе координат объекта, с последующей передачей матриц 3x1 *rotation_vector* (как правило, представление Rodrigues) и *translation_vector*, связывающие две координаты. Если в конкретном случае удобнее работать непосредственно в системе координат камеры, то можно просто передать *object_points* в этой системе координат, а для *rotation_vector* и *translation_vector* установить все

значения в 0 (при этом стоит помнить, что вектор вращения является представлением поворота вида ось-угол, поэтому установка всех значений в 0 означает, что вектор имеет нулевую величину, соответственно "вращения нет").

intrinsic_matrix и *distortion_coeffs* - это матрица внутренних параметров и коэффициенты искажения соответственно, которые приходят от *cvCalibrateCamera2()* (глава 11). Аргумент *image_points* это матрица Nx2, в которую будет записан вычисленный результат.

И наконец, длинный список дополнительных аргументов *dprerot*, *dprdt*, *dprdf*, *dprdc* и *dprddist* - это все частные производные матрицы Jacobian. Эти матрицы связывают точки изображения с каждым из различных входных параметров. В частности: *dprerot* это матрица Nx3 частных производных точек изображения по отношению к вектору вращения; *dprdt* это матрица Nx3 частных производных точек изображения по отношению к вектору перемещения; *dprdf* это матрица Nx2 частных производных точек изображения по отношению к f_x и f_y ; *dprdc* это матрица Nx2 частных производных точек изображения по отношению к c_x и c_y ; *dprddist* это матрица Nx4 частных производных точек изображения по отношению к коэффициентам искажения. В большинстве случаев они не используются (значение NULL). Последний параметр *aspectRatio* также необязателен; он используется для производных только тогда, когда соотношение сторон фиксировано в *cvCalibrateCamera2()* или *cvStereoCalibrate()*. Если этот параметр не 0, то производные *dprdf* будут скорректированы.

[П]|[РС]|(РП) Аффинные и перспективные преобразования

Двумя наиболее используемыми преобразованиями являются аффинные и перспективные преобразования. Впервые об этом было сказано в главе 6. В зависимости от реализации соответствующих функций OpenCV оказывает влияние на список точек или на целое изображение, отображая точки, расположенные в одном месте, в другое место, при этом зачастую выполняя субпиксельную интерполяцию по пути. В результате аффинных преобразований из прямоугольников можно получить параллограммы любой формы, а в результате перспективных преобразований (более общих) из прямоугольника можно получить трапецию любой формы.

Перспективные преобразования тесно связаны с *перспективным проецированием*. При перспективном проецировании карта точек трехмерного материального мира проецируется на двумерную плоскость изображения вдоль набора проекционных линий, сходящихся в так называемом центре проецирования. Перспективные преобразования, которые являются разновидностью *гомографии*, связывают два различных изображения, которые являются альтернативными проекциями одного и того же трехмерного объекта на двух различных *проекционных плоскостях* (и, таким образом, невырожденные формы, такие как материальная плоскость пересекающихся трехмерных объектов, как правило, имеют два различных центра проекции).

Эти функции проекционно связанных преобразований подробно обсуждались в главе 6; для удобства все полученные знания обобщены в таблице 12-1.

Таблица 12-1. Функции аффинных и перспективных преобразований

Функция	Использование
cvTransform()	Аффинные преобразования списка точек
cvWarpAffine()	Аффинные преобразования целого изображения
cvGetAffineTransform()	Заполнение параметров матрицы аффинных преобразований
cv2DRotationMatrix()	Заполнение параметров матрицы аффинных преобразований
cvGetQuadrangleSubPix()	Низко затратные аффинные преобразования целого изображения
cvPerspectiveTransform()	Перспективные преобразования списка точек
cvWarpPerspective()	Перспективные преобразования целого изображения
cvGetPerspectiveTransform()	Заполнение параметров матрицы перспективных преобразований

Пример преобразования с высоты птичьего полета

Общей задачей по навигации в робототехнике, обычно используемой для планирования целей, является преобразование сцены взгляда камеры на роботе в представление с "высоты птичьего полета". На рисунке 12-1 представлено данное преобразование, которое впоследствии может быть покрыто альтернативным представлением мира, созданного при помощи сканирования лазерным дальномером. Далее, при использовании ранее полученных знаний, будет более детально показано, как использовать откалиброванную камеру для вычисления данного представления ("с высоты птичьего полёта").

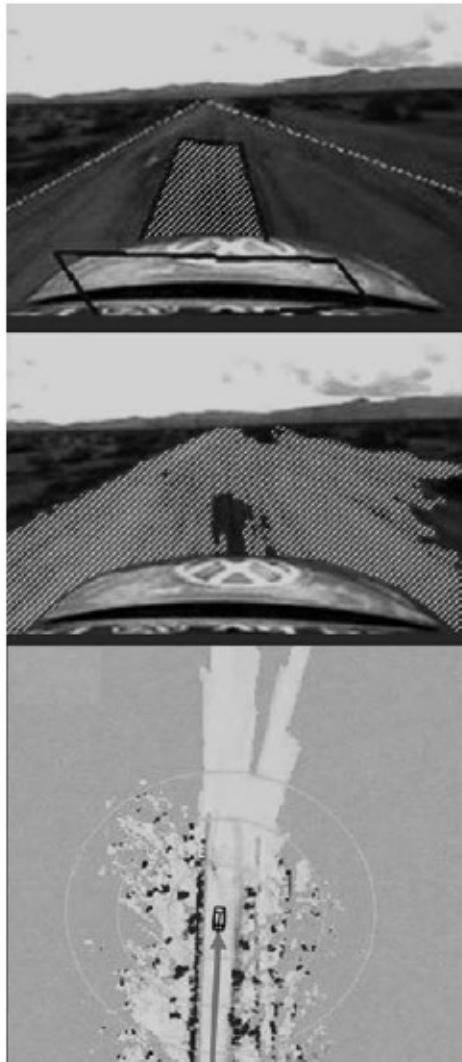


Рисунок 12-1. Представление с высоты птичьего полета: автомобильная камера смотрит на дорогу, где лазерный дальномер определяет область "дороги" в передней части автомобиля и выделяет участок прямоугольником (верхняя часть изображения); алгоритмы компьютерного зрения сегментируют данный участок (центральная часть изображения); сегментируемые участки дороги преобразуются в представление с высоты птичьего полёта, которое объединяется с представлением с высоты птичьего полета карты лазера (нижняя часть изображения)

Для получения представления с высоты птичьего полета (данная техника так же работает и для перспективных преобразований представлений любых плоскостей, например, стены или потолка, во фронтально параллельные представления) необходимы вычисленные в функции калибровки внутренние параметры камеры и матрицы искажений. Для разнообразия данные параметры будут браться из файла на диске. Доска, положенная на пол, будет использована для получения горизонтальной плоскости изображения робота-повоzки; в дальнейшем данное изображение преобразуется в изображение с высоты птичьего полета. Алгоритм работает следующим образом:

1. Чтение внутренних параметров и модели искажений для камеры.
2. Поиск известного объекта на горизонтальной плоскости (в данном случае на шахматной доске). Получение как минимум четырех точек с субпиксельной точностью.
3. Ввод найденных точек в `cvGetPerspectiveTransform()` (глава 6) для вычисления матрицы гомографии H представления горизонтальной плоскости.
4. Использование `cvWarpPerspective()` (глава 6) с флагами `CV_INTER_LINEAR + CV_WARP_INVERSE_MAP + CV_WARP_FILL_OUTLIERS` для получения представления фронтально параллельной горизонтальной плоскости.

В примере 12-1 показан весь процесс получения представления с высоты птичьего полета.

Пример 12-1. Представление с высоты птичьего полёта

```
// Вызов:
// birds-eye board_w board_h instrinsics distortion image_file
// Изменение высоты осуществляется при помощи клавиш:
// 'u' - отдалиться, 'd' - приблизиться, 'ESC' - выход
//
int main( int argc, char* argv[] ) {

    if( argc != 6 ) return -1;

    // Входные параметры:
    //
    int      board_w     = atoi( argv[1] );
    int      board_h     = atoi( argv[2] );
    int      board_n     = board_w * board_h;
    CvSize   board_sz    = cvSize( board_w, board_h );
    CvMat*  intrinsic   = (CvMat*)cvLoad(argv[3]);
    CvMat*  distortion   = (CvMat*)cvLoad(argv[4]);
    IplImage* image       = 0;
    IplImage* gray_image  = 0;

    if( (image = cvLoadImage(argv[5])) == 0 ) {
        printf("Error: Couldn't load %s\n", argv[5]);
        return -1;
    }

    gray_image = cvCreateImage( cvGetSize(image), 8, 1 );
    cvCvtColor( image, gray_image, CV_BGR2GRAY );

    // Удаление искажений
    //
    IplImage* mapx = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
    IplImage* mapy = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
```

```

// Инициализация матриц исправления
//
cvInitUndistortMap(
    intrinsic
    , distortion
    , mapx
    , mapy
);
IplImage *t = cvCloneImage(image);

// Исправление изображений
//
cvRemap( t, image, mapx, mapy );

// Получение плоскости шахматной доски
//
cvNamedWindow("Chessboard");
CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
int corner_count = 0;
int found = cvFindChessboardCorners(
    image
    ,board_sz
    ,corners
    ,&corner_count
    ,CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS
);

if(!found){
    printf("Couldn't acquire chessboard on %s, only found %d of %d corners\n"
        ,argv[5],corner_count,board_n
    );
    return -1;
}

// Субпиксельная коррекция найденных углов
//
cvFindCornerSubPix(
    gray_image
    ,corners
    ,corner_count
    ,cvSize(11,11)
    ,cvSize(-1,-1)
    ,cvTermCriteria( CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 30, 0.1 )
);

// Получение точек изображения и объекта:
// Выбор точек объекта на шахматной доске (r,c):
// (0,0), (board_w-1,0), (0,board_h-1), (board_w-1,board_h-1).
//
CvPoint2D32f objPts[4], imgPts[4];
objPts[0].x = 0;           objPts[0].y = 0;
objPts[1].x = board_w - 1; objPts[1].y = 0;

```

```

objPts[2].x = 0;           objPts[2].y = board_h - 1;
objPts[3].x = board_w - 1; objPts[3].y = board_h - 1;
imgPts[0] = corners[0];
imgPts[1] = corners[board_w - 1];
imgPts[2] = corners[(board_h - 1)*board_w];
imgPts[3] = corners[(board_h - 1)*board_w + board_w - 1];

// Отображене точек в порядке: B, G, R, YELLOW
//
cvCircle( image, cvPointFrom32f(imgPts[0]), 9, CV_RGB(0,0,255), 3 );
cvCircle( image, cvPointFrom32f(imgPts[1]), 9, CV_RGB(0,255,0), 3 );
cvCircle( image, cvPointFrom32f(imgPts[2]), 9, CV_RGB(255,0,0), 3 );
cvCircle( image, cvPointFrom32f(imgPts[3]), 9, CV_RGB(255,255,0), 3 );

// Отображение шахматной доски
//
cvDrawChessboardCorners(
    image
    ,board_sz
    ,corners
    ,corner_count
    ,found
);
cvShowImage( "Chessboard", image );

// Поиск матрицы гомографии
//
CvMat *H = cvCreateMat( 3, 3, CV_32F );
cvGetPerspectiveTransform( objPts, imgPts, H );

// Позволить пользователю регулировать высоту представления Z
//
float Z = 25;
int key = 0;
IplImage *birds_image = cvCloneImage(image);
cvNamedWindow("Birds_Eye");

// Цикл, позволяющий пользователю играться с высотой
//
// ESC - выход
//
while(key != 27) {
    // Установка высоты
    //
    CV_MAT_ELEM(*H, float, 2, 2) = Z;

    // Вычисление фронтально параллельного или с высоты птичьего
    // полета представления за счет использования матрицы гомографии
    //
    cvWarpPerspective(
        image
        ,birds_image
        ,H

```

```
    , CV_INTER_LINEAR | CV_WARP_INVERSE_MAP | CV_WARP_FILL_OUTLIERS
);
cvShowImage( "Birds_Eye", birds_image );

key = cvWaitKey();
if(key == 'u') Z += 0.5;
if(key == 'd') Z -= 0.5;
}

cvSave("H.xml", H); // Сохранение матрицы гомографии для повторного использования
return 0;
}
```

После получения матрицы гомографии и установки параметра высоты, можно убрать доску и приступить к перемещению повозки, тем самым делая видео пути с высоты птичьего полёта, однако, на данный момент выполнение данной задачи остаётся за читателем. На рисунке 12-2 слева показано исходное изображение, а справа с высоты птичьего полёта.

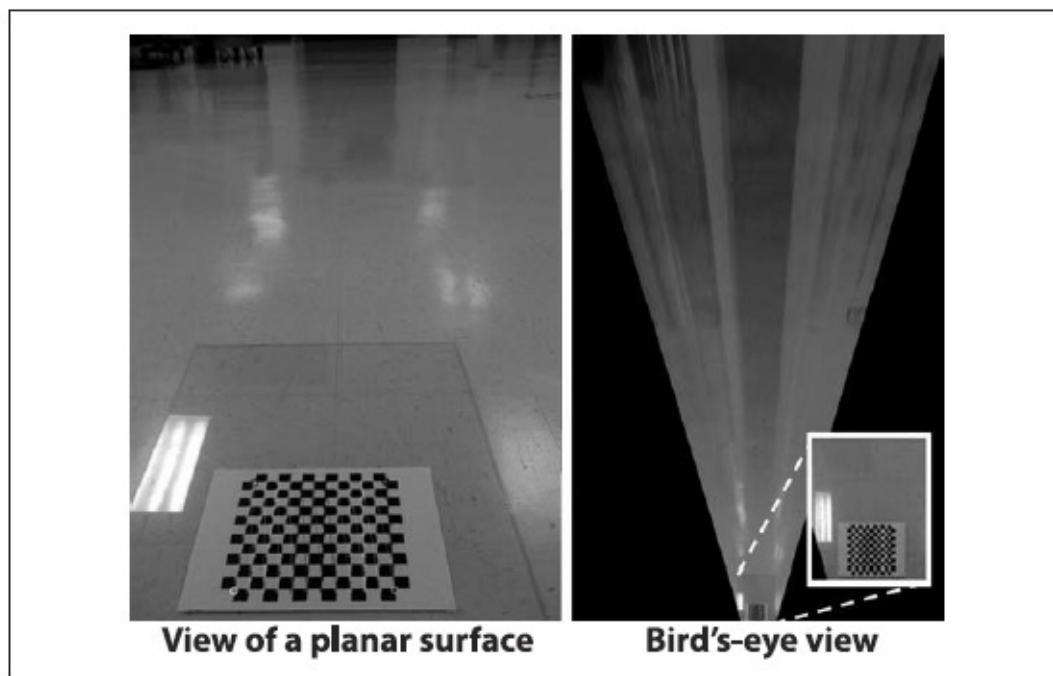


Рисунок 12-2. Пример представления с высоты птичьего полёта

[П]|[РС]|(РП) POSIT: оценивание трехмерного представления

Прежде, чем переходить к стерео зрению, необходимо рассмотреть довольно таки интересный алгоритм, который может оценивать положения известного трехмерного объекта. POSIT (или "Pose from Orthography and Scaling with Iteration") – это алгоритм, первоначально предложенный в 1992, для вычисления представления (положение T и ориентация R описываются шестью параметрами) трехмерного объекта, размеры которого точно известны. Для вычисления этого представления необходимо найти на изображении соответствующие места, по крайней мере, четыре некомпланарные точки на поверхности объекта. В первой части алгоритма, *pose from orthography and scaling*(POS), предполагается, что все точки объекта располагаются на одной и той же глубине и что изменения размера зависят только от расстояния до камеры. В этом случае существует закрытая форма решения для трехмерного представления объекта на основе масштабирования. Предположение, что точки объекта расположены на одной и той же глубине фактически означает, что объект находится достаточно далеко от камеры, поэтому можно пренебречь любыми внутренними разногласиями глубины в пределах объекта; это предположение более известно, как *слабо-перспективное приближение*.

Учитывая знания о внутренних параметрах камеры, можно найти перспективное масштабирование известного объекта и, следовательно, вычислить его приблизительное представление. Эти вычисления не будут особо точными, однако они помогут найти четыре точки, как если бы истинный трехмерный объект имел представление, вычисленное POS. В последующем эти точки будут входами для алгоритма POS. Процесс сходимости к истинному положению объекта, как правило, наступает в четыре или пять итераций – отсюда и название "итерационный алгоритм POS". При этом стоит не забывать, что внутренняя глубина объекта мала по сравнению с расстоянием до камеры. Если это предположение не верно, то алгоритм будет либо не сходиться, либо сходиться к "плохому представлению". Реализация алгоритма в OpenCV предполагает отслеживание более четырех (некомпланарных) точек объекта для улучшения точности оценки.

Алгоритм POSIT в OpenCV представлен тремя функциями: одна выделяет память под структуры данных положения отдельного объекта, одна освобождает память этой же структуры данных, и оставшаяся реализует сам алгоритм.

```
CvPOSITObject* cvCreatePOSITObject(
    CvPoint3D32f* points
    , int point_count
);

void cvReleasePOSITObject(
    CvPOSITObject** posit_object
);

```

Функция *cvCreatePOSITObject()* просто берет *points* (трехмерные точки) и *point_count* (целое число, указывающее на количество точек) и возвращает указатель на выделенную структуру POSIT объекта. *cvReleasePOSITObject()* принимает указатель на указатель структуры и освобождает занимаемую им память (в процессе указатель устанавливается в NULL).

```
void cvPOSIT(
    CvPOSITObject* posit_object
    , CvPoint2D32f* image_points
    , double focal_length
    , CvTermCriteria criteria
    , float* rotation_matrix
    , float* translation_vector
);

```

Стилистически список аргументов в функции *cvPOSIT()* несколько отличается от того, что был представлен многократно ранее, т.к. тут используется "старый стиль" аргументов, распространенный в старых версиях OpenCV (как было не трудно заметить, многие имена функций оканчивались на "2"; это в большинстве случаев связано с тем, что в новых версиях библиотеки меняется стилистика записи списка аргументов). Аргумент *posit_object* это указатель на POSIT объект, который необходимо отследить, а *image_points* это список положений соответствующих точек на плоскости изображения (при этом это 32-битные значения, позволяющие задать положения субпикселей). Текущая реализация *cvPOSIT()* предполагает передачу квадрата пикселей и тем самым передачу только одного параметра *focal_length* вместо двух параметров для направлений x и y. Именно поэтому *cvPOSIT()* является итерационным алгоритмом с критерием завершения *criteria* - критерием достижения довольно таки "хорошей подгонки". Последние два параметра *rotation_matrix* и *translation_vector* аналогичны таким же параметрам из предыдущих функций; при этом стоит отметить, что эти указатели имеют тип *float* и представляют из себя часть матриц, полученных в результате вызова (например) функции *cvCalibrateCamera2()*. В этом случае, полученная матрица M, может быть использована в качестве входного параметра для *cvPOSIT()* в виде *M->data.fl*.

При использовании POSIT, необходимо иметь в виду, что алгоритм не извлекает выгоду из дополнительных точек поверхности, компланарных с другими точками этой же поверхности. Любая точка, лежащая на поверхности и определяемая тремя другими точками, не будет приносить алгоритму пользы. На самом деле, дополнительные компланарные точки могут стать причиной вырождения, что причиняет ущерб производительности алгоритма. Дополнительные некомпланарные точки наоборот помогают алгоритму. На рисунке 12-3 представлен пример реализации алгоритма POSIT с использованием игрушечного самолета. Самолет имеет линиуметку, благодаря которой и происходит определение четырех некомпланарных точек. Эти точки передаются в `cvPOSIT()` и в результате `rotation_matrix` и `translation_vector` используются для управления симуляцией самолёта.

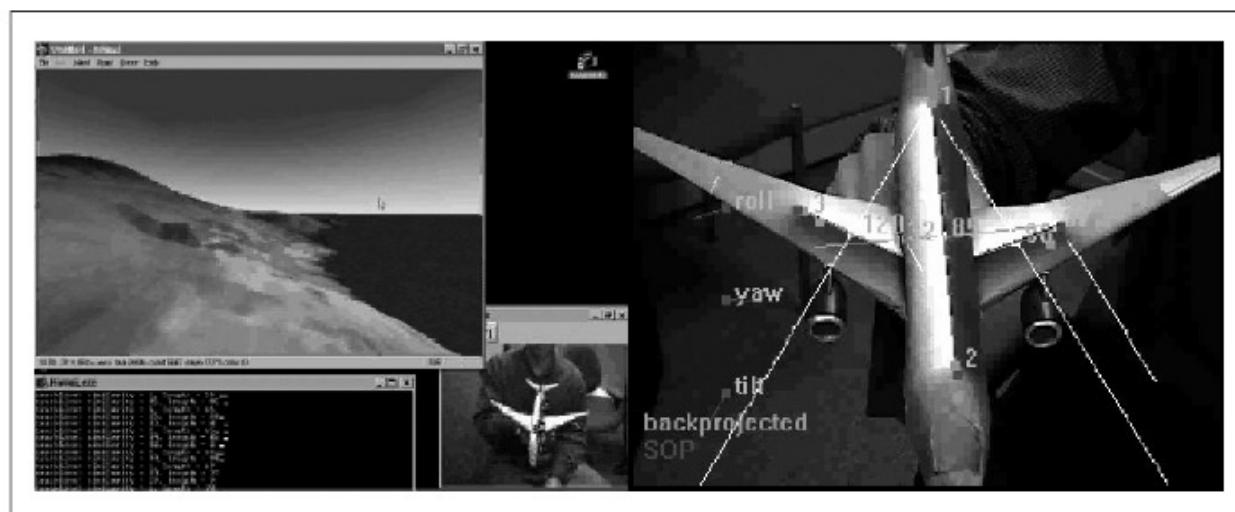


Рисунок 12-3. Использование алгоритма POSIT: четыре некомпланарные точки игрушки используются для управления симуляцией полёта

[П]|[РС]|(РП) Стерео зрение

Теперь все готово для рассмотрения *стерео зрения*. (В данном разделе будет рассмотрена только "верхушка айсберга". Для получения более подробной информации лучше всего обратиться к статьям: Trucco и Verri, Hartley и Zisserman, Forsyth и Ponce, Shapiro и Stockman.) Всем нам знакомы возможности стерео зрения, которые дают нам наши глаза. Возникает вопрос: до какой степени возможно подражать этой возможности в вычислительных системах? Компьютеры выполняют данную задачу за счет нахождения соответствия между точками, которые видны на одном фотоприёмнике и тем же точками на другом фотоприёмнике. За счет этого соответствия и заранее известного базового разделения между камерами, возможно вычислить трехмерное расположение точек. Хотя поиск соответствующих точек вычислительно дорогая операция, можно воспользоваться знаниями о геометрии системы для наиболее возможного сужения пространства поиска. На практике, стерео зрение реализуется в четыре этапа при помощи двух камер.

1. Математически удаляются радиальные и тангенциальные искажения объектива (глава 11). На выходе будет получено неискаженное изображение.
2. Настраиваются углы и расстояния между камерами, так называемый процесс *уточнения*. На выходе будут получены выровненные по строкам (это означает, что два изображения плоскости компланарны и что строки изображения будут выровнены точно, т.е. в одном направлении имеют одинаковую у-координату) и уточненные изображения.
3. Ищутся особенности на представлениях левой и правой камеры (каждый раз при упоминании левой и правой камеры также подразумевается вертикально ориентированные верхняя и нижняя камеры, где диспропорция возникает по у-координате, а не по x), так называемый процесс *сопоставления*. На выходе будет получена *карта несоответствий*, где различия будут соответствовать различиям в x-координате плоскости изображения для одного и того же рассматриваемого признака левой и правой камер: $x^l - x^r$.
4. Зная геометрическое расположение камер, развертывается карта несоответствий за счет *триангуляции*. Это так называемое *перепроектирование*, в результате чего получается карта глубины.

Для начала будет рассмотрен последний шаг, являющийся причиной появления первых трех.

Триангуляция

Пусть имеется совершенно неискаженная, выравненная и измеренная стерео установка, показанная на рисунке 12-4: две камеры, чьи плоскости изображения компланарны с точно параллельными оптическими осями (оптическая ось - это луч, выходящий из центра проекции O и проходящий через основные точки c , более известный как *основной луч*), для которых известно расстояние между ними, и с одинаковыми фокальными расстояниями $f_l = f_r$. Кроме того, пусть *основные точки* c_x^{left} и c_x^{right} уже откалиброваны, чтобы иметь те же координаты пикселей, что и соответствующие левому и правому изображениям. При этом ни в коем случае не нужно путать основные точки с центром изображения. Основной точкой является та, что соответствует пересечению основного луча и плоскости изображения. Это пересечение зависит от оси объектива. Как уже было показано в главе 11, плоскость изображения крайне редко выравнивается именно с объективом, поэтому почти всегда центр фотоприёмника не совпадает с основной точкой.

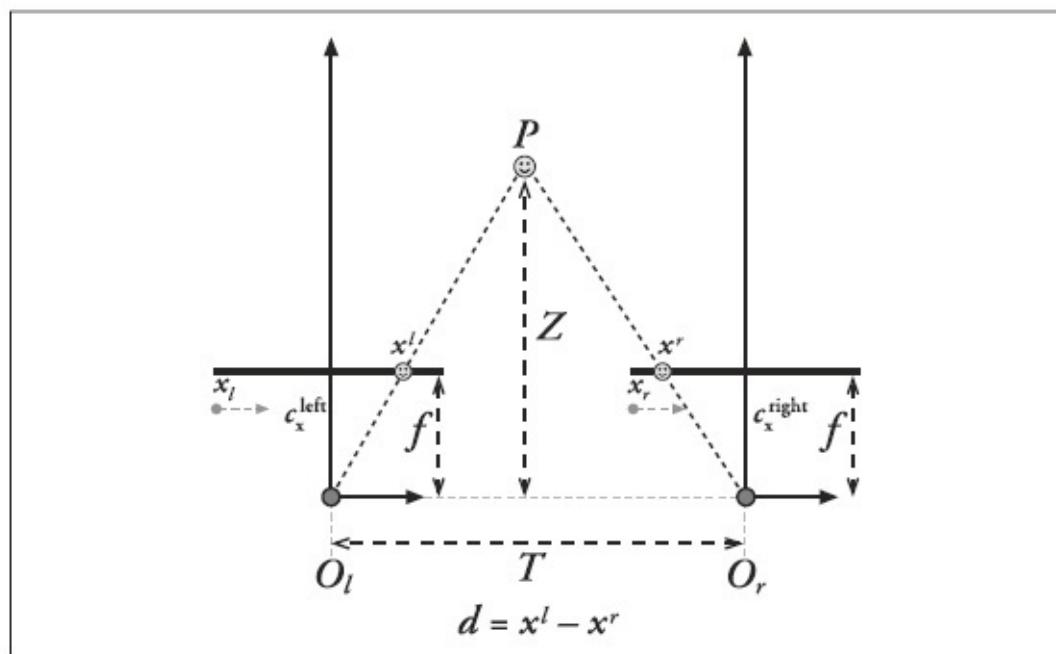


Рисунок 12-4. Совершенно неискаженная, выравненная стерео установка с известными соответствиями, глубина Z может быть найдена при помощи подобия треугольников; основные лучи начинаются от центра проекции O_l и O_r и проходят через основные точки двух плоскостей изображения c_l и c_r .

Теперь пусть строки изображений выровнены, т.е. пиксель строки одной камеры выровнен в соответствии с пикселями строки другой камеры. Таким образом, такая камера именуется *фронтально параллельно расположенной*. Так же пусть имеется возможность найти точку P материального мира на левом и правом изображениях p_l и p_r , которая имеет соответствующие горизонтальные координаты x^l и x^r .

Для данного упрощенного случая x^l и x^r - это горизонтальные позиции точек левого и правого фотоприёмника (соответственно), позволяющие показать, что глубина обратно пропорциональна различию между этими представлениями, где различие определяется довольно таки просто $d = x^l - x^r$. Данный случай показан на рисунке 12-4 и благодаря которому можно с легкостью вывести глубину Z из подобия треугольников. Ссылаясь на рисунок, имеем:

$$\frac{T - (x^l - x^r)}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{fT}{x^l - x^r}$$

(Эта формула основывается на основных лучах, пересекающихся в бесконечности. Однако, как будет показано в следующем разделе, стерео исправления получаются по отношению к основным точкам c_x^{left} и c_x^{right} . Для данной формулы основные лучи пересекаются в бесконечности, а основные точки имеют одинаковые координаты. Тем не менее, если основные лучи пересекаются на конечном расстоянии, то основные точки имеют разные координаты и потому уравнение глубины принимает следующий вид $Z = fT / (d - (c_x^{\text{left}} - c_x^{\text{right}}))$.)

Так как глубина обратно пропорциональна d , то очевидна нелинейная связь между этими двумя элементами. Когда d близко к 0, то небольшие различия приводят к большим различиям глубины. Когда d велико, то небольшие различия не сильно изменяют глубину. Как следствие из всего этого, системы стерео зрения имеют высокое разрешение глубины только в случае объектов, расположенных неподалеку от камеры (рисунок 12-5).

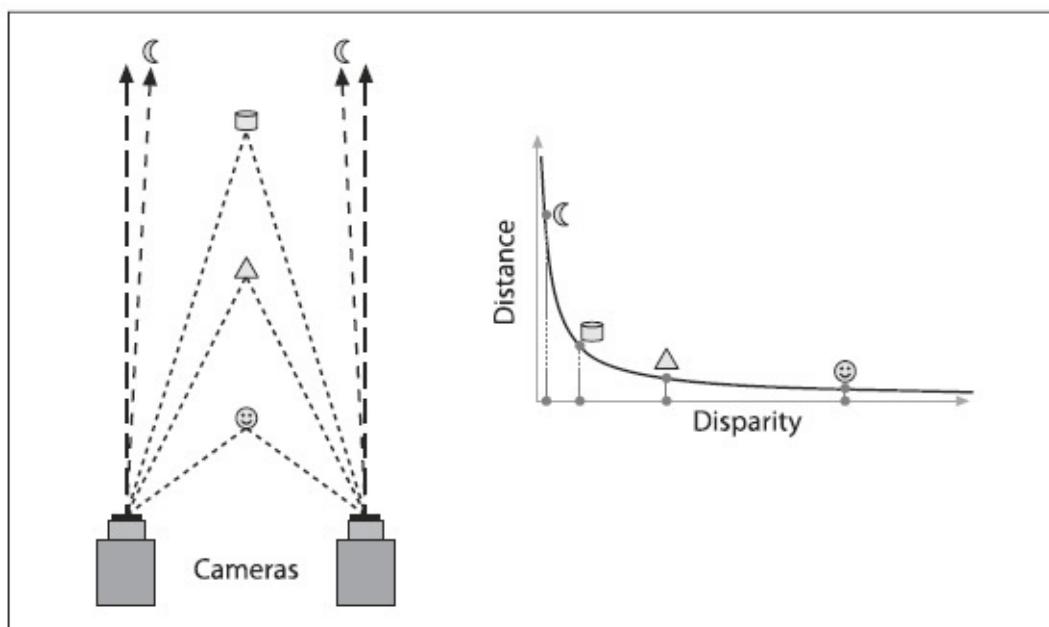


Рисунок 12-5. Глубина и различия связаны обратно пропорционально, поэтому измерения глубины ограничиваются близлежащими объектами

До этого уже были показаны многие системы координат в рамках обсуждения процесса калибровки в главе 11. На рисунке 12-6 показаны двух и трех мерные системы координат, используемые в OpenCV для стерео зрения. При этом стоит обратить внимание на тот факт, что это правые системы координат: если направить указательный палец правой руки вдоль оси X, а средний палец правой руки согнуть в направлении оси Y, то большой палец будет указывать в направлении основного луча. Пиксели левого и правого изображений фотоприёмников представлены на рисунке в верхнем левом углу и обозначаются координатами (x_l, y_l) и (x_r, y_r) , соответственно. Центры проекции O_l и O_r с основными лучами пересекают плоскость изображения в основной точке (не в центре) (c_x, c_y) . После математических исправлений, камеры становятся выровненными по строкам (компланарными и горизонтально выровненными) и отдаленными друг от друга на расстояние T и с фокусным расстоянием f .

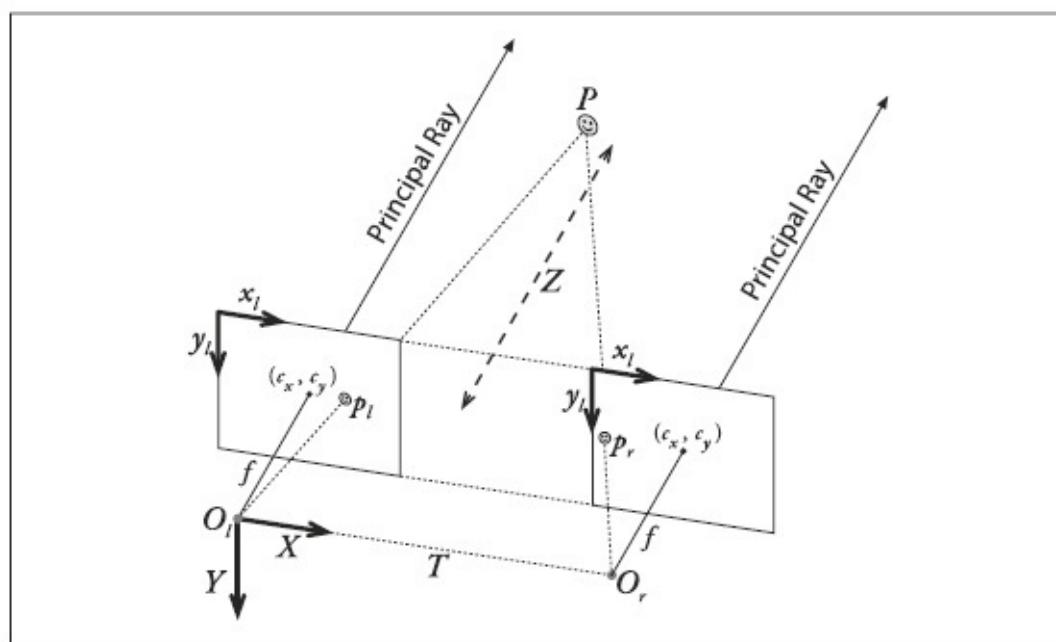


Рисунок 12-6. Системы координат, используемые в OpenCV для неискаженных выпрямленных камер: координаты пикселей относительно верхнего левого угла изображения и две выровненные плоскости; координаты камеры относительно центра проекции левой камеры

При таком расположении относительно легко найти расстояние. Теперь необходимо приложить немного усилий для понимания того, как сопоставить настройки геометрии камеры в реальном времени. В реальном мире камеры почти никогда не выровнены точно фронтально параллельно (рисунок 12-4). Вместо этого, необходимо математически рассчитать проекции изображения и карты искажений, которые поправят левое и правое изображения до фронтально параллельного расположения. При проектировании стерео устройства лучше всего расположить камеры примерно фронтально параллельно и, насколько это возможно, горизонтально ровно.

Физические преобразования благоприятно сказываются на математических вычислениях. Если камеры не выровнены хотя бы приблизительно, то в результате математического выравнивания может быть получено крайне искаженное изображение, тем самым снизиться или пропадет зона стерео перекрытия в конечном изображении. Для получения хорошего результата так же необходимо синхронизировать камеры. Если это не выполняется, то будут возникать проблемы при появлении движений в сцене (в том числе самих камер).

Рисунок 12-7 отражает реальную ситуацию между двумя камерами и необходимость математического выравнивания. Для выполнения математического выравнивания необходима в большей степени информация о геометрии представлений сцен двух камер. После получения данной геометрии и некой терминологии и обозначений, можно вернуться к проблеме выравнивания.

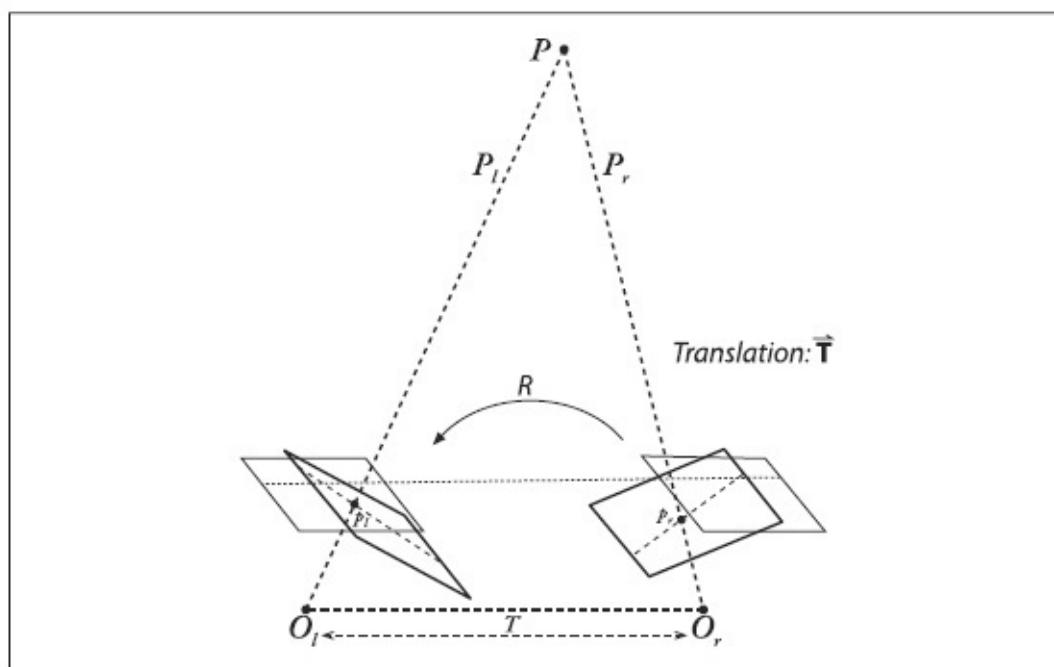


Рисунок 12-7. Целью данного изображения является показать, как математически (а не физически) совместить две камеры в одном представлении плоскости так, чтобы пиксели строк между камерами были в точности выровнены друг относительно друга

Эпиполярная геометрия

В основе геометрии системы компьютерного зрения лежит **эпиполярная геометрия**. В сущности, эта геометрия сочетает в себе две модели камеры обскуры (по одной для каждой камеры) и некоторые интересные новые точки, именуемые как **керновые точки** (рисунок 12-8). Перед объяснением того, чем хороши эти точки, будет рассмотрен процесс их определения и добавлена связанная с ними терминология. После всего

этого, будет кратко изложено общее понимание этой геометрии, а также значительно сужен набор точек, соответствующих двум камера姆. На практике это дополнение очень важное.

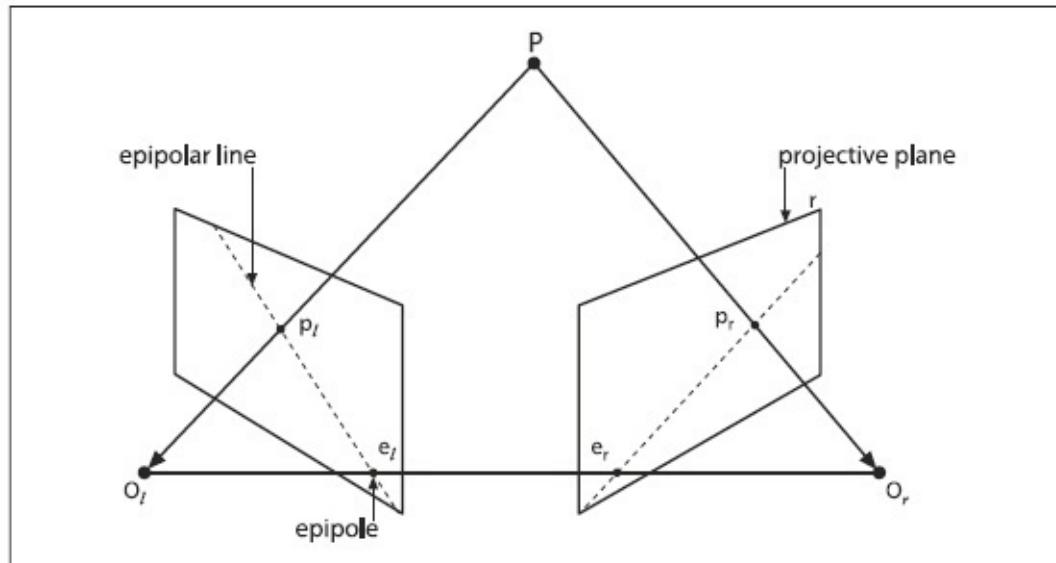


Рисунок 12-8. Эпиполярная плоскость определяется наблюдаемой точкой Р и двумя центрами проекции O_l и O_r ; керновые точки располагаются в точках пересечения линии, соединяющей центры проекции и две проекционные плоскости

Для каждой камеры имеется отдельный центр проекции O_l и O_r и пара соответствующих проекционных плоскостей Π_l и Π_r . Точка Р материального мира проецируется на каждую из проекционных плоскостей p_l и p_r . Наибольший интерес представляют новые точки – керновые точки. Керновая точка e_l (e_r) на плоскости Π_l (Π_r) определяется как центр проекции изображения другой камеры O_r (O_l). Плоскость пространства, образованная фактическими представлениями точки Р и двумя керновыми точками e_l и e_r (или, что эквивалентно, двумя центрами проекции O_r и O_l), называется эпиполярной плоскостью, а линии p_l e_l и p_r e_r (от точки проекции до соответствующей эпиполярной точки) эпиполярными линиями.

Для осознания того, насколько полезна эпиполярность, необходимо вспомнить следующий факт: на самом деле точка материального мира, проецирующаяся на правую (или левую) плоскость изображения, может быть расположена в любом месте вдоль всей линии точек луча, идущего от O_r через p_r (или от O_l через p_l), т.к. для одной из камер не известно расстояние до рассматриваемой точки. Например, имеется точка Р, которую видно на правой камере. Т.к. камера видит только p_r (проекцию Р на Π_r), по факту точка Р может быть расположена в любом месте на линии, образованной p_r и O_r . Очевидно, что эта линия содержит Р, но она так же содержит и другие точки. Однако, что интересно, как выглядит проекция этой линии на левой плоскости изображения Π_l ; по факту это эпиполярная линия, определяемая p_l и e_l . Изображение все возможных

точек, отображенных на одном фотоприёмнике, это *линия*, проходящая через соответствующую точку и керновую точку другого фотоприёмника.

Краткий список некоторых фактов об эпиполярной геометрии стереокамер (и почему этому удалено внимание).

- Каждая трехмерная точка представлений камер располагается на эпиполярной плоскости, которая пересекает каждое изображение по эпиполярной линии.
- Учитывая особенности одного изображения, соответствующее представление другого изображения должно лежать вдоль соответствующей эпиполярной линии. Это известно, как *эпиполярное ограничение*.
- Эпиполярное ограничение означает, что возможно преобразовать двумерный поиск для сопоставления особенностей двух фотоприёмников в одномерный поиск вдоль эпиполярных линий при условии наличия знаний о геометрии стереоустановки. Это не только огромное сокращение операций вычисления, но и отбрасывание из рассмотрения точек, которые могли бы привести к ложным соответствиям.
- Порядок сохранения. Если точки A и B видны на обоих изображениях и горизонтально попадают в указанном порядке на один фотоприёмник, то они горизонтально попадают в указанном порядке и на другой фотоприёмник. (Из-за преград и областей перекрытия представления, важно, чтобы обе камеры не видели одни и те же точки. Тем не менее, порядок должен поддерживаться. Так, если точки A, B и C располагаются слева направо на левом фотоприёмнике и если B не видно на правом фотоприёмнике из-за наличия перекрытия, то правый фотоприёмник по-прежнему должен видеть точки A и C слева направо.)

Существенная и фундаментальная матрицы

Осталось рассмотреть ещё два ингредиента, прежде чем переходить к функциям OpenCV, которые вычисляют эпиполярные линии. Это существенная матрица E и фундаментальная матрица F. Матрица E содержит информацию о перемещении и вращении двух связанных камер в материальном пространстве (рисунок 12-9), а F содержит такую же информацию, как и E плюс информацию о внутренних параметрах, относящихся к двум камерам в пиксельных координатах. Т.к. F содержит информацию о внутренних параметрах, то связь между двумя камерами выражается в пиксельных координатах.

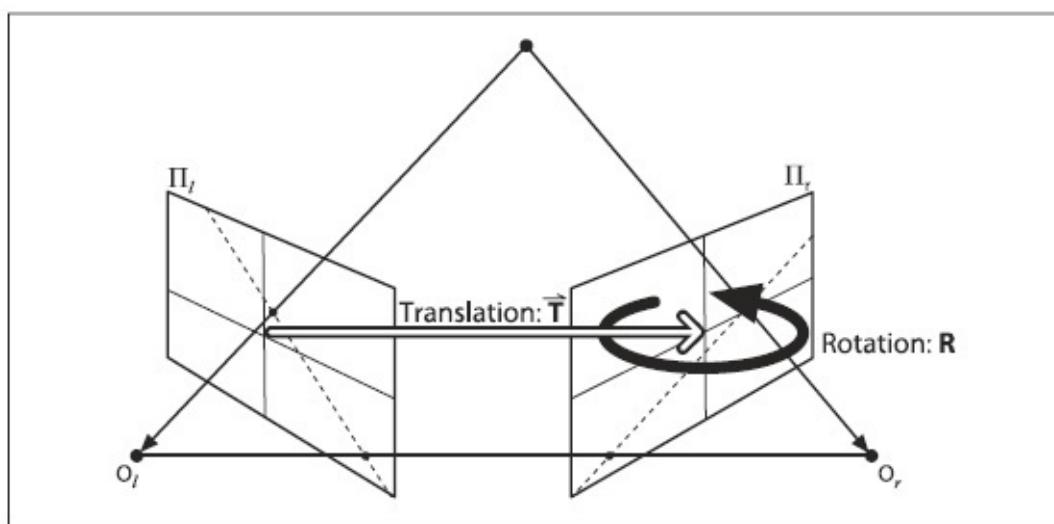


Рисунок 12-9. В основе геометрии стерео зрения лежит существенная матрица E , содержащая всю информацию о перемещении T и вращении R , при помощи которой в свою очередь можно описать положение второй камеры относительно первой в глобальных координатах

Существенная матрица E является чисто геометрической, при этом ничего не знающая о фотоприёмнике. Она сопоставляет расположения, в материальных координатах, точки P на левой камере с расположением той же точки на правой камере (т.е. связь P_l к P_r). Фундаментальная матрица F связывает точки плоскости изображения одной камеры в координатах изображения (в пикселях) с точками плоскости изображения другой камеры в координатах изображения (в дальнейшем будут использоваться обозначения q_l и q_r).

Математические действия над существенной матрицей

Пусть имеется точка P и необходимо получить соотношение, соединяющее отслеживаемые положения P_l и P_r точки P на двух фотоприёмниках. Эта связь служит оберткой определения существенной матрицы. Для начала будет рассмотрена связь между P_l и P_r , при этом материальное расположение точки будет рассматриваться в координатах двух камер. Эта связь может быть представлена при помощи эпиполярной геометрии, как уже было показано ранее. (При этом не стоит путать P_l и P_r , которые являются точками на проекционной плоскости изображения, с p_l и p_r , и которые являются положениями точки P в системе координат двух камер.)

Теперь необходимо выбрать один набор координат, левый или правый, для работы и выполнения расчетов. Пусть выбор падет на координаты сосредоточенные на O_l левой камере. Согласно этим координатам положение отслеживаемой точки будет P_l , а тогда начальное положение другой камеры будет располагаться на расстоянии T . Точка P , как видно на правой камере P_r , имеет координаты этой камеры, где $P_r = R(P_l)$

- Т). Ключевым шагом является введение эпиполярной плоскости, которая как уже известно связывает все эти вещи. Конечно возможно представить плоскость множеством способов, но для рассматриваемого случая необходимо вспомнить, что уравнение для всех точек \mathbf{x} на плоскости с нормальным вектором \mathbf{n} , проходящим через точку \mathbf{a} , необходимо соблюдать следующее ограничение:

$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0$$

Эпиполярная плоскость содержит векторы P_l и T ; таким образом, если есть перпендикулярный (векторное произведение векторов дает третий вектор, ортогональный к двум первым; направление определяется "правилом правой руки": если указательный палец соответствует направлению a , а средний палец направлению b , то векторное произведение ab перпендикулярно к a и b и соответствует направлению большого пальца) к обоим векторам вектор (например $P_l \times T$), то можно использовать его в \mathbf{n} уравнениях плоскости. Так, уравнение для всех возможных точек P_l проходящих через точку T и содержащей оба вектора имеет вид (за счет замены скалярного произведения на матричное перемножение транспонированного и нормального векторов):

$$(P_l - T)^T (T \times P_l) = 0$$

Не стоит забывать, что основная цель заключается в том, чтобы связать q_l и q_r для начала в соотношение P_l и P_r . Отрисовка P_r на рисунке осуществляется за счет равенства $P_r = R(P_l - T)$, которое для удобства лучше переписать следующим образом $(P_l - T) = R^{-1} P_r$. Выполнив подстановку и используя $R^T = R^{-1}$, уравнение примет следующий вид:

$$(R^T P_r)^T (T \times P_l) = 0$$

Векторное произведение можно переписать в (несколько громоздкое) перемножение матриц. Таким образом матрица S примет следующий вид:

$$T \times P_l = S P_l \Rightarrow S = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

После подстановки, уравнение примет следующий вид:

$$(P_r)^T R S P_l = 0$$

RS - это существенная матрица E , тогда уравнение принимает компактный вид:

$$(P_r)^T E P_l = 0$$

В результате получаем связь между точками, которые наблюдаются на фотоприёмниках, но это только первый шаг. За счет выполнения замены при помощи проекционных уравнений $p_l = f_l P_l / Z_l$ и $p_r = f_r P_r / Z_r$ и последующего разделения на $Z_l Z_r / f_l f_r$, уравнение примет окончательный вид:

$$p_r^T E p_l = 0$$

На первый взгляд может показаться, что уравнение определяется элементом p , если остальное известно, но E это обертка ранг-неполной матрицы (существенная матрица 3x3 ранга 2) и как следствие все заканчивается линейным уравнением. (Для квадратной $n \times n$ матрицы E , ранг неполная оценка означает, что имеется менее чем n ненулевых собственных значений. В результате система линейных уравнений, указанных в ранг-неполной матрице, не имеет единственного решения. Если ранг (число ненулевых собственных значений) $n-1$, то это будет строка, образованная набором точек, которые все без исключения удовлетворяют системе уравнений. Система определяется матрицей ранга $n-2$, образуя плоскость, и так далее.) Существенная матрица содержит пять параметров - три для вращения и два для направленного перемещения (без установки масштаба) - и два ограничения: (1) определитель равен 0, т.к. это ранг-неполная матрица (3x3 с рангом 2); (2) два ненулевых сингулярных разложения равны, потому что матрица S кососимметричная, а R это матрица вращения. В общей сложности это даёт семь ограничений. При этом, ещё раз, не стоит путать данную матрицу E с внутренними параметрами E камеры; таким образом, связанные точки выражаются в материальных координатах или координатах камеры, а не в пиксельных координатах.

Математические действия над фундаментальной матрицей

Матрица E содержит всю информацию о геометрии двух камер относительно друг друга, но не о самих камерах. На практике, наибольший интерес вызывают пиксельные координаты. Для поиска связи между пикселием одного изображения и соответствующей эпиполярной линией другого изображения, необходимо ознакомиться с внутренней информацией о двух камерах. Для того, чтобы это сделать необходимо подставить вместо p (в пиксельных координатах) q и матрицу внутренних параметров камеры. $q = Mp$ (где M - это матрица внутренних параметров камеры) или, что эквивалентно, $p = M^{-1} q$. В результате уравнение для E принимает следующий вид:

$$q_r^T (M_r^{-1})^T E M_l^{-1} q_l = 0$$

Это выглядит несколько беспорядочно, однако введение фундаментальной матрицы:

$$F = (M_r^{-1})^T E M_l^{-1}$$

устраняет этот беспорядок

$$q_r^T F q_l = 0$$

Вкратце: фундаментальная матрица F это существенна матрица E с той лишь разницей, что F работает с пиксельными координатами изображения, тогда как E работает с материальными координатами (при этом стоит обратить внимание на уравнение, связывающее фундаментальную и существенную матрицы; если выровнять изображение и нормализовать точки за счет деления на фокусное расстояние, то матрица внутренних параметров M становится единичной и тогда $F = E$). Матрицы E и F имеют ранг 2. Фундаментальная матрица содержит семь параметров, два для каждой эпиполярной точки и три для гомографии, которая связывает две плоскости изображения (как правило масштаб не влияет на четыре параметра).

То, как обрабатывает OpenCV всё, что было рассмотрено ранее

Можно вычислить F точно так же, как и гомографию изображения в предыдущем разделе за счет предоставления набора известных соответствий. В этом случае не требуется выполнять отдельную калибровку камеры, потому что можно найти решение непосредственно для F , которая неявно содержит фундаментальные матрицы обеих камер. Функция, выполняющая все эти вещи, именуется `cvFindFundamentalMat()`:

```
int cvFindFundamentalMat(
    const CvMat*  points1
    ,const CvMat*  points2
    ,CvMat*        fundamental_matrix
    ,int           method = CV_FM_RANSAC
    ,double        param1 = 1.0
    ,double        param2 = 0.99
    ,CvMat*        status = NULL
);
```

Первые два аргумента это вещественные указатели (одинарной точности) Nx2 или Nx3 матрицы, содержащие соответствующие собранные N точки (это так же могут быть $N-1$ многоканальные матрицы с двумя или тремя каналами). Результирующая матрица *fundamental_matrix* должна быть матрицей 3x3 той же точностью, что и точки (в частности размерность может быть 9x3).

Следующий аргумент определяет метод, который будет использоваться при вычислении фундаментальной матрицы по соответствующим точкам; может принимать одно из четырех значений. Для каждого значения есть определенное ограничение по количеству необходимых (или разрешенных) точек в *points1* и *points2*, как показано в таблице 12-2.

Таблица 12-2. Ограничения на аргумент метода

Значение метода	Количество точек	Алгоритм
CV_FM_7POINT	$N = 7$	Семи-точечный алгоритм
CV_FM_8POINT	$N \geq 8$	Восьми-точечный алгоритм
CV_FM_RANSAC	$N \geq 8$	Алгоритм RANSAC
CV_FM_LMEDS	$N \geq 8$	Алгоритм LMedS

Семи-точечный алгоритм задействует ровно семь точек, а также тот факт, что матрица F имеет ранг 2 для полного ограничения матрицы. Преимущество этого ограничения в том, что F всегда имеет ранг 2, поэтому не имеет очень малое собственное число, близкое к 0. Недостатком является то, что это ограничение не является абсолютно уникальным, и потому результирующая матрица может быть трех экземпляров (в этом случае *fundamental_matrix* должна быть матрицей 9x3, чтобы вместить все три варианта). Восьми-точечный алгоритм находит F в виде линейной системы уравнений. Если задействуются более 8 точек, то возникающие ошибки устраняются по методу наименьших квадратов совместно с минимизацией всех точек. Проблема семи-точечного и восьми-точечного алгоритмов в том, что они чрезвычайно чувствительны к выбросам (даже, если имеется более восьми точек в случае с восьми точечным алгоритмом). Алгоритмы RANSAC и LMedS как правило классифицируются как надежные методы, потому что они способны распознавать и удалять выбросы. Для каждого алгоритма важно иметь более восьми точек.

Следующие два аргумента - это параметры, используемые только RANSAC и LMedS. Первый *param1* - это максимальное расстояние от точки до эпиполярной линии (в пикселях), за пределами которой точка считается выбросом. Второй параметр *param2* является желаемым доверительным интервалом (значение между 0 и 1), который по существу сообщает алгоритму количество итераций.

Последний аргумент *status* не обязателен; если используется, то это должна быть матрица Nx1 типа CV_8UC1, где N соответствует *points1* и *points2*. Если эта матрица не NULL, то RANSAC и LMedS используют данную матрицу для хранения информации о том, какие точки являются выбросами, а какие нет. В частности, соответствующая запись будет установлена в 0, если точка является выбросом и в 1 в противном случае.

Возвращаемое значение - это целое число, указывающее на количество найденных матриц. Это будет 1 или 0 для всех случаев, кроме семи-точечного алгоритма, где также возможно значение 3. Значение 0 указывает на невозможность вычисления матрицы. Ниже представленный пример, взятый из руководства по OpenCV, поможет окончательно разобраться с этой функцией.

Пример 12-2. Вычисление фундаментальной матрицы при помощи RANSAC

```

int      point_count = 100;
CvMat*  points1;
CvMat*  points2;
CvMat*  status;
CvMat*  fundamental_matrix;

points1 = cvCreateMat( 1, point_count, CV_32FC2 );
points2 = cvCreateMat( 1, point_count, CV_32FC2 );
status  = cvCreateMat( 1, point_count, CV_8UC1 );

/* Заполнение точек ... */
for( int i = 0; i < point_count; i++ ) {
    points1->data.fl[i*2]   = <x1, i>;
    points1->data.fl[i*2+1] = <y1, i>;
    points2->data.fl[i*2]   = <x2, i>;
    points2->data.fl[i*2+1] = <y2, i>;
}

fundamental_matrix = cvCreateMat( 3, 3, CV_32FC1 );
int fm_count = cvFindFundamentalMat(
    points1
    ,points2
    ,fundamental_matrix
    ,CV_FM_RANSAC
    ,1.0
    ,0.99
    ,status
);

```

Пару слов предупреждений - связанных с возвращением 0 - алгоритмы могут быть провальными, если точки поставляются в *вырожденной форме*. Вырожденная форма возникает тогда, когда поставляемые точки предоставляют меньше информации, чем требуется, например, когда одна точка повторяется более одного раза или, когда несколько точек коллинеарны или компланарны с чересчур многими другими точками. Важно всегда проверять возвращаемое значение *cvFindFundamentalMat()*.

Вычисление эпиполярных линий

Теперь, имея фундаментальную матрицу, появляется возможность для вычисления эпиполярных линий. Функция OpenCV *cvComputeCorrespondEpilines()* вычисляет по списку точек одного изображения эпиполярную линию на другом изображении. При этом стоит отметить, что для любой заданной точки одного изображения существует соответствующая эпиполярная линия на другом изображении. Каждая вычисляемая линия кодируется в виде вектора из трех точек (a, b, c), который определяется следующим уравнением:

$$ax + by + c = 0$$

Для вычисления эпиполярных линий, функция запрашивает фундаментальную матрицу, которая вычисляется *cvFindFundamentalMat()*.

```
void cvComputeCorrespondEpilines(
    const CvMat*  points
    , int          which_image
    , const CvMat* fundamental_matrix
    , CvMat*       correspondent_lines
);

```

Первый аргумент *points*, как правило, Nx2 или Nx3 массив точек (который может быть Nx1 для многоканального массива с двумя или тремя каналами). Аргумент *which_image* может быть либо 1, либо 2 и указывать на то, какие точки определяются на изображении (относительно массивов *points1* и *points2* функции *cvFindFundamentalMat()*). *fundamental_matrix* это матрица 3x3, возвращаемая *cvFindFundamentalMatrix()*. *correspondent_lines* является массивом вещественных чисел Nx3, в который записывается результат. Легко заметить, что уравнение линии $a x + b y + c = 0$ не зависит от общей нормализации параметров a , b и c . По умолчанию они нормированы так, что $a^2 + b^2 = 1$.

Стерео калибровка

Уже было представлено достаточно теории и техник, касаюмо камер и трехмерных точек, для рассмотрения стерео калибровки (в этом разделе) и стерео исправления (в следующем разделе). *Стерео калибровка* - это процесс вычисления геометрической связи между двумя пространствами камер. В отличии от этого, *стерео исправления* - это процесс *исправления* отдельно взятых изображений так, чтобы казалось будто бы они взяты от двух камер с выравненными плоскостями изображения (рисунок 12-4 и 12-7). При таком исправлении, оптические оси (или основные лучи) двух камер параллельны и потому они пересекаются в бесконечности. Можно было бы конечно калибровать две камеры иначе, но в данном случае (и в OpenCV) используется более общий и простой случай установки основных лучей, пересекающихся в бесконечности.

Стерео калибровка зависит от матрицы вращения R и вектора перемещения T между двумя камерами, как показано на рисунке 12-9. И R и T вычисляются функцией *cvStereoCalibrate()*, которая схожа с *cvCalibrateCamera2()*, и которая была рассмотрена в главе 11, за исключением того факта, что теперь имеется две камеры и новая функция может вычислить (или использовать какие-либо предварительные вычисления) для камеры искажения существенной или фундаментальной матрицы. Другое отличие стерео калибровки от калибровки одной камеры в том, что в *cvCalibrateCamera2()* используется законченный список вращений и перемещений

между камерой и представлением шахматной доски. В случае с `cvStereoCalibrate()` ищется одна матрица вращения и один вектор перемещения, которые связывают правую и левую камеры.

Уже было показано, как вычислять существенную и фундаментальную матрицы, однако не было показано как вычислять R и T, связывающие левую и правую камеры. Для любой заданной трехмерной точки P в координатах объекта можно отдельно использовать единичную калибровку двух камер для перевода P в координаты камеры $P_l = R_l P + T_l$ и $P_r = R_r P + T_r$ для левой и правой камер, соответственно. Из рисунка 12-9 так же видно, что два представления P (от двух камер) можно связать уравнением $P_l = R^T(P_r - T)$, где R и T матрица вращения и вектор перемещения между камерами, соответственно. Имея эти три уравнения получаем решение для матрицы вращения и вектора перемещения по отдельности:

$$R = R_r(R_l)^T$$

$$T = T_r - RT_l$$

Имея множество совместных представлений углов шахматной доски, `cvStereoCalibrate()` использует `cvCalibrateCamera2()` чтобы найти решение для параметров матрицы вращения и вектора перемещения представления шахматной доски для каждой камеры в отдельности (глава 11, раздел "Что под капотом?"). Затем найденные решения подставляются в ранее рассмотренные уравнения для нахождения матрицы вращения и вектора перемещения между двумя камерами. Из-за присутствия шума в изображении и ошибок округления, каждая шахматная доска попарно объединяет различия в значениях для R и T. Функция `cvStereoCalibrate()` принимает найденные средние значения R и T в качестве начального приближения для поиска истинных значений за счет использования надежного *Levenberg-Marquardt* итеративного алгоритма поиска (локального) минимума ошибки перепроектирования углов шахматной доски для обоих представлений камер и возвращает решение для R и T. Итак, чтобы было ясно, что дает стерео калибровка: матрица вращения размещается в правой камере в той же плоскости, что и в левой камере; это объединяет две плоскости в одну не выровненную плоскость (выравнивание будет рассматриваться в следующем разделе "Стерео исправления").

Функция `cvStereoCalibrate()` имеет много параметров, но они все довольно таки просты и к тому же многие схожи с параметрами функции `cvCalibrateCamera2()` из главы 11.

```

bool cvStereoCalibrate(
    const CvMat* objectPoints
    ,const CvMat* imagePoints1
    ,const CvMat* imagePoints2
    ,const CvMat* npoints
    ,CvMat* cameraMatrix1
    ,CvMat* distCoeffs1
    ,CvMat* cameraMatrix2
    ,CvMat* distCoeffs2
    ,CvSize imageSize
    ,CvMat* R
    ,CvMat* T
    ,CvMat* E
    ,CvMat* F
    ,CvTermCriteria termCrit
    ,int flags = CV_CALIB_FIX_INTRINSIC
);

```

Первый параметр *objectPoints* это матрица Nx3, содержащая материальные координаты каждой из K точек на каждом M изображении трехмерного объекта таким образом, что N = KxM. Когда используется шахматная доска как трехмерный объект, эти точки находятся в системе координат, связанной с объектом - при установки необходимо сообщить, что левый верхний угол шахматной доски является исходным (и, как правило, выбранный так, что бы координата Z была равна 0), при этом какие-либо известные трехмерные точки могут быть использованы, как описано в *cvCalibrateCamera2()*.

Для разграничения одинаковых по смыслу параметров двух камер параметры оканчиваются на "1" и "2". В результате, следующие параметры *imagePoints1* и *imagePoints2* - это матрицы Nx2, содержащие левые и правые координаты пикселей (соответственно) всех опорных точек объекта из *objectPoints*. Если выполнить калибровку при помощи шахматной доски для двух камер, то *imagePoints1* и *imagePoints2* это соответственно возвращаемые значения для M, которые получаются после вызова *cvFindChessboardCorners()* для левого и правого представления камер.

Аргумент *npoints* содержит число точек для каждого изображения и является матрицей Mx1.

Параметры *cameraMatrix1* и *cameraMatrix2* это матрицы проекций камер 3x3, а *distCoeffs1* и *distCoeffs2* это матрицы искажения 5x1 для 1 и 2 камер, соответственно. При этом два параметра радиальных искажений содержаться в первой матрице, а два параметра тангенциального искажения и два радиальных искажений во второй (глава 11, коэффициенты искажения). Три параметра радиальных искажений завершают матрицу, т.к. были добавлены по ходу развития OpenCV; эти параметры, как правило, используются в случае с широкоугольными (с эффектом рыбий глаз) объективами.

Использование матриц внутренних параметров камер контролируется параметром *flags*. Если *flags* = *CV_CALIB_FIX_INTRINSIC*, то эти матрицы используются в процессе калибровки. Если *flags* = *CV_CALIB_USE_INTRINSIC_GUESS*, то эти матрицы используются в качестве отправной точки по оптимизации матриц внутренних параметров и искажений для каждой камеры и будут установлены в более точные значения после выполнения *cvStereoCalibrate()*. Другие значения параметра *flags*, которые соответствуют аналогичному параметру функции *cvCalibrateCamera2()*, можно аддитивно объединить; в этом случае эти параметры будут вычисляться с нуля в *cvStereoCalibrate()*. Т.е. можно вычислить матрицу внутренних параметров, матрицу внешних параметров и стерео параметры за один проход функцией *cvStereoCalibrate()*.

Параметр *imageSize* - это размер изображения в пикселях. Используется только если необходимо уточнить или вычислить внутренние параметры, когда *flags* != *CV_CALIB_FIX_INTRINSIC*.

Элементы R и T - это выходные параметры, которые заполняются функцией и которые являются матрицей вращения и вектором перемещения (связывающие левую и правую камеры), соответственно. Параметры E и F являются необязательными. Если они не установлены в NULL, то функция будет вычислять и заполнять их 3x3 существенной и фундаментальной матрицами. Параметр *termCrit* уже неоднократно рассматривался. Он отвечает за внутреннюю оптимизацию за счет сообщения о прекращении после определенного числа итераций или, когда вычисляемые параметры становятся меньше заданного порога в структуре *termCrit*. Типичный пример использования данного параметра выглядит следующим образом:
cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5).

И в заключении, параметр *flags*, о котором уже было сказано пару слов ранее. Если камеры откалиброваны и результат внушает доверие, то можно "закрепить неподвижно" результаты калибровки одной из камер при помощи *CV_CALIB_FIX_INTRINSIC*. Если начальные результаты калибровки хорошие, но не замечательные, то можно выполнить уточнение внутренних параметров и параметров искажения при помощи *CV_CALIB_USE_INTRINSIC_GUESS*. Если раздельная калибровка камер не выполнялась, то можно использовать те же настройки параметра *flags*, что и в *cvCalibrateCamera2()* главы 11.

После получения матрицы вращения и вектора перемещения (R, T) или фундаментальной матрицы F, можно их использовать для исправления двух стереоизображений таким образом, чтобы эпиполярные линии расположились вдоль строк изображения, а линии сканирования были одинаковыми на обоих изображениях. Несмотря на то, что R и T не определяют уникальность стерео исправлений, в следующем разделе будет показано как их (совместно с другими ограничениями) можно использовать.

Стерео исправления

Легче всего вычислить стерео несоответствия, когда обе плоскости изображения точно выровнены (рисунок 12-4). К сожалению, как уже было сказано ранее, редко удаётся добиться точно выравненной в реальности стерео системы, т.к. две камеры почти никогда не имеют точно компланарные, построчно-выровненные плоскости изображений. На рисунке 12-7 показана цель стерео исправлений: необходимо перепроектировать плоскость изображения двух камер так, чтобы они находились точно в одной и той же плоскости с совершенно выравненными строками во фронтально-параллельной форме. Возникает вопрос, как выбрать конкретную плоскость, в которой математическое выравнивание камеры зависит от используемого алгоритма? В дальнейшем будут показаны два случая, используемые в OpenCV.

Необходимо, чтобы строки изображения между двумя камерами были выровнены после исправлений так, чтобы стерео соответствия (поиск одной и то же точки на двух различных представлениях камеры) были более надежными и легко вычисляемыми. Надежность и эффективность вычислений можно повысить за счет выполнения поиска только по одной строке для сопоставления с точкой другого изображения. Результатом горизонтального выравнивания строк в пределах общей плоскости изображения, которые содержит каждое изображение, является то, что керновые точки перемещаются в бесконечность. Т.е. изображение центра проекции одного изображения параллельно другой плоскости изображения. Но т.к. имеет место бесконечное число возможных фронтально-параллельных плоскостей для выбора, то необходимо большее количество ограничений: максимизация перекрывающих представлений и/или минимизация искажений, выбор которых зависит от используемого алгоритма.

В результате выравнивания двух плоскостей изображения будут получены восемь элементов, по четыре для каждой камеры. Для каждой камеры также вычисляется вектор искажений *distCoeffs*, матрица вращения R_{rect} и выравненная и неочищенная матрицы (M_{rect} и M , соответственно). Благодаря этим элементам можно построить карту за счет *cvInitUndistortRectifyMap()*, где применяется интерполяция пикселей исходного изображения для создания нового исправленного изображения. (Стерео исправления в OpenCV возможны только тогда, когда эпиполярные точки находятся за пределами изображения прямоугольника. Следовательно, алгоритмы исправления могут не работать со стерео конфигурациями, которые характеризуются либо очень широкими базисными линиями, либо тем, что камеры могут указывать друг на друга в значительной степени).

Есть множество путей вычисления элементов исправления, но OpenCV использует только два: (1) алгоритм Hartley, который на выходе дает неоткалиброванное стерео за счет использования только фундаментальной матрицы; (2) алгоритм Bouguet, который

использует параметры вращения и перемещения от двух откалиброванных камер. Алгоритм Hartley может быть использован для получения структуры движения, записанного одной камерой, но вместе с тем может (при стерео исправлениях) привести к появлению большого количества искаженных изображений, чем откалиброванный алгоритм Bouguet. В ситуациях, когда возможно использование откалиброванных образцов - таких, как рука робота или изображение с камеры безопасности - наиболее естественно использование алгоритма Bouguet.

Неоткалиброванное стерео исправление: алгоритм Hartley

Алгоритм Hartley пытается найти матрицу гомографии, отображающую эпиполярные точки в бесконечности и сводящая к минимуму расчетные несоответствия между двумя стерео изображениями; это достигается за счет сопоставления точек между двумя парами изображения. Таким образом, необходимость в вычислении внутренних параметров камеры отпадает, т.к. эта информация в неявном виде содержится в сопоставлениях. Следовательно необходимо вычислить только фундаментальную матрицу, которая может быть получена от любого набора сопоставлений из семи или более точек между двумя представлениями сцены за счет `cvFindFundamentalMat()`. Как вариант, фундаментальная матрица может быть вычислена при помощи `cvStereoCalibrate()`.

Преимущество от использования алгоритма Hartley заключается в том, что стерео калибровка в реальном времени может быть выполнена за счет наблюдения только за точками сцены. Недостаток заключается в том, что отсутствует чувствительность к изменению масштаба изображения. Например, если использовать шахматную доску для генерации точек сопоставления, то не имеется возможности сказать далеко (если доска размером 100 метров) или близко (если доска размером 100 см). Так же не имеется явного определения матрицы внутренних параметров без которой камеры могут иметь разные фокусные расстояния, перекошенные пиксели, различные центры проекций и/или различные основные точки. Как результат, можно определить трехмерные объекты, реконструировав только проекционные преобразования. Это означает, что различные масштабы или проекции объекта могут показаться одним и тем же (т.е. особые точки будут иметь одинаковые двумерные координаты при различных трехмерных объектах). Обе проблемы показаны на рисунке 12-10.

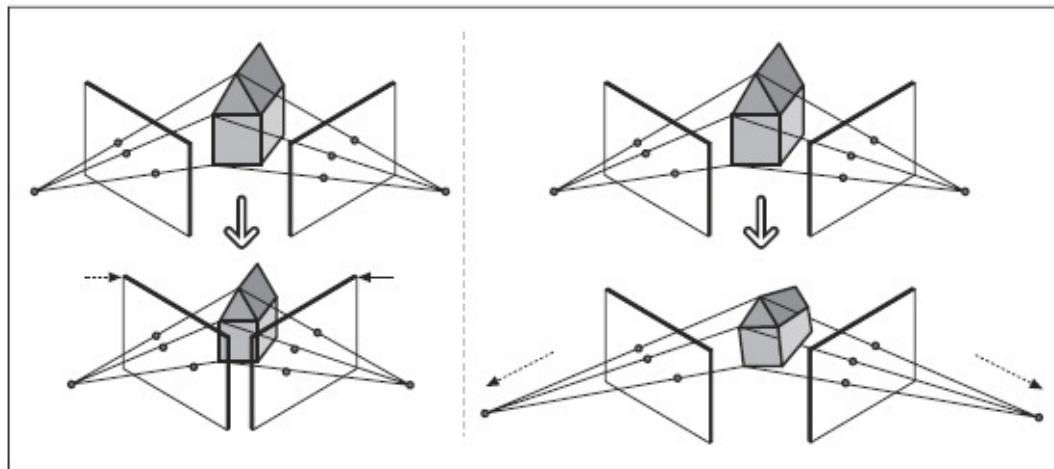


Рисунок 12-10. Реконструкция стерео неоднозначности: если размер объекта не известен, то различные размеры объектов могут дать один и тот же результат независимо от расстояния до камеры (слева); если матрица внутренних параметров камеры не известна, то различные проекции могут выглядеть одинаково - например, имеющие различные фокусные расстояния, основные точки

При наличии фундаментальной матрицы F , которой требуется семь или более точек, алгоритм Hartley будет выглядеть следующим образом:

1. Использование фундаментальной матрицы для вычисления двух эпиполярных точек при помощи соотношения $F e_l = 0$ и $(e_r)^T F = 0$ для левой и правой эпиполярной точки, соответственно.

2. Получение первой матрицы гомографии H_r , которая будет отображать правую эпиполярную точку в двумерной однородной точке в бесконечности $(1, 0, 0)^T$. Матрица гомографии имеет семь ограничений (для масштаба отсутствует), три для отображения в бесконечности, четыре для левой точки H_r . Последние четыре ограничения в основном приводят к беспорядку, т.к. большинство H_r приводят к весьма искаженным изображениям. Для поиска хорошей H_r , необходимо выбрать точку на изображении, где минимальны искажения и разрешены только строгие вращения и перемещения без сдвига. Разумным местом выбора такой точки является исходное изображение и в дальнейшем нужно считать, что эпиполярная точка $(e_r)^T = (f, 0, 1)$ лежит на оси x . Учитывая эти координаты, матрица

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/k & 0 & 1 \end{pmatrix}$$

будет использовать эпиполярные точки до бесконечности.

3. Для выбранной точки на правом (исходном) изображении вычисляется вектор перемещения T , который будет воспринимать эту точку как начало координат (0 в данном случае) и матрицу вращения R , которая использует эпиполярные точки $(e_r)^T = (f, 0, 1)$. Тогда матрица гомографии $H_r = GRT$.

4. Далее ищутся соответствия для матрицы гомографии H_l , которая отправляет левые эпиполярные точки в бесконечность и выравнивает строки двух изображений. Отправку левых эпиполярных точек в бесконечность легко выполнить, используя три ограничения из шага 2. Для выравнивания строк необходимо воспользоваться фактом о том, что выравнивание строк минимизирует общее расстояние между всеми сопоставляемыми точками двух изображений. То есть, найденная H_l минимизирует общее неравенство лево-правого сопоставления точек $\sum_i d(H_l p_i^l, H_r p_i^r)$. Эти две матрицы гомографии определяют стерео исправления.

Несмотря на то, что детали алгоритма несколько сложнее, `cvStereoRectifyUncalibrated()` выполняет всю эту работу за нас. Функция немного неверно названа, т.к. она не исправляет неоткалиброванные стерео изображения, она лишь вычисляет матрицы гомографии, которые в дальнейшем могут быть использованы для исправления.

```
int cvStereoRectifyUncalibrated(
    const CvMat*    points1
, const CvMat*    points2
, const CvMat*    F
, CvSize          imageSize
, CvMat*          Hl
, CvMat*          Hr
, double          threshold
);
```

Входным параметром функции является массив $2 \times K$, соответствующий точкам между левым и правым изображениями массивов *point1* и *point2*. Заранее рассчитанная фундаментальная матрица передается как массив *F*. Параметр *imageSize* указывает на ширину и высоту изображения, которое будет использовано во время калибровки. Функция возвращает матрицы гомографии в *Hl* и *Hr*. И в заключении, если расстояние от этих точек до соответствующих эпиполярных точек превышает установленное пороговое значение *threshold*, то соответствующая точка исключается из алгоритма.

Если камеры имеют примерно одинаковые параметры и устанавливаются в приблизительно горизонтально выравненную фронтально-параллельную форму, то конечные исправления алгоритма Hartley будут схожи со случаем калибровки, который

описан далее. Если известен размер или трехмерная геометрия объекта сцены, то можно получить те же результаты, как и в случае с калибровкой.

Откалиброванные стерео исправления: алгоритм Bouguet

Учитывая матрицу вращения и вектор перемещения (R , T) между стерео изображениями, алгоритм Bouguet для стерео исправлений просто пытается минимизировать количество изменений перепроектирования для каждого из двух изображений (тем самым свести к минимуму результирующие перепроектирования искажений) при максимальной площади охвата.

Для сведения к минимуму искажений перепроектирования изображения, матрица вращения R , которая вращает плоскость изображения правой камеры относительно плоскости изображения левой камеры, делится пополам между двумя камерами; как результат две матрицы вращения r_l и r_r для левой и правой камеры, соответственно. Каждая камера вращается в пол вращения так, что каждый основной луч в конечном итоге становится параллелен суммарному вектору, где указаны их исходные основные лучи. Как уже было сказано ранее, такое вращение выставляет камеры компланарно, но при этом не выравнивает строки. Вычисленная матрица R_{rect} , которая принимает эпиполярные точки левой камеры в бесконечности и совмещает эпиполярные линии по горизонтали, это матрица вращения, начинающаяся с направления эпиполярной точки e_1 . Принимая основную точку (c_x , c_y) за исходное левое изображение, направление эпиполярной точки совпадает с вектором перемещения между двумя центрами проекции камер:

$$e_1 = \frac{T}{\|T\|}$$

Следующий вектор e_2 должен быть ортогонален e_1 , в остальном этот вектор ограничений не имеет. Для e_2 выбранное направление ортогонально основному лучу (который имеет тенденцию к расположению вдоль плоскости изображения), что является хорошим выбором. Это достигается за счет векторного произведения e_1 в направлении основного луча и последующей нормализации так, что получается ещё один единичный вектор:

$$e_2 = \frac{[-T_y \ T_x \ 0]^T}{\sqrt{T_x^2 + T_y^2}}$$

Третий вектор должен быть ортогонален e_1 и e_2 ; он может быть найден при помощи векторного произведения:

$$e_3 = e_1 \times e_2$$

В результате матрица, принимающая эпиполярные точки левой камеры в бесконечности, примет следующий вид:

$$R_{\text{rect}} = \begin{bmatrix} (e_1)^T \\ (e_2)^T \\ (e_3)^T \end{bmatrix}$$

Эта матрица вращает левую камеру относительно центра проекции так, что эпиполярные линии принимают горизонтальное положение, а эпиполярные точки располагаются в бесконечности. Последующее выравнивание строк двух камер достигается следующим образом:

$$R_l = R_{\text{rect}} r_l$$

$$R_r = R_{\text{rect}} r_r$$

Так же необходимо вычислить исправления для левой и правой матриц камеры $M_{\text{rect_l}}$ и $M_{\text{rect_r}}$, при этом результат должен содержать так же проекционные матрицы P_l и P_r

$$P_l = M_{\text{rect_l}} P'_l = \begin{bmatrix} f_{x_l} & \alpha_l & c_{x_l} \\ 0 & f_{y_l} & c_{y_l} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

и

$$P_r = M_{\text{rect_r}} P'_r = \begin{bmatrix} f_{x_r} & \alpha_r & c_{x_r} \\ 0 & f_{y_r} & c_{y_r} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

где α_l и α_r коэффициенты смещения пикселей, которые в современных камерах почти всегда равны 0. Проекционная матрица преобразует трехмерные точки в однородных координатах в двумерные точки в однородных координатах следующим образом:

$$P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

где экранные координаты могут быть вычислены как $(x/w, y/w)$. Двумерные точки могут быть так же перепроецированы в трехмерные за счет координат экрана и матрицы внутренних параметров камеры. Матрица перепроецирования:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/T_x & (c_x - c'_x)/T_x \end{bmatrix}$$

где все параметры соответствуют левому изображению, кроме c'_x , который является координатой x основной точки правого изображения. Если основные лучи пересекаются в бесконечности, то $c_x = c'_x$, а элемент в правом нижнем углу равен 0. При наличии двумерной однородной точки и связанного с ней разрыва d можно спроектировать трехмерную точку при помощи:

$$Q \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

Трехмерные координаты тогда будут выглядеть следующим образом ($X/W, Y/W, Z/W$).

Применение описанного метода исправления Bouguet, дающий идеальную стерео конфигурацию, показано на рисунке 12-4. В последующем, новые центры и оценки изображения выбираются для вращающихся изображений таким образом, чтобы максимизировать области перекрытия представлений. В основном происходит просто установление единого центра камеры и общей максимальной высоты и ширины от двух областей изображения как новых стерео представлений плоскостей.

```
void cvStereoRectify(
    const CvMat* cameraMatrix1
    , const CvMat* cameraMatrix2
    , const CvMat* distCoeffs1
    , const CvMat* distCoeffs2
    , CvSize imageSize
    , const CvMat* R
    , const CvMat* T
    , CvMat* Rl
    , CvMat* Rr
    , CvMat* Pl
    , CvMat* Pr
    , CvMat* Q = 0
    , int flags = CV_CALIB_ZERO_DISPARITY
);
```

Входными значениями `cvStereoRectify()` (опять же не совсем точное именование функции; на самом деле происходит вычисление элементов за счет которых в дальнейшем можно произвести исправление, а не само исправление) являются уже знакомые матрицы камер и векторы искажения, возвращаемые `cvStereoCalibrate()`.

Параметр *imageSize* содержит размеры шахматной доски, используемые при калибровке камеры. Так же передаются матрица вращения R и вектор перемещения T между правой и левой камерами, возвращаемые *cvStereoCalibrate()*.

Возвращаемые параметры *Rl* и *Rr* - это 3x3 построчно выравненные исправленные матрицы вращения для левой и правой плоскостей изображения, а также связанные с ними 3x4 проекционные матрицы *Pl* и *Pr*. Необязательный возвращаемый параметр Q - это 4x4 матрица перепроектирования, описанная ранее.

Параметр *flags* по умолчанию устанавливает разрыв в бесконечности, что соответствует нормальному случаю, показанному на рисунке 12-4. Сброс *flags* означает, что камеры сближены относительно друг друга (что-то типа "косоглазия"), так что нулевой разрыв происходит на конечном расстоянии (это может быть полезно при большой глубине разрешения в непосредственной близости на конкретном расстоянии).

Если параметр *flags* не установлен в *CV_CALIB_ZERO_DISPARITY*, то необходимо соблюдать осторожность при достижении исправленной системы. А исправляется система по отношению к основным точкам (c_x , c_y) левой и правой камер. Таким образом, измерения на рисунке 12-4 должны так же зависеть и от положения. В принципе, необходимо изменить расстояния следующим образом $\tilde{x}^r = x^r - c_x^{\text{right}}$ и $\tilde{x}^l = x^l - c_x^{\text{left}}$. При установке разрыва в бесконечности $c_x^{\text{left}} = c_x^{\text{right}}$ (т.е. когда *flags* = *CV_CALIB_ZERO_DISPARITY*) обычные координаты пикселя (или разрыв) передаются с формулой глубины. Но если *cvStereoRectify()* вызывается без *CV_CALIB_ZERO_DISPARITY*, то $c_x^{\text{left}} \neq c_x^{\text{right}}$. Поэтому даже если формула $Z = fT / (x^l - x^r)$ остается той же, следует иметь ввиду, что x^l и x^r рассчитываются не от центра изображения, а от соответствующих основных точек c_x^{left} и c_x^{right} , которые могут отличаться от x^l и x^r . Поэтому, если разрыв вычислен как $d = x^l - x^r$, то он должен быть отрегулирован перед вычислением Z: $Z = fT / (d - (c_x^{\text{left}} - c_x^{\text{right}}))$.

Карта исправлений

После получения элементов стерео калибровки, можно предварительно вычислить левую и правую карты исправлений для левого и правого представления камер, используя *cvInitUndistortRectifyMap()*. Как и в любой другой функции отображения изображения к изображению, переднего отображения (в котором происходит вычисление перехода пикселей от исходного изображения к целевому) не будет в связи с тем, что конечное положение имеет тип *float*, а все положения попадающих пикселей на конечное изображение придают вид швейцарского сыра этому (конечному) изображению. В результате работа выполняется в обратном порядке: для каждого целочисленного положения пикселя конечного изображения смотрится какая вещественная координата поступила от исходного изображения, а затем

интерполируется в зависимости от окружающих значений исходных пикселей для получения целочисленного положения на конечном изображении. Данное преобразование обычно использует билинейную интерполяцию, которая уже была рассмотрена в главе 6 при обсуждении *cvRemap()*.

Процесс исправления показан на рисунке 12-11. На рисунке уравнение потока фактически показывает, что процесс исправления выполняется в обратном порядке от (c) к (a) и именуется как *процесс обратного отображения*. Для каждого целочисленного пикселя на исправленном изображении (c) ищутся соответствующие координаты на неискаженном изображении (b), которые в последующем используются для показа фактических (вещественных) координат на необработанном изображении (a). Затем эти вещественные координаты пикселя интерполируются за счет ближайших целочисленных положений пикселей на исходном изображении; полученное значение используется для заполнения конечного изображения (c). После заполнения исправленного изображения, оно, как правило, обрезается с акцентом на область пересечения между левым и правым изображениями.

cvInitUndistortRectifyMap() - это функция, реализующая математику, представленную на рисунке 12-11. Данную функцию необходимо применить дважды для левой и один раз для правой стерео пары.

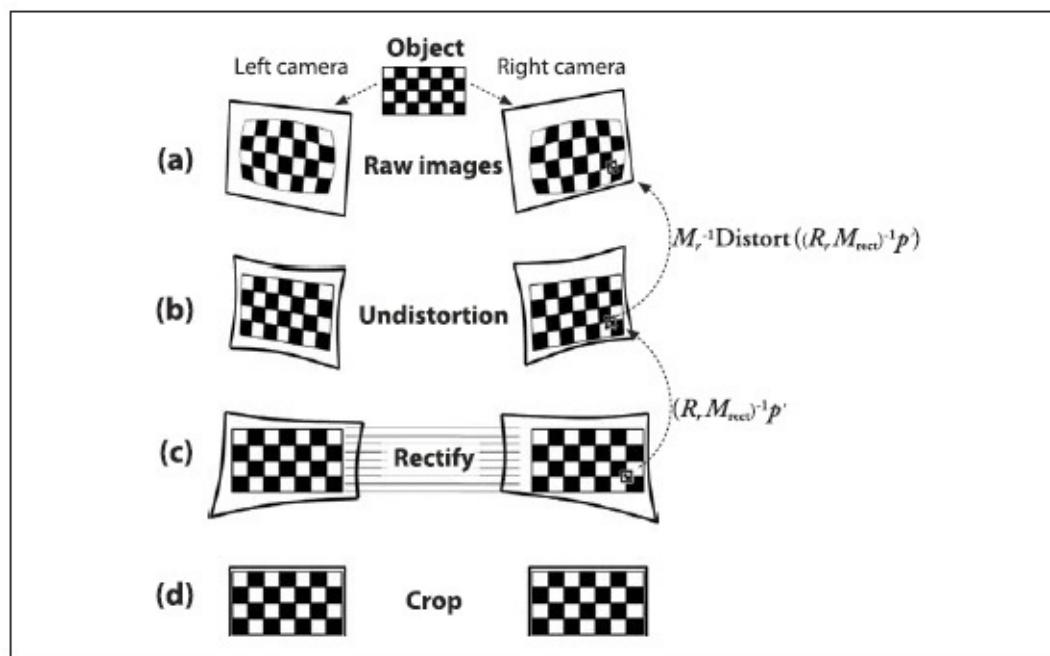


Рисунок 12-11. Стерео исправление: для левой и правой камер необработанное изображение (a) становится неискаженным (b), исправленным (c) и в конечном счете обрезанным (d) для сосредоточения на области перекрытия между двумя камерами; вычисление исправлений на самом деле работает в обратном порядке от (c) к (a)

```

void cvInitUndistortRectifyMap(
    const CvMat*   M
    ,const CvMat* distCoeffs
    ,const CvMat* Rrect
    ,const CvMat* Mrect
    ,CvArr*        mapx
    ,CvArr*        mapy
);

```

Входными параметрами функции являются матрица камеры M размера 3x3, исправленная матрица камеры $Mrect$ размера 3x3 и матрица вращения $Rrect$ размера 3x3, и параметр искажений камеры $distCoeffs$ размера 5x1.

Если стерео камеры откалиброваны при помощи $cvStereoRectify()$, то можно указать входные параметры $cvInitUndistortRectifyMap()$ полученные от $cvStereoRectify()$, используя при этом сначала левые параметры для исправления левой камеры, а затем правые параметры для исправления правой камеры. Для $Rrect$ используется Rl или Rr от $cvStereoRectify()$; для M используется $cameraMatrix1$ или $cameraMatrix2$. Для $Mrect$ используются первые три колонки 3x4 P_l или P_r от $cvStereoRectify()$, но для удобства функция позволяет передать прямо P_l или P_r и $Mrect$ будет читаться из них.

С другой стороны, если используется $cvStereoRectifyUncalibrated()$ для калибровки стерео камер, то необходимо предварительно немного обработать матрицу гомографии. Хотя можно - и на практике, в принципе, тоже - выполнить стерео исправления и без использования внутренних параметров камеры, однако, в OpenCV нет для такого случая подходящей функции. Если не имеется возможности получения $Mrect$ в результате выполнения предварительной калибровки, то правильнее всего будет установить $Mrect = M$. Тогда для $Rrect$ в $cvInitUndistortRectifyMap()$ необходимо вычислить $R_{rect_l} = M_{rect_l}^{-1} H_l M_l$ (или просто $R_{rect_l} = M_l^{-1} H_l M_l$, если не имеется $M_{rect_l}^{-1}$) и $R_{rect_r} = M_{rect_r}^{-1} H_r M_r$ (или просто $R_{rect_r} = M_r^{-1} H_r M_r$, если не имеется $M_{rect_r}^{-1}$) для левого и правого исправления, соответственно. И в заключении, необходимо заполнить матрицу коэффициентов искажения $distCoeffs$ размера 5x1 для каждой камеры.

Функция $cvInitUndistortRectifyMap()$ возвращает найденные карты в $mapx$ и $mapy$. Эти карты указывают на то, как необходимо интерполировать исходные пиксели относительно каждого пикселя конечного изображения; карты могут быть заглушками непосредственно в $cvRemap()$. Так как функция $cvInitUndistortRectifyMap()$ вызывается непосредственно для левой и правой камеры по отдельности, то параметры $mapx$ и $mapy$ можно получить по отдельности. Функция $cvRemap()$ может быть вызвана, используя левую, а затем и правую карты каждый раз, когда появляются новые левые и правые стереоизображения для исправления. На рисунке 12-12 показаны

результатирующие неискаженное стерео и исправленная стереопара изображения. При этом стоит обратить внимание на то, как особенные точки становятся горизонтально выпрямленными в неискаженном исправленном изображении.

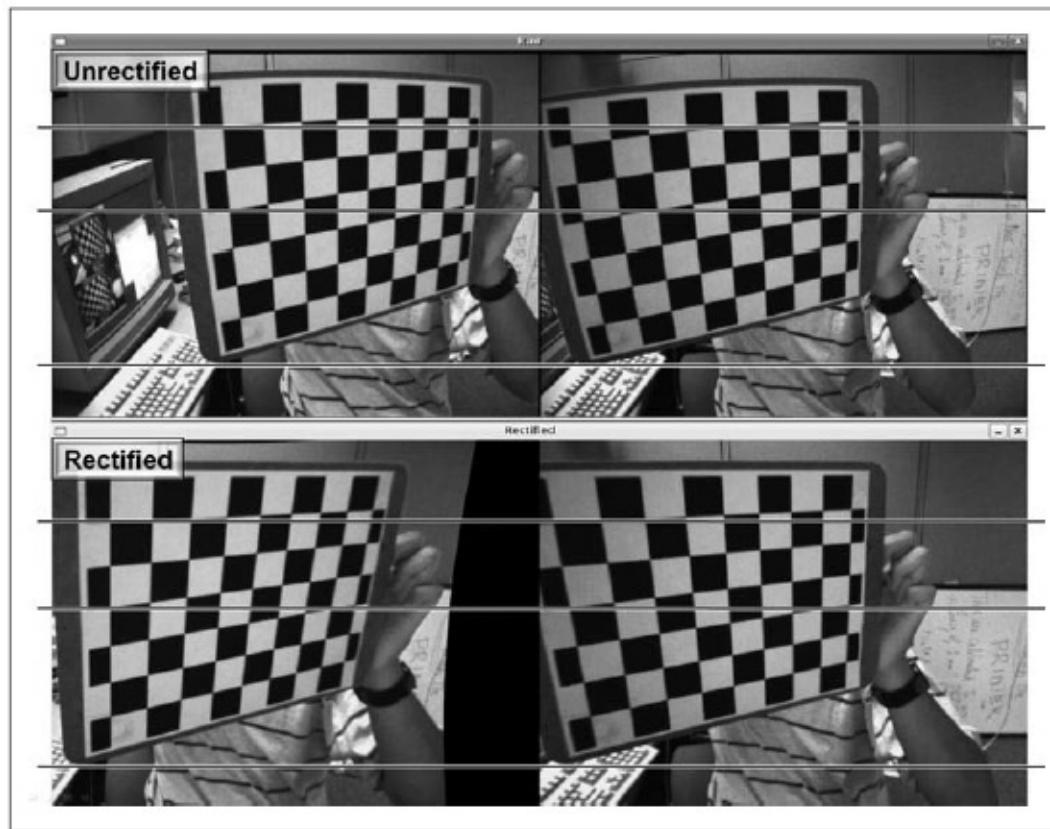


Рисунок 12-12. Стерео исправление: исходная пара левого и правого изображений (верхняя часть рисунка) и стерео исправленная пара левого и правого изображений (нижняя часть рисунка); при этом стоит отметить тот факт, что исправлено искажение "бочка", а линии сканирования выравнены в исправленных изображениях

Стерео соответствие

Стерео соответствие - сопоставление трехмерных точек с двух различных представлений камеры - может быть вычислено только в визуальных областях, где представления двух камер совпадают. Опять же, это одна из причин, почему для получения наилучших результатов необходимо добиваться того, чтобы камеры были максимально фронтально параллельными. Тогда, при знании физических координат камер или размеров объектов на сцене, можно получить размерность глубины за счет триангуляции измерений разрыва $d = x^l - x^r$ (или $d = x^l - x^r - (c_x^{\text{left}} - c_x^{\text{right}})$) при условии, что основные лучи пересекаются на конечном расстоянии) между соответствующими точками с двух различных представлений камер. Не имея такой информации, вычисление глубины возможно только до момента формирования коэффициентов

масштабирования. Если нет той матрицы внутренних параметров камеры, которая использовалась в алгоритме Hartley, то вычисление расположения точки возможно только до момента проекционных преобразований (рисунок 12-10).

В OpenCV реализован быстрый и эффективный алгоритм блочного стерео сопоставления `cvFindStereoCorrespondenceBM()`, который схож с тем, что разработал Kurt Konolige; он работает, используя небольшое "суммирование абсолютных разностей" (SAD) окон для поиска соответствующих точек между левым и правым стерео исправленными изображениями. Этот алгоритм находит только сильно соответствующие (высоко текстурированные) точки между двумя изображениями. Таким образом, в сильно текстурированной сцене, такой как лес на открытом воздухе, для каждого пикселя может быть вычислена глубина. В низко текстурированной сцене, такой как коридор внутри дома, для немногих точек можно вычислить глубину. Данный алгоритм выполняется в три этапа, при этом применяется к неискаженной, выпрямленной паре изображений:

1. Предварительная фильтрация для нормализации яркости изображения и повышения текстурности.
2. Поиск сопоставлений вдоль горизонтальной эпиполярной линии за счет использования SAD окон.
3. Пост-фильтрация для устранения плохих сопоставлений.

На первом шаге исходное изображение нормализуется за счет снижения различий в освещении и повышении текстурности на изображении. Это достигается за счет применения окна - размера 5x5, 7x7 (по умолчанию), ..., 21x21 (максимально) - на изображении. Центр пикселя I_c под окном заменяется минимумом $\min[\max(I_c - \bar{I} - I_{cap}, 0)]$, где \bar{I} среднее значение окна, а I_{cap} - это положительный числовой предел, значение которого по умолчанию равно 30. Данный подход достигается за счет использования флага `CV_NORMALIZED_RESPONSE`. Помимо этого, флаг также может принимать значение `CV_LAPLACIAN_OF_GAUSSIAN`, что в свою очередь приводит в обнаружению пикового значения на сглаженном изображении.

Соответствия вычисляются за счет скольжения SAD окна. Для каждой особенности на левом изображении ищется соответствующая строка на правом изображении для наилучшего совпадения. После исправления, каждая строка становится эпиполярной линией, так что положения соответствий на правом изображении должны расположиться вдоль той же строки (одинаковые у-координаты) что и на левом изображении; эти положения совпадений могут быть найдены, если особенность достаточно текстурирована для обнаружения и если она не перекрыта на правом представлении камеры (рисунок 12-16). Если особенность левого пикселя имеет координаты (x_0, y_0) , то для горизонтально параллельно расположенной

камеры совпадения (если такие имеются) должны быть найдены в той же строке и в, или слева от, x_0 , рисунок 12-13. Для фронтально параллельных камер, x_0 нулевое несоответствие, а все что слева соответствует огромным несоответствиям. Для камер, расположенных под углом друг к другу, совпадения могут возникнуть при отрицательных несоответствиях (т.е. справа от x_0). Первый параметр, который контролирует поиск совпадений это *minDisparity*, откуда начинается поиск совпадений. По умолчанию *minDisparity* равен 0. Поиск несоответствий продолжается до момента *numberOfDisparities* подсчета пикселей (по умолчанию 64 пикселя). Различия являются дискретными, разрешение субпикселей которых устанавливается параметром *subPixelDisparities* (по умолчанию 16 субразличий на пиксель). Уменьшение количества различий при поиске может помочь сократить время вычисления за счёт ограничения длины поиска для совпадающих точек вдоль эпиполярной линии. При этом стоит помнить, что чем ближе расстояние, тем больше различия.

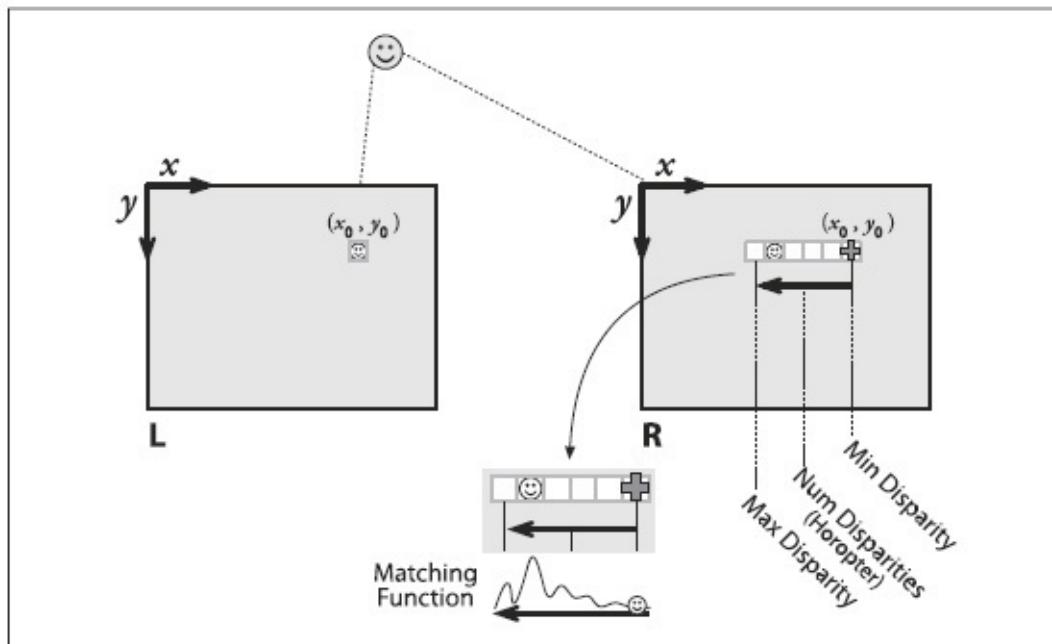


Рисунок 12-13. Любые особенности левого представления должны совпадать с особенностями на правом представлении в одних и тех же строках и в (или слева от) тех же координатах точки, при этом поиск совпадений начинается с точки *minDisparity* (в данном случае 0) и движется влево до заданного количества различий; характеристика функции, основанная на окнах, сопоставления особенностей показана в нижней части рисунка

Установку минимума различий и количества шагов поиска различий задает *horopter*, 3D объём которого покрывает диапазон поиска стерео алгоритма. На рисунок 12-14 показан предел поиска различий в 5 пикселей, начиная с трех различных пределов различий: 20, 17 и 16. Каждый предел различий определяет плоскость на фиксированной глубине от камеры (рисунок 12-15). Как показано на рисунке 12-14, каждый предел различий - вместе с числом различий - устанавливает различия

horopter при которых глубина может быть определена. За пределами данного диапазона теряется возможность определения глубины и как результат "дыра" на карте глубины, где глубина не известна. *Horopters* могут быть больше за счет уменьшения расстояния базисной линии T между камерами, уменьшения фокусного расстояния за счет увеличения диапазона поиска стерео различий или за счет увеличения ширины пикселя.

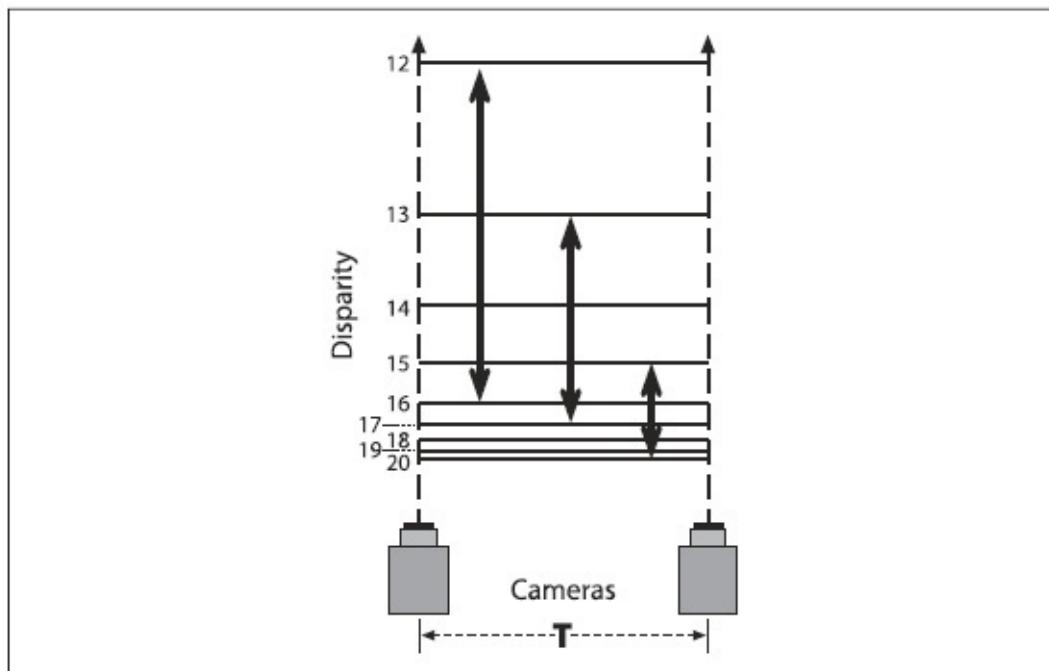


Рисунок 12-14. Каждая строка представляет собой плоскость константных различий целых пикселей от 20 до 12; поиск диапазона различий в пять пикселей охватывает различные диапазоны *horopter*, что показано вертикальными стрелками, а также различные максимальные пределы различий за счет установки различных *horopters*

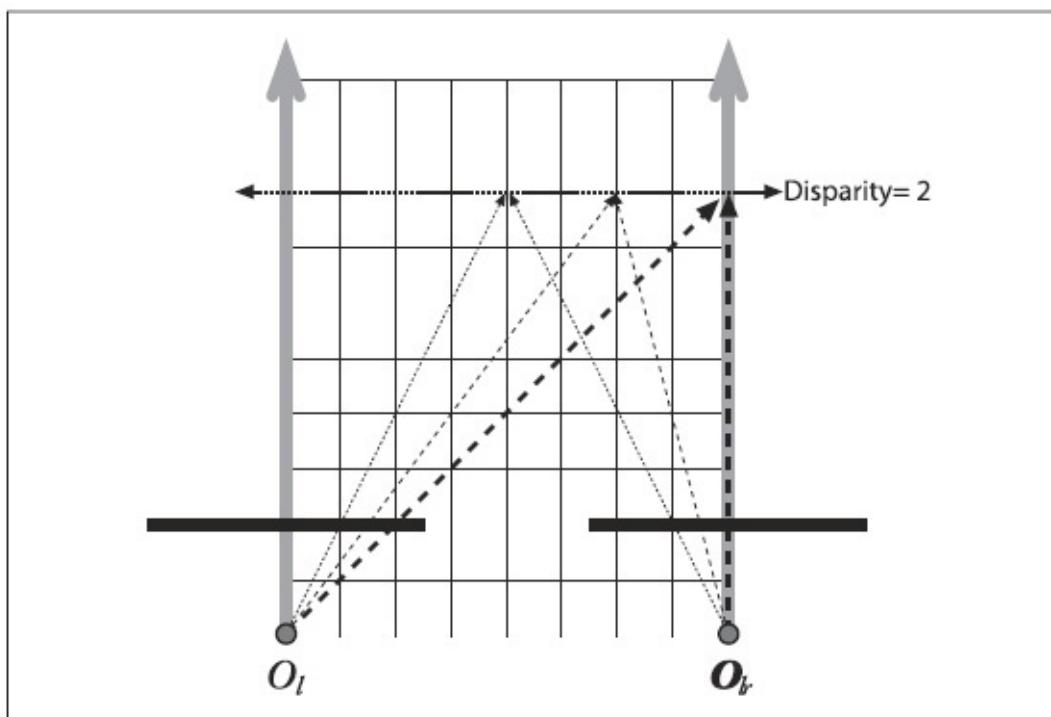


Рисунок 12-15. Фиксированные различия формируют плоскость на фиксированном расстоянии от камеры

Соответствия в пределах *horopter* имеют одно встроенное ограничение, именуемое *ограничением порядка*, согласно которому порядок особенностей не может быть изменен от левого представления до правого. Особенности могут быть *пропущены* - где благодаря перекрытию или шуму некоторые особенности, найденные на левом представлении, не могут быть найдены на правом - но порядок найденных особенностей остается прежним. Точно так же возможно сформировать особенности справа, которые в свою очередь не были определены слева (это именуется *вставкой*), при этом вставка не изменяет *порядок* особенностей, хоть и может распространять эти внешние особенности. Процедура, показанная на рисунке 12-16, отражает ограничение порядка при сопоставлении особенностей на горизонтальной линии сканирования.

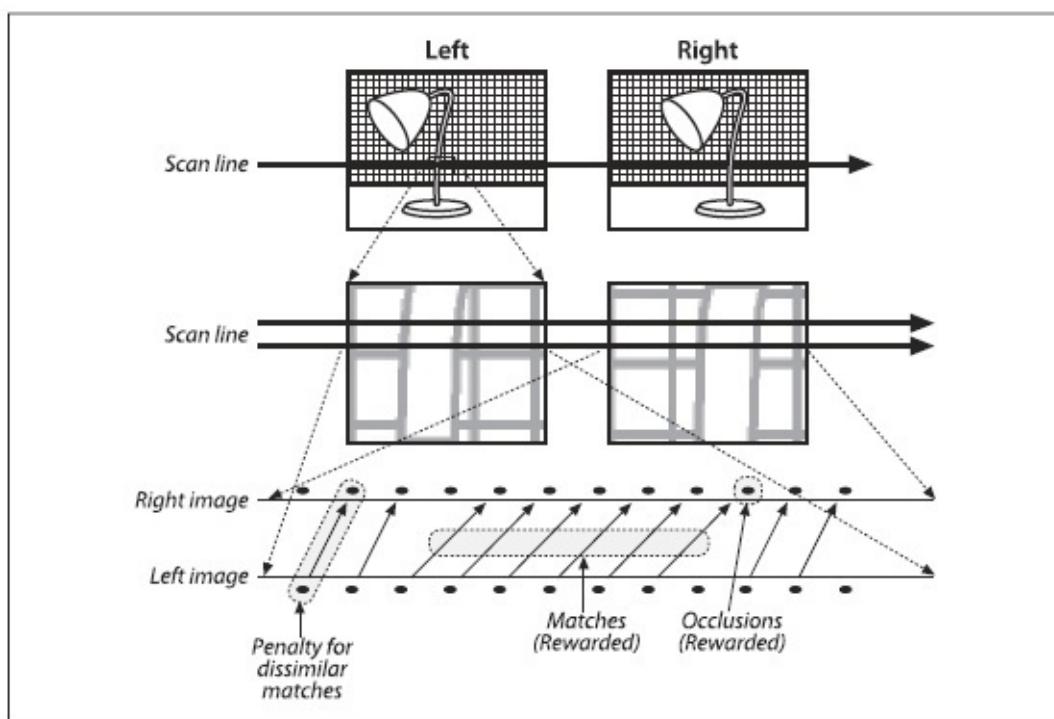


Рисунок 12-16. Стерео сопоставление начинается с назначения точки совпадений между соответствующими строками на левом и правом изображениях: левое и правое изображения лампы (верхняя часть); увеличенная линия сканирования (средняя часть); визуализация сопоставлений (нижняя часть)

Учитывая минимально допустимый прирост различий Δd , можно определить наименее достижимую глубину разрешенного диапазона ΔZ по следующей формуле:

$$\Delta Z = \frac{Z^2}{fT} \Delta d$$

Эту формулу полезно запомнить, т.к. благодаря ей определяется и устанавливается глубина стерео установки.

После сопоставления выполняется пост-фильтрация. В нижней части рисунка 12-13 показана типичная функция сопоставления особенностей как "развертка" от минимального различия к максимальному. Стоит обратить внимание на тот факт, что сопоставление характеризуется наличием сильного центрального пика в окружении боковых долей. После получения кандидатов в особенности, за счет сопоставления двух представлений, пост-фильтрация используется для предотвращения ложных сопоставлений. В OpenCV используется шаблонная функция сопоставления при помощи параметра *uniquenessRatio* (значение по умолчанию 12), которая отфильтровывает совпадения по условию *uniquenessRatio > (match_val - min_match)/min_match*.

Чтобы убедиться в том, что имеется достаточно текстур для преодоления случайного шума во время сопоставления, OpenCV также использует *textureThreshold*. Это просто ограничивает отклик окна SAD так, что не учитываются все сравнения, у которых отклик ниже *textureThreshold* (по умолчанию значение равно 12). И в заключении, при блочном сопоставлении имеются проблемы вблизи границ объектов, т.к. окно ловит сопоставления на переднем плане с одной стороны и на заднем плане с другой. В результате это приводит к большим и малым различиям, именуемые *speckle*. Для предотвращения пограничных сопоставлений, можно установить детектор *speckle* над окном *speckle* (размера от 5x5 до 21x21) и *speckleWindowSize* (по умолчанию равно 9 для окна 9x9). В пределах окна *speckle*, пока минимально и максимально обнаруженные различия укладываются в пределах *speckleRange*, сопоставление допустимо (диапазон по умолчанию равен 4).

Стерео зрение является важной частью систем видеонаблюдения, навигации и робототехники, а так же систем с высокими требованиями к производительности в режиме реального времени. Таким образом процедуры стерео соответствия предназначены для быстрого запуска. Следовательно, не имеется возможности удержать все выделенные внутренние временные буферы, которые соответствуют процедуре *cvFindStereoCorrespondenceBM()*.

Параметры блочного сопоставления и внутренние временные буферы хранятся в структуре данных *CvStereoBMState*:

```

typedef struct CvStereoBMState {
    // предварительная фильтрация (нормализация исходных изображений):
    int preFilterType;
    int preFilterSize; // от 5x5 до 21x21
    int preFilterCap;

    // при сопоставлении используется Sum of Absolute Difference (SAD):
    int SADWindowSize;           // Может быть 5x5, 7x7, ..., 21x21
    int minDisparity;
    int numberOfDisparities;     // Число пикселей при поиске

    // пост фильтрация (исключение плохих сопоставлений):
    int textureThreshold;       // Минимально допустимый порог
    float uniquenessRatio;      // Фильтрация при условии:
                                // [ match_val - min_match <
                                //   uniqRatio*min_match ]
                                // над областью corr окна
    int speckleWindowSize;      // Окно изменения различий
    int speckleRange;           // Приемлемый диапазон изменения в окне

    // временные буфера
    CvMat* preFilteredImg0;
    CvMat* preFilteredImg1;
    CvMat* slidingSumBuf;
} CvStereoBMState;

```

Состояние структуры выделяется и возвращается функцией `cvCreateStereoBMState()`. Эта функция принимает параметр *present*, который может принимать одно из следующих значений:

`CV_STEREO_BM_BASIC` Установка всех параметров в значения по умолчанию

`CV_STEREO_BM_FISH_EYE` Установка параметров для работы с широкоугольными объективами

`CV_STEREO_BM_NARROW` Установка параметров для стерео камер с узким полем зрения

Эта функция так же имеет необязательный параметр *numberOfDisparities*; если он не равен нулю, то перекрывает значение по умолчанию для *preset*.

```

CvStereoBMState* cvCreateStereoBMState(
    int     presetFlag      = CV_STEREO_BM_BASIC
    , int    numberOfDisparities = 0
);

```

Освобождение занимаемой структурой памяти выполняется при помощи `cvReleaseBMState()`:

```
void cvReleaseBMState(
    CvStereoBMState **BMState
);
```

Любые параметры стерео соответствия могут быть скорректированы в любой момент между вызовом `cvFindStereoCorrespondenceBM()` и непосредственным присвоением новых значений полям структуры. Функция сопоставления позаботится о выделении/перераспределении памяти для необходимых внутренних буферов.

`cvFindStereoCorrespondenceBM()` принимает исправленную пару изображений и отдает карту различий согласно структуре состояния:

```
void cvFindStereoCorrespondenceBM(
    const CvArr      *leftImage
    , const CvArr     *rightImage
    , CvArr          *disparityResult
    , CvStereoBMState *BMState
);
```

Пример стерео калибровки, исправления и сопоставления

Теперь настало время рассмотреть пример реализации всего ранее рассмотренного материала. В качестве примера будет рассмотрена шахматная доска с прочтением её структуры из файла `list.txt`. Этот файл содержит список чередующихся левых и правых стерео пар изображений (шахматной доски), которые используются при калибровке камеры и последующем исправлении изображений. При этом камеры расположены таким образом, что их линии сканирования изображения приблизительно физически выравнены, а также каждая камера имеет по существу одно и тоже поле зрения. Благодаря этому можно избежать эпиполярных проблем, начиная с изображения, а также максимизировать площадь стерео перекрытия при минимизации искажений от перепроекции.

В далее представленном коде (пример 12-3) вначале происходит чтение левых и правых пар изображений, поиск углов шахматной доски с субпиксельной точностью и установка объекта и изображения точек для изображений, где все шахматные доски могут быть найдены. Существует возможность скрыть данный процесс (т.е. не отображать на экране). Учитывая список найденных точек на найденных хороших изображениях шахматной доски, вызывается `cvStereoCalibrate()` для калибровки камеры. В результате калибровки будут получены матрица камеры `_M` и вектор искажения `_D` для двух камер; так же матрица вращения `_R`, вектор перемещения `_T`, существенная матрица `_E` и фундаментальная матрица `_F`.

Далее следует небольшая проверка точности калибровки за счет оценки попадания точек одного изображения на эпиполярную линию другого изображения. Для этого исходные точки исправляются при помощи `cvUndistortPoints()` (глава 11), эпиполярные линии вычисляются при помощи `cvComputeCorrespondEpilines()` и затем вычисляется скалярное произведение точек с линиями (в идеальном случае, почти все результаты перемножения равны 0). Накопленное абсолютное расстояние формирует ошибку.

Затем необязательное вычисление карты исправлений при помощи неоткалиброванного (Hartley) метода `cvStereoRectifyUncalibrated()` или откалиброванного (Bouguet) метода `cvStereoRectify()`. Если задействуется неоткалиброванный метод, то в коде дополнительно предусмотрено вычисление необходимой фундаментальной матрицы с нуля или использование фундаментальной матрицы от стерео калибровки. Исправленные изображения затем вычисляются при помощи `cvRemap()`. В рассматриваемом примере линии рисуются сквозь пару изображений, тем самым помогая увидеть, как исправлено выравненное изображение. Пример возможного результата показан на рисунке 12-12, где собственно можно увидеть исправленное исходное изображение с искажением типа "бочка" сверху вниз и то, как выравнены изображения горизонтальными линиями сканирования.

И в заключении, если изображения исправлены, инициализируется блочное сопоставление (внутреннее выделение памяти и инициализация параметров) при помощи `cvCreateBMState()`. Далее имеется возможность вычисления карт искажения при помощи `cvFindStereoCorrespondenceBM()`. Рассматриваемый пример позволяет использовать либо горизонтально выравненные (слева направо), либо вертикально выравненные (сверху вниз) камеры; при этом в случае вертикально выравненных камер функция `cvFindStereoCorrespondenceBM()` может вычислить искажения только для неоткалиброванного исправленного случая, если конечно не добавить код транспонирования изображения. В случае горизонтально выравненных камер, `cvFindStereoCorrespondenceBM()` может найти искажения для откалиброванных или неоткалиброванных исправленных пар стерео изображений.

Пример 12-3. Пример стерео калибровки, исправления и сопоставления

```
#include "cv.h"
#include "cxmisc.h"
#include "highgui.h"
#include "cvaux.h"
#include <vector>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <ctype.h>

using namespace std;
```

```

// Значение useCalibrated (0 для Hartley или 1 для Bouguet стерео методов)
// зависит от списка изображений шахматной доски, количество углов (nx, ny)
// и флага. Калибровка камеры и отображение результатов исправления
// соотносятся с рассчитанным различиями изображений
//

static void StereoCalib(
    const char* imageList
    ,int nx
    ,int ny
    ,int useUncalibrated
) {
    int      displayCorners      = 0;
    int      showUndistorted     = 1;
    bool     isVerticalStereo    = false;      // Выравнивание сверху-вниз
                                                // или слева-направо
    const int      maxScale      = 1;
    const float     squareSize   = 1.f;        // Установка реальных размеров квадрата

    FILE* f = fopen(imageList, "rt");
    int i, j, lr, nframes, n = nx*ny, N = 0;

    vector<string>      imageNames[2];
    vector<CvPoint3D32f> objectPoints;
    vector<CvPoint2D32f> points[2];
    vector<int>          npoints;
    vector<uchar>         active[2];
    vector<CvPoint2D32f> temp(n);

    CvSize imageSize = {0,0};

    // Хранилища под массивы и вектора:
    //

    double M1[3][3], M2[3][3], D1[5], D2[5];
    double R[3][3], T[3], E[3][3], F[3][3];
    CvMat _M1 = cvMat( 3, 3, CV_64F, M1 );
    CvMat _M2 = cvMat( 3, 3, CV_64F, M2 );
    CvMat _D1 = cvMat( 1, 5, CV_64F, D1 );
    CvMat _D2 = cvMat( 1, 5, CV_64F, D2 );
    CvMat _R = cvMat( 3, 3, CV_64F, R );
    CvMat _T = cvMat( 3, 1, CV_64F, T );
    CvMat _E = cvMat( 3, 3, CV_64F, E );
    CvMat _F = cvMat( 3, 3, CV_64F, F );

    if( displayCorners ) {
        cvNamedWindow( "corners", 1 );
    }

    // Чтение файла со структурой шахматной доски
    //

    if( !f ) {
        fprintf(stderr, "can not open file %s\n", imageList );
        return;
    }
}

```

```
}

for( i = 0; ;i++ ) {
    char buf[1024];
    int count = 0, result=0;
    lr = i % 2;
    vector<CvPoint2D32f>& pts = points[lr];

    if( !fgets( buf, sizeof(buf)-3, f ) ) {
        break;
    }

    size_t len = strlen(buf);
    while( len > 0 && isspace(buf[len-1]) ) {
        buf[--len] = '\0';
    }

    if( buf[0] == '#' ) {
        continue;
    }

    IplImage* img = cvLoadImage( buf, 0 );

    if( !img ) {
        break;
    }

    imageSize = cvGetSize(img);
    imageNames[lr].push_back(buf);

    // Поиск углов шахматной доски
    //
    for( int s = 1; s <= maxScale; s++ ) {
        IplImage* timg = img;

        if( s > 1 ) {
            timg = cvCreateImage(
                cvSize(img->width*s, img->height*s)
                ,img->depth
                ,img->nChannels
            );
            cvResize( img, timg, CV_INTER_CUBIC );
        }

        result = cvFindChessboardCorners(
            timg
            ,cvSize(nx, ny)
            ,&temp[0]
            ,&count
            ,CV_CALIB_CB_ADAPTIVE_THRESH |
            CV_CALIB_CB_NORMALIZE_IMAGE
        );
    }
}
```

```

    if( timg != img ) {
        cvReleaseImage( &timg );
    }

    if( result || s == maxScale ) {
        for( j = 0; j < count; j++ ) {
            temp[j].x /= s;
            temp[j].y /= s;
        }
    }

    if( result ) {
        break;
    }
}

if( displayCorners ) {
    printf("%s\n", buf);
    IplImage* cimg = cvCreateImage( imageSize, 8, 3 );
    cvCvtColor( img, cimg, CV_GRAY2BGR );

    cvDrawChessboardCorners(
        cimg
        ,cvSize(nx, ny)
        ,&temp[0]
        ,count
        ,result
    );

    cvShowImage( "corners", cimg );
    cvReleaseImage( &cimg );
}

if( cvWaitKey(0) == 27 ) { // ESC для выхода
    exit(-1);
}
else {
    putchar('.');
}
}

N = pts.size();
pts.resize( N + n, cvPoint2D32f(0,0) );
active[lr].push_back( (uchar)result );

// assert( result != 0 );

if( result ) {
    // Калибровка будет не совсем корректной без субпиксельной интерполяции
    //
    cvFindCornerSubPix(
        img
        ,&temp[0]
        ,count

```

```

        ,cvSize(11, 11)
        ,cvSize(-1,-1)
        ,cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 30, 0.01)
    );

    copy( temp.begin(), temp.end(), pts.begin() + N );
}

cvReleaseImage( &img );
}

fclose(f);
printf("\n");

// Сбор трехмерных точек шахматной доски в список
//
nframes = active[0].size(); // Количество найденных хороших шахматных досок
objectPoints.resize(nframes*n);
for( i = 0; i < ny; i++ ) {
    for( j = 0; j < nx; j++ ) {
        objectPoints[i*nx + j] = cvPoint3D32f(i*squareSize, j*squareSize, 0);
    }
}

for( i = 1; i < nframes; i++ ) {
    copy(
        objectPoints.begin()
        ,objectPoints.begin() + n
        ,objectPoints.begin() + i*n
    );
}

npoints.resize(nframes,n);
N = nframes*n;

CvMat _objectPoints = cvMat( 1, N, CV_32FC3, &objectPoints[0] );
CvMat _imagePoints1 = cvMat( 1, N, CV_32FC2, &points[0][0] );
CvMat _imagePoints2 = cvMat( 1, N, CV_32FC2, &points[1][0] );
CvMat _npoints = cvMat( 1, npoints.size(), CV_32S, &npoints[0] );

cvSetIdentity(&_M1);
cvSetIdentity(&_M2);
cvZero(&_D1);
cvZero(&_D2);

// Калибровка стерео камер
//
printf("Running stereo calibration ...");
fflush(stdout);

cvStereoCalibrate(
    &_objectPoints
    ,&_imagePoints1

```

```

        ,&_imagePoints2
        ,&npoints
        ,&_M1
        ,&_D1
        ,&_M2
        ,&_D2
        ,imageSize
        ,&R
        ,&T
        ,&E
        ,&F
        ,cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5)
        ,CV_CALIB_FIX_ASPECT_RATIO
        + CV_CALIB_ZERO_TANGENT_DIST
        + CV_CALIB_SAME_FOCAL_LENGTH
    );

printf(" done\n");

// Проверка качества калибровки.
// Т.к. конечная фундаментальная матрица неявно содержит
// всю конечную информацию, то оценка качества калибровки
// может быть проведена при помощи эпиполярной геометрии:
// m2^t * F * m1 = 0
//
vector<CvPoint3D32f> lines[2];
points[0].resize(N);
points[1].resize(N);
_imagePoints1 = cvMat(1, N, CV_32FC2, &points[0][0] );
_imagePoints2 = cvMat(1, N, CV_32FC2, &points[1][0] );
lines[0].resize(N);
lines[1].resize(N);
CvMat _L1 = cvMat(1, N, CV_32FC3, &lines[0][0]);
CvMat _L2 = cvMat(1, N, CV_32FC3, &lines[1][0]);

// Исправление искажений
//
cvUndistortPoints(
    &_imagePoints1
    ,&_imagePoints1
    ,&_M1
    ,&_D1
    ,0
    ,&_M1
);

cvUndistortPoints(
    &_imagePoints2
    ,&_imagePoints2
    ,&_M2
    ,&_D2
    ,0
    ,&_M2
);

```

```

);

cvComputeCorrespondEpilines( &_imagePoints1, 1, &_F, &_L1 );
cvComputeCorrespondEpilines( &_imagePoints2, 2, &_F, &_L2 );

double avgErr = 0;
for( i = 0; i < N; i++ ) {
    double err = fabs(
        points[0][i].x*lines[1][i].x
        + points[0][i].y*lines[1][i].y
        + lines[1][i].z
    ) +
    fabs(
        points[1][i].x*lines[0][i].x
        + points[1][i].y*lines[0][i].y
        + lines[0][i].z
    );
    avgErr += err;
}

printf( "avg err = %g\n", avgErr/(nframes*n) );

// Вычисление и отображение исправленных изображений
//
if( showUndistorted ) {
    CvMat* mx1      = cvCreateMat( imageSize.height, imageSize.width, CV_32F );
    CvMat* my1      = cvCreateMat( imageSize.height, imageSize.width, CV_32F );
    CvMat* mx2      = cvCreateMat( imageSize.height, imageSize.width, CV_32F );
    CvMat* my2      = cvCreateMat( imageSize.height, imageSize.width, CV_32F );
    CvMat* img1r    = cvCreateMat( imageSize.height, imageSize.width, CV_8U );
    CvMat* img2r    = cvCreateMat( imageSize.height, imageSize.width, CV_8U );
    CvMat* disp     = cvCreateMat( imageSize.height, imageSize.width, CV_16S );
    CvMat* vdisp    = cvCreateMat( imageSize.height, imageSize.width, CV_8U );

    CvMat* pair;
    double R1[3][3], R2[3][3], P1[3][4], P2[3][4];
    CvMat _R1 = cvMat(3, 3, CV_64F, R1);
    CvMat _R2 = cvMat(3, 3, CV_64F, R2);

    // Если откалибровано (метод Bouguet)
    if( useUncalibrated == 0 ) {
        CvMat _P1 = cvMat(3, 4, CV_64F, P1);
        CvMat _P2 = cvMat(3, 4, CV_64F, P2);

        cvStereoRectify(
            &_M1
            ,&_M2
            ,&_D1
            ,&_D2
            ,imageSize
            ,&_R
            ,&_T

```

```

        ,&_R1
        ,&_R2
        ,&_P1
        ,&_P2
        ,0
        ,0 /*CV_CALIB_ZERO_DISPARITY*/
    );

isVerticalStereo = fabs(P2[1][3]) > fabs(P2[0][3]);

// Предвычисление карт для cvRemap()
//
cvInitUndistortRectifyMap( &_M1, &_D1, &_R1, &_P1, mx1, my1 );
cvInitUndistortRectifyMap( &_M2, &_D2, &_R2, &_P2, mx2, my2 );
}

// иначе метод Hartley
else if( useUncalibrated == 1 || useUncalibrated == 2 ) {
    // использование внутренних параметров камеры, при этом
    // исправленные преобразования вычисляются непосредственно
    // за счет фундаментальной матрицы
    //
    double H1[3][3], H2[3][3], iM[3][3];
    CvMat _H1 = cvMat(3, 3, CV_64F, H1);
    CvMat _H2 = cvMat(3, 3, CV_64F, H2);
    CvMat _iM = cvMat(3, 3, CV_64F, iM);

    // Просто, чтобы показать независимое использование F
    //
    if( useUncalibrated == 2 ) {
        cvFindFundamentalMat(
            &_imagePoints1
            ,&_imagePoints2
            ,&_F
        );
    }
}

cvStereoRectifyUncalibrated(
    &_imagePoints1
    ,&_imagePoints2
    ,&_F
    ,imageSize
    ,&_H1
    ,&_H2
    ,3
);

cvInvert( &_M1, &_iM );
cvMatMul( &_H1, &_M1, &_R1 );
cvMatMul( &_iM, &_R1, &_R1 );
cvInvert( &_M2, &_iM );
cvMatMul( &_H2, &_M2, &_R2 );
cvMatMul( &_iM, &_R2, &_R2 );

```

```

    // Пред вычисление карт для cvRemap()
    //
    cvInitUndistortRectifyMap( &_M1, &_D1, &_R1, &_M1, mx1, my1 );
    cvInitUndistortRectifyMap( &_M2, &_D1, &_R2, &_M2, mx2, my2 );
} else {
    assert(0);
}

cvNamedWindow( "rectified", 1 );

// Исправление изображений и поиск карт искажений
//
if( !isVerticalStereo ) {
    pair = cvCreateMat( imageSize.height, imageSize.width*2, CV_8UC3 );
} else {
    pair = cvCreateMat( imageSize.height*2, imageSize.width, CV_8UC3 );
}

// Установки для поиска стерео соответствий
//
CvStereoBMState *BMState = cvCreateStereoBMState();
assert( BMState != 0 );

BMState->preFilterSize      = 41;
BMState->preFilterCap       = 31;
BMState->SADWindowSize      = 41;
BMState->minDisparity        = -64;
BMState->numberOfDisparities = 128;
BMState->textureThreshold   = 10;
BMState->uniquenessRatio    = 15;

for( i = 0; i < nframes; i++ ) {
    IplImage* img1=cvLoadImage(imageNames[0][i].c_str(),0);
    IplImage* img2=cvLoadImage(imageNames[1][i].c_str(),0);

    if( img1 && img2 ) {
        CvMat part;
        cvRemap( img1, img1r, mx1, my1 );
        cvRemap( img2, img2r, mx2, my2 );

        if( !isVerticalStereo || useUncalibrated != 0 ) {
            // При вертикальной ориентации стерео камеры,
            // useUncalibrated == 0, а изображение не транспонировано,
            // то эпиполярные линии исправленных изображений
            // вертикальны. Функция стерео соответствия в таком случае
            // не поддерживается
            //
            cvFindStereoCorrespondenceBM(
                img1r
                ,img2r
                ,disp
                ,BMState
            );
        }
    }
}

```

```

        cvNormalize( disp, vdisp, 0, 256, CV_MINMAX );
        cvNamedWindow( "disparity" );
        cvShowImage( "disparity", vdisp );
    }

    if( !isVerticalStereo ) {
        cvGetCols( pair, &part, 0, imageSize.width );
        cvCvtColor( img1r, &part, CV_GRAY2BGR );
        cvGetCols( pair, &part, imageSize.width, imageSize.width*2 );
        cvCvtColor( img2r, &part, CV_GRAY2BGR );

        for( j = 0; j < imageSize.height; j += 16 ) {
            cvLine(
                pair
                ,cvPoint(0, j)
                ,cvPoint(imageSize.width*2, j)
                ,CV_RGB(0, 255, 0)
            );
        }
    } else {
        cvGetRows( pair, &part, 0, imageSize.height );
        cvCvtColor( img1r, &part, CV_GRAY2BGR );
        cvGetRows( pair, &part, imageSize.height, imageSize.height*2 );
        cvCvtColor( img2r, &part, CV_GRAY2BGR );

        for( j = 0; j < imageSize.width; j += 16 ) {
            cvLine(
                pair
                ,cvPoint(j,0)
                ,cvPoint(j,imageSize.height*2)
                ,CV_RGB(0,255,0)
            );
        }
    }

    cvShowImage( "rectified", pair );

    if( cvWaitKey() == 27 ) {
        break;
    }
}

cvReleaseImage( &img1 );
cvReleaseImage( &img2 );
}

cvReleaseStereoBMState(&BMState);
cvReleaseMat( &mx1 );
cvReleaseMat( &my1 );
cvReleaseMat( &mx2 );
cvReleaseMat( &my2 );
cvReleaseMat( &img1r );

```

```

        cvReleaseMat( &img2r );
        cvReleaseMat( &disp );
    }

int main(void) {
    Stereocalib( "list.txt", 9, 6, 1 );
    return 0;
}

```

Карты глубины трехмерного перепроектирования

Многие алгоритмы напрямую работают с картами различий - например, для обнаружения наличия объекта на столе. Но для сопоставления трехмерных форм, обучение трехмерной модели, обучение захвату робота и так далее необходимо трехмерное реконструирование или карта глубины. К счастью, все ранее рассматриваемые стерео оборудования с легкостью с этим справляются. Учитывая матрицу перепроектирования Q размера 4x4, различия d и двумерную точку (x, y) можно получить трехмерную глубину при помощи:

$$Q \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

где трехмерные координаты представляют собой (X/W, Y/W, Z/W). Примечательно, что матрица Q может кодировать, а может и не кодировать, линии камер, где сходится поле зрения (перекрестный взгляд), так же как и базовую линию камеры и основные точки на обоих изображениях. Как результат, не нужно явно учитывать сходящиеся или параллельно фронтальные камеры, а вместо этого можно просто извлечь глубину в результате перемножения матриц. В OpenCV для всего этого имеется две функции. Первая, которая уже должна быть знакома, работает с массивом точек и связанными с ними различиями.

```

void cvPerspectiveTransform(
    const CvArr*    pointsXYD
    ,CvArr*         result3DPoints
    ,const CvMat*   Q
);

```

Вторая функция работает с целыми изображениями:

```
void cvReprojectImageTo3D(
    CvArr* disparityImage
    ,CvArr* result3DImage
    ,CvArr* Q
);

```

Эта функция принимает одноканальное изображение *disparityImage* и преобразует координаты (x, y) каждого пикселя наряду с пикселями различий (т.е. вектором $[x \ y \ d]^T$) в соответствующие трехмерные точки (X/W , Y/W , Z/W) при помощи матрицы перепроектирования Q размера 4x4. Конечное трехканальное вещественное (или 16-ти битное целое) изображение получается того же размера, что и исходное изображение.

При этом, обе функции позволяют передавать произвольные перспективные преобразования, вычисленные *cvStereoRectify()*, или совмещать их для трехмерного вращения, перемещения и т.д.

Результаты использования *cvReprojectImageTo3D()* на изображении кружки и кресла показаны на рисунке 12-17.

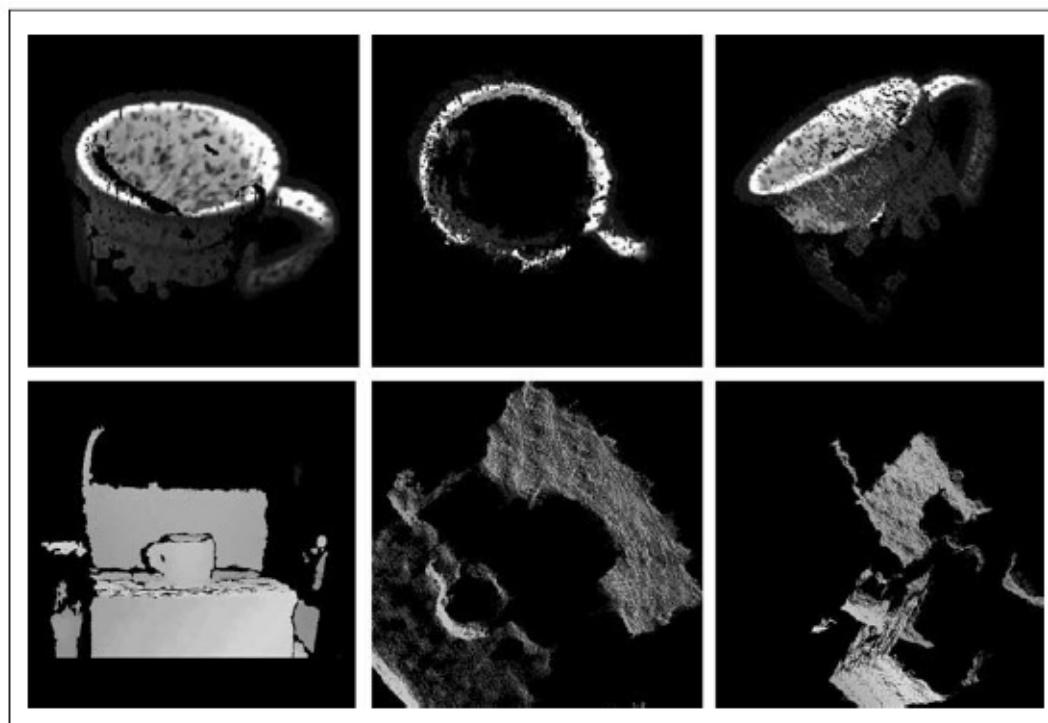


Рисунок 12-17. Пример карт глубины (для кружки и кресла), вычисленные при помощи *cvFindStereoCorrespondenceBM()* и *cvReprojectImageTo3D()*

[П]||[РС]||(РП) Структура движения

Структура движения является важной темой в мобильной робототехнике, а также при анализе видеоизображения, получаемого, например, с переносной видеокамеры. Тема структуры движения является обширной и уже многое было исследовано в этом направлении. Вместе с тем многое может быть достигнуто за счет одного простого наблюдения: изображение статической сцены, снятое при помощи перемещающейся камеры, ничем не отличается от изображения, снятого со второй камеры. Таким образом все ранее рассмотренные математические и алгоритмические техники необходимо перенести в эту ситуацию. При этом термин "статический" имеет решающее значение, а во многих практических ситуациях, где имеется статическая или достаточно статическая сцена, несколько перемещающихся точек можно рассматривать как выбросы надежных методов аппроксимации.

Пусть камера перемещается внутри здания. Если окружающая среда относительно богата на распознаваемые особенности, которые могут быть найдены при помощи методов оптического потока, например, такого, как `cvCalcOpticalFlowPyrLK()`, то появляется возможность для вычисления соответствий между достаточным количеством точек - от кадра к кадру – не только для реконструирования траектории камеры (эта информация кодируется существенной матрицей E , которая в свою очередь может быть вычислена при помощи фундаментальной матрицы F и внутренней матрицы камеры M), но также, косвенно, обобщать трехмерную структуру здания и положения всех вышеупомянутых особенностей в этом здании. Функция `cvStereoRectifyUncalibrated()` запрашивает только фундаментальную матрицу для того, чтобы вычислить основную структуру сцены вплоть до масштабирования.

[П]|[РС]|(РП) Конструирование двух и трехмерных линий

В заключении будет рассмотрено конструирование линий. Данная задача может возникнуть по многих причинам и во многих контекстах. Данная тема выбрана не случайно, т.к. одной из наиболее часто возникающей ситуацией, с которой связано конструирование линии, является анализ трехмерных точек (хотя описываемые в данном разделе функции могут конструировать и двумерные линии). Алгоритмы конструирования линии как правило используют статистически надежные методы. В OpenCV алгоритм конструирования линии представлен функцией *cvFitLine()*.

```
void cvFitLine(
    const CvArr*  points
, int          dist_type
, double       param
, double       reps
, double       aeps
, float*       line
);
```

Массив *points* может быть матрицей вещественных значений размера Nx2 или Nx3 (вмещающая двух или трех мерные точки) или последовательностью структур *cvPointXXX*. (В данном случае *XXX* может быть заменено на *2D32f* или *3D64f*).

Аргумент *dist_type* это метрика расстояния, которая должна быть сведена к минимуму для всех точек (таблица 12-3).

Таблица 12-3. Метрики, используемые для вычисления значений *dist_type*

Значение dist_type	Метрика
CV_DIST_L2	$\rho(r) = \frac{r^2}{2}$
CV_DIST_L1	$\rho(r) = r$
CV_DIST_L12	$\rho(r) = \left[\sqrt{1 + \frac{r^2}{2}} - 1 \right]$
CV_DIST_FAIR	$\rho(r) = C^2 \left[\frac{r}{C} - \log\left(1 + \frac{r}{C}\right) \right], C = 1.3998$
CV_DIST_WELSCH	$\rho(r) = \frac{C^2}{2} \left[1 - \exp\left(-\frac{r^2}{C^2}\right) \right], C = 2.9846$
CV_DIST_HUBER	$\rho(r) = \begin{cases} r^2/2 & r < C \\ C(r-C/2) & r \geq C \end{cases}, C = 1.345$

Параметр *param* используется для установки параметра С из таблицы 12-3. В случае установки данного параметра в 0, значение будет соответствовать табличному в зависимости от выбора.

Аргумент *line* это место, куда будет сохранен результат. Если *points* это массив Nx2, то *line* должен быть указателем на вещественный массив размера четыре (т.е. *float array[4]*). Если *points* это массив Nx3, то *line* должен быть указателем на вещественный массив размера шесть (т.е. *float array[6]*). В первом случае возвращаемыми значениями будут (v_x, v_y, x_0, y_0), где (v_x, v_y) это нормированный вектор параллельный конструируемой линии, а (x_0, y_0) это точка на линии. Аналогичным образом, для трехмерного случая, возвращаемыми значениями будут ($v_x, v_y, v_z, x_0, y_0, z_0$), где (v_x, v_y, v_z) это нормированный вектор параллельный конструируемой линии, а (x_0, y_0, z_0) является точкой на этой линии. Учитывая представление линии, оценка точности параметров *reps* и *aeps* следующая: *reps* требуется оценки точности $x0, y0, z0$, а *aeps* требуется угловая точность vx, vy, vz . В документации по OpenCV рекомендуется использовать значение точности 0.01 в обоих случаях.

cvFitLine() может конструировать двух или трех мерные линии. Конструирование двумерных линий обычно важно в общем (пример 12-4, автор книги выражает благодарность за предоставленный пример **Vadim Pisarevsky**), а конструирование трехмерных линий крайне важно в OpenCV в частности (глава 14). В начале представленного примера синтезируются некоторые двумерные шумовые точки вокруг линии, а затем добавляются случайные точки, которые не имеют ничего общего с линией (и именуемые *выбросами*), а в заключении конструируется линия по

полученным точкам и отображается на экране. Функция *cvFitLine()* хорошо справляется с выбросами; это крайне важно в реальных приложениях, где некоторые измерения могут быть сильно повреждены из-за шума, отказа датчика или по каким-либо другим причинам.

Пример 12-4. Конструирование двумерной линии

```
#include "cv.h"
#include "highgui.h"
#include <math.h>

int main( int argc, char** argv ) {
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG( -1 );

    cvNamedWindow( "fitline", 1 );

    for(;;) {
        char key;
        int i;
        int count      = cvRandInt(&rng)%100 + 1;
        int outliers   = count/5;
        float a         = cvRandReal(&rng)*200;
        float b         = cvRandReal(&rng)*40;
        float angle     = cvRandReal(&rng)*CV_PI;
        float cos_a     = cos(angle);
        float sin_a     = sin(angle);
        CvPoint pt1, pt2;
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]) );
        CvMat pointMat = cvMat( 1, count, CV_32SC2, points );
        float line[4];
        float d, t;

        b = MIN( a*0.3, b );

        // Генерация точек, близких к линии
        //
        for( i = 0; i < count - outliers; i++ ) {
            float x = (cvRandReal(&rng)*2-1)*a;
            float y = (cvRandReal(&rng)*2-1)*b;
            points[i].x = cvRound(x*cos_a - y*sin_a + img->width/2);
            points[i].y = cvRound(x*sin_a + y*cos_a + img->height/2);
        }

        // Генерация "completely off" точек
        //
        for( ; i < count; i++ ) {
            points[i].x = cvRandInt(&rng) % img->width;
            points[i].y = cvRandInt(&rng) % img->height;
        }
    }
}
```

```
// Поиск оптимальной линии
//
cvFitLine( &pointMat, CV_DIST_L1, 1, 0.001, 0.001, line );
cvZero( img );

// Отображение точек
//
for( i = 0; i < count; i++ ) {
    cvCircle(
        img
        ,points[i]
        ,2
        ,(i < count - outliers) ? CV_RGB(255, 0, 0) : CV_RGB(255, 255, 0)
        ,CV_FILLED
        ,CV_AA
        ,0
    );
}

// ... и достаточно длинной линии, чтобы пересечь все изображение
//
d = sqrt( (double)line[0]*line[0] + (double)line[1]*line[1] );
line[0] /= d;
line[1] /= d;
t = (float)(img->width + img->height);
pt1.x = cvRound(line[2] - line[0]*t);
pt1.y = cvRound(line[3] - line[1]*t);
pt2.x = cvRound(line[2] + line[0]*t);
pt2.y = cvRound(line[3] + line[1]*t);
cvLine( img, pt1, pt2, CV_RGB(0,255,0), 3, CV_AA, 0 );

cvShowImage("fitline", img );

key = (char) cvWaitKey(0);

if( key == 27 || key == 'q' || key == 'Q' ) { // 'ESC'
    break;
}

free( points );
}

cvDestroyWindow( "fitline" );

return 0;
}
```

[П]|[РС]|(РП) Упражнения

1. Откалибруйте камеру, используя `cvCalibrateCamera2()` и не менее 15 изображений шахматной доски. Затем, используя `cvProjectPoints2()`, спроектируйте стрелку, ортогональную к каждому изображению шахматной доски (нормаль к поверхности) при помощи векторов вращения и перемещения, полученных в результате калибровки камеры.
2. *Трехмерный джойстик.* Используйте простой заранее известный объект, по крайней мере с четырьмя измерениями, некомпланарными, отслеживаемыми особенными точками, являющимися входами для алгоритма POSIT. Используйте данный объект, как 3D джойстик для перемещения небольшой фигурки на изображении.
3. В примере "с высоты птичьего полёта" с камерой, располагающейся над плоскостью и смотрящей горизонтально вдоль плоскости, было показано, что гомография плоскости грунта является действительной до линии горизонта. Как можно обойти это ограничение? И почему это так сложно сделать?

Подсказка: Нарисуйте линии на равном расстоянии последовательности точек на плоскости, выходящие от камеры. Какое влияние оказывает угол камеры на каждую последующую точку в сравнении с предшествующей до неё точки?

4. Реализуйте пример "с высоты птичьего полета" с камерой, смотрящей на плоскость грунта. Запустите пример в режиме реального времени и изучите, что происходит, когда перемещаются объекты в нормальном изображении по сравнению с изображением с высоты птичьего полёта.
5. Установите две или одну камеру так, чтобы они (она) захватывала две картинки.
 - a. Вычислите, сохраните и изучите фундаментальную матрицу.
 - b. Повторите вычисление фундаментальной матрицы несколько раз. Насколько стабильны вычисления?
6. Имея откалиброванную стерео камеру и отслеживая перемещения точек на обеих камерах, придумайте способ с использованием фундаментальной матрицы, чтобы найти ошибки отслеживания.
7. Вычислите и нарисуйте эпиполяные линии двух камер, установленные для получения стерео.

8. Установите две камеры так, чтобы можно было осуществлять стерео исправления и эксперименты с точностью глубины.
 - a. Что произойдёт при размещении зеркала в сцене?
 - b. Измените количество текстур в сцене и огласите результаты.
 - c. Поэкспериментируйте с различными методами *disparity* и огласите результаты.
9. Настройте стерео камеру и направьте в место, соответствующее текстурам одной из ваших рук. Установите линию на руке, используя все методы *dist_type*. Сравните точность и надежность различных методов.

Машинное обучение

[П]|[РС]|(РП) Что такое машинное обучение

Целью *машинного обучения (ML)* является преобразование данных в информацию. После изучения набора данных, у машин появляется возможность ответить на вопросы о данных: Какие данные наиболее близки к имеющимся данным? Присутствует ли автомобиль на изображении? На какую рекламу будут реагировать покупатели? Наиболее часто используемой компонентой является стоимость, соответственно возникает следующий вопрос: "Какой продукт, из имеющегося набора, с наивысшей стоимостью будет выбран покупателем при показе рекламы?" Машинное обучение преобразует данные в информацию, извлекая правила и шаблоны из этих данных.

Тема машинного обучения является довольно таки обширной. OpenCV в основном занимается статистикой, а не такими вещами, как "Байесовская сеть", "Марковское случайное поле" или "графические модели". Довольно-таки хорошие статьи по данной теме можно найти у: Hastie, Tibshirani, и Friedman; Duda и Hart; Duda, Hart, и Stork; и Bishop. Можно изучить работы Ranger et al. и Chu et al для того, чтобы узнать, как распараллелить машинное обучение.

Обучение и тестирование

Машинное обучение работает с такими данными, как температурные значения, цены на акции, интенсивность окраски и т.д. Зачастую данные предварительно перерабатываются в особенности. Можно, например, имея базу данных из 10000 изображений лица, запустить определитель контура лица на этих лицах с целью накопления таких особенностей, как постановка и четкость контура, с целью определения центра лица для каждого лица в отдельности. В результате можно получить, например, 500 таких значений/лиц или вектор особенностей с 500 записями. Таким образом, методы машинного обучения могут быть использованы для построения своего рода модели по данному набору данных. Если же необходимо только сгруппировать лица (широкие, узкие и т.д.), то лучше всего использовать *алгоритм кластеризации*. Если же необходимо научиться предсказывать возраст человека по шаблону контура лица или лица в целом, то лучше всего использовать *алгоритм классификации*. Для достижения всех этих целей, алгоритмы машинного обучения анализируют набор особенностей и регулируют соответствующие веса, пороги и другие параметры для получения наилучшего результата. Этот процесс корректировки параметров для удовлетворения цели именуется *обучением*.

Всегда важно знать, как работают методы машинного обучения, но это ещё и своего рода "ювелирная" работа. Как правило, исходный набор данных разбивается на большую выборку для обучения (например, 9000 лиц из ранее рассмотренного примера) и на малую выборку для тестов (из оставшихся 1000 лиц). В начале необходимо будет запустить классификатор по выборке для обучения с целью получения модели прогнозирования возраста по полученному вектору особенностей. А затем протестировать полученный классификатор на оставшейся выборке для тестов.

Выборка для тестов не используется в обучении. Классификатор запускается на каждом лице (всего 1000 лиц) из выборки для тестов с последующей фиксацией того, насколько хорошим получается предсказание при сопоставлении полученного варианта возраста с реальным показателем возраста. Если классификатор работает плохо, то можно попробовать добавить новые особенности или рассмотреть другой тип классификатора. Далее в данной главе будут рассмотрены некоторые виды классификаторов и алгоритмы для их обучения.

Если классификатор хорошо справляется с поставленной задачей, то можно судить о получении потенциально ценной модели, которая может быть использована на реальных данных. Возможно использование полученной системы, например, для установки режима в видеоигре в соответствии с возрастом. Перед игрой, его или её лицо обрабатывается для получения 500 (постановка и четкость контура, центр лица) особенностей. Затем эти данные передаются в классификатор; возвращаемое значение возраста приводит к установке соответствующего поведения в игре. При развертывании модели, классификатор рассматривает лица, которые не видел ранее и принимает решение в соответствии с тем, что узнал по выборке для обучения.

В заключении, зачастую при развертывании системы классификации, задействуется набор данных для проверки. Иногда испытание системы в самом конце — это слишком трудоёмкая задача. Зачастую требуется настроить параметры именно перед отправкой классификатора на окончательное тестирование. Сделать это можно, разбив исходное множество данных 10000 лиц на три части: на выборку для обучения из 8000 лиц, на выборку для проверки из 1000 лиц и на выборку для тестирования из 1000 лиц. Теперь при запуске классификатора на выборке для обучения, можно "в фоновом режиме" пройтись и по выборке для проверки. И только после подтверждения верности проделанной работы на выборке для проверки можно будет запустить классификатор на выборке для тестирования для получения окончательного решения.

Контролируемые и неконтролируемые данные

Иногда данные не имеют меток; все что требуется, это просто посмотреть каким образом можно сгруппировать лица, основываясь на информации о крае. А иногда данные имеют метки, например, такую, как возраст. В случае с машинным обучением это означает, что обучение может быть *контролируемым* (т.е. используется обучение "сигнала" или "метки", поступающие от вектора особенностей). Если вектор данных немеченый, то машинное обучение *неконтролируемое*.

Контролируемое обучение может быть *категоричным*, т.к. обучение ассоциируется с именованием лиц, или данные могут быть *пронумерованы* или *упорядочены* метками, такой как возраст. Когда данные поименованы (категоричны), то можно сказать, что выполнена *классификация*. Когда данные пронумерованы, то можно сказать, что выполнена *регрессия*.

Контролируемое обучение сопряжено с преобразованиями в оттенки серого. При таком виде обучения используется попарное сопоставление один к одному меток вектора данных; в добавок к этому может быть использовано *отложенное обучение* (иногда так же именуемое как *подкрепленное обучение*). В подкрепленном обучении, метка данных (так же именуемая *наградой* или *наказанием*) может быть получена намного позже после исследования искомого вектора данных предмета наблюдения.

Например, в случае перемещения мыши по лабиринту в поисках пищи, она может сделать несколько кругов, прежде чем найдет пищу, т.е. вознаграждение. В свою очередь, это вознаграждение должно каким-то образом оказать влияние на все предыдущие мнения и действия, которые совершила мышь в процессе поиска еды. Подкрепленное обучение работает аналогичным образом: система получает сигнал с задержкой (награду или наказание) и пытается предсказать поведение для будущих запусков (способ принятия решений; например, в какую сторону идти в лабиринте на каждом последующем шаге). Контролируемое обучение может так же использовать частичную маркировку, т.е. когда не хватает некоторых меток (это так называемое *полуконтролируемое обучение*), или шумовые метки, т.е. когда некоторые метки ошибочны. Большинство алгоритмов машинного обучения обрабатывают только одну или две из описанных ситуаций. Например, алгоритмы машинного обучения могут выполнить классификацию, но не могут выполнить регрессию; алгоритм в состоянии выполнить полу контролируемое обучение, но не подкрепленное обучение; алгоритм в состоянии иметь дело с числовыми данными, но не с категоричными; и так далее.

В реальных случаях приходиться иметь дело с немечеными данными и проверкой естественности попадания данных в группы. Алгоритмы, использующие неконтролируемое обучение, именуются *алгоритмами кластеризации*. В этом случае, целью является группировка немеченых данных, которые "близки". В простом случае может возникнуть необходимость просто посмотреть, как распределяются лица: Какие из них тонкие или широкие, длинные или короткие? Например, рассматривая данные, поступающие о раках, возникает вопрос: как кластеризовать различные виды раков на

группы, имеющие различные химические сигналы? Неконтролируемые кластеризованные данные зачастую используются при формировании вектора особенностей для высокоуровневых контролируемых классификаторов. Для начала можно сформировать кластер лиц по типу лица (широкие, узкие, длинные, короткие), а затем использовать его в качестве исходных данных для обработки других данных, например, таких, как набор средних вокальных частот для предсказания пола человека.

Эти две общие задачи машинного обучения, классификация и кластеризация, пересекаются с двумя наиболее общими задачами в компьютерном зрении: распознаванием и сегментацией. Эти задачи иногда так же называют как "что" и "где". То есть, зачастую необходимо, чтобы компьютер назвал объекты на изображении (распознал или "что"), а также указал место этого объекта (сегментация или "где"). Так как в компьютерном зрении интенсивно используется машинное обучение, в OpenCV включено множество мощных алгоритмов машинного обучения в виде библиотеки, расположенной в `../opencv/ml`.

Код в OpenCV, отвечающий за машинное обучение, является обобщенным. Т.е. хотя этот код является полезным для задач компьютерного зрения, сам по себе код не специфичен для компьютерного зрения. С его помощью, например, можно узнать геномные последовательности за счет соответствующих процедур. При этом основная задача сводится к управлению объектом, заданный вектором особенностей и полученный от изображения.

Генеративные и Дискриминативные модели

Многие алгоритмы были разработаны для выполнения обучения и кластеризации. OpenCV поддерживает некоторые наиболее полезные и доступные в настоящее время статистические подходы машинного обучения. Вероятностные подходы машинного обучения, такие как Байесовые сети или графические модели, менее хорошо поддерживаются в OpenCV прежде всего из-за того, что являются более новыми и до сих пор в стадии активного развития. OpenCV стремиться поддерживать *дискриминативные алгоритмы*, для которых характерна вероятность метки относительно данных ($P(L|D)$), а не *генеративные алгоритмы*, для которых характерна вероятность данных относительно метки ($P(D|L)$). Хотя различия и не всегда понятны, дискриминативные модели хороши для высокопроизводительных прогнозов по заданным данным, в то время, как генеративные модели хороши для более мощного представления данных или для условного синтеза новых данных (имея "воображаемого" слона, можно сгенерировать данные по условию "слон").

Зачастую проще объяснить генеративную модель, т.к. эти модели являются причиной (правильных или неправильных) данных. Дискриминативное обучение зачастую сводится к принятию решения на основе некоторого порогового значения, которое может быть произвольным. Например, если предположить, что участок дороги определяется в сцене по большому счету потому, что составляющая "красного" цвета меньше 125. При этом возникает вопрос: будет ли участок, составляющая "красного" цвета которого равна 126, соответствовать дороге? Такого рода вопросы довольно таки трудно интерпретировать. В случае с генеративными моделями, как правило, приходиться иметь дело с условными распределениями данных по заданным категориям, что позволит развить чувство "близости" к полученному распределению.

Алгоритмы машинного обучения в OpenCV

Алгоритмы машинного обучения, реализованные в OpenCV, приведены в таблице 13-1. Все эти алгоритмы находятся в библиотеке **ML**, за исключением *Mahalanobis* и *K-means*, которые находятся в **CVCORE**, и *face detection*, который располагается в **CV**.

Таблица 13-1. Алгоритмы машинного обучения, реализованные в OpenCV

Алгоритм	Описание
Mahalanobis	Мера расстояния, рассчитываемая для "тягучего" пространства данных, разделенное на ковариационные данные. Если ковариация является единичной матрицей (идентичная дисперсии), то эта мера совпадает с мерой Евклидова расстояния
K-means	Неконтролируемый алгоритм кластеризации, который представляет собой распределение данных с использованием K центров, где K задается пользователем. Разница между этим алгоритмом и expectation maximization заключается в том, что в этом алгоритме центры не гауссова и как результат кластеры похожи на мыльные пузыри, т.к. центры конкурируют за "владение" ближайшими точками данных. Эти кластерные области зачастую используются как разряженные гистограммы бинов для представления данных. Изобретен Steinhaus, использован Lloyd
Normal/Naïve Bayes classifier	Генеративный классификатор, в котором допускаются особенности с гауссовым распределением и статически независимые друг от друга; сильное предположение, как правило, не верно. По этой причине данный алгоритм зачастую называют "наивно Баесовым" классификатором. Тем не менее этот метод зачастую работает на удивление хорошо
Decision trees	Дискриминативный классификатор. В дереве ищется одна особенность и пороговое значение текущего узла, которое наилучшим образом разделяет данные на отдельные классы. Данные разделяются, и данная процедура рекурсивно повторяется вниз слева направо по ветвям дерева. Не так часто результат можно получить на самой вершине дерева, однако, это первое,

	что необходимо попробовать сделать, т.к. это быстро и имеет высокую функциональность
Boosting	Группа дискриминативных классификаторов. Обобщенное классификационное решение получается за счет комбинирования взвешенных классификационных решений группы классификаторов. В ходе обучения изучается группа классификаторов (по одному за раз). Каждый классификатор из группы является "слабым" классификатором (с неким шансом на повышение производительности). Эти слабые классификаторы, как правило, состоят из единой переменной деревьев решений, называемой "stumps". В ходе обучения, найденный stump обучает решения классификатора, а также определяет вес его "голоса". Между обучениями каждый классификатор, один за одним, повторно взвешивает точки так, что наибольшее внимание уделяется точкам, где были сделаны ошибки. Этот процесс продолжается до тех пор, пока суммарное значение ошибок, возникающих от комбинирования взвешенных голосов дерева решений на множестве данных, не станет меньше установленного порога. Этот алгоритм зачастую эффективен, при наличии большого объема данных
Random trees	Дискриминативный лес множества decision trees, каждое из которых построено с большой или максимально расщепленной глубиной. В ходе обучения для каждого узла каждого дерева разрешено выбирать переменные расщепления только из случайного подмножества особенностей. Это гарантирует то, что каждое дерево становится статистически независимым в принятии решения. Во время применения, каждое дерево получает взвешенный голос. Этот алгоритм зачастую очень эффективен, а также позволяет выполнять регрессию за счет усреднения выходных значений каждого дерева
Face detector / Haar classifier	Приложение, распознающее некий объект и базирующееся на основе разумного использования алгоритма boosting. OpenCV содержит алгоритм обучения для определения лица, который на удивление хорошо работает. Алгоритм обучения так же можно применить на других объектах за счет прилагаемого ПО. Данный алгоритм хорошо работает в случае твердых объектов и характерных представлений
Expectation maximization (EM)	Генеративный бесконтрольный алгоритм, который используется для кластеризации. Ему соответствует N многомерных Гауссиан, где N задается пользователем. Данный алгоритм может быть эффективным способом представления более сложного распределения у которого только несколько параметров (среднее значение и дисперсия). Зачастую используется в сегментации
K-nearest neighbors	Самый простой дискриминативный классификатор. Обучение состоит в простом сохранении меток. После этого контрольная точка классифицируется в соответствии с большинством голосов её ближайших K других точек (в евклидовом смысле близости). Это, скорее всего, самое простое, что можно сделать. Зачастую данный алгоритм является эффективным, но вместе с тем он медленно работает и требует много памяти

Neural networks / Multilayer perceptron (MLP)	Дискриминативный алгоритм, который (почти всегда) имеет "скрытые единицы" между выходными и входными узлами для наилучшего представления входного сигнала. Обучение может быть медленным, однако, во время применения данный алгоритм очень быстрый. Тем не менее, главной областью применения является распознавание букв
Support vector machine (SVM)	Дискриминативный классификатор, который выполняет регрессию. Функция расстояния определяется между любыми двумя точками многомерного пространства. Алгоритм "узнает" разделенные гиперплоскости, которые максимально разделяют классы в верхнем измерении. Алгоритм стремится быть в числе лучших на ограниченном наборе данных, проигрывая boosting или random trees в случае большого объема данных

Использование машинного обучения в компьютерном зрении

В общем все алгоритмы из таблицы 13-1 в качестве входного значения принимают вектор данных, состоящий из множества особенностей, где число особенностей может исчисляться тысячами. Предположим, имеется задача распознать определенного типа объект, например, человека. Первая проблема, с которой придётся столкнуться, заключается в том, чтобы собрать и обучить метки, которые соответствуют положительным (человек присутствует в сцене) и отрицательным (человек отсутствует в сцене) случаям. Вскоре будет показано, что представление людей может быть разных масштабов: в виде нескольких пикселей на изображении или какой-то части тела, например, ухо. И даже хуже, люди зачастую будут перекрыты: мужчина внутри автомобиля; одна нога, видимая из-за дерева; лицо женщины. К тому же необходимо определение того, что именно имеется ввиду, когда говорят о человеке в сцене.

Сбор данных является следующей проблемой. Возникают следующие вопросы: собирать ли информацию о движениях; собирать ли другую информацию (такую, как время, сезон, температура, открытые ворота в сцене). Алгоритм, который ищет людей на пляже, может неверно работать в случае поиска людей на горнолыжном склоне. Необходимо фиксировать изменчивые данные: различные взгляды людей, различные источники света, погодные условия, тени и т.д.

После выполнения сбора большого объема данных, возникает вопрос как выделить метки? В начале необходимо определить, что такое "метка". При этом необходимо ли знать, где находится человек в сцене? Какие действия (бег, ходьба, ползанье, порядок следования) важны? В конечном счёте может потребоваться обработать миллион или более изображений. И как собственно маркировать всё это? Есть множество способов

для выполнения исключения фона в контролируемых условиях и сбора сегментированных передних планов людей в сцене. Для выполнения классификации можно использовать платные или бесплатные сервисы.

После выполнения маркировки данных, необходимо решить, какие особенности необходимо выделять на объектах. Опять же необходимо знать, что делать после этого. Если люди всегда располагаются с правой стороны, то нет причин для использования инвариантно-вращающихся особенностей и выполнения вращения объектов заранее. В общем, необходимо искать особенности, которые выражают некоторую инвариантность объектов, такие как масштабно-независимые гистограммы градиентов или цветов или популярные SIFT особенности (Lowe's SIFT feature demo (<http://www.cs.ubc.ca/~lowe/keypoints/>)). Если в сцене присутствует информация о фоне, то в начале может потребоваться удалить его, чтобы потом можно было выделить другие объекты. Затем может потребоваться выполнение обработки изображения, которая в свою очередь может состоять из нормализации изображения (масштабирование, вращение, гистограммы выравнивания и т.д.) и вычисление множества различных типов особенностей. Каждый из полученных векторов данных имеет метку, связанную с объектом, действием или сценой.

Зачастую после того, как данные собраны и размещены в векторах особенностей, возникает необходимость в разбиении полученных данных во время обучения, проверки и тестирования. Это "лучший способ" выполнить обучение, проверку и тестирование в рамках перекрестной проверки. Т.е. данные разбиваются на K подмножеств и выполняется множество итераций обучения (возможно проверки) и тестирования, где каждая итерация состоит из различного набора данных, принимающие на себя роль обучения (проверки) и тестирования (как правило, одно обучение (возможно проверка) и цикл тестирования от 5 до 10 раз). Затем результаты тестирования отдельных итераций усредняются для получения окончательного результата. Перекрёстная проверка даёт более точную картину того, как будет работать классификатор на новом наборе данных. (В дальнейшем об этом будет рассказано более подробно.)

Теперь, когда данные сформированы, необходимо выбрать классификатор. Зачастую выбор классификатора продиктован соображениями в пользу вычислений, данных или памяти. Для некоторых приложений, таких как онлайн строитель модели предпочтений пользователя, необходимо, чтобы обучение классификатора происходило быстро. В таком случае, такие алгоритмы, как nearest neighbors, normal Bayes или decision trees будут хорошим выбором. Если делается упор на память, то decision trees или neural networks наиболее эффективны. Если компонент времени обучения классификатора не критичен, но при этом важна скорость его работы, то neural networks хороший выбор, так же, как и normal Bayes classifiers и support vector machines. Если компонент времени обучения классификатора не критичен, но при этом важна точность, то

boosting и random trees то что нужно. Если же необходим простой и понятный способ проверки выбранных особенностей, то decision trees или nearest neighbors хороший выбор. В первую очередь "из коробки" стоит попробовать boosting или random trees.

Усреднив все возможные типы распределения данных, все классификаторы выполняют одно и тоже. Таким образом, нельзя сказать, какой алгоритм из таблицы 13-1 является "лучшим". Для любого заданного распределения данных имеется наилучший классификатор. Поэтому на реальном наборе данных лучше всего испробовать максимально возможное количество классификаторов. В зависимости от преследуемых целей - произвести верный или быстрый расчёт; интерпретировать данные - различные классификаторы ведут себя по разному.

Переменная важности

Два алгоритма из таблицы 13-1 позволяют оценить эту переменную (более известную как "переменная важности"). Возникает вопрос: как, учитывая вектор особенностей, определить важность этих особенностей для точности классификатора? Binary decision trees делают это напрямую: они обучаются за счет выбора переменной, которая наилучшим образом разделяет данные каждого узла. Переменная верхнего узла является наиболее важной переменной; переменные следующего уровня являются следующими важными переменными и так далее. Random trees могут вычислить важную переменную, используя технику Leo Breiman (данная техника описана в работе "Looking Inside the Black Box" (www.stat.berkeley.edu/~breiman/wald2002-2.pdf)); эта техника может быть применима к любому классификатору, но на данный момент в OpenCV реализована только для decision и random trees.

Один из возможных вариантов использования важной переменной сводиться к уменьшению числа особенностей, которые классификатор должен учитывать. Начиная с большого количества особенностей, классификатор обучается и впоследствии ищется важная переменная для каждой особенности относительно других особенностей. Впоследствии можно отбросить незначительные особенности. Исключение незначительных особенностей увеличит скоростные характеристики (т.к. будет исключена обработка для вычисления этих особенностей), тем самым обучение и тестирование становится быстрее. Кроме того, в случае недостаточного объема данных, что зачастую и бывает, за счет исключения незначительных переменных можно повысить точность классификации; это ускоряет обработку с последующим получением лучшего результата.

Алгоритм получения важной переменной способом Breiman состоит из следующих шагов:

1. Обучение классификатора на множестве для обучения.
2. Использование множеств для проверки или тестирования для повышения точности классификатора.
3. Для каждой точки данных и выбранной особенности случайным образом выбирается новое значение для этой особенности среди значений особенностей из оставшегося набора данных (так называемая "выборка с заменой"). Это гарантирует, что распределение для этого признака будет оставаться тем же, как и в оригинальном наборе данных, только теперь актуальная структура или значение этой особенности стирается (потому что значение выбирается случайным образом из оставшегося набора данных).
4. Обучение классификатора на измененном множестве для обучения и последующего изменения точности классификатора на измененном множестве для тестирования или проверки. Если случайная особенность оказывает сильное влияние на точность, то эта особенность очень важна. В ином случае случайная особенность не так важна и является кандидатом для удаления.
5. Восстановление исходного набора для тестов или проверки и переход к следующей особенности, пока они не закончились. Результатом является сортировка особенностей в порядке их важности.

Описанный алгоритм базируется на random trees и decision trees. Таким образом, random trees и decision trees можно использовать для определения переменных, которые действительно необходимо использовать в качестве особенностей; в последующем можно использовать облегченный вектор особенностей для обучения того же (или другого) классификатора.

Диагностика проблем машинного обучения

Хорошо работающее машинное обучение — это скорее искусство, чем наука. Алгоритмы зачастую "вроде" работают, но не так хорошо, как требуется. Именно в этот момент и проявляется искусство — происходит выяснение и исправление того, что происходит не так. Данного раздела не хватит для разъяснения всех подробностей, однако, далее будет дан краткий обзор некоторых из наиболее распространенных проблем (профессор Andrew Ng из Stanford University предоставляет более подробную информацию в веб-лекциях, озаглавленные как "Советы по применению машинного обучения" (<http://www.stanford.edu/class/cs229/materials/ML-advice.pdf>)). Для начала некоторые эмпирические правила: большой объем данных разбивается на малые объемы данных, а лучшие особенности распределяются по лучшим алгоритмам. В случае введения собственных особенностей — при их максимальной независимости

друг от друга и минимальном изменении в различных условиях - почти любой алгоритм будет работать хорошо. Помимо этого, существуют две наиболее распространённые проблемы:

Необъективность

Предполагаемая модель слишком устойчивая, поэтому соответствующая модель не очень хорошая.

Изменчивость

Алгоритм запоминает данные, *включающие* шум, поэтому он не может быть обобщен

На рисунке 13-1 показана базовая настройка для статистического машинного обучения. Работа сводится к моделированию истинной функции f , которая преобразует основные входы в некий выход. Эта функция может иметь проблемы с регрессией (например, предсказание возраста человека по лицу) или проблему прогнозирования категории (например, идентификация личности по чертам лица). Проблемы, связанные с реальным миром, шум и непродуманные эффекты могут привести к тому, что получаемый выход может не соответствовать теоретическому. Например, при распознавании лица можно изучить модель за счет расстояния между глазами, рта и носа, идентифицирующие лицо. При этом изменения освещения, поступающие от соседней мерцающей лампы, приводят к шуму в измерениях, или плохо изготовленный объектив камеры приводят к систематическому искажению в измерениях, которые не будут рассмотрены как часть модели. Все это так же влияет и на точность.

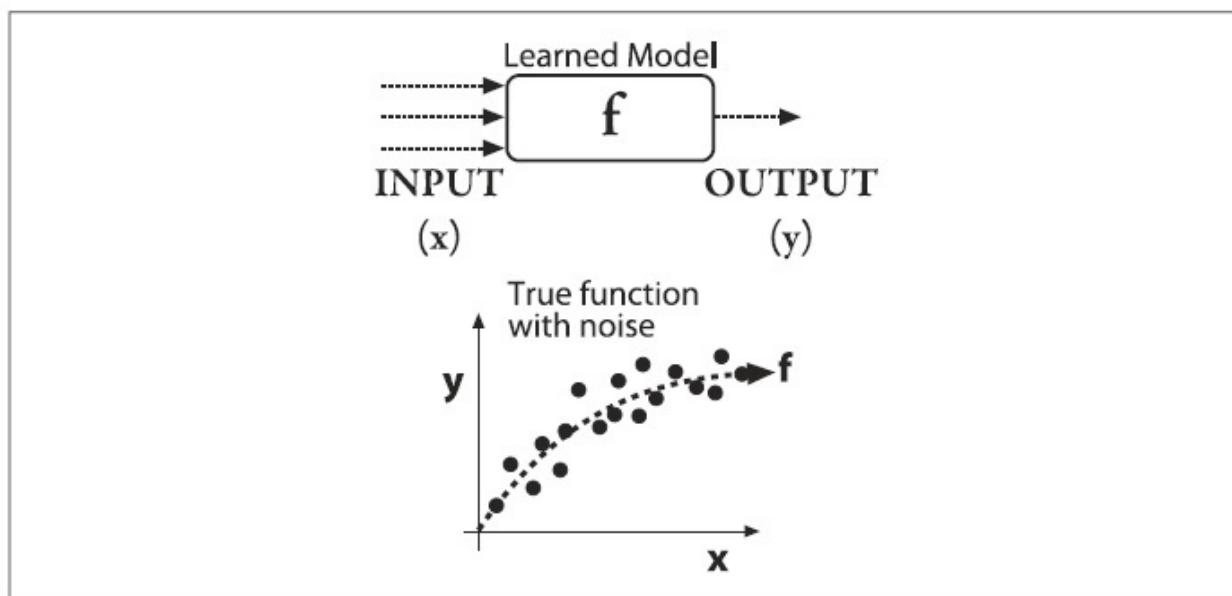


Рисунок 13-1. Настройки для статистического машинного обучения: классификатор обучается, чтобы соответствовать набору данных; истинная модель f почти всегда повреждена шумом или неизвестным воздействием

В двух верхних областях рисунка 13-2 показано недостаточное и чрезмерное обучение, а в двух нижних областях этого же рисунка показаны последствия с точки зрения ошибки в процессе подготовки заданного размера. В левой части рисунка 13-2 показана попытка обучения классификатора для прогнозирования данных из нижней области рисунка 13-1. Если использовать слишком ограниченную модель - показанная четко видимой прямой пунктирной линией - то будет невозможно соответствовать базовой истинной параболе f , которая показана неясно видимой пунктирной линией. Таким образом, данные предназначенные для обучения и тестирования будут плохими, даже при наличии большого объема данных. В этом случае возникает **необъективность**, т.к. оба набора данных (для обучения и для тестирования) прогнозируются плохо. Правая часть рисунка 13-2 соответствует точному набору для обучения, но это приводит к бессмысленной функции, которой соответствует немного шума. Таким образом, запоминание набора для обучения приводит и к запоминанию данных, содержащих шум. Как результат, набор данных для тестирования оставляет желать лучшего. Низкая погрешность при обучении в сочетании с большой погрешностью при тестировании указывает на проблему *изменчивости*.

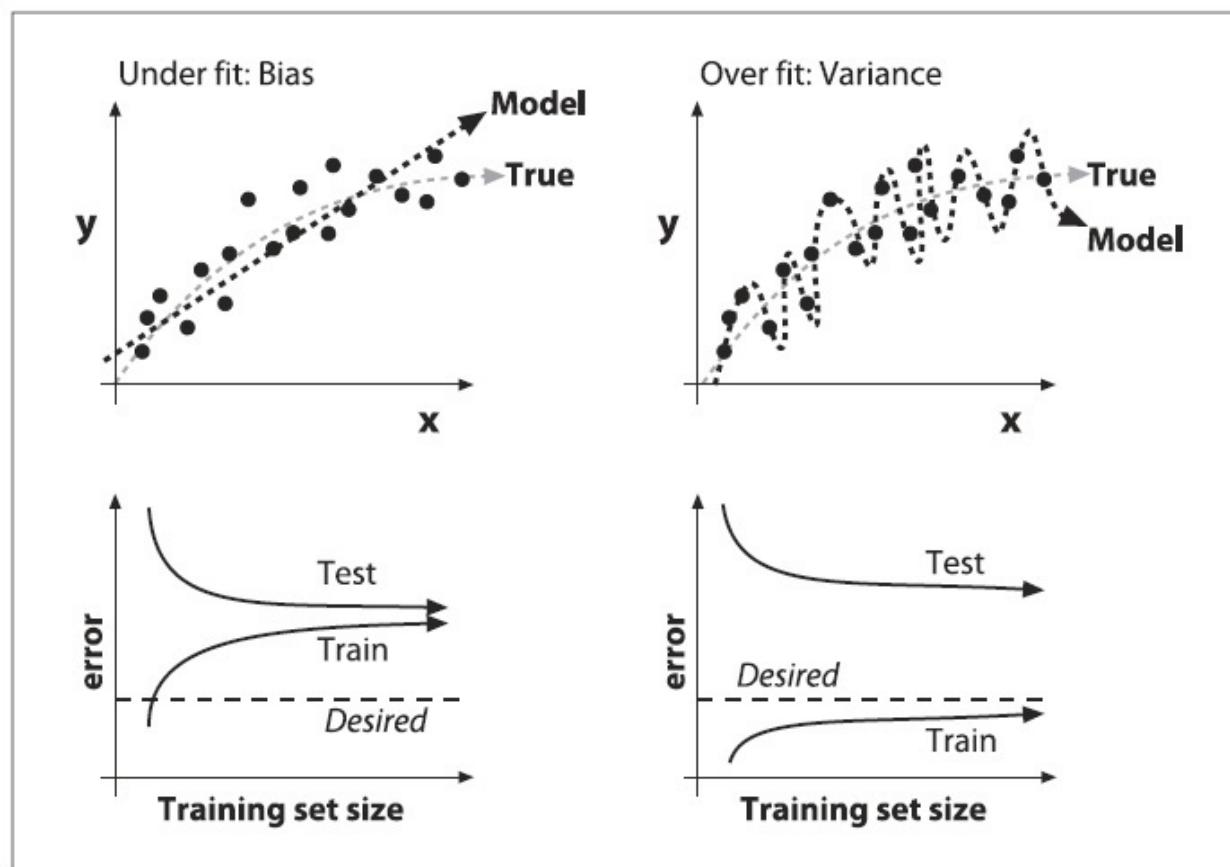


Рисунок 13-2. Плохая модель машинного обучения и её влияние на производительность процессов обучения и тестирования, где истинная функция графически показана неясно видимой пунктирной линией в верхней части рисунка: underfit модель (сверху слева) приводит к большой ошибке прогнозирования наборов

данных для обучения и тестирования (снизу слева), в то время как overfit модель (сверху справа) приводит к малой ошибке набора для обучения и большой ошибке набора для тестирования (снизу справа)

Иногда необходимо соблюдать осторожность для нахождения верного решения проблемы. Если наборы данных для обучения и тестирования имеют малую ошибку, а алгоритм работает плохо в реальных условиях, то соответствующие наборы данных выбраны согласно нереальным условиям - возможно из-за того, что согласно этим условиям процессы сбора или моделирования осуществить проще. Если же алгоритм просто не может воспроизвести наборы для обучения или тестирования, то, возможно, используется не тот алгоритм, или не те особенности, которые после извлечения являются неэффективными, или просто нет "связей" в собранных данных. В таблице 13-2 приведены возможные варианты решения описанных проблем. Разумеется, это не полный список возможных проблем и их решений. Для того, чтобы машинное обучение давало хорошие результаты, необходимо более тщательное проектирование и осмысление процесса сбора данных, а также более тщательный подбор функций для выполнения вычислений. Также необходимо систематическое мышление для диагностики проблем машинного обучения.

Таблица 13-2. Проблемы машинного обучения и возможный вариант их решения; достижение наилучших особенностей поможет решить любую проблему

Проблема	Возможное решение
Необъективность	Добавление особенностей для достижения лучшего соответствия. Использование более мощного алгоритма.
Изменчивость	Использование большего объема данных для обучения поможет сгладить модель. Использование меньшего количества особенностей для уменьшения переобучения. Использование менее мощного алгоритма
Хорошие результаты при тестировании/ обучении, плохие в реальных условиях	Сбор более реальных данных
Модель не распознает наборы для тестов или обучения	Перепроектирование особенностей для лучшего отслеживания изменчивости данных. Сбор новых, более реальных данных. Использование более мощного алгоритма

Перекрестная проверка, самонастройка, ROC-кривая и таблица сопряженности

В заключении будет рассмотрен инструментарий для оценки результатов машинного обучения. В случае контролируемого обучения, одной из основных проблем является знание того, насколько хорошо алгоритм работает: насколько точна классификация

или пригодны ли данные? Возможно, может возникнуть мысль: "Спокойно, можно ведь просто испытать данный алгоритм на наборах для тестов или проверки и получить результат." Однако в реальных условиях необходимо так же учитывать шум, колебания и ошибки выборки. Проще говоря, наборы для тестов или проверки могут не точно отражать фактическое распределение данных. Для того, чтобы приблизиться к истинной производительности классификатора, необходимо использовать технику *перекрестной проверки* и/или тесно связанную с ней технику *самонастройки*.

Основу перекрестной проверки составляет разделение данных на K различных подмножеств. В обучение используется K-1 подмножеств, а оставшееся подмножество используется для тестирования. Данное действие выполняется K раз, где каждое из K подмножеств совершают "виток" становясь подмножеством для проверки, а результат в конечном счете является средней величиной.

Техника самонастройки схожа с техникой перекрестной проверки, за исключением того факта, что набор для проверки выбирается случайным образом из набора для обучения. Выбранные точки для итерации используются для тестового набора, а не для набора для обучения. Затем процесс начинается с нуля. Описанное действие необходимо выполнить N раз, где каждый раз происходит выбор, случайным образом, набора для проверки, а результат в конечном счете является средней величиной. Это означает, что некоторые и/или многие точки повторяются в различных наборах для проверки, но результат зачастую получается лучше в сравнении с результатом от использования техники перекрестной проверки.

Использование одной из данных техник позволит улучшить результаты машинного обучения. При этом увеличение точности позволит задействовать параметры для настройки системы обучения, которые в последствии можно будет неоднократно менять, обучать и измерять.

Двумя другими очень полезными способами оценки, характеристики и настройки классификатора является построение ROC-кривой - *рабочей характеристики приёмника* - и заполнение таблицы сопряженности (рисунок 13-3). ROC-кривая соответствует соотношению измерений реакции и параметра производительности классификатора на протяжении всего диапазона заданного параметра. Можно сказать, что параметр — это порог. Для понимания данного факта, представьте, что необходимо распознать желтый цвет на изображении при наличии порога, указывающий на желтый цвет, выступающий в роли детектора. Чрезвычайно высокий показатель порога будет означать, что классификатору не удалось распознать желтый цвет, т.е. имеем ложно положительный нулевой уровень, при этом и истинно положительный уровень так же нулевой (нижняя левая часть кривой на рисунке 13-3). С другой стороны, если желтый порог равен 0, то любой сигнал на протяжении всей кривой будет распознан. Это означает, что все истинно положительные (желтые цветы)

распознаются так же, как и все должно положительные тоже (оранжевые или красные цветы); таким образом, имеется 100% показатель ложных срабатываний (верхняя правая часть кривой на рисунке 13-3). Лучшей ROC-кривой является та, что соответствует у-оси на 100%, с последующим измельчанием в правом верхнем углу. В противном случае, чем ближе кривая к верхнему левому углу, тем лучше. Так же можно вычислить долю площади под ROC-кривой относительно общей площади ROC участка как сводную статистику заслуг: чем ближе данное соотношение к 1, тем лучше классификатор.

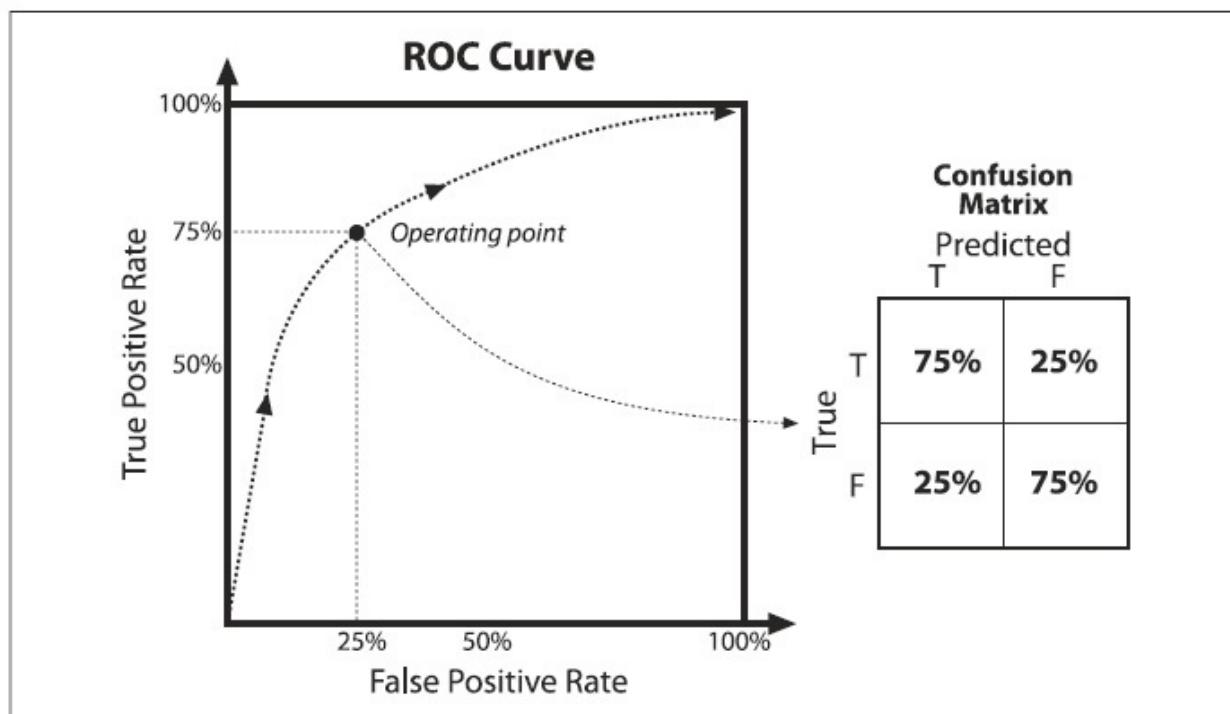


Рисунок 13-3. Рабочая характеристика приёмника (ROC) и связанная с ней таблица сопряженности: в начале показан отклик корректной классификации должно положительной оценки вдоль всего диапазона изменений параметра производительности классификатора; затем показана ложно положительная оценка (ложные распознавания) и ложно негативная оценка (пропущенные распознавания)

На рисунке 13-3 также показана *таблица сопряженности*. Это просто схема положительных и ложных срабатываний с ложно положительной и ложно негативной оценками. А так же это еще один быстрый способ оценки производительности классификатора: в идеале значения диагонали NW-SE равны 100% и 0% в иных случаях. Если имеется классификатор, который может изучить более одного типа (например, многослойный персепtron или random forest классификатор, которые могут изучить множество различных типов меток за раз), то таблица сопряженности обобщает множество типов и остается просто отслеживать тип соответствующей метки.

Оценка ошибочной классификации. Собственно, это единственное, что ещё не обсуждалось. Что это значит: например, имеется классификатор для обнаружения ядовитых грибов (в дальнейшем будет показан пример, использующий такой набор данных), имеющий большую ложную негативную оценку (съедобные грибы определяются как ядовитые) до тех пор, пока не минимизируется ложная положительная оценка (ядовитые грибы определяются как съедобные). ROC-кривая может помочь справиться с этим; можно установить ROC-параметр, выбрав рабочую точку ниже кривой - в направлении левой нижней части графика на рисунке 13-3. Другой способ сделать тоже самое заключается в том, чтобы вес ложной положительной ошибки стал больше, чем вес ложной отрицательной при генерации ROC-кривой. Например, можно задать каждую ложную положительную ошибку, рассчитав десять ложных негативных (Это особенно полезно при наличии конкретного априори понятия относительно оценки двух типов ошибок. Например, оценка ошибочной классификации одного продукта как другого продукта поможет определить количественную оценку такого продукта заранее). Некоторые алгоритмы машинного обучения в OpenCV, такие как decision trees и SVM, могут регулировать баланс "успешная оценка относительно ложного предупреждения", указав приоритет вероятностей самих типов (каких типов больше, а каких меньше) или указав веса отдельных обучающих выборок.

Дисперсия несочетающихся особенностей. Ещё одна распространенная ошибка при обучении некоторого классификатора возникает, когда вектор особенностей включает особенности с широко различными дисперсиями. Например, если одна особенность представлена символом в нижнем регистре ASCII, то диапазон всех возможных значений равен 26. В противопоставление, если особенность представляет число биологических клеток на предметном стекле микроскопа, диапазон возможных значений может измеряться миллиардами. Такой алгоритм, как K-nearest neighbors может рассмотреть первую особенность (с символом) как относительную постоянную (ничего не изучив) по сравнению со второй особенностью (с клетками). Вариант исправления данной проблемы заключается в предварительной обработке переменной нормализации дисперсии каждой особенности. Данная практика является приемлемой при условии, что особенности не коррелируют друг с другом; когда особенности взаимосвязаны, то можно нормализовать их среднюю дисперсию или ковариацию. Некоторые алгоритмы, такие как decision trees (Decision trees не оказывают влияния на различия дисперсии в переменных особенности, т.к. каждая переменная ищется только для эффективно разделяющих порогов. Другими словами, не имеет значения насколько большой диапазон переменной для выяснения значения разделения), не страдают от широких различий дисперсии. Правило гласит, что если алгоритм зависит в какой-то степени от измерения расстояния (например, взвешенные значения), то необходимо нормализовать дисперсию. Можно нормализовать все особенности за раз и рассчитать для них ковариацию при помощи расстояния

Mahalanobis, о котором далее в главе будет рассказано более подробно (читатель, знакомый с машинным обучением или обработкой сигнала может узнать в этой технике "отбеливание данных").

Далее в этой главе будет рассказано более подробно о некоторых представленных алгоритмах машинного обучения, реализованных в OpenCV (большинство из них можно найти в `.../opencv/ml/`). В начале будет рассказано о некоторых методах, которые являются универсальными для подбиблиотеки ML.

[П]|[РС]|(РП) Универсальные методы библиотеки ML

Эта глава содержит описание того, как работают алгоритмы машинного обучения.

Будут рассмотрены удобные универсальные методы, описание которых можно найти в документации (.../opencv/docs/ref/opencvref_ml.htm), устанавливаемой вместе с OpenCV и/или в онлайн документации OpenCV Wiki

(<http://opencvlibrary.sourceforge.net>). Данная под библиотека на момент написания книги находилась в активной разработке.

Все методы библиотеки ML (алгоритмы Haar classifier, Mahalanobis и K-means были реализованы до появления библиотеки ML и потому находятся в библиотеках *cv* и *cvcvcore*) написаны как классы C++, наследующиеся от класса *CvStatModel*, содержащий универсальные методы для всех алгоритмов. Эти методы перечислены в таблице 13-3. Стоит обратить внимание на то, что в *CvStatModel* реализованы две модели сохранения и загрузки данных: *save()* и *write()* для сохранения; *load()* и *read()* для загрузки. Для моделей машинного обучения необходимо использовать более простые *save()* и *load()*, которые по существу являются обертками более сложных функций *write()* и *read()* интерфейса, записывающего и читающего XML и YAML на и с диска. Реализация двух других наиболее важных функций, *predict()* и *train()*, зависит от алгоритма и будет рассматриваться чуть позже.

Таблица 13-3. Методы базового класса библиотеки ML

CvStatModel::Methods	Описание
save(const char <i>filename</i> , const char <i>name</i> = 0)	Сохранение обученной модели в XML или YMAL. Метод используется для сохранения
load(const char <i>filename</i> , const char <i>name</i> = 0)	Вызов <i>clear()</i> с последующей загрузкой XML или YMAL модели. Метод используется для загрузки
clear()	Освобождение всей занимаемой памяти. Возможно повторное использование
bool train(—data points—, [flags] —responses—, [flags etc])	Функция обучения модели данных. Обучение специфично для каждого алгоритма и потому входные параметры различны
float predict(const CvMat* sample [,]) const	Функция используется после обучения для предсказания метки или значения новой обучаемой точки или точек
Конструктор, Деструктор	
CvStatModel(); CvStatModel(const CvMat* <i>train_data</i> ...);	Конструктор по умолчанию и конструктор, позволяющий создавать и обучать модель в одном кадре
CvStatModel::~CvStatModel();	Деструктор
Поддерживаемые Write/Read (использовать save/load)	
write(CvFileStorage <i>storage</i> , const char <i>name</i>)	CvFileStorage - общая структура для записи на диск (обсуждалась в главе 3), расположенная в библиотеке <i>cvcore</i> . Вызывается из <i>save()</i>
read(CvFileStorage <i>storage</i> , CvFileNode <i>node</i>)	CvFileStorage - общая структура для чтения с диска (обсуждалась в главе 3), расположенная в библиотеке <i>cvcore</i> . Вызывается из <i>load()</i>

Обучение

Прототип метода обучения:

```
bool CvStatModel::train(
    const CvMat* train_data,
    [int tflag,]           ...,
    const CvMat* responses, ...,
    [const CvMat* var_idx,] ...,
    [const CvMat* sample_idx,] ...,
    [const CvMat* var_type,] ...,
    [const CvMat* missing_mask,]
    <misc_training_alg_params> ...
);
```

Метод *train()* может принимать различные формы в зависимости от используемого алгоритма. Все алгоритмы принимают в качестве входных данных (для обучения) указатель на матрицу типа CvMat. Эта матрица должна быть типа 32FC1 (32-битной, вещественной, одноканальной). CvMat можно использовать и для многоканальных изображений, но алгоритмы машинного обучения могут работать только с одним каналом - т.е. только с двумерной матрицей чисел. Как правило, эта матрица организована в виде строк точек, где каждая "точка" представлена как вектор особенностей. Следовательно, столбцы содержат конкретные особенности для каждой точки, а все вместе точки образуют двумерную одноканальную матрицу для обучения. Таким образом типичная матрица данных состоит из (строка, столбец) = (точки, особенности). Вместе с тем некоторые алгоритмы могут обрабатывать транспонированную (обсуждалось ранее) матрицу. Для таких алгоритмов можно использовать параметр *tflag*, чтобы сообщить алгоритму о том, что точки находятся в столбцах, а не в строках. Это удобно, т.к. не нужно самостоятельно выполнять транспонирование. Когда алгоритм может обрабатывать данные, представленные обоими способами, необходимо использовать следующие флаги:

tflag = CV_ROW_SAMPLE

Это значит, что вектор особенностей хранится в строке (по умолчанию)

tflag = CV_COL_SAMPLE

Это значит, что вектор особенностей храниться в столбце

Может возникнуть вопрос: а что, если обучающий набор данных представлен не вещественными числами, а буквами алфавита или целыми числами, представляющие музыкальные ноты или названия растений? Ответ следующий: просто используйте 32-битные вещественные числа при заполнении CvMat. В случае с буквами, особенности или метки ASCII символов можно расценивать как вещественные числа при заполнении массива данных. Тоже самое относится и к целым числам. До тех пор, пока переход уникален, всё работает - однако, не стоит забывать, что некоторые методы чувствительны к широким различиям дисперсии между особенностями. В большинстве случаев, лучше нормализовать дисперсию особенностей, как было описано ранее. Лишь за исключением алгоритмов, основанных на деревьях (decision trees, random trees и boosting), которые поддерживают и недвусмысленные и упорядоченные входные переменные; все остальные алгоритмы OpenCV ML работают только с упорядоченными входами. Популярный метод создания упорядоченного входа для алгоритма также работает и с недвусмысленными данными, чтобы представить их в единичной системе счисления; например, если входная переменная цвета может иметь семь различных значений, то она может быть заменена на семь двоичных переменных, где одна и только одна из переменных может быть установлена в 1.

Параметр ответа может быть либо категорической меткой, такой как "ядовитый" или "неядовитый" в случае определения гриба, либо регрессионным значением (числом), таким как температура тела, измеренная при помощи термометра. Значение ответа или "метки", как правило, это одномерный вектор с одним значением на точку - за исключением нейронных сетей, которые могут иметь вектор ответов для каждой точки. Значения ответа могут иметь один из двух типов: для категорических ответов тип может быть целым (`32SC1`); для регрессионных значений ответ может быть 32-битным вещественным (`32FC1`). При этом некоторые алгоритмы могут иметь проблемы только с классификационными проблемами, а некоторые только с регрессионными; а некоторые могут справиться и с первой, и со второй. В последнем случае, тип выходной переменной передается либо в виде отдельной переменной, либо как последний элемент вектора `var_type`, который может быть установлен следующим образом:

`CV_VAR_CATEGORICAL`

Это означает, что выходные значения являются раздельными классами меток

`CV_VAR_ORDERED (=CV_VAR_NUMERICAL)`

Это означает, что выходные значения упорядочены; т.е. различные значения можно сравнивать как числа, т.к. это регрессионная проблема

Типы входных переменных также можно задать при помощи `var_type`. Однако, алгоритмы регрессионного типа могут обрабатывать только упорядоченные входные переменные. Иногда возможно упорядочивание категорических переменных до тех пор, пока порядок сохраняется последовательным, но иногда это может вызвать регрессионные затруднения, т.к. создается только видимость "упорядочивания" значений, которые будут иметь большой разброс, не имея при этом никакой физической основы для их упорядочивания.

Многие модели библиотеки ML могут быть обучены на выбранном подмножестве особенностей и/или выбранном простом подмножестве обучающего множества. Для упрощения данной ситуации для конечного пользователя, метод `train()`, как правило, включает в себя вектора `var_idx` и `sample_idx` в качестве параметров. По умолчанию можно использовать "все данные", установив значения этих параметров в `NULL`, но через `var_idx` можно передать только те переменные (особенности), которые вызывают интерес, так же как и через `sample_idx` можно передать только те точки, которые вызывают интерес. При помощи этих параметров собственно можно указать какие особенности и какие образцы точек необходимо использовать в процессе обучения. Оба вектора либо одноканальные целые (`CV_32SC1`) вектора - т.е. списки, индексация которых начинается с 0 - либо одноканальные 8-битные (`CV_8UC1`) маски активных переменных/образцов, где ненулевые значения означают активы. Параметр

sample_idx особенно полезен в случае чтения данных порциями, при этом какая-то часть используется для обучения, а какая-то для тестирования, не нарушая при этом предназначения этих векторов.

Более того, некоторые алгоритмы могут обрабатывать недостающие измерения. Например, когда авторы работают с показателями хода производственного процесса, некоторые измерения особенностей могут пропасть навсегда во время перерыва на кофе. Иногда экспериментальные данные могут быть просто забыты, например, температура пациента за один день медицинского эксперимента. В этом случае, параметр *missing_mask* является 8-битной матрицей того же размера, что и *train_data*, и используемая для обозначения пропущенных значений (соответствующие ненулевым значениям маски). Некоторые алгоритмы не могут обработать недостающие значения, поэтому недостающие точки должны быть либо интерполированы пользователем перед обучением, либо исключены заранее. Другие алгоритмы, такие как decision tree и naïve Bayes, обрабатывают недостающие значения по-разному. В алгоритме decision tree используются альтернативные разделители (так называемые "суррогатные разделители" Breiman); алгоритм naïve Bayes выводит значения.

Как правило, ранее описанная модель очищается при помощи *clear()* перед запуском процесса обучения. Вместе с тем, некоторые алгоритмы могут (необязательно) обновлять обучаемые модели при поступлении нового набора для обучения, вместо запуска процесса с нуля.

Прогнозирование

Метод *predict()* формирует параметр *var_idx*, который определяет какие особенности использовать в методе *train()*, для последующего извлечения только необходимых компонент из исходной выборки. Общий вид метода *predict()* следующий:

```
float CvStatMode::predict(
    const CvMat* sample
    [, <prediction_params>]
) const;
```

Этот метод используется для прогнозирования ответа при поступлении нового вектора данных. При использовании классификатора, *predict()* возвращает метку класса. В случае с регрессией, этот метод возвращает числовое значение. При этом исходная выборка должна иметь столько же компонентов, как и параметр *train_data*, используемый в обучении. Дополнительный параметр *prediction_params* специфичен для конкретного алгоритма и позволяет обрабатывать недостающие особенности методов, базирующихся на основе деревьев. Суффикс *const* на конце функции говорит

о том, что прогнозирование не влияет на внутреннее состояние модели, поэтому данный метод является потокобезопасным и может работать параллельно, что полезно в случае с веб-сервером, выполняющего поиск изображений для нескольких пользователей и роботов, необходимых для ускорения процесса сканирования сцены.

Контролирование итераций обучения

Контролируемая итерации структура *CvTermCriteria*, которая используется некоторыми методами машинного обучения, уже обсуждалась ранее в предыдущих главах. В качестве напоминания данная структура приведена чуть ниже:

```
typedef struct CvTermCriteria {
    int      type;      /* CV_TERMCRIT_ITER и/или CV_TERMCRIT_EPS */
    int      max_iter;   /* максимальное число итераций */
    double  epsilon;    /* значения для остановки процесса */
}
```

Целочисленное значение *max_iter* задает общее число итераций, которое алгоритм должен выполнить. Параметр *epsilon* задает пороговое значение, по достижении которого процесс прекращается. И наконец, параметр *type* сообщает о том, какой из этих двух критериев использовать, при этом возможно одновременное использование обоих критериев (*CV_TERMCRIT_ITER | CV_TERMCRIT_EPS*). Определения значений для *term_crit.type* следующие:

```
#define CV_TERMCRIT_ITER     1
#define CV_TERMCRIT_NUMBER   CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS       2
```

Теперь можно перейти к описанию алгоритмов, реализованных в OpenCV. Для начала будет рассмотрен наиболее часто используемый алгоритм *Mahalanobis distance*, а затем неконтролируемый алгоритм *K-means*; оба алгоритма можно найти в библиотеке *cxcore*. Потом будет рассмотрена библиотека *ML*, начиная с *normal Bayes classifier* и заканчивая алгоритмами, основанные на *decision-tree* (*decision trees, boosting, random trees* и *Haar cascade*). Для всех других алгоритмов будут представлены краткое описание и примеры использования.

[П] | [PC] | (РП) Расстояние Mahalanobis

Расстояние Mahalanobis является мерой расстояния, которая рассчитывается для ковариации или для "растягивания" пространства, в котором лежат данные. Если известна Z-оценка, то можно представить расстояние Mahalanobis как многомерный аналог Z-оценки. На рисунке 13-4(a) показано начальное распределение между тремя наборами данных, при этом по вертикале наборы располагаются ближе друг к другу. После нормализации пространства данных ковариации, как показано на рисунке 13-4(b), оказывается, что на самом деле ближе друг к другу горизонтальные наборы. Такого рода вещи встречаются часто; например, если сравнить рост людей в метрах с их возрастом в днях, то можно заметить очень малое отклонение в росте относительно большой дисперсии в возрасте. Нормализовав дисперсию, можно получить более реалистичное сравнение переменных. Некоторые классификаторы, такие как K-nearest neighbors плохо обрабатывают различия в дисперсии, в то время как другие алгоритмы (такие как decision trees) не обращают на это внимания.

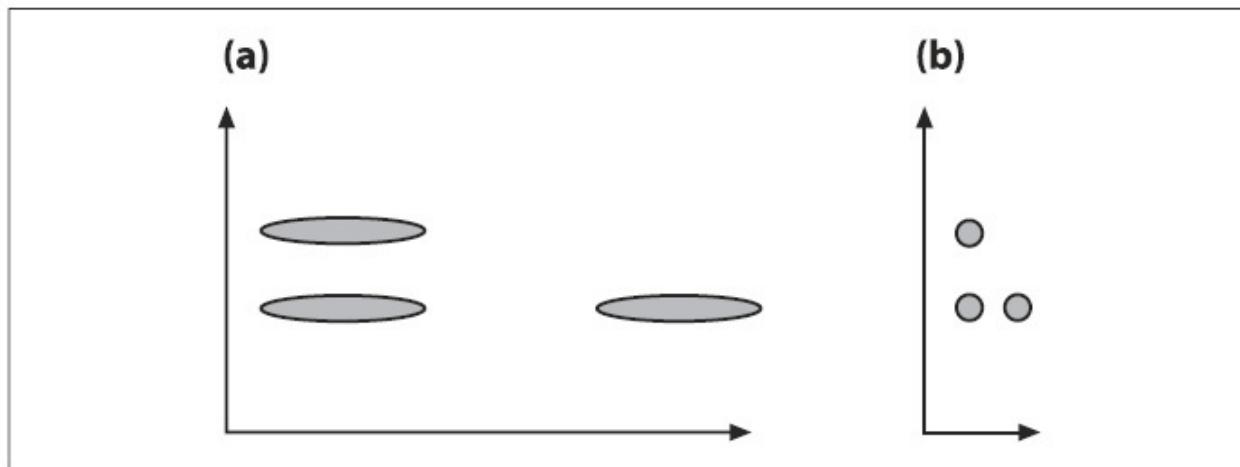


Рисунок 13-4. Вычисленное расстояние Mahalanobis позволяет по-новому интерпретировать ковариационные данные - как "растяжение" пространства: (а) расстояние по вертикале между строками меньше горизонтального расстояния; (б) после нормализации расстояние по горизонтали меньше расстояния по вертикале

Собственно, на рисунке 13-4 уже дана подсказка относительно того, как получить расстояние Mahalanobis (На рисунке 13-4 представлены диагональные матрицы ковариации, что означает независимые x и y дисперсии. Это сделано с целью предоставления наиболее простого объяснения. В действительности, данные зачастую "растягиваются" значительно более интенсивными способами.); необходимо каким-то образом делить ковариацию пока измеряется расстояние. Для начала необходимо понять, что такая ковариация. Имея список X из N точек, где каждая точка

предоставления наиболее простого объяснения. В действительности, данные зачастую "растягиваются" значительно более интенсивными способами.); необходимо каким-то образом делить ковариацию пока измеряется расстояние. Для начала необходимо понять, что такое ковариация. Имея список X из N точек, где каждая точка может иметь размерность (длину вектора) K , и средний вектор μ_x (состоящий из отдельных средних $\mu_{1,\dots,K}$), получаем, что ковариацией является матрица $K \times K$, определяемая по следующей формуле:

$$\sum = E[(X - \mu_x)(X - \mu_x)^T]$$

где $E[\cdot]$ является оператором математического ожидания. При помощи OpenCV легко вычислить матрицу ковариации за счет следующей функции:

```
void cvCalcCovarMatrix(
    const CvArr** vvects
    , int count
    , CvArr* cov_mat
    , CvArr* avg
    , int flags
);
```

Эта функция в определенной степени мудреная. Параметр *vvects* является указателем на указатель *CvMat*. Это означает, что имеется *vvects[0]* до *vvects[count-1]*, но на самом деле данный факт зависит от значения *flags*, описание которого будет представлено чуть позже. В принципе, имеется два случая:

1. *vvects* является одномерным вектором указателей на одномерную или двумерную матрицы (два измерения для изображений). Т.е. каждый *vvects[i]* может указывать на одномерный или двумерный вектор, если не установлены и не *CV_COV_ROWS* и не *CV_COV_COLS*. Вычисленное накопление ковариации масштабируется или разделяется на число точек в соответствии с *count*, если установлено *CV_COVAR_SCALE*.
2. Зачастую имеется только один входной вектор, поэтому используется только *vvects[0]*, если задано *CV_COVAR_ROWS* или *CV_COVAR_COLS*. Если этот параметр установлен, то величина масштабирования не зависит от *count*, а только от числа векторов, содержащихся в *vvects[0]*. Тогда все точки располагаются в:
 - a. строках *vvects[0]*, если задано *CV_COVAR_ROWS*
 - b. столбцах *vvects[0]*, если задано *CV_COVAR_COLS*. Невозможно одновременное использование двух флагов (более подробная информация представлена в описании флага).

векторов в `vects[]`; в случае 2a и 2b (задано `CV_COVAR_ROWS` или `CV_COVAR_COLS`), параметр `count` не используется, а используется действительное число векторов в `vects[0]`. Результатом является ковариационная матрица KxK, возвращаемая в `cov_mat` и имеющая тип `CV_32FC1` или `CV_64FC1`. Не зависимо от того будет ли использован вектор `avg`, он зависит от настройки `flags`. Если параметр `avg` используется, то он имеет тот же тип, что и `vects` и содержит средние значения K-особенностей `vects`. Параметр `flags` может иметь множество различных комбинаций настройки, образующихся за счет добавления значений друг к другу (для более сложных ситуаций, обратитесь к документации

`.../opencv/docs/ref/opencvref_cxcore.htm`). В общем, `flags` может иметь одно из следующих значений:

`CV_COVAR_NORMAL`

Обычно ковариация вычисляется согласно ранее показанному уравнению. Среднее значение результата зависит от параметра `count`, если не задано `CV_COVAR_SCALE`; в противном случае, результат зависит от среднего числа точек в `vects[0]`.

`CV_COVAR_SCALE`

Нормализация вычисленной ковариационной матрицы.

`CV_COVAR_USE_AVG`

Используется матрица `avg` вместо автоматически вычисляемого среднего значения каждой особенности. Установка этого значения экономит время вычислений при наличии средних значений (например, после использования `cvAvg()`); в противном случае, функции придётся вычислять эти средние значения (должно быть передано предварительно вычисленное среднее значение, если пользователь имеет более статистически обоснованное среднее значение или если матрица ковариации вычисляется по частям).

Чаще всего данные объединяются построчно в одну большую матрицу; тогда `flags` должен иметь следующее значение: `CV_COVAR_NORMAL | CV_COVAR_SCALE | CV_COVAR_ROWS`.

Не достаточно просто иметь ковариационную матрицу для нахождения расстояния Mahalanobis, потому что необходимо ещё разделить пространство дисперсии и найти обратную матрицу ковариации. Последняя операция легко выполнима за счет использования следующий функции:

```
double cvInvert(
    const CvArr* src
    ,CvArr* dst
    ,int method = CV_LU
);
```

Матрица *src* должна быть матрицей ковариации, вычисленная до передачи в функцию, а матрица (предназначенная для размещения обратной матрицы) *dst* должна быть того же размера, что и *src*. Можно использовать значение *method* по умолчанию - *CV_LU* -, но лучше использовать *method = CV_SVD_SYM* (Так же может быть использован метод *CV_SVD*, но он медленнее *CV_SVD_SYM*. Должен быть использован метод *CV_SVD_SYM* даже если он медленнее *CV_LU*, при условии, что размерность пространства намного меньше числа точек. В таком случае общее время вычислений согласно *cvCalcCovarMatrix()* должно преобладать в любом случае. Так что возможно стоит потратить немного времени на вычисление более точной (при условии, что множество точек сосредоточено в подпространстве меньшей размерности) обратной матрицы ковариации. В общем, метод *CV_SVD_SYM* является лучшим выбором для выполнения данной задачи).

Теперь, после определения обратной матрицы ковариации Σ^{-1} , можно перейти к определению расстояния Mahalanobis. Эта мера определяется почти также, как и Евклидовая метрика - квадратный корень из суммы квадратов разности между двумя векторами *x* и *y* - плюс деление на матрицу ковариации:

$$D_{\text{Mahalanobis}}(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^T \Sigma^{-1} (\mathbf{x} - \mathbf{y})}$$

Данное расстояние — это просто число. Если матрица ковариации является единичной матрицей, то расстояние Mahalanobis равно Евклидовой метрике. Теперь осталось рассмотреть непосредственно саму функцию, которая вычисляет расстояние Mahalanobis. Данная функция принимает два вектора - *vec1* и *vec2* - и обратную ковариационную матрицу *mat*, а возвращает расстояние как число типа *double*:

```
double cvMahalanobis(
    const CvArr* vec1
    ,const CvArr* vec2
    ,CvArr* mat
);
```

Расстояние Mahalanobis является важной мерой сходства между двумя различными точками в многомерном пространстве, а не алгоритмом кластеризации или классификатором. Далее будет рассмотрен наиболее часто используемый алгоритм кластеризации: **K-means**.

[П]|[РС]|(РП) K-Means

Алгоритм кластеризации *K-Means* был реализован в библиотеке *cvScore* задолго до появления библиотеки ML. K-means пытается найти естественные кластеры или "сгустки" данных. Пользователь задает желаемое число кластеров, а K-Means затем быстро находит хорошие расположения центров для этих кластеров, где термин "хороший" означает, что центр кластера располагается в середине сгустка данных. Данный алгоритм является наиболее часто используемым методом кластеризации и сильно схож с EM-алгоритмом для смеси Гаусса (реализован в библиотеке ML как *CvEM()*), а так же в какой то степени с алгоритмом mean-shift, обсуждаемый в главе 9 (реализован в библиотеке cv как *cvMeanShift()*). Алгоритм K-means является итерационным и, согласно реализации в OpenCV, так же известен как алгоритм Lloyd (S. P. Lloyd, "Least Squares Quantization in PCM", IEEE Transactions on Information Theory 28 (1982), 129–137) или (что эквивалентно) "Итерации Voronoi". Алгоритм работает следующим образом:

1. Берётся (a) исходный набор и (b) требуемое число кластеров K (выбранное пользователем).
2. Случайным образом выбираются расположения центров кластера.
3. Каждая из точек связывается с ближайшим центром кластера.
4. Перемещаются центры кластера в центры тяжести точек.
5. Происходит возвращение к пункту 3 до полной сходимости (без перемещения центра тяжести).

На рисунке 13-5 показаны диаграммы работы K-Means; для данного случая, сходимость наступит после двух итераций. В реальных случаях алгоритм зачастую быстро сходиться, при этом в некоторых случаях для этого потребуется большое число итераций.

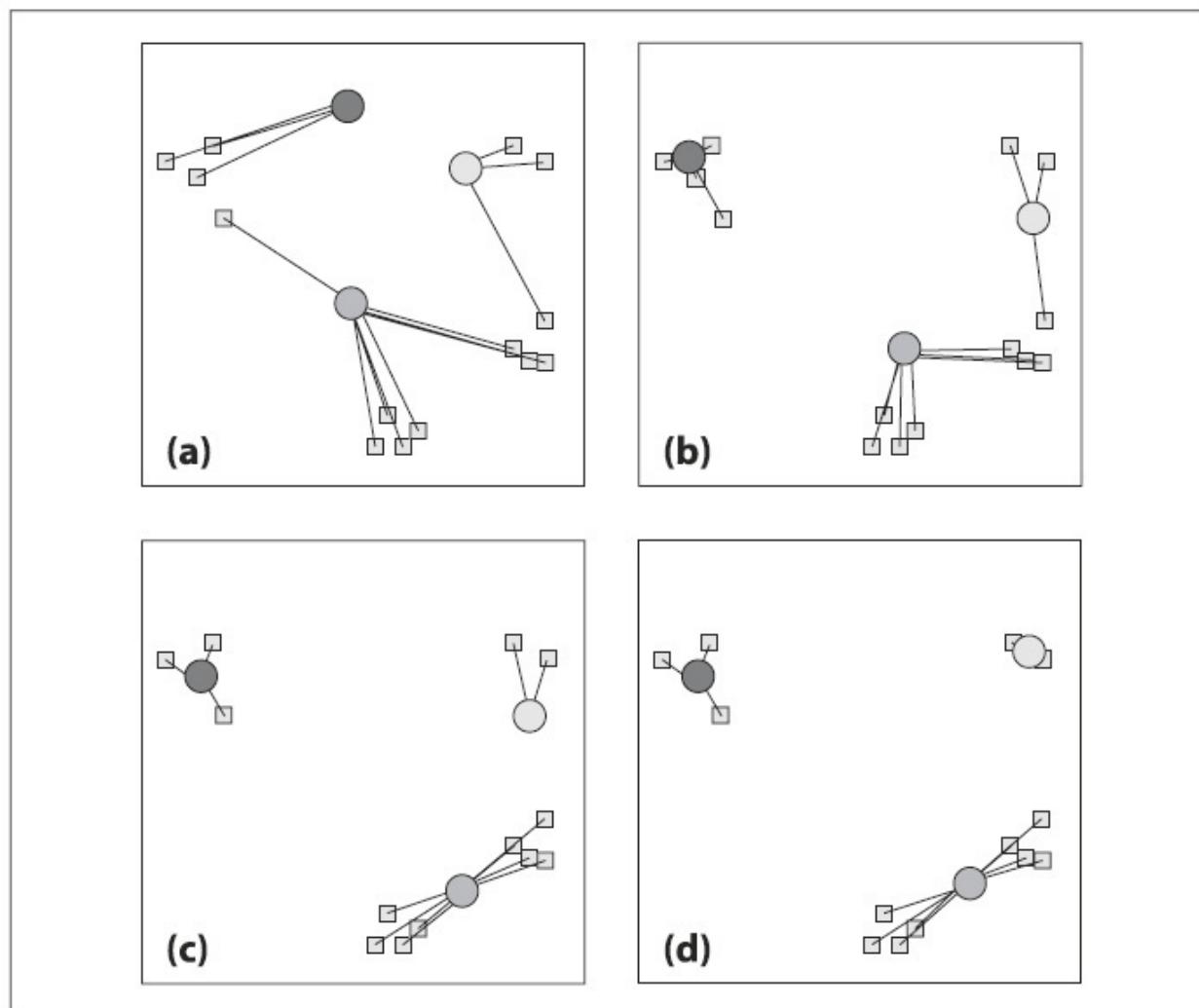


Рисунок 13-5. Работа K-means для двух итераций: (а) центры кластера расположены случайным образом и каждой точке присваивается ближайший центр кластера; (б) центры кластера перемещаются в центр тяжести точек; (с) точкам вновь назначаются ближайшие центры кластера; (д) центры кластера вновь перемещаются в центры тяжести этих точек

Проблемы и их решения

K-means является чрезвычайно эффективным алгоритмом, но у него есть три проблемы:

1. K-means не гарантирует, что будет найдено лучшее решение для размещения центров кластера. Тем не менее гарантируется сходимость (т.е. число итераций ограничено).
2. K-means не сообщает о том, сколько центров кластера необходимо использовать. Если выбрано 2 или 4 кластера, как показано на примере на рисунке 13-5, то результаты будут различны и, возможно, не интуитивно понятными.

3. K-means предполагает, что, либо нет смысла в ковариации пространства, либо пространство уже нормализовано (более подробную информацию можно найти в разделе "Расстояние Mahalanobis").

Для каждой проблемы имеется "решение" или, по крайней мере, приблизительное решение. Решения первых двух проблем зависят от "объяснения дисперсии". В K-means каждый центр кластера "владеет" точками и вычисляемой дисперсией этих точек. Лучший кластер минимизирует дисперсию не вызывая больших сложностей. Имея это в виду, перечисленные проблемы могут быть "смягчены" следующим образом:

1. За счёт применения K-means несколько раз и каждый раз с различным размещением центров кластера (это легко выполнимо, т.к. OpenCV располагает центры случайным образом) с последующим выбором того применения, чьи результаты дали лучшую дисперсию.
2. Начиная с одного кластера с последующим перебором определенного числа кластеров, использовать на всех кластерах метод #1. Как правило, общая дисперсия будет довольно таки быстро сокращаться, после чего появится "локоть" на кривой дисперсии; это указывает на то, что новый центр кластера не существенно уменьшит общую дисперсию.
3. За счёт умножения на обратную матрицу ковариации (как было описано в разделе "Расстояние Mahalanobis"). Например, имея исходные векторы данных D, организованные в виде строк с одной точкой на строку, нормализовать "растянутое" пространство за счет вычисления нового вектора $D^* = D\Sigma^{-1/2}$

Пример использования K-means

Функция, реализующая алгоритм K-means, выглядит следующим образом:

```
void cvKMeans2(
    const CvArr* samples
    , int cluster_count
    , CvArr* labels
    , CvTermCriteria termcrit
);
```

Массив *samples* является матрицей многомерных точек, по одной на строку. Есть один тонкий момент относительно этого параметра и заключается он в том, что каждый элемент-точка может быть регулярным вектором вещественных чисел типа *CV_32FC1* или многомерной точкой типа *CV_32FC2* или *CV_32FC3* или даже *CV_32FC(K)* (это

эквивалентно матрице NxK типа `32FC1`, в которой N строк точек и K колонок для отдельных компонент каждого расположения точки). Параметр `cluster_count` является выходным и содержит вектор меток, указывающих на окончательный индекс кластера каждой точки. Описание параметра `termcount` можно найти в: раздел "Универсальные методы библиотеки ML" -> подраздел "Контролирование итераций обучения".

Далее представлен пример (пример 13-1) использования K-means, который полезен для понимания и освоения других подходов машинного обучения.

Пример 13-1. Пример использования алгоритма K-means

```
#include "cxcore.h"
#include "highgui.h"

void main( int argc, char** argv ) {
#define MAX_CLUSTERS 5

    CvScalar      color_tab[MAX_CLUSTERS];
    IplImage*     img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG        rng = cvRNG( 0xffffffff );

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );

    for(;;) {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        // генерация случайной выборки по многомерному
        // распределению Гаусса
        for( k = 0; k < cluster_count; k++ ) {
            CvPoint center;
            CvMat point_chunk;

            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;

            cvGetRows(
                points
                ,&point_chunk
                ,k*sample_count/cluster_count
                ,k == cluster_count - 1 ? sample_count
                : (k+1)*sample_count/cluster_count
            );
        }
    }
}
```

```

        cvRandArr(
            &rng
            ,&point_chunk
            ,CV_RAND_NORMAL
            ,cvScalar( center.x,center.y, 0, 0 )
            ,cvScalar( img->width/6, img->height/6, 0, 0 )
        );
    }

    // перемешивание образцов
    for( i = 0; i < sample_count/2; i++ ) {
        CvPoint2D32f* pt1 = (CvPoint2D32f*)points->data.fl
            + cvRandInt(&rng)%sample_count;
        CvPoint2D32f* pt2 = (CvPoint2D32f*)points->data.fl
            + cvRandInt(&rng)%sample_count;
        CvPoint2D32f temp;
        CV_SWAP( *pt1, *pt2, temp );
    }

    cvKMeans2(
        points
        ,cluster_count
        ,clusters
        ,cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 )
    );

    cvZero( img );

    for( i = 0; i < sample_count; i++ ) {
        CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
        int cluster_idx = clusters->data.i[i];

        cvCircle(
            img
            ,cvPointFrom32f(pt)
            ,2
            ,color_tab[cluster_idx]
            ,CV_FILLED
        );
    }

    cvReleaseMat( &points );
    cvReleaseMat( &clusters );

    cvShowImage( "clusters", img );

    int key = cvWaitKey(0);
    if( key == 27 ) { // 'ESC'
        break;
    }
}

```

В данном примере используется *highgui.h* для управления интерфейсом окна и *cxcore.h*, содержащий *Kmeans2()*. В функции *main()* задаются цвета отображения кластеров, верхний предел максимально выбираемых случайным образом возможных центров кластера *MAX_CLUSTERS* (равный 5) в *cluster_count* и 1000 точек, случайным образом выбираемые и сохраняемые в *sample_count*. Внешний цикл *for* повторяется до тех пор, пока не будет нажата клавиша Esc, внутри цикла происходит выделение памяти под указатель на матрицу *points* вещественного типа, содержащая точки *sample_count* (в данном случае одну колонку двумерных точек типа *CV_32FC2*) и выделение памяти под указатель на матрицу *clusters* целочисленного типа, содержащая получаемые метки кластера, от 0 до *cluster_count* - 1.

Следующий вложенный цикл *for* генерирует вводимые данные, которые могут быть повторно использованы для тестирования других алгоритмов. Для каждого кластера происходит последовательное порционное заполнение массива точек размера *size sample_count / cluster_count*. Каждая порция заполняется в соответствии с нормальным распределением *CV_RAND_NORMAL*, двумерные (*CV_32FC2*) точки сосредоточены в случайно выбранном двумерном центре.

Следующий цикл *for* просто перемешивает общий результат "пачки" точек. Затем происходит вызов функции *cvKMeans2()*, которая выполняется до тех пор, пока наибольшее перемещение центра кластера не станет меньше 1 (при этом число итераций ограничено 10).

Последний цикл *for* формирует изображение результата. Затем происходит освобождение занимаемой памяти и отображение результата за счет изображения "clusters". В конце управление передается пользователю, который может продолжить выполнение программы или завершить её при помощи нажатия клавиши Esc.

[П]|[РС]|(РП) Naïve/Normal Bayes Classifier

Ранее обсуждались алгоритмы из библиотеки `cvcore`. Теперь настало время перейти к обсуждению библиотеки машинного обучения (ML). Для начала будет рассмотрен относительно простой классификатор, который именуется как *normal Bayes classifier* (нормальный классификатор Bayes) и *naïve Bayes classifier* (наивный классификатор Bayes) и реализован при помощи класса `CvNormalBayesClassifier`. "Наивный", т.к. предполагается, что все особенности независимы относительно друг друга, хотя иногда случается и обратное (например, при поиске одного глаза, предполагается, что другой глаз располагается где-то поблизости). В статьях *Zhang* представлены возможные причины (иногда) удивительно хорошей производительности этого классификатора. Naïve Bayes не используется для регрессии, при этом является эффективным классификатором, который может обрабатывать несколько классов, а не только два. Этот классификатор является простейшим из всех возможных, что в настоящее время способствует бурному развитию байесовых сетей или "вероятностных графических моделей". Байесовые сети являются причинно-следственными моделями; на рисунке 13-6, например, особенности лица получены в результате наличия лица. Во время использования, переменная лицо считается скрытой переменной, а особенности лица - после обработки исходного изображения - составляют доказательство существования лица. Это называется "порождающей" моделью, т.к. лицо является причиной образования особенностей лица. И наоборот, можно начать с предположения, что узел лицо является активом и затем случайным образом формировать выборку из возможно генерируемых особенностей, учитывая, что лицо является активом (Глупо создавать лицо, используя наивный байесовый алгоритм, т.к. он предполагает независимость особенностей. Однако, более общие байесовые сети можно легко построить, по мере необходимости, с зависимыми особенностями). Генерация данных сверху вниз по той же статистике, что и для причинно-следственной модели (лицо), порождает полезные особенности, чем в свою очередь чисто дискриминационные модели не обладают. Например, можно генерировать лица для компьютерной графики или предоставлять роботу возможность буквально "представлять" что он должен делать дальше, генерируя сцены, объекты и взаимодействия. Для понимания отличий (при использовании рисунка 13-6), дискриминационная модель будет иметь направления стрелок в обратную сторону.

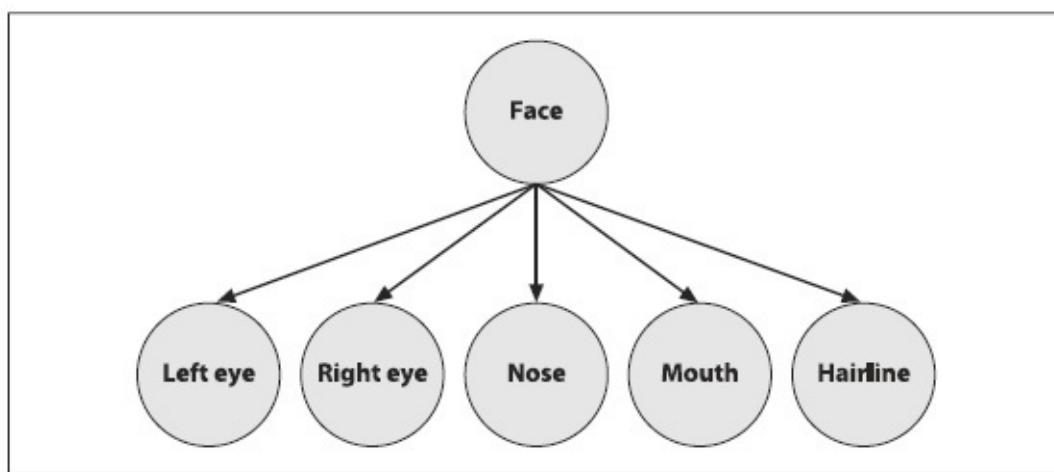


Рисунок 13-6. (Наивная) Байесовая сеть, где особенности нижнего уровня образуются в результате наличия объекта (лица)

Байесовые сети являются глубокими и первоначально сложно уяснить поле деятельности, но при этом алгоритм naïve Bayes основан на простом применении закона Байеса. В этом случае, вероятность (обозначено p) определения лица по особенностям (показанные на рисунке 13-6 и обозначенные слева направо как LE, RE, N, M, H) определяется по следующей формуле:

$$p(\text{face} | \text{LE, RE, N, M, H}) = \frac{p(\text{LE, RE, N, M, H} | \text{face}) p(\text{face})}{p(\text{LE, RE, N, M, H})}$$

Просто, чтобы знать, по-английски это уравнение означает следующее:

posterior probability = (likelihood x prior probability) / evidence

апостериорная вероятность = (вероятность x предварительную вероятность) / признаки

На практике, вычисляется член *evidence*, а затем решается, какой объект стал его причиной. Если вычисляемый член *evidence* остается постоянным для данного объекта, то его можно отбросить. Если же имеется множество моделей, то нужно найти только один максимальный числитель *evidence*. Данный числитель объединяет вероятность модели с данными: $p(\text{face}, \text{LE, RE, N, M, H})$. Соответственно можно использовать определенную условную вероятность для получения объединенной вероятности:

$$p(\text{face, LE, RE, N, M, H}) = p(\text{face}) p(\text{LE} | \text{face}) p(\text{RE} | \text{face, LE}) p(\text{N} | \text{face, LE, RE}) \times p(\text{M} | \text{face, LE, RE, N}) p(\text{H} | \text{face, LE, RE, N, M})$$

Применяя предположение о независимости особенностей, отпадает необходимость в условных особенностях. Таким образом, обобщая лицо в "object" и особенные особенности в "all features", получается следующее уравнение:

$$p(\text{object}, \text{all features}) = p(\text{object}) \prod_{i=1}^{\text{all features}} p(\text{feature}_i | \text{object})$$

Для использования этого в качестве обобщенного классификатора, необходимо изучить модели требуемых объектов. В режиме использования вычисляются особенности и ищется объект, который максимизирует это уравнение. Как правило, во время тестов происходит определение объекта "победителя" при помощи использования порогового значения. Если условие, согласно пороговому значению, выполняется, то можно считать, что объект найден; в противном случае, объявляется, что объект не распознан.

Если (как это часто бывает) требуется найти только один объект, то может возникнуть вопрос: "Вычисляемая вероятность - это вероятность относительно чего?". В представленном случае неявно имеется ещё один объект - фон, объединяющий в себе все то, что не является объектом, который требуется найти.

Процесс изучения модели довольно таки прост. Собирается множество изображений объектов; затем вычисляются особенности этих объектов и доля того, как часто встречается особенность каждого объекта в наборе для обучения. На практике, стараются избегать нулевой вероятности, т.к. это будет означать, что объект не существует; поэтому нулевая вероятность, как правило, задается очень малым значением. В общем, при ограниченном наборе данных, простые модели, такие как naïve Bayes, будут превосходить более сложные модели, которые "выдумывают" о данных больше, чем есть на самом деле (проявляется необъективность).

Функции, реализующие Naïve/Normal Bayes

Метод обучения классификатора normal Bayes в OpenCV представлен следующей функцией:

```
bool CvNormalBayesClassifier::train(
    const CvMat* _train_data
    ,const CvMat* _responses
    ,const CvMat* _var_idx = 0
    ,const CvMat* _sample_idx = 0
    ,bool update = false
);
```

Данная функция реализует ранее описанный метод обучения, при этом позволяет использовать только те данные, для которых точка обучения храниться в строке (т.е. используется флаг `tflag = CV_ROW_SAMPLE`). Кроме того, входной параметр `_train_data` является вектором-столбцом типа `CV_32FC1`, который должен быть только упорядоченного типа `CV_VAR_ORDERED` (пронумерованным). Выходной параметр

`_responses` содержит метки и является вектором-столбцом, который должен быть только типа категории `CV_VAR_CATEGORICAL` (содержит целочисленные значения, даже если вектор является типа `float`). Параметры `_var_idx` и `_sample_idx` являются необязательными; они позволяют отметить (соответственно) особенности и точки данных, которые требуется использовать. В большинстве случаев достаточно использовать только первые два параметра, а данные векторы можно просто установить в `NULL`, однако, стоит помнить, что параметр `_sample_idx` может быть использован, например, для разделения наборов для обучения и тестирования. Оба вектора либо одноканальные целочисленные (`CV_32SC1`), либо 8-битные значения маски (`CV_8UC1`), где 0 означает пропуск. И наконец, параметр `update` позволяет просто обновить обученную модель `normal Bayes`, вместо того, чтобы обучать новую модель с нуля.

Метод прогноза `CvNormalBayesClassifier` вычисляет наиболее возможный класс исходного вектора. Один или более исходных векторов хранятся в строках матрицы `samples`. Предсказания возвращаются в соответствующих строках вектора `results`. Если имеется только один исходный вектор, то предсказание является возвращаемым значением метода `predict` типа `float`, а матрица `results` в таком случае устанавливается в `NULL` (значение по умолчанию). Метод прогнозирования выглядит следующим образом:

```
float CvNormalBayesClassifier::predict(
    const CvMat*   samples
    ,CvMat*        results = 0
) const;
```

[П]|[РС]|(РП) Binary Decision Trees

Decision trees будут рассмотрены более подробно, т.к. они весьма полезны и используют большую часть функциональности библиотеки машинного обучения (можно сказать являются таким неким учебным пособием для изучения библиотеки ML). Binary decision trees были придуманы Leo Breiman и его коллегами (L. Breiman, J. Friedman, R. Olshen b C. Stone, Classification and Regression Trees (1984), Wadsworth), которые назвали их алгоритмами *classification and regression tree* (CART). В OpenCV реализован алгоритм *decision tree*. Суть алгоритма заключается в определении нечистой метрики (примеси) относительно данных каждого узла дерева. Например, для использования подходящей функции регрессии, можно использовать сумму квадратов разности между истинным и прогнозируемым значениями. В дальнейшем может потребоваться свести к минимуму эту сумму разности ("примеси") каждого узла дерева. В случае категориальных меток необходимо определить меру, которая будет минимальна, когда большинство значений в узле имеют один и тот же класс. Как правило, используют три меры - энтропию, индекс Gini и неправильную классификацию (далее все эти меры будут более детально рассмотрены в этом разделе). После получения метрики, binary decision tree выполняет поиск по вектору особенностей в поисках объединенной особенности, которая совместно с пороговым значением наилучшим образом очищает данные. Согласно заранее оговоренным условиям, особенности, которые выше порогового значения являются "правдивыми" и, таким образом, классифицируются в левую ветку; в противном случае в правую. В последующем данная процедура выполняется рекурсивно вниз по каждой ветке дерева до тех пор, пока данные не станут достаточно чистыми или пока не будет достигнуто минимально установленное число точек в узле.

Далее будет рассмотрено уравнение узла примеси $i(N)$ для двух случаев: регрессия и классификация.

Регрессия примеси

Для регрессии или функции соответствия уравнения узла примеси - это просто квадрат разности между значением узла y и значением данных x :

$$i(N) = \sum_j (y_j - x_j)^2$$

Классификация примеси

В случае классификации, decision trees зачастую используют один из трех методов: *entropy impurity*, *Gini impurity* или *misclassification impurity*. Для разъяснения этих методов будут использоваться следующие обозначения: $P(\omega_j)$ обозначает долю шаблонов в узле N класса ω_j . Каждая примесь имеет несколько различных эффектов в момент принятия решения о разделении. Метод Gini является наиболее часто используемым, при этом все методы без исключения стремятся свести к минимуму примеси в узле. На рисунке 13-7 показан график примесей, которые необходимо свести к минимуму.

Entropy impurity

$$i(N) = -\sum_j P(\omega_j) \log P(\omega_j)$$

Gini impurity

$$i(N) = \sum_{j \neq i} P(\omega_i)P(\omega_j)$$

Misclassification impurity

$$i(N) = 1 - \max_j P(\omega_j)$$

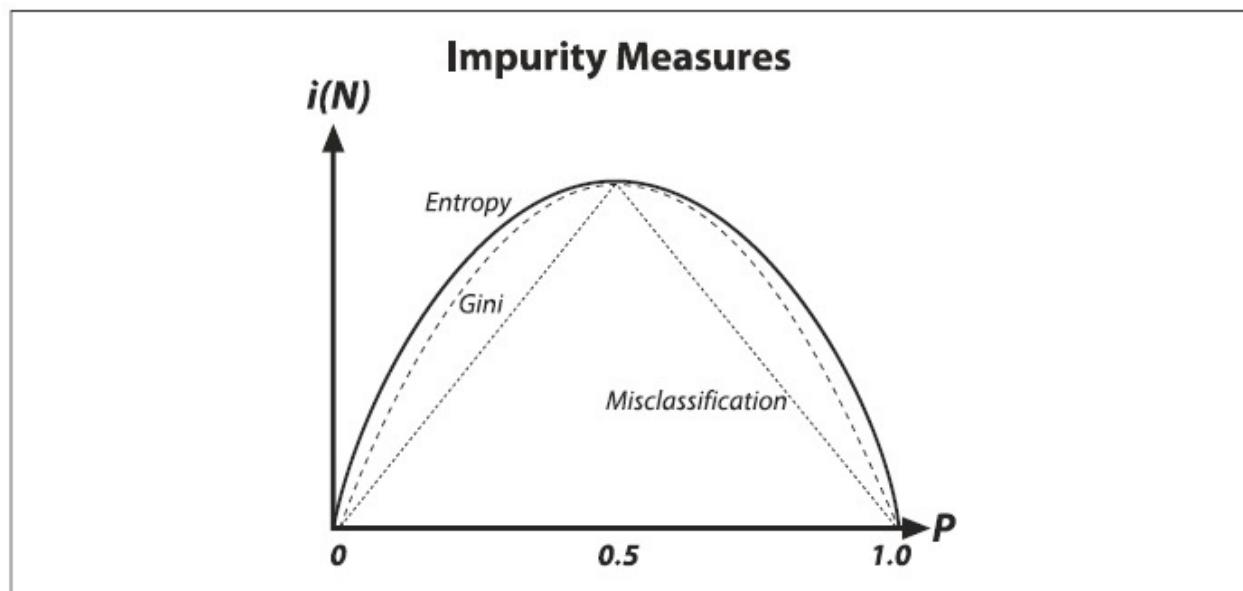


Рисунок 13-7. Примеси decision tree

Decision trees является, пожалуй, наиболее широко используемой технологией классификации. Это связано с простотой реализации и интерпретации результатов, гибкостью к различным типам данных (категоричные, численные и ненормализованные + их сочетания), способностью обрабатывать недостающие

данные за счет суррогатного разделения и естественного способа присвоения важности особенностям в порядке разделения. Decision trees является основой для других алгоритмов: *boosting* и *random trees*, которые будут рассмотрены чуть позже.

Использование decision tree

Далее будет представлено более чем достаточное описание работы decision trees. При этом стоит знать, что существует множество других способов получения доступа к узлам, изменения разделителей и так далее. Для сформирования наиболее полного уровня (который, возможно, когда то может потребоваться читателю) представления, необходимо обратиться к руководству пользователя

`.../opencv/docs/ref/opencvref_ml.htm`, в частности, к классу `CvDTree{}`, классу обучения `CvDTreeTrainData{}` и к классам, ответственных за узлы и разделители `CvTreeNode{}` и `CvTreeSplit{}`, соответственно.

При переходе от теории к практике, стоит начать с рассмотрения конкретного примера. В папке `.../opencv/samples/c` имеется файл `mushroom.cpp`, в котором представлена реализация *decision trees* на данных, расположенных в файле `agaricus-lepiota.data`. Файл с данными состоит из меток "р" или "е" (обозначающие *poisonous* (ядовитые) или *edible* (съедобные), соответственно) в совокупности с 22 категориями (каждая представлена одной буквой) признаков. При этом стоит заметить, что файл с данными имеет формат CVS, где значения особенностей разделены точкой с запятой. Файл `mushroom.cpp` содержит не совсем корректную функцию `mushroom_read_database()`, которая читает содержимое файла с данными. Эта функция довольно-таки специфична и "хрупка", при этом с основной задачей - заполнить три массива - она справляется прекрасно:

1. Матрица вещественного типа `data[][]`, которая имеет размерность: число строк = числу точек, число колонок = числу особенностей (в рассматриваемом случае 22), при этом все буквенные обозначения особенностей преобразуются в вещественное число
2. Матрица символьного типа `missing[][]`, где значения "true" или 1 указывают на недостающие значения, которые помечены в строке файла с данными вопросительным знаком, а 0 на все оставшиеся.
3. Вектор вещественного типа `responses[]` содержит ответ "р" (ядовитый) или "е" (съедобный).

Далее будет представлен подробный разбор всего того, что используется, напрямую или косвенно, в файле `mushroom.cpp` из главной функции `main()`.

Обучение дерева

Для выполнения обучения дерева необходимо заполнить структуру `CvDTreeParams`:

```

struct CvDTreeParams {
    int max_categories; // Граница предварительной кластеризации
    int max_depth; // Максимальный уровень дерева
    int min_sample_count; // Не делить узел, если меньше
    int cv_folds; // Обрезка дерева с К кратной
                    // перекрестной проверкой
    bool use_surrogates; // Альтернативные разделители для
                        // отсутствующих данных
    bool use_1se_rule; // Грубое удаление
    bool truncate_pruned_tree; // Не "запоминать" отброшенные ветки
    float regression_accuracy; // Один из критериев "стоп-разделителя"
    const float* priors; // Вес каждой предсказанной категории

    CvDTreeParams():
        max_categories(10)
        ,max_depth(INT_MAX)
        ,min_sample_count(10)
        ,cv_folds(10)
        ,use_surrogates(true)
        ,use_1se_rule(true)
        ,truncate_pruned_tree(true)
        ,regression_accuracy(0.01f)
        ,priors(NULL)
    { ; }

    CvDTreeParams(
        int _max_depth
        ,int _min_sample_count
        ,float _regression_accuracy
        ,bool _use_surrogates
        ,int _max_categories
        ,int _cv_folds
        ,bool _use_1se_rule
        ,bool _truncate_pruned_tree
        ,const float* _priors
    );
}

```

В данной структуре параметр `max_categories` имеет значение по умолчанию равное 10. Это ограничение для категориальных значений до которого decision tree будет выполнять предварительную кластеризацию так, чтобы проверить не более $2^{\max_categories} - 2$ возможных значений подмножества. Это не проблема для упорядоченных или пронумерованных особенностей, где алгоритм просто должен найти порог, согласно которому производилось бы разделение налево и направо. Переменные, которые в большей степени категориальны, чем `max_categories`, будут равны кластеру значений, которые меньше `max_categories` возможных значений. Таким

образом, *decision trees* должны протестировать не более чем *max_categories* уровней за раз. Если данный параметр будет иметь малое значение, то это приводёт к уменьшению вычислений за счет понижения точности.

(На заметку. Более детальное сопоставление категориальных и упорядоченных разделителей: в то время, как разделение упорядоченной переменной происходит следующим образом: "если $x < a$, то перемещаем влево, иначе вправо", разделение категориальной переменной происходит следующим образом: "если

$x \in \{v_1, v_2, v_3, \dots, v_k\}$, то перемещаем влево, иначе вправо", где v_i это некоторое возможное значение переменной. Таким образом, если категориальная переменная имеет N возможных значений, то для наилучшего разделения необходимо попробовать $2^N - 2$ подмножеств (пустое или полное подмножества необходимо исключить). Таким образом, алгоритм аппроксимации определяет каким образом все N значений нужно сгруппировать в $K \leq max_categories$ кластеров (алгоритм K-mean), основываясь на статистике образца текущего анализируемого узла. После этого алгоритм попытается произвести различные сочетания кластеров и выбрать наилучший разделитель, который даёт хорошие результаты чаще других. При этом для двух наиболее распространенных задач, классификация и регрессия с двумя классами, оптимальный категориальный разделитель (т.е. наилучшее подмножество значений) может быть эффективно найден без выполнения кластеризации. Соответственно кластеризацию необходимо применять только в случае $n >$ двух классов проблем классификации для категориальных переменных при $N > max_categories$ возможных значений. Таким образом, необходимо подумать дважды, прежде чем устанавливать *max_categories* более чем 20, т.к. это будет означать более миллиона операций для каждого разделителя!)

Остальные параметры имеют комментарии, которых вполне достаточно для объяснения их смысла. Последний параметр *priors* в некоторых случаях может быть ключевым. Он задает относительный вес, который будет задавать неверную классификацию. Т.е. если вес первой категории имеет 1, а вес второй категории равен 10, то каждая ошибка предсказаний во второй категории дает эквивалентные 10 ошибок при прогнозировании в первой категории. В рассматриваемом примере имеются съедобные и ядовитые грибы; таким образом, "наказание" для ошибки в случае, когда ядовитый гриб определился как съедобный в 10 раз больше "наказания" для ошибки в случае, когда съедобный гриб определился как ядовитый.

Далее показан шаблон метода обучения *decision tree*. В общем имеется два варианта метода: первый используется непосредственно для работы с *decision trees*; второй используется для формирования ансамблей (используются в *boosting*) или лесов (используются в *random trees*).

```

// Работа непосредственно с decision trees:
bool CvDTree::train(
    const CvMat*    _train_data
, int             _tflag
, const CvMat*   _responses
, const CvMat*   _var_idx      = 0
, const CvMat*   _sample_idx   = 0
, const CvMat*   _var_type     = 0
, const CvMat*   _missing_mask = 0
, CvDTreeParams  params       = CvDTreeParams()
);

// Метод использует ансамбль decision trees,
// перебирая в процессе обучения каждое дерево
// из ансамбля
bool CvDTree::train(
    CvDTreeTrainData* _train_data
, const CvMat*     _subsample_idx
);

```

Первый параметр метода `_train_data[]/[]` является матрицей вещественного типа. Если `_tflag = CV_ROW_SAMPLE`, то каждая строка состоит из указателя, который содержит вектор особенностей, формирующие столбцы матрицы. Если `tflag = CV_COL_SAMPLE`, то значения строк и столбцов меняются местами. Аргумент `_responses[]/[]` является вектором вещественного типа, значения которого могут быть предсказаны с учетом особенностей. Остальные аргументы являются необязательными. Вектор `_var_idx` содержит особенности, а вектор `_sample_idx` содержит наблюдения; оба вектора либо целочисленные списки значений пропусков (0), либо 8-битные маски активов (1) или пропусков (0) (ранее в данной главе это уже обсуждалось при рассмотрении метода `train()`). Вектор `_var_type` типа `byte` (`CV_8UC1`) - это маска для каждого типа особенности (`CV_VAR_CATEGORICAL` или `CV_VAR_ORDERED`, где `CV_VAR_ORDERED` тоже самое, что и `CV_VAR_NUMERICAL`); размерность данного вектора равна числу особенностей плюс 1 (для записи типа ответа). Матрица `_missing_mask[]/[]` типа `byte` используется для фиксации отсутствующих значений как 1 (в противном случае как 0). В примере 13-2 детально показан процесс создания и обучения *decision tree*.

Пример 13-2. Создание и обучение *decision tree*

```

float    priors[] = { 1.0, 10.0}; // Вес съедобный относительно веса ядовитый
CvMat*  var_type;
var_type = cvCreateMat( data->cols + 1, 1, CV_8U );
cvSet( var_type, cvScalarAll(CV_VAR_CATEGORICAL) ); // все значения категориальные

CvDTree* dtree;
dtree = new CvDTree;
dtree->train(
    data
    ,CV_ROW_SAMPLE
    ,responses
    ,0
    ,0
    ,var_type
    ,missing
    ,CvDTreeParams(
        8          // максимальная глубина
        ,10         // минимальное число образцов
        ,0          // точность регрессии: используется N/A
        ,true       // вычислять ли суррогатный разделитель
                    // из-за наличия недостающих данных
        ,15         // максимальное число категорий
                    // (алгоритм не оптимален в случае
                    // больших чисел)
        ,10         // перекрестная проверка
        ,true       // использовать ли правило 1SE => небольшое дерево
        ,true       // отбрасывать ли отброшенные ветви дерева
        ,priors    // массив priors, где чем больше
                    // p_weight, тем больше вероятность того, что
                    // ядовитый
    )
);

```

В начале представленного примера происходит объявление *decision tree* как *dtree* и выделение под него необходимого объема памяти. Затем происходит вызов метода *dtree->train()*. Для рассматриваемого случая вектор *responses* будет заполнен соответствующими ASCII символами "р" (ядовитый) или "е" (съедобный) для каждого наблюдения. После выполнения метода *train()*, *dtree* готово к использованию для прогнозирования новых данных. Так же *decision tree* может быть сохранен на диск или прочитан с него. В промежутке, между сохранением и загрузкой, дерево необходимо сбрасывать и обнулять при помощи метода *clear()*.

```

dtree->save( "tree.xml", "MyTree" );
dtree->clear();
dtree->load( "tree.xml", "MyTree" );

```

Методы сохранения и загрузки используют файл *tree.xml*. (Расширение .xml указывает на XML формат файла; если бы использовалось расширение .yml или .yaml, то это указывало бы на YAML формат файла). Дополнительный параметр "MyTree" является тегом, который маркирует дерево. Как и в других статистических моделях, в модуле "машинное обучение" есть несколько объектов, которые не могут быть сохранены в одном .xml или .yml файле при использовании *save()*; для сохранения таких объектов необходимо использовать *cvOpenFileStorage()* и *write()*. В случае с функцией *load()* это совсем другая история: данная функция может загрузить объект по его имени, даже если в файле присутствуют другие данные, не относящиеся к объекту.

Функция предсказания выглядит следующим образом:

```
CvDTreeNode* CvDTree::predict(
    const CvMat*   _sample
    ,const CvMat*   _missing_data_mask = 0
    ,bool           raw_mode         = false
) const;
```

Параметр *_sample* является вектором вещественного типа и содержит особенности, используемые в предсказании; параметр *_missing_data_mask* является вектором типа *byte* той же длины и ориентации (под "та же ... ориентация" имеется ввиду, что если образец является вектором 1xN, то маска должна быть размера 1xN, а если образец имеет размерность Nx1, то маска должна быть размера Nx1), как и вектор *_sample*, в котором (*_missing_data_mask*) ненулевые значения указывают на недостающие особенности. Параметр *raw_mode* указывает на нормализацию входных категориальных данных: "false" – не нормализовать (по умолчанию), "true" – нормализовать. Данный параметр, как правило, используется в случае ансамблей деревьев для ускорения процесса предсказания. Данные нормализуются в пределах интервала (0, 1) для ускорения процесса вычисления, т.к. таким образом алгоритму будут известны пределы, в которых данные могут колебаться. Такая нормализация никак не сказывается на точности. Метод *predict()* возвращает узел дерева через который можно получить доступ к значению предсказания при помощи (*CvDTreeNode*)->*value*:

```
double r = dtree->predict( &sample, &mask )->value;
```

И в заключении можно вызвать метод *var_importance()* для определения важности каждой особенности. Эта функция возвращает вектор размера Nx1 типа *double* (*CV_64FC1*), содержащий каждую особенность относительно важности для

прогнозирования, где значение 1 указывает на высокий уровень важности, а 0 на абсолютную неважность или полезность для предсказания. В дальнейшем неважные особенности могут быть отброшены при втором проходе обучения.

```
const CvMat* var_importance = dtree->get_var_importance();
```

Как показано в файле `.../opencv/samples/c/mushroom.cpp`, доступ к отдельному элементу вектора значимости можно получить за счёт следующей конструкции

```
double val = var_importance->data.db[i];
```

Большинство пользователей используют только метод `train()` при использовании *decision tree*, однако продвинутым пользователям или исследователям может иногда потребоваться изучить и/или модифицировать узлы дерева или критерии разделения. Как уже было сказано в начале данного раздела, информация о том, как это сделать при помощи ML описано в поставляемой с OpenCV документации [.../opencv/docs/ref/opencvref_ml.htm#ch_dtrees](#), доступ к которой можно так же получить через [OpenCV Wiki](#). Разделы представляющие интерес в случае выполнения анализа: структура класса `CvDTree{}`, структура обучения `CvDTreeTrainData{}`, структура узла `CvDTreeNode{}` и структура, содержащая разделитель `CvDTreeSplit{}`.

Результаты decision tree

Используя ранее описанный код, можно узнать несколько интересных вещей о съедобных и ядовитых грибах, исходная информация о которых храниться в файле `agaricus-lepiota.data`. Если выполнить просто обучение decision tree (без отсечения), то можно получить вполне хорошие данные (дерево для данного случая представлено на рисунке 13-8). Хоть выполнение полного цикла обучения decision tree на наборе для обучения и дает лучшие результаты, не стоит забывать и об уроке, показанного на рисунке 13-2 (переобучение). Сделанное и показанное на рисунке 13-8 выполнено с целью запоминания данных вместе со своими ошибками и шумом. Таким образом, маловероятно, что будут хорошие результаты на реальных данных. Вот почему decision trees и деревья типа CART в OpenCV, как правило, включают в себя дополнительные шаги "наказывания" сложных деревьев и их обрезку до момента достижения должного уровня сложности. Есть и другие варианты реализации decision tree, которые "выращивают" деревья до тех пор, пока не будет сбалансирована сложность, и за счет объединения фазы обрезания с фазой обучения. Тем не менее, в ходе разработки библиотеки ML было обнаружено, что деревья, которые полностью выращены, а затем обрезаны (так реализовано в OpenCV), лучше тех деревьев, которые сочетают обучение с обрезкой в фазе generation.

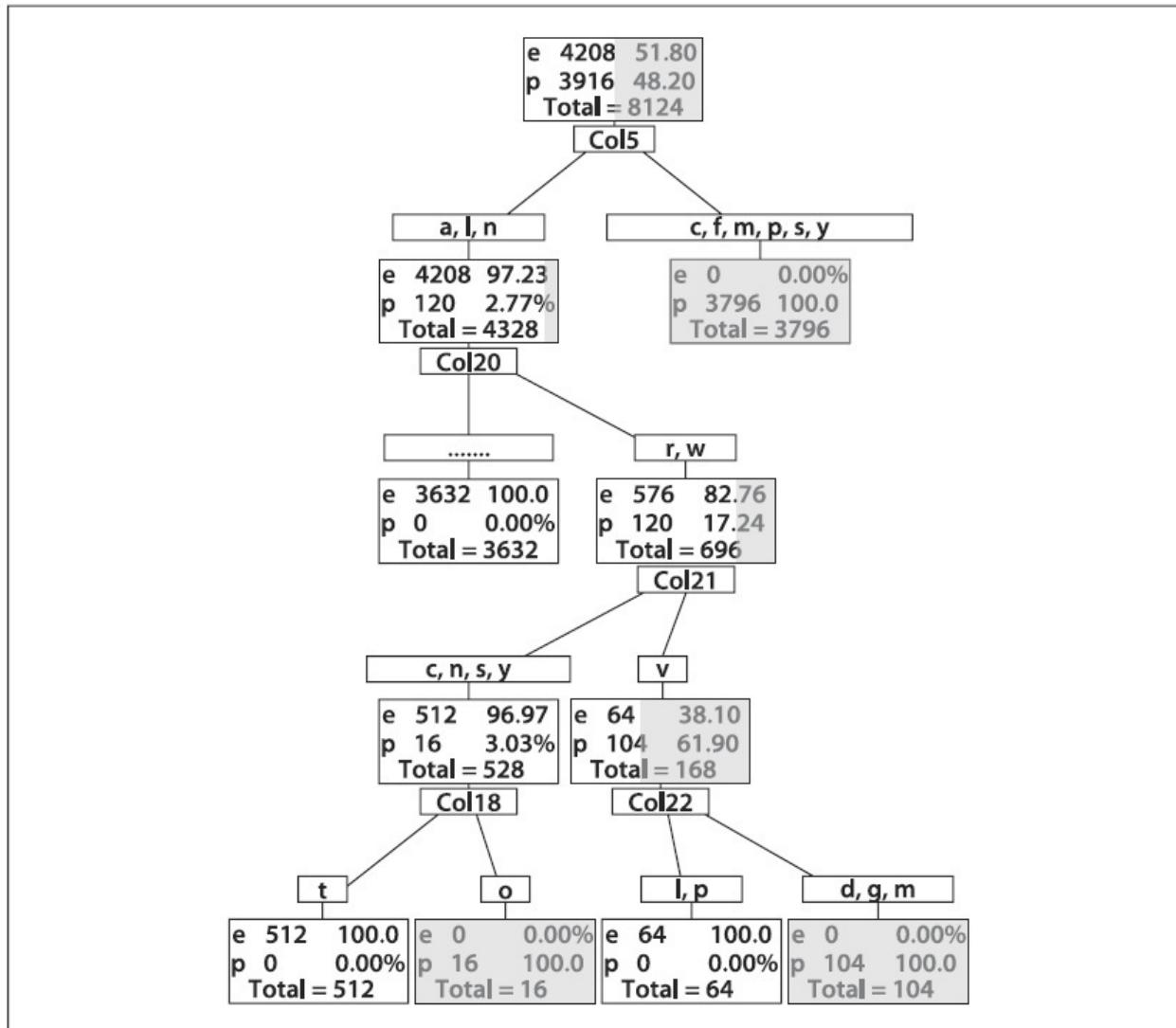


Рисунок 13-8. Полное decision tree для ядовитых (p) или съедобных (e) грибов: данное дерево было построено с 0% ошибок на наборе для обучения, которое, возможно, страдает от проблем с дисперсией на наборах для тестирования или на реальных данных (темная часть треугольника соответствует ядовитым грибам на данном этапе категоризации)

На рисунке 13-9 показано обрезанное дерево, которое достаточно хорошо (но не идеально) справляется с набором для обучения, но, возможно, ещё лучше справляется с реальными данными, т.к. дерево обладает лучшим балансом между смещением и дисперсией. Однако, данный классификатор обладает серьёзным недостатком: ядовитые грибы определяются съедобными в 1.23% времени. Возможно, можно быть более счастливым, имея классификатор похуже, который помечал бы многие съедобные грибы как ядовитые, при условии, что никогда бы не предлагал есть ядовитые грибы! Данный классификатор может быть специально создан для умышленного смещения классификатора и/или данных. Иногда это называется *добавленной стоимостью* к классификатору. В рассматриваемом случае (с грибами) необходимо повысить стоимость неправильной оценки ядовитых грибов относительно

стоимости неправильной оценки съедобных грибов. Дополнительная стоимость может быть "наложена внутри" классификатора за счёт изменения весов, а именно изменения соотношения: несколько "плохих" : одного "хорошего". OpenCV позволяет выполнить это за счет регулирования вектора *priors* в структуре *CvDTreeParams*, передаваемого методу *train()*, как уже было показано ранее. Даже не вдаваясь в подробности кода классификатора, можно предложить вариант с наложением приоритетной стоимости путем дублирования (или повторной выборки) "плохих" данных. Дублирование "плохих" данных неявно приводит к увеличению веса "плохих" данных (данная техника может работать с любым классификатором).

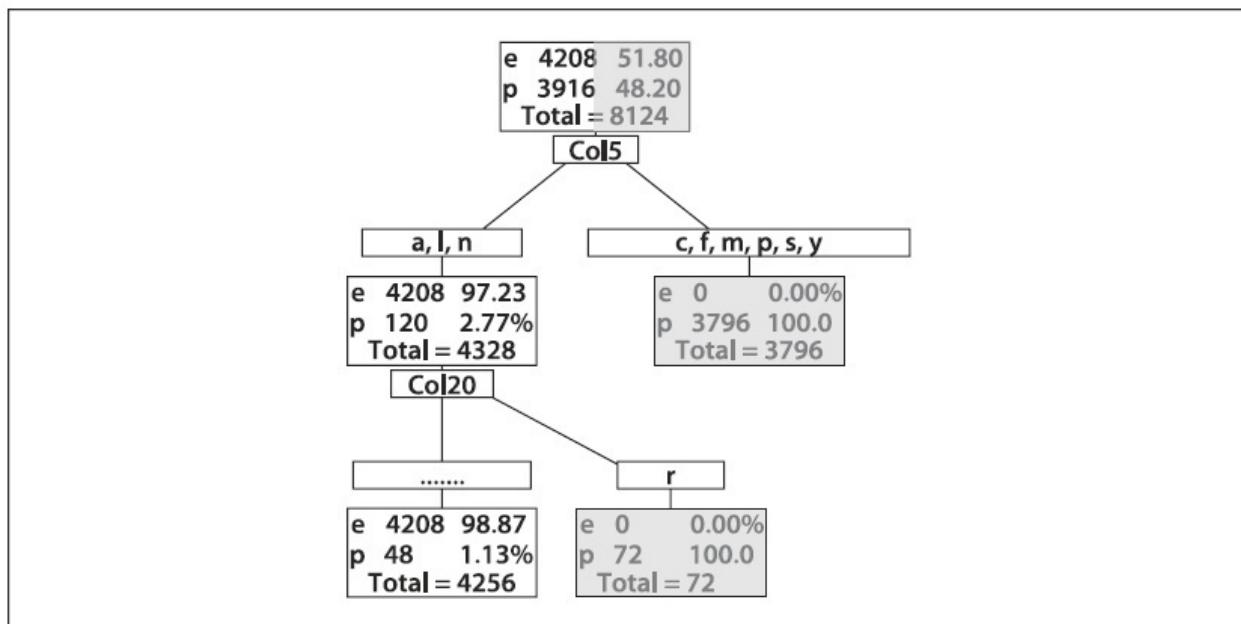


Рисунок 13-9. Обрезанное decision tree для ядовитых (p) и съедобных (e) грибов: несмотря на то, что это обрезанное дерево, оно дает низкий процент ошибок на наборе для обучения и, скорее всего, хорошо отработает на реальных данных

На рисунке 13-10 показано дерево, где навязано 10 кратное смещение вопреки ядовитым грибам. Это дерево не совершил ошибок при выборе ядовитых грибов, т.к. стоимость ядовитых грибов намного превышает стоимость съедобных грибов - случай "береженого бог бережет". Матрицы неточностей для (обрезанных) деревьев, имеющие и не имеющие смещение, показаны на рисунке 13-11.

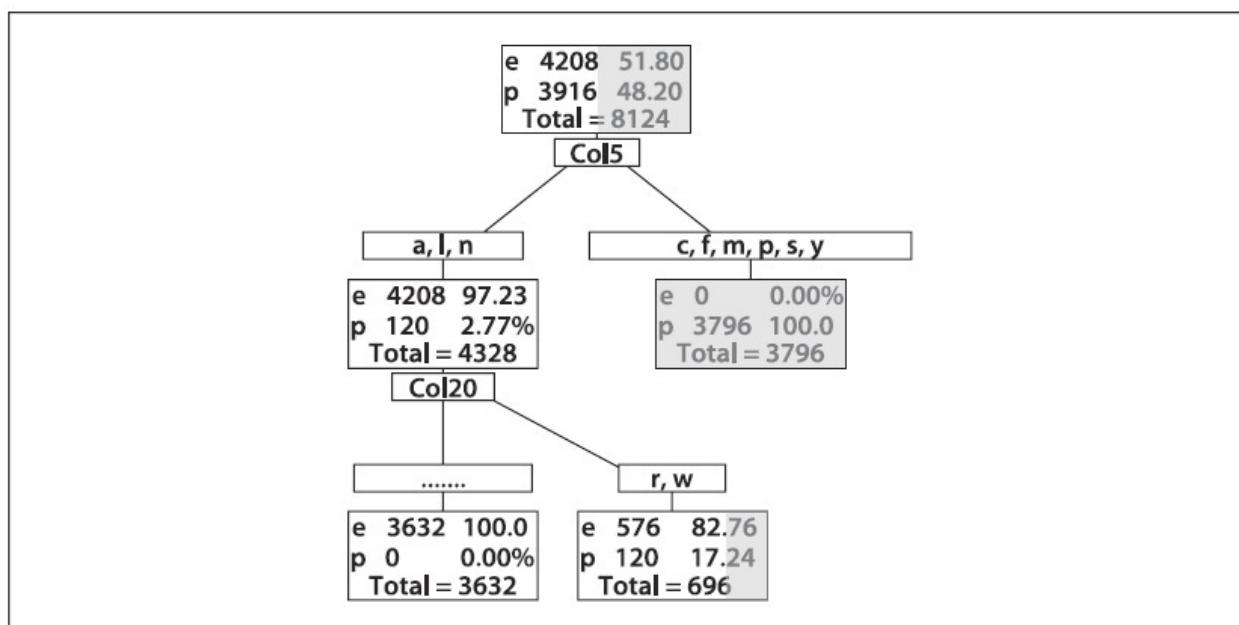


Рисунок 13-10. Decision tree съедобного гриба с 10 кратным смещением вопреки неверному определению ядовитых грибов как съедобные; нижний правый прямоугольник хотя и содержит в подавляющем большинстве съедобные грибы, он все же не содержит 10 кратное большинство, которое классифицирует как несъедобное

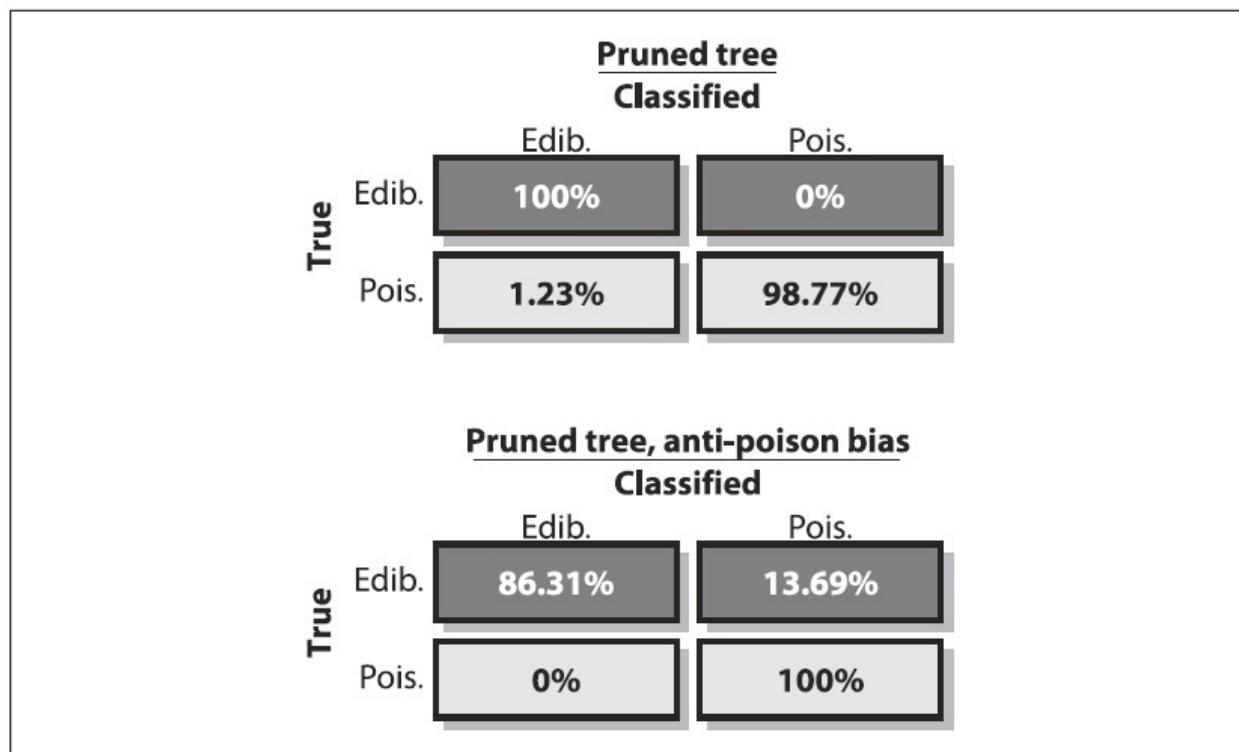


Рисунок 13-11. Матрицы неточностей для (обрезанных) decision trees съедобных грибов: дерево без смещения дает лучшую общую производительность (верхняя часть рисунка), при этом иногда происходит неверная классификация съедобных грибов как ядовитые; дерево со смещением не влияет на общую производительность (нижняя часть рисунка) и никогда не классифицирует ядовитые грибы ошибочно

Итого: о данных можно узнать чуть больше при помощи переменной важности, которая содержится в OpenCV классификаторах, основанных на деревьях (переменная важности может быть использована с любым классификатором, но на момент написания книги в OpenCV была реализована только в методах, основывающиеся на деревьях). Методы измерения переменной важности уже были рассмотрены в предыдущем подразделе, и суть их сводиться к последовательному перебору каждой особенности и последующему измерению влияния на производительность классификатора. Особенности, которые в большей степени влияют на производительность, важнее. Кроме того, важность в decision trees показана за счет разделителей: первые разделители, скорее всего, более важные, чем более поздние. Разделители могут быть полезным индикатором важности, но они действуют в "алчной" манере - нахождения разделяют наиболее чистые данные в текущий момент времени. Зачастую случается так, что плохое разделение в начале приводит к лучшим разделениям в дальнейшем, но деревья об этом уже не узнают (OpenCV вычисляет важную переменную сквозь все разделения, включая и суррогатные, стремящиеся снизить негативный эффект, который имеет алчный алгоритм разделения CART, оценивающий важную переменную). Переменная важности для ядовитых грибов показана на рисунке 13-12 для деревьев, имеющих и не имеющих смещение. Порядок переменных важности изменяется в зависимости от смещения деревьев.

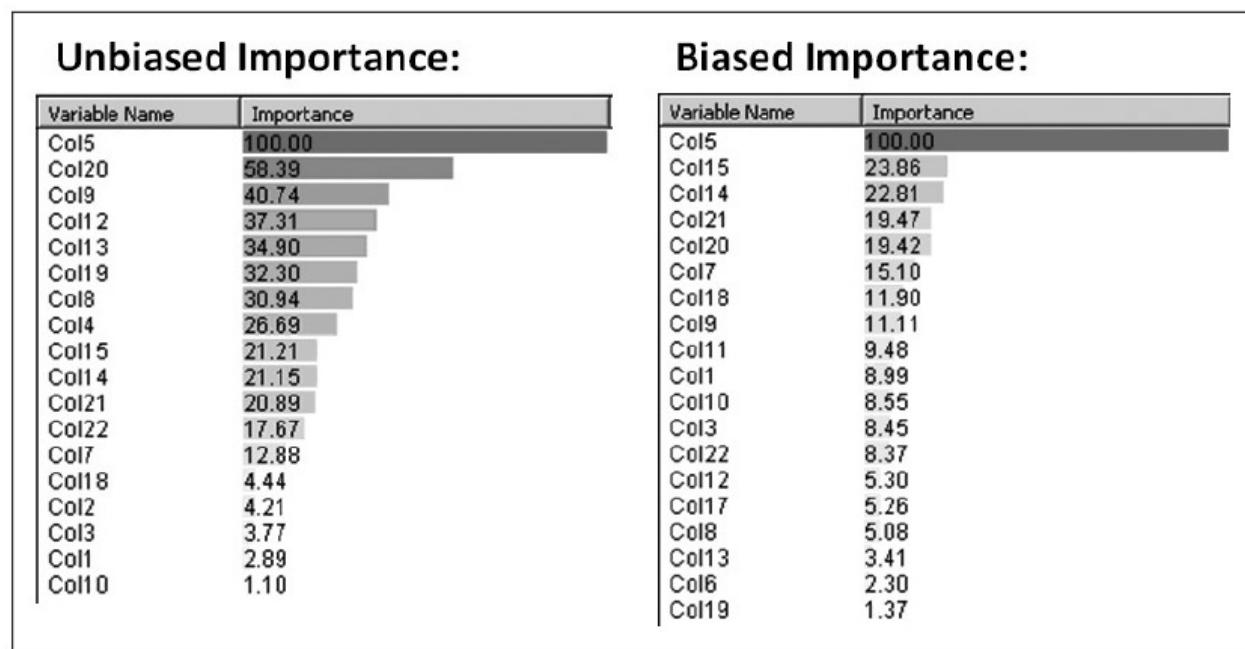


Рисунок 13-12. Переменная важности для съедобных грибов, измеренная при помощи дерева без смещения (левая часть рисунка), и дерева со смещением по отношению к ядовитым (правая часть рисунка)

[П]|[РС]|(РП) Boosting

Decision trees являются крайне полезными, однако, они зачастую не являются наиболее эффективными классификаторами. В этом и следующем разделах будут представлены два метода, *boosting* и *random trees*, которые используют деревья в их внутреннем цикле, тем самым наследуя многие полезные свойства деревьев (например, имеется возможность работать со смешанными и ненормализованными типами данных и отсутствующими особенностями). Реализация этих методов сопоставимо с искусством; поэтому зачастую лучше использовать реализацию этих методов "из коробки" (реализация данных методов в библиотеке присутствует). (На самом деле стоит понимать, что не существует "лучшего" классификатора. При этом на многих наборах данных, в которых возникает интерес в vision, boosting и random trees работают достаточно хорошо).

В области контролируемого обучения существует алгоритм *мета-обучения* (впервые описан Michael Kerns в 1988), именуемый *statistical boosting*. Изначально Kerns интересовало: возможно ли получить сильный классификатор из набора слабых классификаторов (Мощность "слабого классификатора" слабо коррелирует с истинной классификацией, в то время, как "сильный классификатор" сильно коррелирует с истинной классификацией). Таким образом, понятия слабый и сильный определены в статистическом смысле). Вскоре после этого был сформулирован первый boosting алгоритм Freund и Schapire, известный как *AdaBoost* (Y. Freund и R. E. Schapire, "Эксперименты с новым алгоритмом Boosting", Машинное обучение: 30 международная конференция (Morgan Kauman, San Francisco, 1996), 148–156.). С OpenCV поставляются четыре типа boosting:

- CvBoost :: DISCRETE (discrete AdaBoost)
- CvBoost :: REAL (real AdaBoost)
- CvBoost :: LOGIT (LogitBoost)
- CvBoost :: GENTLE (gentle AdaBoost)

Каждый является вариантом оригинального AdaBoost и зачастую *real AdaBoost* и *gentle AdaBoost* работают лучше всего. *Real AdaBoost* использует доверительно-рейтинговые прогнозы и работает с категориальными данными. *Gentle AdaBoost* устанавливает меньшие веса для выбросов и как следствие зачастую получаются хорошие данные регрессии. *LogitBoost* так же может приводить к получению хороших данных регрессии. Так как для выбора одного из методов необходимо использовать только флаг, то стоит на этапе разработки попробовать все четыре варианта и выбрать тот, который дает

наилучший результат (Данный подход использует метатехнику, известную как *voodoo обучение* или *voodoo программирование*. Несмотря на беспринципность, данный подход является эффективным способом достижения наилучшей производительности. Иногда за счет более тщательного осмысления можно понять почему тот или иной метод даёт наилучший результат и как следствие более глубокое понимание данных. А иногда нет). Данный раздел посвящен описанию оригинального AdaBoost. С точки зрения классификации стоит отметить тот факт, что в OpenCV boosting реализован как двуклассовый (да-или-нет) классификатор (в отличие от decision tree или random tree, которые могут обрабатывать несколько классов за раз). (Существует трюк, именуемый как *развертка*, который может быть использован для адаптации любого двоичного классификатора (включая boosting) для N-классовой классификации проблемы, при этом операции обучения и предсказания не становятся дороже (.../opencv/samples/c/letter_recog.cpp)). Из различных boosting методов, *LogitBoost* и *GentleBoost* могут быть использованы (подраздел "Boosting Code") для выполнения регрессии в дополнение к двоичной классификации.

AdaBoost

Алгоритм boosting используется для обучения Т слабых классификаторов h_t , $t \in \{1, \dots, T\}$. Эти классификаторы по отдельности, как правило, просты. В большинстве случаев данные классификаторы получаются в результате использования алгоритма decision trees с одним единственным разделителем (именуемый *decision stumps*) или с разделителем с небольшими уровнями разделения (возможно до трех). Каждому из классификаторов назначается вес α_t в момент окончательного принятия решения. Используется меченный набор исходных векторов особенностей x_i , каждый из которых содержит скалярную метку y_i (где $i=1, \dots, M$). Для AdaBoost метки бинарные, $y_i \in \{-1, +1\}$, в случае с другими алгоритмами метка может быть любым вещественным числом. Наблюдения инициализируются весами из распределения $D_t(i)$, которые сообщают алгоритму "стоимость" неправильного наблюдения. Ключевой особенностью boosting является то, что в ходе выполнения алгоритма эта стоимость должна меняться таким образом, чтобы обучаемые позже слабые классификаторы сосредоточились на наблюдениях, которые ранее обученные слабые классификаторы обработали плохо. Алгоритм выглядит следующим образом:

1. $D_1(i) = 1/m$, $i = 1, \dots, m$
2. Для $t = 1, \dots, T$:
 - a. Ищется классификатор h_t , который минимизирует взвешенную ошибку $D_t(i)$

b. $h_t = \arg \min_{h_j \in H} \varepsilon_j$, где $\varepsilon_j = \sum_{i=1}^m D_t(i)$ (для $y_i \neq h_j(x_i)$) до тех пор, пока $\varepsilon_j < 0.5$, иначе выход

c. Для h_t задается вес $\alpha_t = \frac{1}{2} \log[(1 - \varepsilon_t)/\varepsilon_t]$, где ε_t - это средняя минимальная ошибка на шаге 2b

d. Обновляются веса наблюдений: $D_{t+1}(i) = [D_t(i) \exp(-\alpha_t y_i h_t(x_i))] / Z_t$, где Z_t уравнение нормализации всех наблюдений i .

Если на шаге 2b не будет найден классификатор с ошибкой менее 50%, то алгоритм завершает свою работу (скорее всего необходимо предоставить более лучшие особенности).

Когда алгоритм завершается, полученный сильный классификатор принимает новый исходный вектор x и классифицирует его, используя взвешенную сумму слабых классификаторов h_t :

$$H(x) = \text{sign}\left(\sum_{t=1}^T \alpha_t h_t(x)\right)$$

Функция *sign* преобразует положительное значение в 1, а отрицательное значение в -1 (значение 0 преобразованию не подвергается).

Код boosting

Код из примера `.../opencv/samples/c/letter_recog.cpp` показывает, как использовать boosting, random trees и back-propagation (multilayer perception, MLP). Код boosting схож с кодом decision trees, но только имеет собственные параметры управления:

```

struct CvBoostParams : public CvDTreeParams {
    int boost_type;      // CvBoost:: DISCRETE, REAL, LOGIT, GENTLE
    int weak_count;     // Количество слабых классификаторов
    int split_criteria; // CvBoost:: DEFAULT, GINI, MISCLASS, SQERR
    double weight_trim_rate;

    CvBoostParams();

    CvBoostParams(
        int boost_type
        ,int weak_count
        ,double weight_trim_rate
        ,int max_depth
        ,bool use_surrogates
        ,const float* priors
    );
};

};

```

В CvDTreeParams *boost_type* задаёт один из четырех алгоритмов boosting, описанных ранее. Параметр *split_criteria* может быть одним из четырех:

- CvBoost :: DEFAULT (используется разделитель специфичный для конкретного метода *boosting*)
- CvBoost :: GINI (значение по умолчанию для *real AdaBoost*)
- CvBoost :: MISCLASS (значение по умолчанию для *discrete AdaBoost*)
- CvBoost :: SQERR (ошибка метода наименьших квадратов; опция доступна только для *LogitBoost* и *gentle AdaBoost*)

Последний параметр *weight_trim_rate* используется для вычислительной экономии и его использование будет описано чуть далее. В процессе обучения многие наблюдения становятся несущественными. Тем самым вес для $D_{t(i)}$ *i*-ого наблюдения становится очень малым. Параметр *weight_trim_rate* является порогом между 0 и 1 (включительно), который неявно используется для отсечения некоторых обучающих образцов на текущей итерации *boosting*. Например, *weight_trim_rate* = 0.95. Это означает, что образцы с суммарным весом $\leq 1.0 - 0.95 = 0.05$ (5%) не участвуют в следующей итерации обучения. При этом стоит понимать смысл выражения "не участвуют в следующей итерации". Это означает, что образцы не отбрасываются навсегда. Когда следующий слабый классификатор подвергнется обучению, веса вычисляются для всех образцов и потому некоторые ранее отброшенные незначительные образцы могут быть возвращены в следующий набор для обучения. Для отключения данного параметра необходимо, чтобы *weight_trim_rate* = 0.

`CvBoostParams` наследуется от `CvDTreeParams`, что позволяет устанавливать другие параметры, связанные с *decision trees*. В частности, если обрабатываются особенности, которые могут быть исключены (Что касается компьютерного зрения, то особенности вычисляются по изображению, а затем передаются классификатору; следовательно, они почти никогда не "теряются". Потерянные особенности зачастую появляются в данных, собранных людьми - например, при сборе измерений температуры пациента по дням), то можно задать `use_surrogates`

`CvDTreeParams::use_surrogates`, который гарантирует, что альтернативные особенности будут разделены и сохранены в каждом узле. Важным моментом является установление "стоимости" должно положительных определений. Опять же, если рассматривать пример с грибами, то можно установить `priors` как `float priors[] = {1.0, 10.0}`; тогда стоимость каждой ошибки маркировки ядовитых грибов как съедобные в 10 раз больше стоимости маркировки съедобных грибов как ядовитые.

Класс `CvBoost` содержит поле `weak`, которое является указателем `CvSeq` на слабый классификатор, который наследуется от *decision trees* `CvDTree` (Имена у данных объектов несколько не интуитивны. Объект типа `CvBoost` является boosted tree классификатором. Объекты типа `CvBoosTree` являются слабыми классификаторами, которые образуют сильный классификатор. Предположительно, слабые классификаторы имеют тип `CvBoostTree`, т.к. наследуются от `CvDTree` (т.е. они содержат в себе маленькие деревья, настолько, что являются просто stumps). Поле `weak` это указатель `CvBoost` последовательности пронумерованных слабых классификаторов типа `CvBoostTree`). `LogitBoost` и `GentleBoost` являются деревьями регрессии (деревьями, которые предсказывают значения типа `float`); *decision trees* для других методов возвращают только класс: 0 (для положительного значения) или 1 (для негативного значения). Класс содержащий эту последовательность имеет следующий прототип:

```

class CvBoostTree: public CvDTree {
public:
    CvBoostTree();

    virtual ~CvBoostTree();
    virtual bool train(
        CvDTreeTrainData* _train_data
        , const CvMat*      subsample_idx
        , CvBoost*          ensemble
    );
    virtual void scale( double s );
    virtual void read(
        CvFileStorage*     fs
        , CvTreeNode*       node
        , CvBoost*          ensemble
        , CvDTreeTrainData* _data
    );
    virtual void clear();

protected:
    ...
    CvBoost* ensemble;
};

```

Процесс обучения почти такой же, как и в случае с decision trees, за исключением параметра *update*, который по умолчанию имеет значение false (0). Если используется значение параметра по умолчанию, то в процессе обучения новой ансамбль слабых классификаторов обнуляется. Если параметр *update* установлен в true (1), то новый слабый классификатор просто добавляется в существующую группу. Прототип функции обучения классификатора boosting имеет следующий вид:

```

bool CvBoost::train(
    const CvMat*      _train_data
    , int             _tflag
    , const CvMat*    _responses
    , const CvMat*    _var_idx      = 0
    , const CvMat*    _sample_idx   = 0
    , const CvMat*    _var_type     = 0
    , const CvMat*    _missing_mask = 0
    , CvBoostParams&  params       = CvBoostParams()
    , bool            update       = false
);

```

Пример обучения классификатора boosting можно найти в [.../opencv/samples/c/letter_recog.cpp](#). Фрагмент обучения показан в примере 13-3.

Пример 13-3. Фрагмент обучения классификаторов boosting

```

var_type = cvCreateMat( var_count + 2, 1, CV_8U );

cvSet( var_type, cvScalarAll(CV_VAR_ORDERED) );

// the last indicator variable, as well
// as the new (binary) response are categorical
//
cvSetReal1D( var_type, var_count, CV_VAR_CATEGORICAL );
cvSetReal1D( var_type, var_count+1, CV_VAR_CATEGORICAL );

// Обучение классификатора
//
boost.train(
    new_data
    ,CV_ROW_SAMPLE
    ,responses
    ,0
    ,0
    ,var_type
    ,0
    ,CvBoostParams( CvBoost::REAL, 100, 0.95, 5, false, 0 )
);

cvReleaseMat( &new_data );
cvReleaseMat( &new_responses );

```

Функция предсказания для boosting схожа с функцией предсказания для decision trees:

```

float CvBoost::predict(
    const CvMat*    sample
    ,const CvMat*    missing      = 0
    ,CvMat*          weak_responses = 0
    ,CvSlice         slice        = CV_WHOLE_SEQ
    ,bool            raw_mode     = false
) const;

```

Для выполнения простого прогнозирования, достаточно просто передать вектор особенностей *samples* и функция вернет предсказание. Однако, существуют и множество дополнительных параметров. Первый необязательный параметр *missing* это маска особенностей, точно такая же, как и в случае с decision trees; он состоит из вектора байт той же размерности, что и *samples*, где ненулевые значения указывают на отсутствующие особенности. (Маска не может быть использована, если классификатор обучался с параметром *use_surrogates*.)

Если необходимо получить ответы от каждого слабого классификатора, то необходимо использовать параметр *weak_responses*, длина которого равна числу слабых классификаторов. В процессе выполнения функция *CvBoost::predict* заполнит

weak_responses ответами от каждого слабого классификатора следующим образом:

```
CvMat* weak_responses = cvCreateMat(
    1
    , boostedClassifier.get_weak_predictors()->total
    , CV_32F
);
```

Следующий необязательный параметр *slice* указывает какое смежное подмножество слабых классификаторов использовать; параметр может быть задан следующим образом:

```
inline CvSlice cvSlice( int start, int end );
```

При этом, как правило, используется значение по умолчанию (CV_WHOLE_SEQ). Последний необязательный параметр *row_mode* имеет значение по умолчанию false. Этот параметр точно такой же, как и для decision trees и указывает на то, что данные предварительно нормализованы для экономии времени вычислений. Как правило, данный параметр не должен быть использован. Пример вызова функции предсказания для boosting:

```
boost.predict( temp_sample, 0, weak_responses );
```

Кроме всего ранее представленного, имеется ещё набор вспомогательных функций, которые могут быть использованы время от времени. Удалить слабый классификатор из обучаемой модели можно при помощи:

```
void CvBoost::prune( CvSlice slice );
```

Получить все слабые классификаторы можно при помощи:

```
CvSeq* CvBoost::get_weak_predictors();
```

[П]|[РС]|(РП) Random trees

OpenCV содержит класс *random trees*, который реализует теорию *random forests* Leo Breiman (Большая часть работы Breiman о random trees собрана на одном [сайте](#)). Random trees могут обучать более одного класса в единицу времени просто собирая класс "votes" на листьях каждого из множества деревьев и выбирая класс с максимум "votes". Регрессия получается за счет усреднения значений листьев "леса". Random trees состоят из *randomly perturbed decision trees* и являются одними из наиболее эффективных классификаторов на момент сборки библиотеки ML. Random trees так же обладают потенциалом к параллельной реализации, даже на системах с неразделяемой памятью, что оставляет запас для их более широкого использования в будущем. Основу random trees составляют вновь decision trees. Построение этих decision trees происходит до тех пор, пока не станет чисто. Таким образом (см. верхнюю правую часть рисунка 13-2), каждое дерево является высоко-дисперсионным классификатором, который почти идеально обучается на наборе для обучения. В противовес к высокой дисперсии, множество таких деревьев усредняются (отсюда и такое название random trees).

Конечно, усреднение деревьев не даёт никакой пользы, если все деревья сильно схожи между собой. Для того, чтобы это исправить random trees случайным образом выбирают подмножество особенностей из общего множества особенностей, на основе которого дерево в дальнейшем обучает каждый узел. Например, объект, который необходимо распознать, может иметь длинный список потенциальных особенностей: цвет, текстура, величина градиента, направление градиента, дисперсия, значения соотношений и т.д. Каждому узлу дерева разрешено случайным образом выбрать подмножество таких особенностей для определения как лучше всего разделить данные; в дальнейшем, все последующие узлы дерева получают новое, случайным образом выбранное, подмножество особенностей для дальнейшего разделения. Зачастую размер случайным образом выбранных подмножеств выбирается как квадратный корень из числа особенностей. Так, если имеется 100 потенциальных особенностей, то каждый узел будет случайным образом выбирать 10 особенностей и искать наилучший разделитель данных из числа выбранных 10 особенностей. Для повышения надежности random trees используют меру *out of bag* для подтверждения разделения. То есть любой выбранный узел, обучение которого происходит при использовании нового подмножества данных, выбираемое случайным образом с заменой (это означает, что некоторые случайно выбранные наблюдения могут повторяться), и неиспользуемые данные (не случайно выбранные значения,

именуемые "out of bag" (или ОOB) используются для оценки производительности разделения. ОOB, как правило, устанавливается равным одной трети всех наблюдений.

Как и все методы, основанные на деревьях, random trees наследуют множество полезных свойств деревьев: суррогатные разделители для отсутствующих значений, обработка категориальных и численных значений, отсутствие необходимости в нормализации значений и наличие простых методов для поиска переменных, которые необходимы для выполнения предсказания. Random trees так же используют результаты ошибок ОOB для оценки того, насколько хорошо будут обработаны отсутствующие данные. Если распределения обучаемых данных и тестовых данных совпадают, то даваемое ОOB предсказание может быть достаточно точным.

И наконец, random trees могут быть использованы для определения *близости* (что в данном контексте означает "как одинаковы", а не "как близки") любых двух наблюдений. Алгоритм выполнения данной операции выглядит следующим образом: (1) "сбрасывание" наблюдений в деревья; (2) подсчет количества попаданий на один лист; (3) разделение подсчитанного значения на общее число деревьев. Если результат близости равен 1, то схожи, если 0, то совершенно различны. Данная мера близости может быть использована для идентификации выбросов (данные точки совершенно не похожи на остальные), а также сгруппированных точек (группа близких точек).

Реализация random tree в OpenCV

На данный момент уже должно сложиться некое представление о том, как работает библиотека ML и в частности random trees. Для начала будет рассмотрена структура *CvRTParams*, которая наследуется от *decision trees*:

```

struct CvRTParams : public CvDTreeParams {
    bool          calc_var_importance;
    int           nactive_vars;
    CvTermCriteria term_crit;

    CvRTParams() : CvDTreeParams(
        5, 10, 0, false,
        10, 0, false, false,
        0
    ), calc_var_importance(false)
    , nactive_vars(0) {
        term_crit = cvTermCriteria(
            CV_TERMCRIT_ITER | CV_TERMCRIT_EPS
            , 50
            , 0.1
        );
    }

    CvRTParams(
        int           _max_depth
        , int          _min_sample_count
        , float         _regression_accuracy
        , bool          _use_surrogates
        , int           _max_categories
        , const float* _priors
        , bool          _calc_var_importance
        , int           _nactive_vars
        , int           max_tree_count
        , float         forest_accuracy
        , int           termcrit_type
    );
};


```

Новый ключевой параметр в *CvRTParams calc_var_importance* - это просто переключатель вычислений важной перемененной каждой особенности в момент обучения. На рисунке 13-13 показана важная переменная, вычисленная на основе подмножества грибов, которое поставляется вместе с OpenCV и располагается в *.../opencv/samples/c/agaricus-lepiota.data*. Параметр *nactive_vars* задает размер случайно выбранного подмножества особенностей, которое будет тестироваться в любом выбранном узле и, как правило, имеет значение, равное квадратному корню от общего числа особенностей; параметр *term_crit* управляет максимальным числом деревьев. Для обучения random trees в *term_crit* параметр *max_iter* устанавливает общее число деревьев; *epsilon* задает критерий прекращения обучения для приостановки процесса добавления новых деревьев при достижении значения ошибки ниже значения ошибки ОOB; *type* сообщает какой из двух возможных критериев останова использовать (как правило, используются оба: *CV_TERMCRIT_ITER* | *CV_TERMCRIT_EPS*).

Random trees обучаются точно так же, как и decision trees, за исключением только того, что в рассматриваемом случае используется структура CvRTParam:

```
bool CvRTrees::train(
    const CvMat* train_data
    , int tflag
    , const CvMat* responses
    , const CvMat* comp_idx = 0
    , const CvMat* sample_idx = 0
    , const CvMat* var_type = 0
    , const CvMat* missing_mask = 0
    , CvRTParams params = CvRTParams()
);
```

Variable Name	RandomTrees	Boosting	DecisionTree
Col6	100.00	100.00	100.00
Col20	35.20	58.89	57.37
Col21	18.47	6.11	34.51
Col19	13.35	4.57	26.11
Col9	13.01	43.15	45.96
Col13	10.02	24.47	26.85
Col8	9.52	37.51	42.28
Col12	9.09	27.88	28.90
Col22	8.29	0.28	20.00
Col7	6.08	0.10	21.33
Col15	4.06	1.84	21.41
Col11	3.62	0.44	16.29
Col4	3.12		14.67
Col14	2.98	0.25	20.81
Col18	2.68		0.70
Col3	2.58	0.11	9.15
Col2	2.22	0.39	12.14
Col10	1.79		2.67
Col1	0.41	0.24	7.26
Col17	0.18	0.32	0.54
Col0			
Col6			
Col16			

Рисунок 13-13. Важная переменная из набора данных грибов для random trees, boosting и decision trees: в случае с random trees так же задействованы менее значимые переменные, за счет чего достигнуто лучшее предсказание (100% верных ответов на случайно выбранном тестовом наборе, охватывающий 20% данных)

Пример использования функции обучения для случая с несколькими классами поставляется вместе с OpenCV и располагается в [.../opencv/samples/c/letter_recog.cpp](#), где классификатор random trees именуется *forest*:

```

forest.train(
    data
    , CV_ROW_SAMPLE
    , responses
    , 0
    , sample_idx
    , var_type
    , 0
    , CvRTParams(10, 10, 0, false, 15, 0, true, 4, 100, 0.01f, CV_TERMCRIT_ITER)
);

```

Функция предсказания для random trees схожа с функцией предсказания для decision trees, но вместо возврата указателя *CvDTreeNode*, рассматриваемая функция возвращает среднее возвращаемых значений всех деревьев из леса. Мaska *missing* является необязательным параметром той же размерности, что и вектор *sample*, где ненулевые значения указывают на отсутствующие значения в *sample*.

```

double CvRTrees::predict(
    const CvMat*   sample
    ,const CvMat*   missing = 0
) const;

```

Пример использования функции предсказания из файла *letter_recog.cpp*:

```

double r;
CvMat sample;

cvGetRow( data, &sample, i );

r = forest.predict( &sample );
r = fabs((double)r - responses->data.fl[i]) <= FLT_EPSILON ? 1 : 0;

```

В представленном куске кода происходит преобразование переменной *r* в число верных предсказаний.

И наконец, для random trees имеются функции для анализа и вспомогательные функции. Если, например, переменная *CvRTParams::calc_var_importance* установлена для обучения, то можно получить относительную важность каждой переменной следующим образом:

```

const CvMat* CvRTrees::get_var_importance() const;

```

Пример важной переменной набора данных грибов для random trees показан на рисунке 13-13. Так же можно получить меру близости одного наблюдения относительно другого обученной модели random trees следующим образом:

```
float CvRTrees::get_proximity(
    const CvMat* sample_1
    , const CvMat* sample_2
) const;
```

Как уже было сказано ранее, если возвращаемое значение равно 1, то наблюдения полностью идентичны, если возвращаемое значение равно 0, то совершенно различны. Как правило, это значение находится между 0 и 1 для двух наблюдений, взятых из распределения, которое аналогично набору для обучению.

Имеется ещё две полезные функции, дающие общее количество деревьев и структуру данных, содержащая данное decision trees:

```
int get_tree_count() const;           // Кол-во деревьев в лесу
CvForestTree* get_tree(int i) const; // Получение конкретного decision tree
```

Использование random trees

Как уже было сказано ранее, алгоритм random trees зачастую работает лучше всего (или как минимум входит в число лучших) на наборах для тестирования, но все же лучше всего их применять для объединения множества классификаторов в один, имея определенный набор для обучения. Все ранее представленные алгоритмы random trees, boosting и decision trees рассматривались на примере набора данных грибов. Из 8124 наблюдений случайным образом извлекалось 1624 тестовых наблюдений, а остальные использовались как обучающее множество. После обучения этих классификаторов, основанных на деревьях, были получены результаты, показанные в таблице 13-4, при использовании набора для тестирования. Набор данных о грибах довольно таки прост и потому нельзя однозначно сказать, какой из трех классификаторов лучше всего будет работать с данным набором данных (хотя random trees и показали наилучший результат).

Таблица 13-4. Результаты методов, основанных на деревьях, при использовании набора данных грибов (1624 случайно выбранных наблюдений без каких-либо дополнительных штрафов для неверной оценки ядовитости грибов)

Классификатор	Результаты производительности
Random trees	100%
AdaBoost	99%
Decision trees	98%

Наиболее интересной является важная переменная (которая к тому же измеряется на основе классификатора), показанная на рисунке 13-13. На рисунке показано, что random trees и boosting используют важную переменную значительно меньше, чем используют её decision trees. Значение свыше 15%, имеют в случае с random trees только три переменные, а в случае с boosting шесть, в то время, как в случае с decision trees тридцать. Таким образом можно сократить размер набора особенностей для экономии времени вычислений и памяти, не теряя возможности получать хорошие результаты. Алгоритма decision trees может обрабатывать только одно дерево, в то время как random trees и AdaBoost могут оценить сразу несколько деревьев; таким образом, какой метод затрачивает меньше всего времени на вычисления зависит от используемого набора данных.

[П] | [РС] | (РП) Распознавание лиц или классификатор Хаара

Данный раздел будет посвящен *классификатору Хаара*, который построен на *boosted rejection cascade*. Формат данного классификатора отличается от формата остальной части библиотеки ML, т.к. данный классификатор был написан до её появления как полноценный, готовый к применению распознаватель лиц. Поэтому данный классификатор будет рассмотрен отдельно; будет показано, как его можно обучить для распознавания лиц и других твердых объектов.

Компьютерное зрение - это обширная и быстро меняющаяся область, поэтому OpenCV, реализующая определенные методы, а не компоненты алгоритмической части, подверженная риску устаревать. Детектор лиц, который поставляется вместе с OpenCV, соответственно тоже в категории "риска". Тем не менее, детектор лиц находит более обширное применение за счёт наличия техники базовой линии, которая работает достаточно хорошо; к тому же эта техника основана на известной и часто используемой области *statistical boosting*, что так же указывает на более обширное применение данного детектора. По факту, проектированием детектора "лица" в OpenCV занимаются несколько компаний для обнаружения по большей части "твердых" объектов (лица, автомобилей, мотоциклов, тел человека) за счёт обучения новых детекторов на наборе для обучения, состоящего из множества тысяч изображений представлений каждого рассматриваемого объекта. Данная техника была использована для создания state-of-the-art детекторов, даже не смотря на то, что в обучении детектора используются различные представления или положения каждого рассматриваемого объекта. Таким образом, классификатор Хаара является ценным инструментом в решении подобного рода задач распознавания.

В OpenCV реализован вариант детектора лица, впервые разработанный Paul Viola и Michael Jones (более известный как *детектор Viola-Jones*), а в последующем дополненный Rainer Lienhart и Jochen Maydt, использующий *диагональные особенности* (чуть позже об этом будет рассказано более подробно). OpenCV ссылается на этот детектор как на "классификатор Haara", т.к. используются особенности Haara (Технически это не корректно. Классификатор использует суммарный порог и различные прямоугольные области данных, полученные от любого детектора особенностей, которые может включать Haar в случае прямоугольников с необработанными значениями (серого) изображения. В дальнейшем будет использоваться термин "Haar" для подчеркивания этого различия), или, если быть более точным, то *вейвлеты Haara*, которые формируются за счет сложения и вычитания прямоугольных областей изображения перед выполнением порогового

преобразования над результатом. Дистрибутив OpenCV включает в себя набор файлов с распознанными объектами, но вместе с тем также имеется возможность обучать и сохранять новые модели объекта для данного детектора. Итак, код для обучения (`createsamples()`, `haartraining()`) и распознавания (`cvHaarDetectObjects()`) может работать с любыми объектами (не только с лицами), которые текстурированы и в основном твёрдые.

Изученные объекты, поставляемые вместе с OpenCV, располагаются в `.../opencv/data/haarcascades`, где модель наилучшего распознавания фронтально расположенного лица располагается в файле `haarcascade_frontalface_alt2.xml`. Иные расположения лица распознать при помощи данной техники уже труднее. Если Вам удастся получить хорошо обученную модель объекта, то, возможно, Вы захотите поделиться ей с миром! Если модель будет действительно выдающейся, то она может быть включена в следующую версию библиотеки OpenCV!

Контролируемое обучение и теория Boosting

Классификатор Haar, который включен в OpenCV, является контролируемым классификатором. Это означает, что имеются histogram- и size- скорректируемые патчи изображения для классификатора, помечающиеся в дальнейшем как содержащие (или не содержащие) интересующие объекты, которые для данного классификатора чаще всего являются лицами.

Детектор Viola-Jones это разновидность AdaBoost, только организован как *rejection cascade* узлов, где каждый узел - это много древовидный классификатор AdaBoost, обладающий высоким (99,9%) уровнем распознавания (низкие ложные отрицательные срабатывания или отсутствующие лица) за счёт низкого (около 50%) процента брака (высокие ложные положительные срабатывания или ошибочно классифицированные "не лица"). Для каждого узла, если результат не соответствует "классу", то алгоритм прекращает своё выполнение на любой из стадий каскада и сообщает о том, что в заявлении месте лицо не было найдено. Таким образом, алгоритм вернет верно распознанный класс только тогда, когда в процессе вычислений будет совершен обход всего каскада. В случаях, когда верно распознанный класс редкость (например, лицо на изображение), rejection cascades общее время вычислений может значительно сократиться, т.к. большая часть регионов, в которых ищется лицо, будут отброшены.

Boosting в Haar cascade

Ранее в этой главе уже были рассмотрены boost классификаторы. Для rejection cascade Viola-Jones слабые классификаторы, которые boosts в каждом узле, являются decision trees, которые зачастую имеют один уровень вложенности (т.е. "decision

stumps"). Decision stumps предлагают только одно решение следующего вида: "Узнается выше или ниже значение v конкретной особенности f некоторого порогового значения t "; например, если "да", то лицо найдено, а если "нет", то лицо не найдено":

$$f_i = \begin{cases} +1 & v_i \geq t_i \\ -1 & v_i < t_i \end{cases}$$

Число особенностей Haar, которое классификатор Viola-Jones использует в каждом слабом классификаторе, может быть установлено в процессе обучения, но в большинстве случаев используется единственная особенность (т.е. дерево с единственным разделением) или максимум три. Затем boosting итеративно выстраивает классификатор как взвешенную сумму слабых классификаторов. Классификатор Viola-Jones использует следующую функцию классификации:

$$F = \text{sign}(w_1 f_1 + w_2 f_2 + \dots + w_n f_n)$$

Функция sign вернет -1, если number < 0; 0, если number = 0; +1, если number > 0. При первом проходе по данному набору данных, вычисляются t_i и f_i , которые наилучшим образом классифицируют исходные данные. Затем boosting использует полученные ошибки для вычисления взвешенного vote w_i . Как и в случае стандартного AdaBoost, для каждой особенности вектора (наблюдения) пересчитывается вес в большую или меньшую сторону, в зависимости от корректности классификации на данной итерации (Иногда возникает путаница с boosting: уменьшение веса классификации свидетельствует о корректной классификации, в то время, как увеличение веса свидетельствует о некорректной классификации. Причина в том, что попытка сосредоточиться на тех моментах, в которых есть "проблемы", приводит к игнорированию наблюдений, которые "известно" как классифицировать.). После того, как путь будет вычислен, оставшиеся данные передаются вверх по каскаду для обучения следующего узла и т.д.

Классификатор Viola-Jones. Теория

Классификатор Viola-Jones использует AdaBoost в каждом узле каскада для обучения классификатора с высоким уровнем распознавания при низком multitree (в основном multistump) проценте брака в каждом узле каскада. Данный алгоритм включает в себя несколько инновационных особенностей.

1. Использует исходные особенности Haar: порог применяется для суммирования и чтобы различать прямоугольные области изображения.
2. Использует технику *встраивания изображений*, которая позволяет ускорить процесс вычисления значения прямоугольных областей или разворачивать такие области на 45 градусов (глава 6). Данная структура данных используется для

ускорения процесса вычисления исходных особенностей Haar.

3. Использует статистический boosting для создания двоичной (лицо - не лицо) классификации узлов, которая характеризуется высоким уровнем распознавания и малым процентом брака.
4. Организует слабые классификаторы узлов rejection cascade. Другими словами: первая группа классификаторов выбирается таким образом, чтобы наилучшим образом (в силу возможного) распознать области изображения, содержащие объект, допуская при этом множество ошибок распознавания; следующая выбираемая группа (стоит не забывать, что каждый "узел" в rejection cascade является группой классификаторов AdaBoost) классификаторов является следующей лучшей группой с малым процентом брака; и т.д. Объект будет считаться распознанным только в том случае, если будет произведен обход всего каскада (это позволяет ускорить процесс использования каскада, т.к. он почти сразу отвергает области изображения, не содержащие объект; тем самым потребность в обработке остальной части каскада не возникает).

Особенности Haar используются классификатором, который показан на рисунке 13-14. При любом масштабе, эти особенности образуют "сырьё", которое будет использовано boost классификаторами. Они быстро вычисляются по встроенным изображениям (глава 6), представляющие оригинальные изображения в оттенках серого.

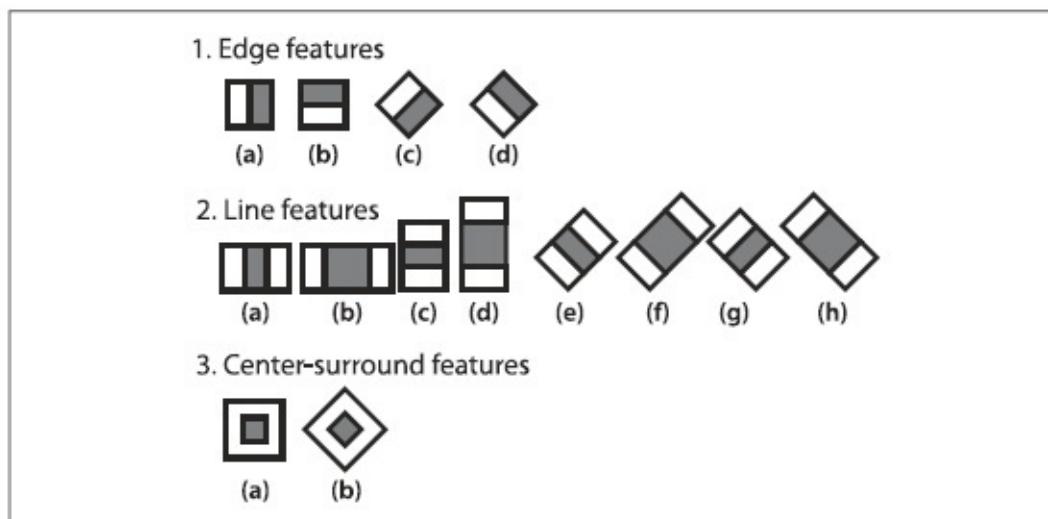


Рисунок 13-14. Особенности Haar из исходного дистрибутива OpenCV (прямоугольные и повернутые регионы легко вычисляются по встроенному изображению): в данных схематично представленных вейвлетах светлая область интерпретируется как "добавить эту область", а темная область как "вычесть эту область"

Viola и Jones организовали каждую группу boost классификаторов в узлах rejection cascade, как показано на рисунке 13-15. На рисунке каждый узел F_j содержит целый boost каскад сгруппированных decision stumps (или trees), обученных на особенностях

Haar, лиц и не лиц (или других объектов, в зависимости от выбора пользователя). Как правило, узлы сортируются от наименее к наиболее комплексному так, чтобы минимизировать процесс вычислений (простые узлы обрабатываются в первую очередь), отбрасывая сначала простые области изображения. Как правило, boosting в каждом узле настраивается на очень высокий уровень обнаружения (при обычной стоимости должно положительных срабатываниях). В случае с лицами, например, распознаются почти все (99.9%) лица, но многие (около 50%) не лица ошибочно "распознаются" в каждом узле. Но это нормально, т.к. при использовании (например) 20 узлов, процент распознавания лица (при обходе всего каскада) по прежнему будет $0.999^{20} \approx 98\%$ при $0.5^{20} \approx 0.0001\%$ должно положительных срабатываниях!

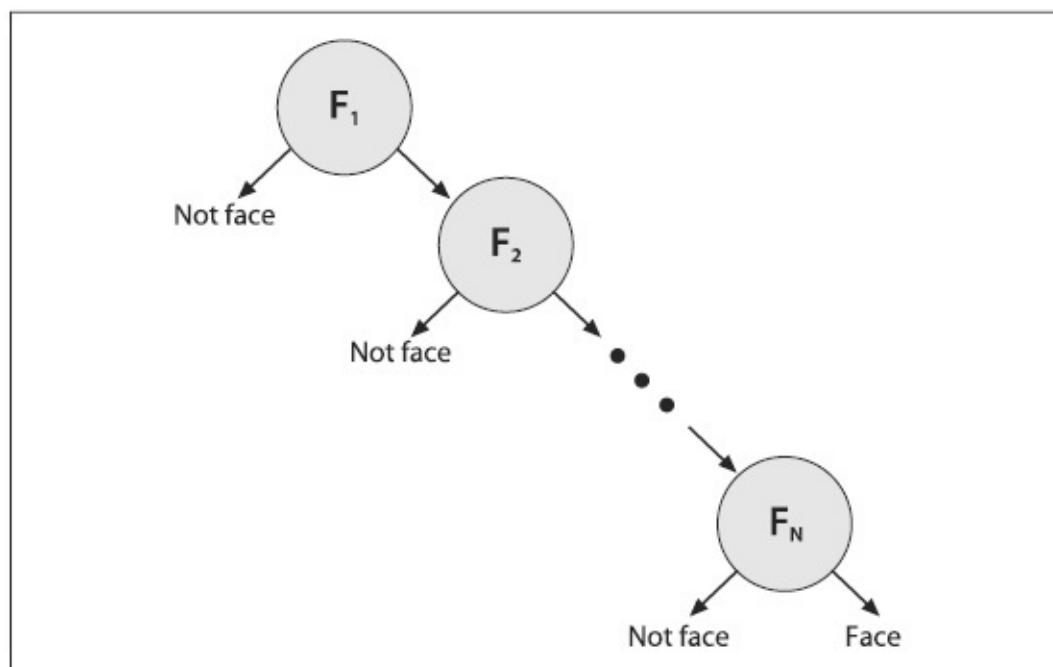


Рисунок 13-15. Использование rejection cascade в классификаторе Viola-Jones: каждый узел представляет из себя ансамбль multitree boost классификатор, настроенный таким образом, чтобы почти никогда не пропускать лица при малом проценте брака; при этом, все что является не лицом отбрасывается в последнем узле, как результат - остаются только лица

В рабочем режиме при поиске областей различных размеров охватывается всё изображение. На практике, 70-80% не лиц отбрасываются первыми двумя узлами rejection cascade, где каждый узел использует около 10 decision stumps. За счет "раннего отсечения" процесс распознавания лиц значительно ускоряется.

Хорошо работает на ...

Представленная техника специализируется на распознавании лиц, однако, существует возможность использования её и для других целей; данная техника так же показывает хорошие результаты и на других (в основном твердых) объектах, которые имеют

характерные представления. То есть, если попытаться распознать фронтальное лицо; машину сзади, сбоку или спереди, то детектор даст хорошие результаты; но если попытаться распознать лицо или машину "под углом", то результаты будут не очень хорошими - это связано с тем, что подобного рода представления вносят изменения в шаблон, состоящий из "блоков" особенностей, используемые данным детектором, который в свою очередь не может справиться с такими изменениями хорошо.

Например, в процессе обучения модели представления лица сбоку необходимо охватить часть меняющегося фона для определения кривой профиля. Для распознавания представлений лиц сбоку, можно попробовать использовать *haarcascade_profileface.xml*, но для получения действительно хорошего результата необходимо гораздо больше данных, чем может дать данный файл и, возможно, так же потребуется расширить данные за счёт добавления представлений с различными вариантами фона. Опять же, представления профиля трудно даются данному классификатору, т.к. используются блоки особенностей из-за чего детектор пытается изучить меняющийся фон для "считывания" информации о крае лица. В процессе обучения, эффективней всего узнавать только (скажем) верно ли определился вид профиля. Затем запустить функцию тестирования (1) для распознавания правой стороны профиля, а затем (2) перевернуть изображение по вертикальной оси и вновь запустить детектор правой стороны для распознавания левой стороны профиля.

Как уже было сказано, детекторы, основанные на особенностях Haar, хорошо работают с "блоками" особенностей - такими, как глаза, рот и волосы - и не очень хорошо с ветвями дерева или, когда очертание объекта является главной отличительной характеристикой (как в случае с кружкой кофе).

Если необходимо собрать много хорошо сегментированных данных с действительно твёрдых объектов, то данный классификатор входит в число лучших для выполнения данной задачи, т.к. он основан на *rejection cascade*, что позволяет ему быстро работать в рабочем режиме (но не в режиме обучения). Фраза "много данных" означает тысячи примеров объекта и десятки тысяч примеров не объектов. Понятие "хорошо" означает, что не следует смешивать, например, лица под наклоном с вертикально расположенными лицами; вместо этого поддерживать разделение данных и использовать два классификатора, один для лиц под наклоном и ещё один для вертикально расположенных лиц. Понятие "сегментированные" данные означает что данные систематизированы в *box* (прямоугольная область на изображении).

Отсутствие должного внимания к границам *boxes* обучаемых данных зачастую приводит к корректировки фиктивной изменчивости данных классификатора.

Например, различие в расположениях мест глаз на лице может привести к тому, что классификатор сочтет расположения глаз геометрически не фиксированной

особенностью и потому будет перемещаться. Производительность почти всегда хуже, когда классификатор пытается приспособиться к вещам, которых на самом деле нет в реальном наборе данных.

Распознавание лиц. Практика

Код представленный в примере 13-4 распознает лица и рисует на изображении места расположения найденных лиц различными цветными прямоугольниками. Как показано в четвертой и шестой строке (комментарии), данный код предполагает, что заранее подготовленный каскад классификатора уже загружен и память под распознаваемые лица уже выделена.

Пример 13-4. Код по распознаванию лиц с отображением результатов

```
// Detect and draw detected object boxes on image
// Presumes 2 Globals:
// Cascade is loaded by:
// cascade = (CvHaarClassifierCascade*)cvLoad( cascade_name, 0, 0, 0 );
// AND that storage is allocated:
// CvMemStorage* storage = cvCreateMemStorage(0);
//
void detect_and_draw( IplImage* img, Double scale = 1.3 ) {
    // Unknown error build gitbook
    // Заменить '(' на две '{' и ')' на две '}'
    // и раскомментировать строки 85-88
    //
    // static CvScalar colors[] = {
    //     (0,0,255), (0,128,255), (0,255,255), (0,255,0),
    //     (255,128,0), (255,255,0), (255,0,0), (255,0,255)
    // };

    // IMAGE PREPARATION:
    //
    IplImage* gray = cvCreateImage( cvSize(img->width,img->height), 8, 1 );
    IplImage* small_img = cvCreateImage(
        cvSize( cvRound(img->width/scale)
        ,cvRound(img->height/scale))
        ,8
        ,1
    );

    cvCvtColor( img, gray, CV_BGR2GRAY );
    cvResize( gray, small_img, CV_INTER_LINEAR );
    cvEqualizeHist( small_img, small_img );

    // DETECT OBJECTS IF ANY
    //
    cvClearMemStorage( storage );
    CvSeq* objects = cvHaarDetectObjects(
```

```

    small_img
    ,cascade
    ,storage
    ,1.1
    ,2
    ,0 /*CV_HAAR_DO_CANNY_PRUNING*/
    ,cvSize(30, 30)
);

// LOOP THROUGH FOUND OBJECTS AND DRAW BOXES AROUND THEM
//
for(int i = 0; i < (objects ? objects->total : 0); i++ ) {
    CvRect* r = (CvRect*)cvGetSeqElem( objects, i );
    cvRectangle(
        img
        ,cvPoint(r.x,r.y)
        ,cvPoint(r.x+r.width,r.y+r.height)
        ,colors[i%8]
    )
}

cvReleaseImage( &graygray );
cvReleaseImage( &small_img );
}

```

Для удобства представленная функция *detect_and_draw()* имеет статичный массив векторов цвета *colors[]*, который задает цветовое выделение для найденных лиц. Классификатор работает с изображением в оттенках серого, поэтому исходное BGR изображение преобразуется при помощи функции *cvCvtColor()*, а затем (необязательно) уменьшается при помощи *cvResize()*. Так же полученное изображение подвергается выравниванию гистограммы при помощи *cvEqualizeHist()*, которая "размазывает" значения яркости - это выполняется из-за того, что особенности встроенного изображения основаны на различных прямоугольных областях и если гистограмма не будет сбалансирована, то различия между областями могут быть искажены общей освещенностью. Классификатор возвращает результат в виде прямоугольников как последовательность типа *CvSeq*, при этом необходимо воспользоваться функцией *cvClearMemStorage()* для очистки используемого глобального хранилища. Фактически распознавание происходит в цикле *for{}*. В рамках одной итерации происходит поиск прямоугольной области, содержащей лицо, и нанесение результатов на конечное изображение индивидуальным цветом при помощи функции *cvRectangle()*. Функция распознавания используется следующим образом:

```
CvSeq* cvHaarDetectObjects(
    const CvArr*           image
    ,CvHaarClassifierCascade* cascade
    ,CvMemStorage*          storage
    ,double                 scale_factor     = 1.1
    ,int                   min_neighbors   = 3
    ,int                   flags            = 0
    ,CvSize                min_size         = cvSize(0, 0)
);
```

CvArr image это изображение в оттенках серого. Если задан регион интересов (ROI), то функция будет работать только с этим регионом. Таким образом использование региона интереса это один из способов ускорения процесса распознавания лиц.

Каскад классификатора это каскад Haar особенностей, который загружается при помощи функции *cvLoad()*. Аргумент *storage* в рамках OpenCV это "рабочий буфер" для данного алгоритма; выделение нового участка памяти производиться при помощи *cvCreateMemStorage(0)*, а освобождение при помощи *cvClearMemStorage(storage)*.

Функция *cvHaarDetectObjects()* сканирует исходное изображение лица всех масштабов. Параметр *scale_factor* задает шаг перемещения между масштабами; чем выше значение, тем быстрее будет процесс вычисления за счёт пропуска возможно распознанных лиц на пропущенных изображениях определенных масштабов.

Параметр *min_neighbors* контролирует процесс предотвращения ложных распознаваний. Реальное расположение лица на изображении, как правило, имеет несколько "hits" в близлежащей области, т.к. соседние пиксели зачастую отчасти указывают на лицо. Установка данного параметра в значение по умолчанию (3) означает, что решение о присутствии лица в определенном месте будет принято только в том случае, если есть не менее трех дублирующих друг друга успешных распознаваний. Параметр *flags* может принимать одно из четырех возможных значений, которые (как обычно) могут быть объединены при помощи *OR*. Значение *CV_HAAR_DO_CANNY_PRUNING* сообщает классификатору о необходимости пропускать ровные регионы (без линий). Значение *CV_HAAR_SCALE_IMAGE* сообщает алгоритму об изменении масштаба изображения, а не детектора (это может повысить производительность с точки зрения использования памяти и кеша). Значение *CV_HAAR_FIND_BIGGEST_OBJECT* сообщает OpenCV о необходимости возвращать только самый большой из найденных объектов (соответственно, число возвращаемых объектов будет равно либо единице, либо нулю). (При этом лучше не использовать *CV_HAAR_DO_CANNY_PRUNING* вместе с *CV_HAAR_FIND_BIGGEST_OBJECT*, т.к. вместе они очень редко дают прирост производительности; зачастую наблюдается диаметрально противоположный эффект). Значение *CV_HAAR_DO_ROUGH_SEARCH* используется только в совокупности с *CV_HAAR_FIND_BIGGEST_OBJECT*. Данное значение флага используется для прекращения поиска при обнаружении первого

кандидата независимо от масштаба (при достаточном количестве соседей, чтобы считать их "hit"). Последний параметр *min_size* задаёт минимально возможный размер региона, в котором можно искать лицо. Установка данного параметра в относительно большое значение уменьшает время вычислений за счет отбрасывания лиц малых размеров. На рисунке 13-16 показаны результаты использования представленного примера с задействованием сцены, на которой присутствуют лица.



Рисунок 13-16. Результаты из примера 13-4: не удалось распознать некоторые лица под наклоном, а так же есть ложные положительные срабатывания (кофта на мужчине в очках); данный результат был получен за 1,5 секунды на 2 GHz машине при использовании базы, состоящей из более миллиона изображений размера 1054x851

Обучение новых объектов

Ранее уже было показано как загрузить и запустить заранее обученный каскад классификатора, хранящийся в файле XML. Для загрузки использовалась функция *cvLoad()* с последующим применением *cvHaarDetectObjects()* для поиска объектов, схожих с теми, на которых выполнялось обучение. Теперь будет рассмотрен процесс обучения собственных классификаторов для обнаружения других объектов таких, как глаза, перемещающиеся люди, машины и т.д. В данном разделе будет показано как это сделать при помощи *haartraining*, который создает классификатор, производящий

набор для обучения, состоящий из положительных и отрицательных образцов. Весь процесс состоит из четырех шагов (более подробную информацию о *haartraining* можно найти в директории *opencv/apps/HaarTraining/doc*):

1) Собрать набор данных, состоящий из примеров объекта, который необходимо изучить (например, фронтальные представления лиц, представления машины сбоку). Собранные данные могут быть сохранены в одном или нескольких директориях, состоящих из индексных файлов следующего формата:

```
<path>/img_name_1 count_1 x11 y11 w11 h11 x12 y12 ...
<path>/img_name_2 count_2 x21 y21 w21 h21 x22 y22 ...
...
```

Каждая строка содержит путь (если есть) и имя файла изображения, содержащего объект(-ы). Далее указывается число объектов на данном изображении и список прямоугольников, содержащих объекты. Формат прямоугольников следующий: координаты (x,y) верхнего левого угла и ширина с высотой в пикселях.

Например, имея набор данных из расположений лиц в директории *data/faces*, файл индекса *faces.idx* мог бы выглядеть следующим образом:

```
data/faces/face_000.jpg 2 73 100 25 37 133 123 30 45
data/faces/face_001.jpg 1 155 200 55 78
...
```

Для того, чтобы классификатор работал хорошо, необходимо собрать большой объём высококачественных данных (1 000 – 10 000 положительных примеров). Понятие "высококачественные" данные означает, что из рассмотрения исключаются все ненужные отклонения от данных. Например, при изучении лиц, необходимо выровнять глаза (желательно нос и рот тоже) насколько это возможно. В противном случае классификатор будет обучен таким образом, что будет искать глаза в любом месте в пределах заданной области, а не в фиксированном месте лица. Однин из возможных вариантов преодоления данной проблемы заключается в следующем: в начале необходимо обучить каскад по частям, например, на "глазах", которые легче выровнять. Затем, используя детектор глаза, найти глаза и вращать/изменять размер лица до тех пор, пока глаза не будут выровнены. Для ассиметричных данных, "фишка" с вращением изображения вдоль вертикальной оси уже была описана ранее в разделе "Хорошо работает на ...".

2) Воспользоваться *createsamples* для создания выходного файла-вектора, содержащего положительные образцы. Используя полученный файл, можно повторить процедуру обучения несколько раз, перебирая различные параметры. Например:

```
createsamples -vec faces.vec -info faces.idx -w 30 -h 40
```

Данная строка означает, что информация читается из файла *faces.idx*, который был получен на шаге 1, и выходной вектор записывается в файл *faces.vec*. *createsamples* извлекает положительные образцы из изображения перед нормализацией и изменяет их в соответствии с заданной шириной и высотой (в примере указано 30 и 40, соответственно). *createsamples* также может быть использован для синтеза данных за счёт применения геометрических преобразований, добавления шума, изменения цвета и т.д. Данная процедура может быть использована (например), чтобы узнать логотип компании, для которого выбирается только одно изображение, подвергаемое различным искажениям, которые возможно могут проявиться на реальном изображении. Более подробную информацию о *haartraining* можно найти в директории *opencv/apps/HaarTraining/doc*.

3) Каскад Viola-Jones является бинарным классификатором: просто принимается решение о схожести ("yes" или "no") объекта на изображении с предложенным вариантом из набора для обучения. Ранее уже было показано как собирать и обрабатывать образцы "yes", содержащие выбираемый объект. Теперь будет рассмотрено как собирать и обрабатывать образцы "no" для того, чтобы классификатор знал на какие образцы не похож выбранный объект. Любое изображение, которое не содержит интересующий объект можно отнести к образцам "no". Лучше всего во время тестирования использовать в качестве образцов "no" однотипные изображения. Например, для получения наилучших результатов при распознавании лиц на онлайн видео необходимо использовать образцы "no" с сопоставимых кадров (т.е. с других кадров из того же видео). Тем не менее действительно хороших результатов можно добиться за счёт использования образцов "no", взятых со стороннего ресурса (например, с CD или из интернета). И вновь можно использовать схему с индексным файлом, который должен содержать список имен файлов изображений, по одному изображению на строку. Например, индексный файл с именем *backgrounds.idx* может иметь следующее содержимое:

```
data/vacations/beach.jpg  
data/nonfaces/img_043.bmp  
data/nonfaces/257-5799_IMG.JPG
```

4) **Обучение.** Вот пример обучения, который можно запустить из командной строки или при помощи batch-файла:

```
Haartraining /  
-data face_classifier_take_3 /  
-vec faces.vec -w 30 -h 40 /  
-bg backgrounds.idx /  
-nstages 20 /  
-nsplits 1 /  
[-nonsym] /  
-minhitrate 0.998 /  
-maxfalsealarm 0.5
```

Результаты классификатора будут сохранены в файле *face_classifier_take_3.xml*. Файл *faces.vec* содержит положительные образцы (размера width-height=30x40), а случайным образом извлеченные из файла *backgrounds.idx* изображения будут использованы в качестве негативных образцов. Задаётся 20 (*-nstages*) уровневый каскад, где каждый уровень обучается с процентом верного обнаружения не ниже 0.998 (*-minhitrate*). Задается процент брака (*-maxfalsealarm*) не превышающий 50% для каждого уровня для достижения показателя верного обнаружения 0.998. Слабые классификаторы заданы как "stumps" - это означает, что они имеют только один разделитель (*-nsplits*); можно попробовать задать больше, т.к. в некоторых случаях это может улучшить результаты. Для более сложных объектов можно использовать до шести разделителей, но в большинстве случаев вполне хватает не более трех.

Даже при наличии хорошего компьютера, на обучение может уйти несколько часов; всё зависит от размера набора данных. Процедура обучения может проверить до 100 000 особенностей в рамках обучения на всех положительных и отрицательных образцах. Данный процесс можно распараллелить на многоядерных машинах (при помощи OpenMP). Версию с распараллеленным обучением можно найти в дистрибутиве OpenCV.

[П]|[РС]|(РП) Другие алгоритмы машинного обучения

Итак, на данный момент должно уже сформироваться некоторое представление о том, как работает библиотека ML в OpenCV. Кроме всего прочего, она устроена таким образом, что позволяет с легкостью внедрить в неё новые алгоритмы и методы. И кстати, в ближайшее время планируется добавление новых алгоритмов. В данном разделе будет представлено краткое описание новых алгоритмов, которые уже добавлены в OpenCV на момент написания книги. Каждый из рассматриваемых алгоритмов реализует уже хорошо известную технику обучения, описание которой для конкретного алгоритма можно найти в других книгах, публикациях или интернете. Более подробную информацию по источникам можно найти в директории [.../opencv/docs/ref/opencvref_ml.htm](#).

Expectation Maximization

Expectation maximization (EM) является одной из наиболее популярной техники кластеризации. OpenCV поддерживает EM только с гауссовыми смесями, хотя сама техника носит более общий характер. Техника состоит из нескольких итераций, каждая из которых принимает наиболее вероятное (среднее или "ожидаемое") предположение текущей модели, с последующей настройкой этой модели для максимизации шансов, что предположение окажется верным. В OpenCV алгоритм EM реализован в классе CvEM, который использует годные гауссовые смеси. Т.к. пользователь сам предоставляет годное количество гауссиан, то можно сказать, что данный алгоритм схож с K-means.

K-Nearest Neighbors

K-Nearest Neighbors (К-ближайших соседей) является одним из наиболее простых методов классификации, который просто сохраняет все изученные наблюдения. При возникновении необходимости классифицировать новое наблюдение, ищутся К её ближайших соседей (где К целое число), и новому наблюдению присваивается тот класс, который наиболее распространён среди найденных соседей. В OpenCV данный алгоритм реализован в классе CvKNearest. Техника KNN может быть очень эффективной, хоть и требует сохранения всего набора для обучения, что в свою очередь может привести к разрастанию памяти и как следствие сильному замедлению. Люди зачастую кластеризуют набор для обучения перед использованием данной техники с целью уменьшения его размера. Для читателей, которым интересно как

методы динамически адаптивных ближайших соседей могут быть использованы в электронном мозгу (и в машинном обучении) могут изучить труд Grossberg или относительно более новый труд Carpenter и Grossberg.

Multilayer Perceptron

Multilayer perceptron (многослойный персептрон, MLP; так же известный как *back-propagation*) - это нейронная сеть, которая на момент написания книги входила в число самых эффективных классификаторов, особенно в случаях распознавания текста. Обучение может быть довольно таки медленным, т.к. используется градиентный спуск для минимизации ошибки путем корректировки взвешенных связей между пронумерованными узлами классификации в пределах слоя. Однако в режиме тестирования, работает довольно таки быстро. Реализован данный классификатор в классе CvANN_MLP; пример использования можно найти в файле `.../opencv/samples/c/letter_recog.cpp`. Читатели, которые заинтересованы в эффективном использовании MLP для распознавания текста и объектов, найдут ответы в трудах LeCun, Bottou, Bengio и Haffner. Подробную информацию о реализации и настройке MLP можно найти у LeCun, Bottou и Muller. Относительно более новый труд о *мозгоподобных иерархических сетях* с распространяющейся вероятностью можно найти в работах Hinton, Osindero и Teh.

Support Vector Machine

Имея большой объем данных, лучше всего использовать boosting или random trees классификатор. Однако, при наличии ограниченного набора данных *support vector machine* (SVM) зачастую работает лучше. Этот N-классовый алгоритм работает путём проецирования данных в многомерном пространстве (создаёт новое выходное измерение, комбинируя особенности) с последующим поиском оптимального линейного разделителя между классами. В исходном пространстве исходных данных этот многомерный линейный классификатор может стать далеко нелинейным. Следовательно, можно использовать методы линейной классификации, основанные на *максимальном межклассовом разделении*, для получения нелинейного классификатора, чтобы в каком-то смысле оптимально разделить классы. При наличии необходимого количества измерений, почти всегда возможно отлично разделить классы. Представленная техника реализована в классе CvSVM.

Представленный инструментарий тесно взаимодействует со многими алгоритмами компьютерного зрения, от поиска особенностей при помощи обученного классификатора для слежения, сегментирования сцен, до более простых задач классификации объектов и кластеризации изображений.

[П]||[РС]||(РП) Упражнения

1. Для начала рассмотрим попытку изучения последующей стоимости акции на основе нескольких последних цен за акцию. Допустим, имеются данные об акциях за последние 20 лет. Оцените эффект от различных вариантов превращения этих данных в наборы для обучения и тестирования. Какие преимущества и недостатки у далее представленных подходов?
 - a. Используйте четные наблюдения как набор для обучения и нечетные наблюдения как набор для тестирования.
 - b. Случайным образом распределите наблюдения по наборам для тестирования и для обучения.
 - c. Разделите наблюдения пополам, одну половину используйте как набор для обучения, а другую половину как набор для тестирования.
 - d. Разделите наблюдения на множество мелких окон, состоящих из нескольких наблюдений из прошлого и одного предсказываемого.
2. На рисунке 13-17 представлено распределение "ложных" и "истинных" типов. На рисунке также показано несколько потенциальных мест (a, b, c, d, e, f, g), где можно было бы задать порог.

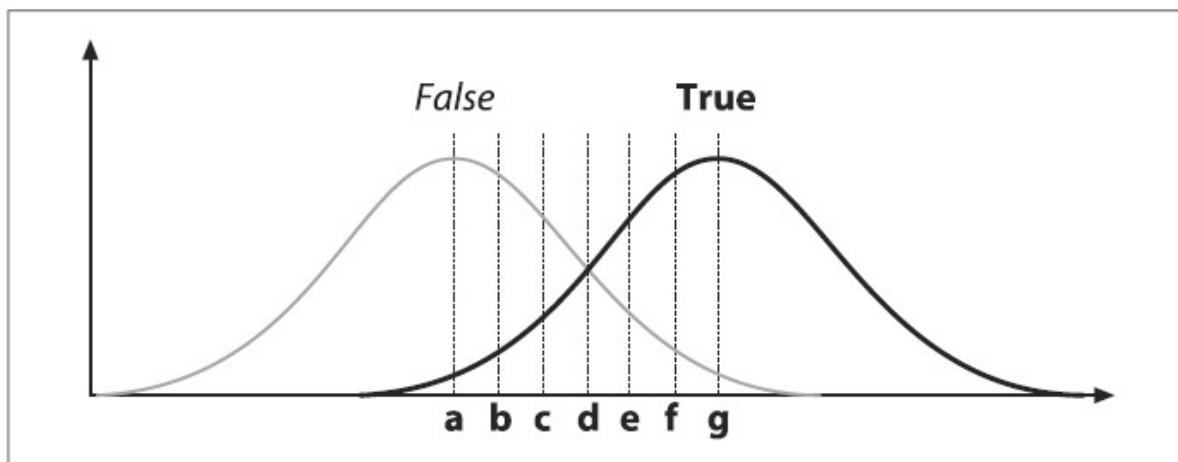


Рисунок 13-17. Гауссово распределение из двух типов, "ложный" и "истинный"

- a. Нарисуйте точки a-g на ROC-кривой
- b. Если класс "истинный" соответствует ядовитым грибам, то какие буквы задают порог?
- c. Как decision tree разделил бы эти данные?

3. Рисунок 13-1.

- a. Нарисуйте как decision tree приблизится к истинной кривой (показана пунктирной линией), имея три разделителя (в данном случае ищется регрессия, а не модель классификации). "Лучший" разделитель для регрессии берет среднее значение значений, содержащихся в листьях, которые являются результатом разделения. Конечные значения регрессионного дерева похожи на лестницу.
 - b. Нарисуйте decision tree, соответствующее истинным данным, с семью разделителями.
 - c. Нарисуйте decision tree, соответствующее зашумленным данным, с семью разделителями.
 - d. Объясните разницу между b и c с точки зрения переобучения.
4. Почему мера разделения (например, Gini) продолжает работать даже когда изучаются множество типов в единственном decision tree?
 5. Изучите рисунок 13-4, на котором изображено двумерное пространство с неровной дисперсией слева и ровной дисперсией справа. Допустим, что это связано с проблемой классификации. Т.е. данные возле одного из "сгустка" принадлежат к одному из двух типов до тех пор, пока данные вблизи другого сгустка принадлежат к тому же или другому из двух типов. Будет ли отличаться важная переменная между левым и правым пространством в случае:
 - a. decision trees?
 - b. K-nearest neighbors?
 - c. naïve Bayes?
 6. Модифицируйте код генерации данных из примера 13-1 - раздел K-means - для создания случайным образом сгенерированного меченного набора. Для этого используйте единое нормальное распределение 10000 наблюдений с координатами пикселя центра (63, 63) на изображении 128x128 со стандартным отклонением ($img->width/6$, $img->height/6$). Чтобы пометить эти данные, разделите пространство на четыре квадранта с центром в пикселе (63, 63). Для получения вероятности маркировки используйте следующую схему. Если $x < 64$, то с вероятностью 20% тип A; иначе если $x \geq 64$, то вероятность тип A 90%. Если $y < 64$, то с вероятностью 40% тип A; иначе если $y \geq 64$, то вероятность тип A 60%. Перемножение вероятностей x и y даёт полную вероятность тип A для квадранта, значения которой показаны в матрице 2x2. Если наблюдение не помечается как A, то по умолчанию оно помечается как B. Например, если $x < 64$ и $y < 64$, то имеется 8% вероятность того, что наблюдение будет помечено как тип A и 92%

вероятность того, что наблюдение будет помечено как *тип B*. Четыре квадранта матрицы вероятности того, что наблюдение будет помечено как *тип A* (или как *тип B*) выглядят следующим образом:

$0.2 \times 0.6 = 0.12$	$0.9 \times 0.6 = 0.54$
$0.2 \times 0.4 = 0.08$	$0.9 \times 0.4 = 0.36$

Используйте эту матрицу для того, чтобы пометить все наблюдения. Для этого в начале определите принадлежность наблюдения к одному из квадранту. Затем случайным образом сгенерируйте число от 0 до 1. Если полученное значение меньше или равно значения из нечетного квадранта, то помечайте наблюдение как *тип A*; иначе помечайте как *тип B*. В результате получите список меченых наблюдений. При этом можно отметить то факт, что ось x более информативнее, чем ось y. Обучение random forests на этом наборе данных и вычисление важной переменной показывает, что x действительно важнее y.

7. Используя набор данных из упражнения 6 и дискретный AdaBoost, обучите две модели: одну с *weak_count = 20 trees*, а другую с *weak_count = 500 trees*. Сгенерируйте случайным образом выбранные 10000 наблюдений для формирования наборов для обучения и тестирования. Выполните обучение и протестируйте на наборе для обучения, состоящее из:
 - a. 150 наблюдений;
 - b. 500 наблюдений;
 - c. 1200 наблюдений;
 - d. 5000 наблюдений.
 - e. Объясните полученные результаты.
8. Повторите упражнение 7, используя классификатор random trees с 50 и 500 trees.
9. Повторите упражнение 7, используя 60 trees и сравните random trees и SVM.
10. В каких случаях алгоритм random tree более устойчив к переобучению, чем decision trees?
11. Рисунок 13-2. При каких условиях, если это возможно, у набора для тестирования ошибок меньше, чем у набора для обучения?
12. Рисунок 13-2 отражает проблему регрессии. Отметьте первую точку на графике как тип A, вторую как тип B, третью как тип A, четвертую как тип B и так далее. Нарисуйте линию разделения для этих двух типов (A и B), которая показывает:

- a. смещение;
 - b. дисперсию.
13. Рисунок 13-3.
- a. Нарисуйте максимально возможную лучшую ROC кривую.
 - b. Нарисуйте максимально возможную худшую ROC кривую.
 - c. Нарисуйте кривую для классификатора, который использует случайнym образом сформированный набор для тестирования.
14. В теореме "no free lunch" говориться, что классификатор не оптимален по всем распределениям меченных данных. Опишите классификатор (который не был представлен в данной главе), хорошо работающий с распределением меченных данных.
- a. Насколько сложным должно быть распределение для обучения naïve Bayes?
 - b. Насколько сложным должно быть распределение для обучения decision trees?
 - c. Как нужно выполнить предобработку распределения на части a и b так, чтобы классификатор можно было обучить максимально легко?
15. Настройте и запустите классификатор Haar для распознавания собственного лица, получаемого с веб-камеры.
- a. В каких пределах изменения масштаба может работать данный классификатор?
 - b. В каких пределах размытия?
 - c. Какой предел наклона головы?
 - d. Какой предел наклона подбородка?
 - e. Какой максимально возможный поворот (влево или вправо) головы?
 - f. Изучите возможность использования 3D поз головы
16. Используйте изображение жеста (статичного) руки с синим или зеленым фоном. Соберите коллекцию изображений, комбинируя различные фоны и жесты. Обучите классификатор Haar для обнаружения жеста. Проведите тестирование полученного классификатора в реальном времени и оцените качество обнаружения.
17. Используя результаты из упражнения 16 и свои знания, попытайтесь улучшить результаты.

Будущее OpenCV

[П]||[РС]||(РП) Прошлое и будущее

В первой главе была представлена история становления библиотеки OpenCV. В следующих главах 2-13 был представлен детальный разбор основных моментов, связанных с библиотекой. Теперь осталось рассказать о том, что планируется сделать в будущем. Количество приложений, использующие компьютерное зрение, быстро растёт; подобного рода приложения находят своё применение для анализа изображений, видео, медицинских снимков и даже для перемещения по Марсу. Совместно с увеличением потребности в подобного рода запросах развивается и сама библиотека.

OpenCV имеет продолжительную историю поддержки от корпорации Intel и относительно недавнюю от [Willow Garage](#), которая финансирует новые разработки в сфере робототехнике и технологический инкубатор. Willow Garage большое внимание уделяет развитию гражданской робототехнике за счёт открытой разработки и поддержки развития аппаратной и программной инфраструктуры, которая на момент написания книги не была частью OpenCV. Благодаря этой поддержке и сохранению связей с несколькими первыми разработчиками, библиотека стала довольно таки быстро развиваться и обновляться. Так же при наличии поддержки стремительно стало развиваться OpenCV сообщество, ускорился процесс оценки и интеграции новых разработок.

Одним из ключевых новых направлений развития в OpenCV является роботизированное восприятие. Сосредоточение происходит на 3D восприятии, а также на распознавании 2D+3D объектов, комбинируя типы для улучшения особенностей, используемые при обнаружении объектов, сегментации и распознавании.

Роботизированное восприятие во многом зависит от 3D сканирования, поэтому ведутся работы по улучшению процесса калибровки камеры, исправлению и сопоставлению картинки с нескольких камер и комбинированию камеры и лазера (рисунок 14-1, на момент написания книги данные разработки ещё были не закончены и потому не были включены в OpenCV).

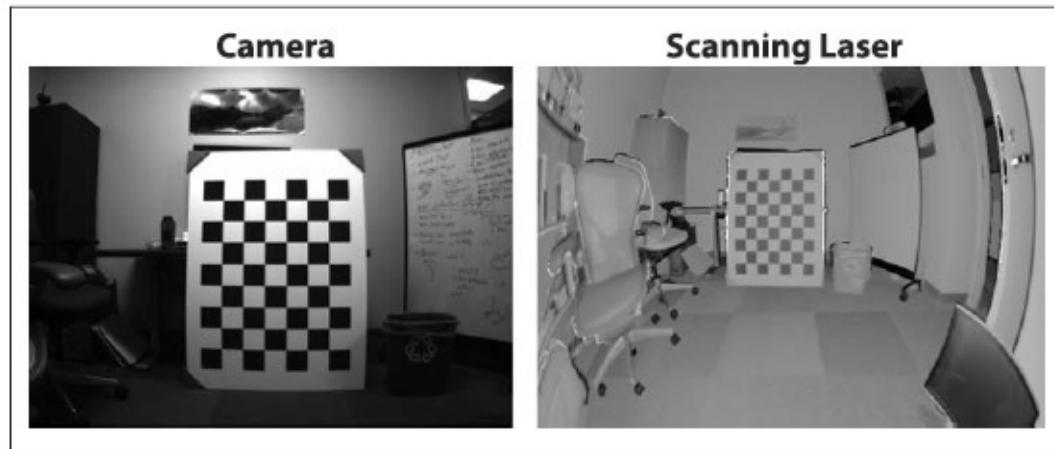


Рисунок 14-1. Новые комбинации 3D сканирования: калибровка камеры (слева) и лазерное сканирование глубины (справа). Фото любезно предоставлено Hai Nguyen и Willow Garage

Если в продажу поступят устройства, имеющие "лазер + камера", то развитие получат flash LIDAR и устройства с инфракрасным излучением. Дополнительные усилия будут направлены на развитие структурированной триангуляции или лазерного света для повышения точности глубины сканирования. Большинство методов после выполнения глубинного сканирования возвращают облако 3D точек. Поэтому дополнительно планируется развивать переработку облака точек, полученных в результате сканирования 3D мира, в 3D сети. 3D сети позволяют 3D моделям захватывать объекты среды, сегментировать объекты и как следствие обучение роботов возможности манипулировать этими объектами. Трехмерные сети также могут быть использованы для того, чтобы роботы могли плавно переходить от внешнего к внутреннему 3D восприятию для планирования и последующего выполнения повторяющихся операций регистрации, манипуляции и перемещения объекта.

Наравне с 3D сканированием объектов, роботам нужно будет распознавать 3D объекты и их 3D представления. Для того, чтобы это было возможно необходимо развивать некоторые масштабируемые методы, распознающие 2D+3D объекты. Процесс создания роботов включает в себя большинство областей компьютерного зрения и искусственного интеллекта, начиная от точного 3D реконструирования слежения, идентификации человека, распознавания объекта, сшивки изображений и заканчивая обучением, контролированием, планированием и принятием решений. Любая высокоуровневая задача, такая, как планирование, во многом облегчает, ускоряет и повышает точность восприятия глубины и распознавание. Именно этим областям в дальнейшем будет уделяться повышенное внимание либо за счёт поощрения открытого сообщества, либо за счёт использования более эффективных методов решения сложных проблем восприятия реального мира, распознавания и обучения.

При этом безусловно OpenCV не перестанет развиваться в других направлениях, от индексации изображений и видео в интернете до систем безопасности и анализа медицинских данных. В целом, дальнейшее развитие OpenCV будет зависеть от потребностей общества.

[П]|[РС]|(РП) Направления

Хотя OpenCV и не фокусируется только на алгоритмах реального времени, в дальнейшем их развитие не прекратиться. Никто не может сказать в каком направлении будет развиваться библиотека, но точно можно сказать, что высокоприоритетные области точно будут развиваться.

Приложения

Уже имеются "потребители" рабочих приложений с ограниченным набором функциональности. Например, всё больше людей используют полностью автоматизированное стерео решение. В дальнейшем ожидается ещё больше рабочих приложений, которые смогут выполнять калибровку и исправление нескольких камер, а также GUI для выполнения глубинного 3D сканирования.

3D

В дальнейшем ожидается улучшенная поддержка датчиков глубины и скомбинированных 2D камер с устройствами измерения 3D. Также ожидается улучшение стерео алгоритмов. Поддержка структурированного света также возможна.

Плотный оптический поток

Т.к. имеется потребность в знании о движении объекта (и в частности поддержка 3D), в OpenCV давно назрела потребность в эффективной реализации плотного оптического потока Block.

Features

В плане улучшения процесса распознавания объекта, можно ожидать полнофункциональный инструментарий, который будет иметь структуру для обнаружения и идентификации взаимозаменяемых точек. Изменения коснутся SURF, HoG, Shape Context, MSER, Geometric Blur, PHOG, PHOW и других. Планируется поддержка 2D и 3D features.

Инфраструктура

Ожидается улучшение классов-обертоок (Daniel Filip и Google поделились быстрым и легковесным классом-оберткой *WImage*, который был разработан для внутреннего использования в OpenCV), хороший интерфейс Python, улучшенный GUI, улучшенная документация и обработка ошибок, улучшенная поддержка Linux и т.д.

Интерфейс камеры

Планируется более бесшовная обработка изображения с камеры наряду с возможной поддержкой камер с более высоким динамичным диапазоном. На момент написания книги большинство камер поддерживали только 8 бит на канал; более новые камеры смогут поддерживать 10 или 12 бит на канал. Чем выше динамический диапазон у камеры, тем лучше процесс распознавания и стерео регистрации, т.к. появляется возможность обнаружения тонких текстур и цветов, которые выдержаны в более узком диапазоне камеры.

Специфика

Многие методы распознавания объектов определяют *заметные области*, которые почти не меняются при переходе от одного представления к другому. Эти области (также известные как *interest points*) могут быть отмечены каким-то особенным способом. Хотя все техники, описанные в данном разделе, могут быть выполнены за счёт использования примитивов OpenCV, на момент написания книги в OpenCV отсутствовала реализация самых популярных техник: *interest-region detectors* и *feature keys*.

OpenCV включает в себя эффективную реализацию *Harris corner interest-point detectors*, но ей не хватает поддержки от популярного "maximal Laplacian over scale" детектора, разработанного *David Lowe*, и детектора *maximally stable extremal region* (MSER).

Также OpenCV не хватает *SURF gradient histogram grids*, которые идентифицируют заметные области. Кроме того, ожидается включение *histogram of oriented gradients* (HoG), *Geometric Blur*, *offset image patches*, *dense rapidly computed Gaussian scale variant gradients* (DAISY), *gradient location* и *orientation histogram* (GLOH). Так же ожидается включение *contextual* или *meta-features* такие, как *pyramid match kernels*, *pyramid histogram* вложенные в другие *features*, *PHOW*, *Shape Context*. Всё это трудновыполнимые задачи, но OpenCV сообщество нацелено на их решение.

Для роботов требуются механизмы распознавания объектов (что) и определения местоположения объектов (где). Для этого необходимо добавление подходов, основанных на работах *Shi* и *Malik*, для выполнения сегментации с, возможно, более быстрой реализацией. При этом недавние разработки, однако, используют обучение для обеспечения возможности совместного использования распознавания и сегментирования.

Вместе с улучшением 3D сканирования должна прийти поддержка для *visual odometry* и *visual SLAM*. С появлением более точного восприятия глубины и идентификации, возникнет потребность в обеспечении более лучшей навигации и манипуляции 3D

объектом. Так же ведутся разговоры о создании специализированного интерфейса для трассировки лучей для генерации более лучших наборов для обучения при работе с 3D объектами.

Роботы, системы безопасности и системы поиска изображений и видео в вебе - всем требуется способность распознавать объекты; таким образом, в OpenCV должны быть усовершенствованы техники шаблонного сопоставления, которые содержит под библиотека ML. В частности, для начала необходимо упростить интерфейс алгоритмов обучения, а затем сделать так, чтобы они работали "из коробки". Могут быть добавлены новые методы обучения, некоторые из которых смогут работать сразу с двумя и более объектами (как это делается уже сейчас в random forest). Так же требуются масштабируемые методы распознавания объектов, чтобы пользователю не приходилось заново обучать совершенно новую модель для каждого типа объекта. В уже имеющиеся в ML классификаторы планируется добавление обработчиков информации о глубине.

Markov random fields (MRFs) и *conditional random fields* (CRFs) становятся всё более популярными в компьютерном зрении. Хотя зачастую эти методы имеют специфичные проблемы, всё же планируется их внедрение.

Так же требуются методы обучения *web-sized* или *automatically collected* при помощи перемещения БД робота, возможно включив предложение Zisserman для методов "approximate nearest neighbor" для случаев с миллионом или миллиардом наблюдений. Вместе с тем требуется ускорить *boosting* и обучение *Haar feature* для поддержки масштабирования больших объектов. Несколько процедур из ML требуют, чтобы все данные находились в памяти, что серьёзно ограничивает их использование в случае большого набора данных. В дальнейшем OpenCV необходимо избавиться от подобного рода ограничений.

OpenCV так же недостаёт хорошей документации. Данная книга конечно поможет покрыть часть запросов, однако руководство к использованию всё ещё нуждается в улучшениях. Высокий приоритет имеет задача по улучшению поддержки Linux и интерфейса для взаимодействия с внешними языками - прежде всего необходимо обеспечить возможность программирования в Python и Numpy. В дополнение к этому было бы неплохо использовать ML напрямую из Python и его пакетов SciPy и Numpy.

Для развития сообщества разработчиков планируется проводить большие конференции по компьютерному зрению. Так же планируется устраивать соревнования "grand challenge" с соизмеримой суммой призовых.

[П]|[РС]|(РП) OpenCV для разработчиков

Существует всемирное сообщество разработчиков, которые используют OpenCV так, что конечные пользователи могут взаимодействовать с их творениями динамичными способами. Чаще всего разработчики используют в своих работах детектор лица, оптический поток и слежение. Авторы книги надеются, что данная книга поможет начинающим разобраться в том, как работает библиотека OpenCV, а улучшения в области сканирования глубины сделает взаимодействие богаче и более надежнее. Целенаправленные улучшения в области распознавания объекта добавят разнообразия в режимы взаимодействия с творениями, т.к. появится возможность использования самих объектов в качестве модального управления. С появлением возможности захвата 3D сетей также станет возможно "импортировать" конечного пользователя в "творение", что даст разработчику более качественную информацию о действиях пользователя; что в свою очередь может быть использовано для повышения динамичного взаимодействия. В дальнейшем планируется уделять больше времени потребностям и пожеланиям, возникающие у сообщества разработчиков.

[П]|[РС]|(РП) Послесловие

В данной книге было представлено достаточное количество теоретического и практического материала для начинающих знакомство с OpenCV разработчиков, а также раскрыты ближайшие планы на дальнейшее развитие библиотеки. Безусловно развитие программного и аппаратного обеспечения не стоит на месте. Камеры дешевеют и становятся более доступными за счет расширения области их применения: от сотовых телефонов до светофоров. Кроме того, группа производителей задалась целью разработать сотовый телефон проектор, что идеально подходит для роботов, т.к. большинство сотовых телефонов легкие, с низким уровнем потребления энергии, и имеют встроенную камеру. Это также «даёт зелёный свет» ближнему портативному структурированному свету; тем самым имеем более точные карты глубины, а это именно то, что нужно для манипуляции роботом и сканирования 3D объекта.

Оба автора книги участвовали в создании видеосистемы для Stanley, Стэнфордского робота, который выиграл в 2005 году DARPA Grand Challenge. В рамках семичасовой гонки разработанная видеосистема в совокупности с лазерным сканером глубины работала безупречно. Для человечества это означает, что поездка на машине может стать более безопасной. Авторы книги надеются, что данная книга поможет читателям в решении собственных задач в области компьютерного зрения. Используя в совокупности камеру и OpenCV, люди смогут решать реальные проблемы. Например, стерео зрение поможет повысить уровень безопасности при использовании автомобиля, появятся новые средства управления в играх и новые системы безопасности.

Впереди компьютерное зрение ждёт хорошее будущее и, скорее всего, оно будет одной из ключевых технологий 21 века. OpenCV в свою очередь также (отчасти) будет одной из перспективных технологий для компьютерного зрения. Изучайте, дерзайте и, возможно, именно Вы поспособствуете развитию библиотеки OpenCV!