Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Учебный курс «Разработка мультимедийных приложений с использованием библиотек OpenCV и IPP»

Лабораторная работа
Базовые операции обработки изображений

Кустикова В.Д.

При поддержке компании Intel

Нижний Новгород
2012

Содержание

Введение

> Великолепный снимок – это вещь в себе, а не компромисс на пути к печатной странице.
>
> *Ирвинг Пенн*

Компьютерное зрение – одна из самых молодых областей, которая использует результаты многих смежных научных направлений таких, как машинное обучение, проективная геометрия, теория вероятности и т.п. Основной круг задач компьютерного зрения лежит в сфере качественного и количественного анализа изображений и потоков видеоданных (поиск и классификация объектов, сопровождение объектов на видео и др.). Большинство методов решения указанных задач тем или образом использует различные подходы для выполнения предобработки изображений или кадров видеопотока с целью дальнейшего повышения качества работы алгоритмов.

В настоящей работе рассматривается ряд методов предобработки, демонстрируется использование соответствующих операций на базе средств открытой библиотеки компьютерного зрения OpenCV [5].

Предобработка подразумевает преобразование исходного изображения в некоторое новое изображение. Безусловно, наиболее популярным способом обработки является *фильтрация* [6], которая в большинстве приложений используется для удаления шумов. Результат фильтрации – изображение того же размера, что и исходное, но содержащее значения интенсивностей пикселей, обновленные в соответствии с некоторым правилом. *Линейные фильтры* – простейшие представители данного класса методов предобработки. Линейная фильтрация сводится к пересчету значений интенсивности каждого пикселя изображения посредством вычисления взвешенной свертки интенсивностей пикселей, принадлежащих некоторой его окрестности. *Размытие* или *сглаживание* – еще один подход к предобработке. Сглаживание подобно линейной фильтрации, в простейшем случае предполагает свертку с равными весовыми коэффициентами, в более сложных приложениях – свертку с дискретными значениями функции распределения Гаусса или выбор медианы среди набора интенсивностей в окрестности.

Морфологические операции (эрозия, дилатация) вместо вычисления сверток выполняют поиск минимального/максимального значения интенсивности в фиксированной окрестности каждого пикселя. Если эрозия и дилатация эффективно работают на черно-белых изображениях, то для полутоновых и цветных существуют более сложные морфологические преобразования такие, как замыкание, размыкание и т.п.

Выделение *краев* (или *ребер*) объектов на изображениях – принципиально другая задача, возникающая в процессе предобработки т.к. на выходе формируется не просто преобразованное изображение, а карта границ объектов. Типичные методы выделения ребер базируются на применении к исходному изображению оператора Собеля (оператор первых производных по направлениям) или дискретного оператора Лапласа (оператор вторых производных). На данный момент наиболее известным детектором ребер является детектор Канни [4, 8, 9]. В работе предлагается описание схемы работы данного детектора.

Слабый контраст – распространенный дефект изображений и кадров видео. Существует три основных метода повышения контраста изображения: *линейная растяжка гистограммы* (линейное контрастирование), *нормализация гистограммы*, *выравнивание* (линеаризация или эквализация, equalization) гистограммы. Далее в работе предлагается краткое описание перечисленных методов, впоследствии основное внимание уделяется выравниванию гистограммы как наиболее эффективному подходу к решению задачи повышения контраста.

Заметим, что множество методов предобработки изображений не ограничивается представленными в настоящей работе. Значительный упор сделан на методы и подходы, которые приобрели практическое применение в задачах компьютерного зрения. Примерами таких задач могут служить:

1. Анализ медицинских изображений, например, рентгеновских снимков, полученных с помощью магнитно-резонансной томографии.
2. Классификация изображений по контексту и организации последующего поиска в базах данных изображений по текстовым запросам.
3. Задачи видеоаналитики, в частности, распознавание людей в охранных системах видеонаблюдения или системах помощи водителю, автомобильных номеров в системах наблюдения за дорожным движением.
4. Качественный анализ сцен и распознавание конкретных классов объектов роботами, оснащенными видеокамерами.
5. Задачи повышения качества и контраста снимков, полученных со спутниковых видеокамер, для последующего их анализа.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучить базовые операции обработки изображений с использованием реализаций соответствующих функций библиотеки компьютерного зрения OpenCV.

Данная цель предполагает решение следующих задач:

1. Изучить принцип работы базовых операций обработки изображений:
   - линейная фильтрация;
   - сглаживание с различными ядрами;
   - морфологические преобразования;
   - применение оператора Собеля;
   - применение оператора Лапласа;
   - определение ребер посредством детектора Канни;
   - вычисление гистограммы;
   - выравнивание гистограммы.
2. Рассмотреть прототипы функций, реализующих перечисленные операции в библиотеке OpenCV.
3. Разработать простые примеры использования указанного набора функций.
4. Разработать консольный графический редактор, поддерживающий все представленные операции работы с изображениями. Предполагается организация простейшего диалога с пользователем.
5. Разработать графический редактор с интерфейсом на базе библиотеки OpenCV, которая предоставляет возможность создания различных компонент средствами Qt.

1.2. Структура работы

В работе предлагается описание базовых операций обработки изображений. Приводятся прототипы функций библиотеки OpenCV, содержащих реализацию рассматриваемых функций с описанием назначения входных параметров. Предлагаются примеры программ, демонстрирующие использование каждой функции. Проводится анализ результатов запуска этих программ на некоторых тестовых изображениях. Разрабатывается структура консольного графического редактора, который

обеспечивает возможность применения рассматриваемых операций обработки изображений. Рассматривается структура графического редактора посредством использования Qt-компонент библиотеки OpenCV.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Операционная система	Microsoft Windows 7
Среда разработки	Microsoft Visual Studio 2010
Библиотека TBB	Intel® Threading Building Blocks 3.0 for Windows, Update 3 (в составе Intel® Parallel Studio XE 2011 SP1)
Библиотеки OpenCV	Версия 2.4.2

1.4. Рекомендации по проведению занятий

При выполнении данной лабораторной работы рекомендуется следующая последовательность действий:

1. Привести примеры задач и приложений компьютерного зрения, которые требуют использования операций обработки изображений. Обосновать необходимость использования таких операций.
2. Последовательно рассмотреть базовые операции работы с изображениями, начиная с линейной фильтрации. В параллели необходимо вводить функции библиотеки OpenCV, реализующие данные операции, и демонстрировать примеры программ, в которых выполняется их применение, анализировать результаты запуска программ на тестовых изображениях, пояснять возникающие эффекты.
3. Разработать структуру консольного графического редактора, поддерживающего все представленные операции работы с изображениями. Пояснить схему организации диалога с пользователем.
4. Разработать структуру графического редактора с кнопочным интерфейсом на базе библиотеки OpenCV, обеспечивающей возможность создания Qt-компонент.

2. Обзор возможностей модуля imgproc библиотеки OpenCV

2.1. Свертка и линейные фильтры

Линейные фильтры – семейство самых простых фильтров изображений с точки зрения математического описания [6]. Предположим, что имеется полутоновое изображение I. Тогда любой линейный фильтр определяется вещественнозначной функцией F, заданной на растре. Данная функция называется *ядром фильтра*, а операция фильтрации выполняется посредством вычисления *дискретной свертки*:

$$I'(x,y) = \sum_i \sum_j F(i,j) \cdot I(x+i, y+j)$$

Как правило, ядро фильтра применяется к некоторой окрестности O точки, поэтому пределы изменения индексов i и j определяются выбранной формой и размером окрестности. Данная окрестность в некоторых источниках называется *шаблоном* или *апертурой*. В процессе вычисления свертки выполняется проход по пикселям всего изображения, шаблон накладывается на каждый текущий пиксель посредством совмещения пикселя с конкретной точкой шаблона – *ведущей позицией шаблона*, после чего вычисляется свертка. Необходимо отдельно обратить внимание на ситуацию, когда текущий пиксель находится на границе изображения. Указанную проблему можно решить несколькими способами:

- Обрезать края, т.е. не проводить фильтрацию для всех граничных пикселей, на которые невозможно наложить шаблон без выхода за пределы изображения.
- Не учитывать в процессе суммирования пиксель, который реально не существует.
- Доопределить окрестности граничных пикселей посредством экстраполяции (например, простым дублированием граничных пикселей).
- Доопределить окрестности граничных пикселей посредством зеркального отражения, т.е. завернуть изображение в тор.

Выбор решения во многом зависит от приложения, так например, зеркальное отражение на практике не совсем естественный способ.

Для вычисления сверток в библиотеке OpenCV присутствует функция **`filter2D`**.

```
void filter2D(const Mat& src, Mat& dst, int ddepth,
```

```
                const Mat& kernel,
                Point anchor=Point(-1, -1), double delta=0,
                int borderType=BORDER_DEFAULT)
```

Рассмотрим подробнее параметры приведенной функции.

- **src** – исходное изображение.
- **dst** – свертка. Имеет такое же количество каналов и глубину, что и исходное изображение.
- **ddepth** – глубина результирующего изображения. Если на вход функции передано отрицательное значение, то глубина совпадает с глубиной входного изображения.
- **kernel** – ядро свертки, одноканальная вещественная матрица.
- **anchor** – ведущая позиция ядра. По умолчанию принимает значение (-1,-1), которое означает, что ведущая позиция расположена в центре ядра.
- **delta** – константа, которая может быть добавлена к значению интенсивности после фильтрации перед непосредственной записью результата.
- **borderType** – параметр, определяющий метод дополнения границы, чтобы можно было применять фильтр к граничным пикселям исходного изображения. Принимает любое значение вида **BORDER_*** за исключением **BORDER_TRANSPARENT** и **BORDER_ISOLATED**.

Функция обеспечивает применение произвольного линейного фильтра с ядром **kernel** к изображению **src**. Результат фильтрации записывается в массив **dst**. Если апертура выходит за пределы изображения, то граничные пиксели дополняются в соответствии с методом, указанным в **borderType**. Новое значение интенсивности пикселя вычисляется по формуле:

$$dst(x,y) = \sum_{\substack{0 \le x' < anchor.x \\ 0 \le y' < anchor.y}} kernel(x',y') \cdot src(x + x' - anchor.x, \\ y + y' - anchor.y)$$

В случае многоканального изображения ядро применяется к каждому каналу в отдельности.

Ниже приведен пример использования функции **filter2D**. Представленная программа обеспечивает загрузку изображения и применение линейного фильтра с вещественным ядром, заданным константой **kernel**. Также выполняется отображение исходного и результирующего изображений.

```
#include <stdlib.h>
#include <stdio.h>
#include <opencv2/opencv.hpp>

using namespace cv;

const char helper[] =
    "Sample_filter2D.exe <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    // константы для определения названия окон
    const char *initialWinName = "Initial Image",
               *resultWinName = "Filter2D";
    // константы для хранения ядра фильтра
    const float kernelData[] = {-0.1f, 0.2f, -0.1f,
                                 0.2f, 3.0f,  0.2f,
                                -0.1f, 0.2f, -0.1f};
    const Mat kernel(3, 3, CV_32FC1, (float *)kernelData);
    // объекты для хранения исходного
    // и результирующего изображений
    Mat src, dst;
    // проверка аргументов командной строки
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
    // загрузка изображения
    src = imread(argv[1], 1);
    // применение фильтра
    filter2D(src, dst, -1, kernel);

    // отображение исходного изображения и
    // результата применения фильтра
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, src);
    namedWindow(resultWinName, CV_WINDOW_AUTOSIZE);
    imshow(resultWinName, dst);
    waitKey();

    // закрытие окон
    destroyAllWindows();
    // освобождение ресурсов
    src.release();
    dst.release();
    return 0;
}
```

Далее на рисунке показан результат работы приведенной программы (рис. 1, справа). Очевидно, что применение фильтра с ядром, зафиксированным в программной коде (выделено полужирным), привело к уменьшению контраста исходного тестового изображения (рис. 1, слева).

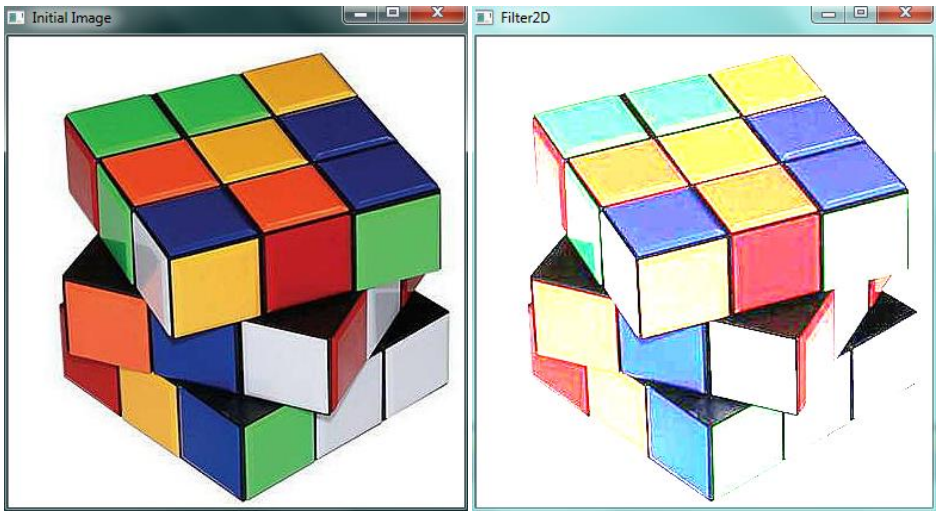


Рис. 1. Результат применения фильтра

Отметим, что в случае больших ядер (размера порядка 11x11 пикселей) для вычисления свертки используется быстрое преобразование Фурье, в случае небольших ядер – прямой алгоритм. Также если ядро *сепарабельное*, т.е. может быть представлено в виде пары ядер, которые могут быть последовательно применены к строкам и столбцам изображения в отдельности, то предусмотрена более эффективная реализация линейного фильтра с использованием функции **`sepFilter2D`**. При вызове данная функция требует явного указания двух одномерных ядер **`rowKernel`** и **`columnKernel`**.

```
void sepFilter2D(const Mat& src, Mat& dst, int ddepth,
                 const Mat& rowKernel,
                 const Mat& columnKernel,
                 Point anchor=Point(-1, -1),
                 double delta=0,
                 int borderType=BORDER_DEFAULT)
```

2.2. Сглаживание изображений

Сглаживание или *размытие* изображения – это одна из самых простых и часто используемых операций обработки изображений. Как правило, размытие применяется, чтобы уменьшить шум или артефакты, которые обусловлены выбором камеры. Также сглаживание играет важную роль

при необходимости уменьшить разрешение изображения и получить пирамиду изображений разного масштаба (image pyramids) [1].

Разработчики OpenCV реализовали несколько функций размытия изображения. Далее остановимся на некоторых из них, а именно рассмотрим функции **blur**, **boxFilter**, **GaussianBlur** и **medianBlur** [7]:

```
void blur(const Mat& src, Mat& dst, Size ksize,
          Point anchor=Point(-1, -1),
          int borderType=BORDER_DEFAULT)

void boxFilter(const Mat& src, Mat& dst, int ddepth,
               Size ksize, Point anchor=Point(-1, -1),
               bool normalize=true,
               int borderType=BORDER_DEFAULT)

void GaussianBlur(const Mat& src, Mat& dst, Size ksize,
                  double sigmaX, double sigmaY=0,
                  int borderType=BORDER_DEFAULT)

void medianBlur(const Mat& src, Mat& dst, int ksize)
```

Приведем назначение общих параметров указанных функций:

- **src** – исходное изображение.
- **dst** – результирующее изображение, имеет такой же размер и тип, как и исходное изображение.
- **kSize** – размер ядра для размытия.
- **anchor** – ведущий элемент ядра. По умолчанию параметр принимает значение (-1, -1), ведущий элемент совпадает с центром ядра.
- **borderType** – способ дополнения границы.

Функция **blur** выполняет размытие посредством вычисления свертки исходного изображения с ядром K:

$$K = \frac{1}{kSize.width \cdot kSize.height}\begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}$$

Функция **boxFilter** использует ядро более общего вида:

$$K = \alpha\begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}, \text{ где } \alpha = \begin{cases} \frac{1}{kSize.width \cdot kSize.height}, & normalize = true \\ 1, \text{в противном случае} \end{cases}$$

Параметр **normalize** по существу представляет флаг, который указывает, является ли ядро нормализованным или нет. Вызов функции **blur**

эквивалентен вызову **boxFilter(src, dst, src.type(), anchor, true, borderType)**.

Функция **GaussianBlur** осуществляет размытия с помощью вычисления свертки изображения с дискретным ядром Гаусса со стандартными отклонениями, равными **sigmaX** и **sigmaY** по осям Ox и Oy соответственно. Заметим, что при вызове данной функции накладывается ограничение на параметр **kSize.** Ширина и высота ядра должны быть положительными и нечетными, либо нулевыми, если размер ядра определяется из стандартных отклонений.

Функция **medianBlur** обеспечивает размытие посредством применения *медианного фильтра*. Медианный фильтр строится подобно линейному фильтру. Выбирается некоторый шаблон, который накладывается на все пиксели изображения. Набор интенсивностей пикселей, которые накрыты шаблоном, сортируются, и выбирается интенсивность, находящаяся в середине отсортированного множества. По сути, определяется медиана в отсортированном наборе данных. Исходное изображение **src** может представляться 1-, 3-, либо 4-канальной матрицей. В случае нескольких каналов медианный фильтр применяется независимо к каждому из них. Размер апертуры определяется параметром **kSize**. Если размер шаблона составляет 3 или 5, то глубина изображения должна иметь тип **CV_8U**, **CV_16U** или **CV_32F**. Для большей величины **kSize** поддерживается только **CV_8U**.

Далее показан пример использования функции **blur** и результат выполнения приведенной программы (рис. 2).

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

using namespace cv;

const char helper[] =
    "Sample_blur.exe  <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
               *blurWinName = "blur";
    Mat img, blurImg;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
```

```
    // загрузка изображения
    img = imread(argv[1], 1);
    // сглаживание
    blur(img, blurImg, Size(5, 5));

    // отображение исходного изображения
    // и результата размытия
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(blurWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, img);
    imshow(blurWinName, blurImg);
    waitKey();

    // закрытие окон
    destroyAllWindows();
    // освобождение ресурсов
    img.release();
    blurImg.release();
    return 0;
}
```

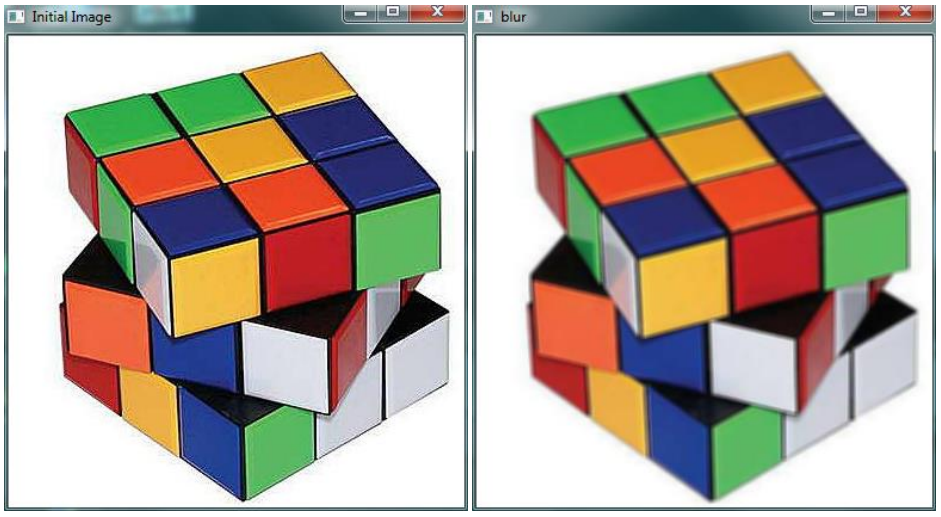


Рис. 2. Результат размытия с квадратным ядром со стороной в 5 пикселей

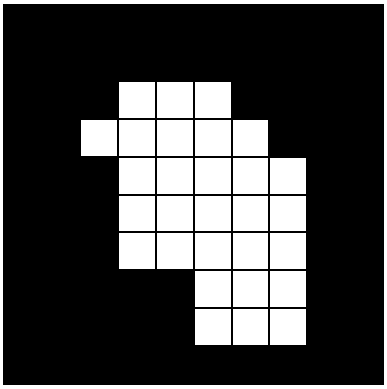
2.3. Морфологические преобразования

2.3.1. Дилатация и эрозия

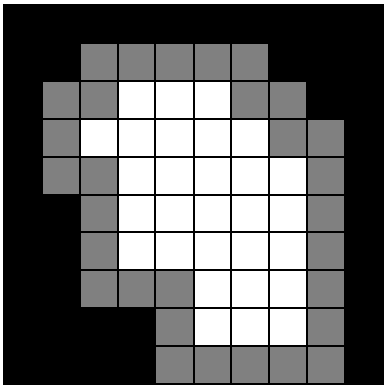
Дилатация (морфологическое расширение) – свертка изображения или выделенной области изображения с некоторым ядром. Ядро может иметь произвольную форму и размер [1, 2]. При этом в ядре выделяется единственная *ведущая позиция* (anchor), которая совмещается с текущим пикселем при вычислении свертки. Во многих случаях в качестве

ядра выбирается квадрат или круг с ведущей позицией в центре. Ядро можно рассматривать как шаблон или маску. Применение дилатации сводится к проходу шаблоном по всему изображению и применению оператора поиска локального максимума к интенсивностям пикселей изображения, которые накрываются шаблоном. Такая операция вызывает рост светлых областей на изображении (рис. 3, c). На рисунке серым цветом отмечены пиксели, которые в результате применения дилатации будут белыми.

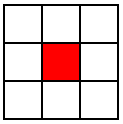
Эрозия (морфологическое сужение) – обратная операция. Действие эрозии подобно дилатации, разница лишь в том, что используется оператор поиска локального минимума (рис. 3, d), серым цветом залиты пиксели, которые станут черными в результате эрозии.



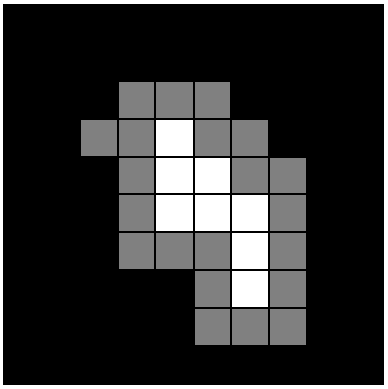
a) исходное изображение



c) результат дилатации



b) шаблон (центр – ведущий элемент)



d) результат эрозии

Рис. 3. Применение морфологических операций

Рассмотрим прототипы соответствующих функций эрозии и дилатации, реализованных в OpenCV [7].

```
void dilate(const Mat& src, Mat& dst, const Mat& element,
            Point anchor=Point(-1, -1), int iterations=1,
```

```
            int borderType=BORDER_CONSTANT,
            const Scalar& borderValue =
                  morphologyDefaultBorderValue())

void erode(const Mat& src, Mat& dst, const Mat& element,
           Point anchor=Point(-1, -1), int iterations=1,
           int borderType=BORDER_CONSTANT,
           const Scalar& borderValue =
                  morphologyDefaultBorderValue())
```

Параметры:

- **src** – исходное изображение.
- **dst** – результирующее изображение, имеет такой же размер, что и входное изображение. Отметим, что результат операции может записываться в исходное изображение.
- **element** – шаблон, который используется в процессе дилатации. Если **element=Mat()**, то применяется квадратный шаблон размером 3x3.
- **anchor** – позиция ведущего пикселя в структурном элементе. Значение по умолчанию $(-1,-1)$ означает, что в качестве ведущего элемента выбирается центр шаблона.
- **interations** – количество раз, которое применяется дилатация/эрозия.
- **borderType** – параметр, определяющий метод дополнения границы, чтобы можно было применять дилатацию/эрозию к граничным пикселям исходного изображения. Принимает любое значение вида **BORDER_*** за исключением **BORDER_TRANSPARENT** и **BORDER_ISOLATED**.
- **borderValue** – размер границы в случае, если она имеет постоянный размер. Значение по умолчанию равно **morphologyDefaultBorderValue**, преобразуется в **-inf** для дилатации и **+inf** для эрозии. При использовании значения по умолчанию операция применяется только к внутренним пикселям изображениям.

Далее приведен пример использования указанных функций. Сначала в программе выполняется чтение исходного изображения, название которого передается через аргументы командной строки. Затем применяются рассматриваемые операции. По окончании выполнения программы выполняется отображение результата применения эрозии и дилатации.

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

```
using namespace cv;

const char helper[] =
    "Sample_erode_dilate.exe  <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
               *erodeWinName = "erode",
               *dilateWinName = "dilate";
    Mat img, erodeImg, dilateImg, element;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
    // загрузка черно-белого изображения
    img = imread(argv[1], 1);
    // вычисление эрозии и дилатации
    element = Mat();
    erode(img, erodeImg, element);
    dilate(img, dilateImg, element);

    // отображение исходного изображения и результата
    // применения морфологических операций "эрозия"
    // и "дилатация"
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(erodeWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(dilateWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, img);
    imshow(erodeWinName, erodeImg);
    imshow(dilateWinName, dilateImg);
    waitKey();

    // освобождение ресурсов
    img.release();
    erodeImg.release();
    dilateImg.release();
    return 0;
}
```

Результат выполнения данной программы показан на примере некоторого черно-белого изображения (рис. 4). Как видно из рисунков, применение эрозии привело к сужению белых областей, дилатации – расширению (рис. 5).

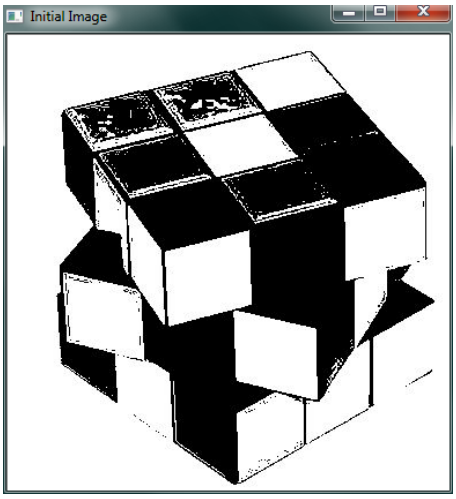


Рис. 4. Исходное черно-белое изображение

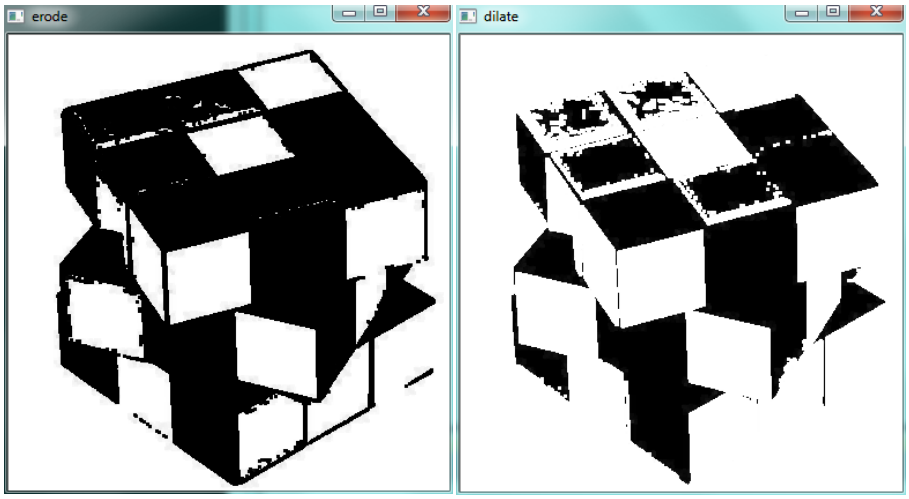


Рис. 5. Результат применения эрозии (слева) и дилатации (справа)

2.3.2. Дополнительные морфологические операции

При обработке бинарных изображений, как правило, базовых операций эрозии и дилатации достаточно. В процессе работы с цветными изображениями или изображениями в оттенках серого могут быть полезными более сложные морфологические операции.

Библиотека OpenCV поддерживает ряд дополнительных морфологических операций, которые реализуется в функции **`morphologyEx`** [7].

```
void morphologyEx(const Mat& src, Mat& dst, int op,
                  const Mat& element,
                  Point anchor=Point(-1, -1),
                  int iterations=1,
```

```
                int borderType=BORDER_CONSTANT,
                const Scalar& borderValue =
                      morphologyDefaultBorderValue())
```

Рассмотрим параметры функции **morphologyEx**.

- **src, dst, element, anchor, borderType, borderValue** имеют такой же смысл, что и в функциях вычисления эрозии и дилатации.
- **op –** тип морфологической операции.

Данная функция обеспечивает выполнение следующих морфологических операций:

- **MORPH_OPEN** – размыкание. Результатом размыкания является дилатации, которая применяется к эрозии исходного изображения. Условно можно записать в виде выражения **dst = open(src, element) = dilate(erode(src, element))**. Операция эрозия позволяет удалить все мелкие объекты и шум на изображении, но ее применение приводит к значительному уменьшению размеров оставшихся объектов. Чтобы увеличить размер объектов, выделенных с помощью эрозии, достаточно применить дилатацию.
- **MORPH_CLOSE** – замыкание. Замыкание – операция обратная размыканию, **dst = close(src, element) = erode(dilate(src, element))**. Операция замыкания позволяет удалить небольшие внутренние «дырки» и убрать зернистость по краям области. «Дырки» удаляются за счет начального применения дилатации, но дилатация приводит к росту границы. Последующее применение эрозии обеспечивает обратное уменьшение границы.
- **MORPH_GRADIENT** – морфологический градиент. Результатом применения данной операции является разница дилатации и эрозии исходного изображения с одинаковым ядром **dst = morph_grad(src, element) = dilate(src, element) - erode(src, element)**. Морфологические градиент обеспечивает поиск контуров объектов, размер которых превышает размер ядра.
- **MORPH_TOPHAT** – «верх шляпы» («top hat»), **dst = tophat(src, element) = src – open(src, element)**. Применение данной операции позволяет выделить наиболее яркие области на изображении.

- **MORPH_BLACKHAT** – «черная шляпа» («black hat»), **dst = tophat(src, element) = close(src, element)- src**. Использование указанного морфологического преобразования обеспечивает выделение наиболее темных областей.

Ниже представлен пример использования функции **morphologyEx**, показывающий применение всех перечисленных продвинутых операций. Приложение обеспечивает загрузку изображения и последовательное применение описанных морфологических преобразований.

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

using namespace cv;

const char helper[] =
    "Sample_morphologyEx.exe  <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
            *morphologyOpenWinName = "MORPH_OPEN",
            *morphologyCloseWinName = "MORPH_CLOSE",
            *morphologyGradientWinName = "MORPH_GRADIENT",
            *morphologyTopHatWinName = "MORPH_TOPHAT",
            *morphologyBlackHatWinName = "MORPH_BLACKHAT";
    Mat img, morphologyOpenImg, morphologyCloseImg,
        morphologyGradientImg, morphologyTopHatImg,
        morphologyBlackHatImg, element;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }

    // загрузка изображения
    img = imread(argv[1], 1);

    // применение морфологических операций
    element = Mat();
    morphologyEx(img, morphologyOpenImg,
                 MORPH_OPEN, element);
    morphologyEx(img, morphologyCloseImg,
                 MORPH_CLOSE, element);
    morphologyEx(img, morphologyGradientImg,
                 MORPH_GRADIENT, element);
    morphologyEx(img, morphologyTopHatImg,
                 MORPH_TOPHAT, element);
    morphologyEx(img, morphologyBlackHatImg,
```

```
                MORPH_BLACKHAT, element);

    // отображение исходного изображения
    //и результата выполнения операций
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(morphologyOpenWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(morphologyCloseWinName,
                                CV_WINDOW_AUTOSIZE);
    namedWindow(morphologyGradientWinName,
                                CV_WINDOW_AUTOSIZE);
    namedWindow(morphologyTopHatWinName,
                                CV_WINDOW_AUTOSIZE);
    namedWindow(morphologyBlackHatWinName,
                                CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, img);
    imshow(morphologyOpenWinName, morphologyOpenImg);
    imshow(morphologyCloseWinName, morphologyCloseImg);
    imshow(morphologyGradientWinName,
                                morphologyGradientImg);
    imshow(morphologyTopHatWinName, morphologyTopHatImg);
    imshow(morphologyBlackHatWinName,
                                morphologyBlackHatImg);
    waitKey();
    // закрытие окон
    destroyAllWindows();
    // осовобождение памяти
    img.release();
    morphologyOpenImg.release();
    morphologyCloseImg.release();
    morphologyGradientImg.release();
    morphologyTopHatImg.release();
    morphologyBlackHatImg.release();
    return 0;
}
```

Далее на рисунке (рис. 6) показаны результаты выполнения данной программы.

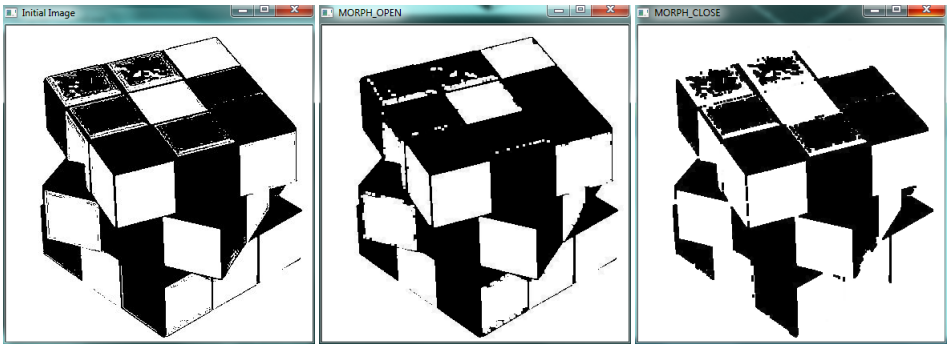


Рис. 6. Результат применения продвинутых морфологических операций (верхний ряд слева направо: исходное изображение, размыкание, замыкание; нижний ряд слева направо: морфологический градиент, «top hat», «black hat»)

2.4. Оператор Собеля

Оператор Собеля – дискретный дифференциальный оператор, вычисляющий приближенные значения производных разного порядка для функции яркости пикселей [1]. Наиболее распространенным примером практического использования является определение границ (ребер) объектов на изображении, т.е. точек резкого изменения яркости.

Данный оператор основан на свертке изображения с целочисленными фильтрами. В простейшем случае оператор построен на вычислении сверток исходного изображения с ядрами G_x и G_y, обеспечивающими вычисление первых производных по направлениям:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Данный оператор используется для приближенного вычисления градиента функции интенсивности пикселей. Применение оператора G_x позволяет определить приближенное значение первой частной производной изменения интенсивности в горизонтальном направлении, G_y – в вертикальном. На основании данной информации можно вычислить магнитуду градиента для пикселя с координатами (i, j) согласно формуле $|G^{ij}| = \sqrt{\left(G_x^{ij}\right)^2 + \left(G_y^{ij}\right)^2}$. Также используя полученные данные, можно определить направление градиента как $\theta^{ij} = \arctan\left(\frac{G_y}{G_x}\right)$.

В библиотеке OpenCV поддерживается вычисление первых, вторых, третьих и смешанных производных функции интенсивности пикселей с использованием расширенного оператора Собеля [7]. Ниже приведен прототип соответствующей программной функции.

```
void Sobel(const Mat& src, Mat& dst, int ddepth,
           int xorder, int yorder, int ksize=3,
           double scale=1, double delta=0,
           int borderType=BORDER_DEFAULT)
```

Перечислим входные параметры функции **Sobel**:

- **src** – исходное изображение.
- **dst** – результирующее изображение.
- **ddepth** – глубина результирующего изображения.
- **xorder** – порядок производной по оси Ox.
- **yorder** – порядок производной по оси Oy.
- **ksize** – размер расширенного ядра оператора Собеля. Принимает одно из значений 1, 3, 5 или 7. Во всех случаях ядро имеет размер **kSize** x **kSize**, кроме ситуации, когда **kSize=1**. При **kSize=1** ядра имеют размер 3x1 или 1x3, по существу применяется фильтр Гаусса. Указанное значение можно использовать только при вычислении первых и вторых частных производных по осям Ox и Oy. По умолчанию ядро имеет размер 3x3. Отметим, что дополнительно предусмотрено специальное значение параметра **kSize = CV_SCHARR = -1**, который соответствует ядру размера 3x3 фильтра Щарра (Scharr) и может давать более точные оценки производных по сравнению с оператором Собеля:

$$G_x = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}, G_y = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix}$$

- **scale** – опциональный параметр, который задает коэффициент масштабирования для вычисляемых значений производных. По умолчанию масштабирование не применяется.
- **delta** – опциональный параметр смещения интенсивности, добавляется перед сохранением результата в матрицу **dst**.
- **borderType** – параметр, определяющий метод дополнения границы.

Далее приведен пример выделения ребер на изображении посредством применения горизонтального и вертикального операторов Собеля и усреднения полученных градиентов по направлениям. Заметим, что предварительно применяется фильтр Гаусса (**GaussianBlur**) для удаления шумов на исходном изображении и выполняется преобразование полученного изображения в оттенки серого (**cvtColor**). В результате применения оператора Собеля получаются изображения, глубина которых

отличается от глубины исходного изображения, поэтому перед дальнейшими операциями выполняется преобразование указанных матриц в 8-битные целочисленные (**convertScaleAbs**).

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

using namespace cv;

const char helper[] =
    "Sample_Sobel.exe  <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
            *xGradWinName = "Gradient in the direction Ox",
            *yGradWinName = "Gradient in the direction Oy",
            *gradWinName = "Gradient";
    int ddepth = CV_16S;
    double alpha = 0.5, beta = 0.5;
    Mat img, grayImg, xGrad, yGrad,
        xGradAbs, yGradAbs, grad;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
    // загрузка изображения
    img = imread(argv[1], 1);
    // сглаживание помощью фильтра Гаусса
    GaussianBlur(img, img, Size(3,3),
                 0, 0, BORDER_DEFAULT);
    // преобразование в оттенки серого
    cvtColor(img, grayImg, CV_RGB2GRAY);
    // вычисление производных по двум направлениям
    Sobel(grayImg, xGrad, ddepth, 1, 0); // по Ox
    Sobel(grayImg, yGrad, ddepth, 0, 1); // по Oy
    // преобразование градиентов в 8-битные
    convertScaleAbs(xGrad, xGradAbs);
    convertScaleAbs(yGrad, yGradAbs);
    // поэлементное вычисление взвешенной
    // суммы двух массивов
    addWeighted(xGradAbs, alpha, yGradAbs, beta, 0, grad);

    // отображение результата
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(xGradWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(yGradWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(gradWinName, CV_WINDOW_AUTOSIZE);
```

```
    imshow(initialWinName, img);
    imshow(xGradWinName, xGradAbs);
    imshow(yGradWinName, yGradAbs);
    imshow(gradWinName, grad);
    waitKey();
     // закрытие окон
    destroyAllWindows();
    // осовобождение памяти
    img.release();
    grayImg.release();
    xGrad.release();
    yGrad.release();
    xGradAbs.release();
    yGradAbs.release();
    return 0;
}
```

Ниже на рисунке (рис. 7) показан результат выполнения приведенной программы на тестовом изображении (рис. 2, слева).

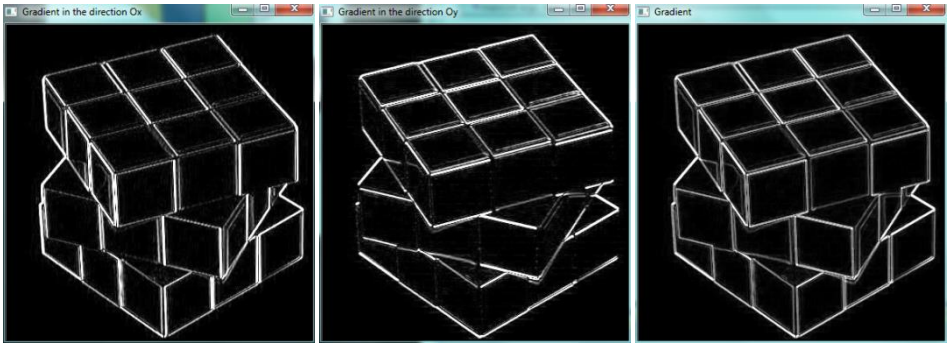


Рис. 7. Результат применения оператора Собеля в горизонтальном и вертикальном направлениях, усредненное значение проекций

Очевидно, что применение горизонтального оператора Собеля позволяет отчетливо выделить вертикальные ребра, а вертикального – горизонтальные ребра. Смесь указанных градиентов с весовыми коэффициентами, равными 0.5 обеспечивает вычисление приближенного значения градиента. Для этого используется вызов функции **`addWeighted`**.

Отметим, что библиотека OpenCV содержит функцию **`getDerivKernels`**, которая позволяет получить ядро для вычисления конкретной частной производной с определенной апертурой. Подробное описание параметров данной функции можно найти в документации [7].

2.5. Оператор Лапласа

Математически *оператор Лапласа* представляет сумму квадратов вторых частных производных функции $\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$. Дискретный аналог оператора Лапласа используется при обработке изображений, в частности, для определения ребер объектов на изображении. Ребра формируются из множества пикселей, в которых оператор Лапласа принимает нулевые значения, т.к. нули вторых производных функции соответствуют экстремальным перепадам интенсивности.

Библиотека OpenCV содержит функцию **`Laplacian`**, обеспечивающую вычисление оператора Лапласа [7].

```
void Laplacian(const Mat& src, Mat& dst, int ddepth,
               int ksize=1, double scale=1, double delta=0,
               int borderType=BORDER_DEFAULT)
```

Параметры функции:

- **`src`**, **`dst`**, **`ddepth`**, **`scale`**, **`delta`**, **`borderType`** имеют тот же смысл, что и при вызове функции **`Sobel`**.
- **`kSize`** – размер апертуры для вычисление второй производной, является положительным четным числом. При использовании значения по умолчанию **`kSize=1`** применяется апертура размером 3x3 и ядро представляется матрицей:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Приведем пример программы, которая обеспечивает поиск ребер объектов на изображения с помощью оператора Лапласа. Перед непосредственным применением оператора выполняется сглаживание с использованием фильтра Гаусса (**`GaussianBlur`**) и преобразование исходного изображения в оттенки серого (**`cvtColor`**). Заметим, что результирующие значения оператора Лапласа записываются в матрицу, глубина которой аналогично примеру с фильтром Собеля отличается от глубины исходного изображения, поэтому перед отображением выполняется преобразование полученной матрицы в 8-битную целочисленную посредством вызова функции **`convertScaleAbs`**.

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

using namespace cv;

const char helper[] =
    "Sample_Laplacian.exe  <img_file>\n\
```

```
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
               *laplacianWinName = "Laplacian";
    Mat img, grayImg, laplacianImg, laplacianImgAbs;
    int ddepth = CV_16S;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
    // загрузка изображения
    img = imread(argv[1], 1);
    // сглаживание с помощью фильтра Гаусса
    GaussianBlur(img, img, Size(3,3),
                 0, 0, BORDER_DEFAULT);
    // преобразование в оттенки серого
    cvtColor(img, grayImg, CV_RGB2GRAY);
    // применение оператора Лапласа
    Laplacian(grayImg, laplacianImg, ddepth);
    convertScaleAbs(laplacianImg, laplacianImgAbs);

    // отображение результата
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(laplacianWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, img);
    imshow(laplacianWinName, laplacianImgAbs);
    waitKey();

     // закрытие окон
    destroyAllWindows();
    // осовобождение памяти
    img.release();
    grayImg.release();
    laplacianImg.release();
    laplacianImgAbs.release();
    return 0;
}
```

Далее показан результат выделения ребер с использованием оператора Лапласа (рис. 8) на тестовом изображении, показанном ранее (рис. 2, слева).

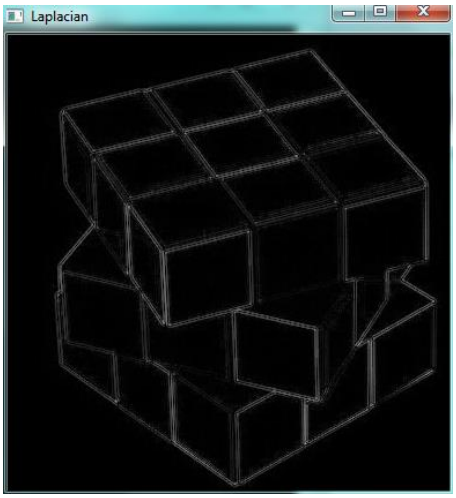


Рис. 8. Результат выделения ребер с использованием оператора Лапласа

2.6. Детектор ребер Канни

Детектор ребер Канни [4, 8, 9] предназначен для поиска границ объектов на изображении. Детектор строится на основании оператора Собеля и включает несколько этапов:

1. Удаление шума на изображении посредством применения фильтра Гаусса с ядром размера 5:

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2. Вычисление первых производных (магнитуд и направлений) функции интенсивности пикселей по горизонтальному и вертикальному направлениям посредством применения оператора Собеля с ядрами G_x и G_y (см. раздел 2.4). Направления градиентов округляются до одного из возможных значений 0^0, 45^0, 90^0, 135^0.

3. Отбор пикселей, которые потенциально принадлежат ребру с использованием процедуры non-maximum suppression [4]. Пиксели, которым соответствуют вектора производных по направлениям, являющиеся локальными максимумами, считаются потенциальными кандидатами на принадлежность ребру.

4. Двойное отсечение (гистерезис). Выделяются «сильные» и «слабые» ребра. Пиксели, интенсивность которых превышает максимальный порог, считаются пикселями, принадлежащими «сильным» ребрам.

Принимается, что пиксели с интенсивностью, входящей в интервал от минимального до максимального порогового значения, принадлежат «слабым» ребрам. Пиксели, интенсивность которых меньше минимального порога, отбрасываются из дальнейшего рассмотрения. Результирующие ребра содержат пиксели всех «сильных» ребер и те пиксели «слабых» ребер, чья окрестность содержит хотя бы один пиксель «сильных» ребер.

Детектор Канни реализован в библиотеке OpenCV [7] в виде отдельной функции, прототип которой приведен далее.

```
void Canny(const Mat& image, Mat& edges, double threshold1,
           double threshold2, int apertureSize=3,
           bool L2gradient=false)
```

Функция принимает на вход следующие параметры:

- **`image`** – одноканальное 8-битное изображение.
- **`edges`** – результирующая карта ребер, представляется матрицей, размер которой совпадает с размером исходного изображения.
- **`threshold1`**, **`threshold2`** – параметры алгоритма, пороговые значения для отсечения.
- **`apertureSize`** – размер апертуры для применения оператора Собеля.
- **`L2gradient`** – флаг, который указывает, по какой норме будет вычисляться магнитуда градиента. Принимает истинное значение, если используется норма L_2 (корень квадратный из суммы квадратов частных производных), в противном случае L_1 (сумма модулей частных производных). Как правило, нормы L_1 достаточно, и вычисляется она быстрее в связи с отсутствием вызова функции **`sqrt`**.

Приведем пример использования детектора Канни. Отметим, что перед непосредственным применением детектора выполняется размытие изображения (**`blur`**) и преобразование в оттенки серого (**`cvtColor`**).

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

using namespace cv;

const char helper[] =
    "Sample_Canny.exe  <img_file>\n\
    \t<img_file> - image file name\n";

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

```
    const char *cannyWinName = "Canny detector";
    Mat img, grayImg, edgesImg;
    double lowThreshold = 70, uppThreshold = 260;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }

    // загрузка изображения
    img = imread(argv[1], 1);
    // удаление шумов
    blur(img, img, Size(3,3));
    // преобразование в оттенки серого
    cvtColor(img, grayImg, CV_RGB2GRAY);
    // применение детектора Канни
    Canny(grayImg, edgesImg, lowThreshold, uppThreshold);

    // отображение результата
    namedWindow(cannyWinName, CV_WINDOW_AUTOSIZE);
    imshow(cannyWinName, edgesImg);
    waitKey();

     // закрытие окон
    destroyAllWindows();

    // осовобождение памяти
    img.release();
    grayImg.release();
    edgesImg.release();
    return 0;
}
```

На рисунке (рис. 9) показан результат применения детектора Канни к тестовому изображению (рис. 2, слева).

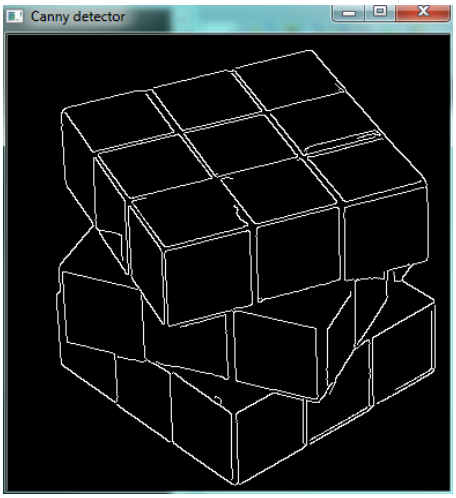


Рис. 9. Результат выделения ребер объекта с использованием детектора Канни

2.7. Вычисление гистограмм

Один из наиболее распространенных дефектов фотографических, сканерных и телевизионных изображений – слабый контраст. Дефект во многом обусловлен ограниченностью диапазона воспроизводимых яркостей. Под *контрастом* понимается разность максимального и минимального значений яркости. Контрастность изображения можно повысить за счет изменения яркости каждого элемента изображения и увеличения диапазона яркостей. Существует несколько методов, основанных на вычислении гистограммы.

Допустим, что имеется изображение в оттенках серого, интенсивность пикселей которого изменяется в пределах значений от a до b, где $a \geq 0$ и $b \leq 255$. Для изображения можно построить гистограмму со столбцами, отвечающими количеству пикселей определенной интенсивности. Такого рода гистограмма позволяет представить распределение оттенков на изображении. В общем случае под *гистограммой* понимается коллекция целочисленных значений, каждое из которых определяет количество точек, обладающих некоторым свойством или принадлежащих определенному *бину*. На практике гистограммы применяются, чтобы получить статистическую картину о распределении каких-либо данных (пикселей, векторов признаков, направлений градиента во всех точках изображения и т.п.).

В данном разделе остановимся на рассмотрении структур данных и функций OpenCV, обеспечивающих вычисление гистограмм. Ниже приведены прототипы доступных функций.

```
void calcHist(const Mat* arrays, int narrays,
```

```
              const int* channels, const Mat& mask,
              MatND& hist, int dims, const int* histSize,
              const float** ranges, bool uniform=true,
              bool accumulate=false)

void calcHist(const Mat* arrays, int narrays,
              const int* channels, const Mat& mask,
              SparseMat& hist, int dims,
              const int* histSize, const float** ranges,
              bool uniform=true, bool accumulate=false)
```

Параметры:

- **arrays** – исходные массивы данных или изображения. Должны иметь одинаковую глубину (**CV_8U** или **CV_32F**) и размер.
- **narrays** – количество исходных массивов данных.
- **channels** – массив индексов каналов в каждом входном массиве, по которым будет вычисляться гистограмма.
- **mask** – маска, на которой считается гистограмма. Опциональный параметр. Если маска не пуста, то она представляется 8-битной матрицей того же размера, что и каждый исходный массив. При построении гистограммы учитываются только элементы массивов, которые соответствуют ненулевым элементам маски. Если маска пуста, то построение гистограммы выполняется на полном наборе данных.
- **hist** – результирующая гистограмма, плотная в случае использования первого прототипа функции, разреженная – в случае второго. Для хранения плотной гистограммы используется структура данных **MatND**, для разреженной – **SparseMat**. **MatND** представляется в виде n-мерного массива, **SparseMat** – хэш-таблицей ненулевых значений [10].
- **dims** – размерность гистограммы. Параметр принимает положительные целочисленные значения, не превышающие **CV_MAX_DIMS = 32**.
- **histSize** – количество бинов по каждой размерности гистограммы.
- **ranges** – интервалы изменения значений по каждой размерности гистограммы. Если гистограмма равномерная (**uniform = true**), то для любой размерности **i** достаточно указать только нижнюю границу изменения (по существу значение, соответствующее первому бину), верхняя граница будет совпадать с **histSize[i]-1**.

- **uniform** – флаг, который определяет тип диаграммы (равномерная или нет).
- **accumulate** – флаг, указывающий на необходимость очищения гистограммы перед непосредственными вычислениями. Использование данного флага позволяет использовать одну и ту же гистограмму для нескольких множеств массивов или обновлять гистограмму во времени.

Рассмотрим пример программы, которая осуществляет построение и отображение гистограмм по каждому каналу цветного изображения. Программа получает в качестве аргументов командной строки название изображения, расщепляет полученную матрицу по каналам (**split**) и вычисляет гистограмму для каждого канала изображения (**calcHist**). Заметим, что в OpenCV каналы изображения хранятся в порядке BGR, а не в RGB. Далее выполняется нормализация гистограмм (**normalize**) для приемлемого отображения в виде ломаных.

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

using namespace cv;

const char helper[] =
    "Sample_calcHist.exe  <img_file>\n\
    \t<img_file> - image file name\n";

int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
               *histWinName = "Histogram";
    Mat img, bgrChannels[3], bHist, gHist, rHist, histImg;
    int kBins = 256; // количество бинов гистограммы
    // интервал изменения значений бинов
    float range[] = {0.0f, 256.0f};
    const float* histRange = { range };
    // равномерное распределение интервала по бинам
    bool uniform = true;
    // запрет очищения перед вычислением гистограммы
    bool accumulate = false;
    // размеры для отображения гистограммы
    int histWidth = 512, histHeight = 400;
    // количество пикселей на бин
    int binWidth = cvRound((double)histWidth / kBins);
    int i, kChannels = 3;
    Scalar colors[] = {Scalar(255, 0, 0),
                Scalar(0, 255, 0), Scalar(0, 0, 255)};
    if (argc < 2)
    {
```

```
        printf("%s", helper);
        return 1;
    }
    // загрузка изображения
    img = imread(argv[1], 1);
    // выделение каналов изображения
    split(img, bgrChannels);
    // вычисление гистограммы для каждого канала
    calcHist(&bgrChannels[0], 1, 0, Mat(), bHist, 1,
             &kBins, &histRange, uniform, accumulate);
    calcHist(&bgrChannels[1], 1, 0, Mat(), gHist, 1,
             &kBins, &histRange, uniform, accumulate);
    calcHist(&bgrChannels[2], 1, 0, Mat(), rHist, 1,
             &kBins, &histRange, uniform, accumulate);

    // построение гистограммы
    histImg = Mat(histHeight, histWidth, CV_8UC3,
                  Scalar(0, 0, 0));
    // нормализация гистограмм в соответствии с размерами
    // окна для отображения
    normalize(bHist, bHist, 0, histImg.rows,
        NORM_MINMAX, -1, Mat());
    normalize(gHist, gHist, 0, histImg.rows,
        NORM_MINMAX, -1, Mat());
    normalize(rHist, rHist, 0, histImg.rows,
        NORM_MINMAX, -1, Mat());
    // отрисовка ломаных
    for (i = 1; i < kBins; i++)
    {
        line(histImg, Point(binWidth * (i-1),
                histHeight-cvRound(bHist.at<float>(i-1))) ,
             Point(binWidth * i,
                histHeight-cvRound(bHist.at<float>(i)) ),
            colors[0], 2, 8, 0);
        line(histImg, Point(binWidth * (i-1),
                histHeight-cvRound(gHist.at<float>(i-1))) ,
            Point(binWidth * i,
                histHeight-cvRound(gHist.at<float>(i)) ),
            colors[1], 2, 8, 0);
        line(histImg, Point(binWidth * (i-1),
               histHeight-cvRound(rHist.at<float>(i-1))) ,
            Point(binWidth * i,
               histHeight-cvRound(rHist.at<float>(i)) ),
            colors[2], 2, 8, 0);
    }
    // отображение исходного изображения и гистограмм
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(histWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, img);
    imshow(histWinName, histImg);
```

```
    waitKey();

    // закрытие окон
    destroyAllWindows();
    // осовобождение памяти
    img.release();
    for (i = 0; i < kChannels; i++)
    {
        bgrChannels[i].release();
    }
    bHist.release();
    gHist.release();
    rHist.release();
    histImg.release();
    return 0;
}
```

Результат запуска программы на тестовом изображении из набора PASCAL VOC 2007 показан на рисунке (рис. 10) ниже.

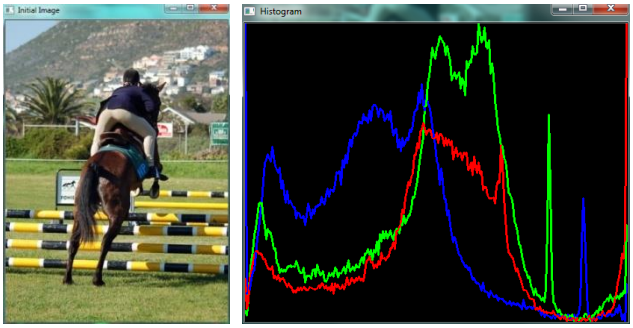


Рис. 10. Гистограммы распределения интенсивностей по каждому каналу исходного изображения

2.8. Выравнивание гистограмм

Существует три основных метода повышения контраста изображения:

- линейная растяжка гистограммы (линейное контрастирование),
- нормализация гистограммы,
- выравнивание (линеаризация или эквализация, equalization) гистограммы.

Линейная растяжка сводится к присваиванию новых значений интенсивности каждому пикселю изображения. Если интенсивности исходного изображения изменялись в диапазоне от i_1 до i_2, тогда необходимо линейно «растянуть» указанный диапазон так, чтобы значения изменялись от 0 до 255. Для этого достаточно пересчитать старые значения интенсивности f_{xy} для всех пикселей (x, y) согласно формуле $g_{xy} = a \cdot$

$f_{xy} + b$, где коэффициенты a, b просто вычисляются, исходя из того, что граница i_1 должна перейти в 0, а i_2 – в 255.

Нормализация гистограммы в отличие от предыдущего метода обеспечивает растяжку не всего диапазона изменения интенсивностей, а только его наиболее информативной части. Под информативной частью понимается набор пиков гистограммы, т.е. интенсивности, которые чаще остальных встречаются на изображении. Бины, соответствующие редко встречающимся интенсивностям, в процессе нормализации отбрасываются, далее выполняется обычная линейная растяжка получившейся гистограммы.

Выравнивание гистограмм – это один из наиболее распространенных способов. Цель выравнивания состоит в том, чтобы все уровни яркости имели бы одинаковую частоту, а гистограмма соответствовала равномерному закону распределения. Допустим, что задано изображение в оттенках серого, которое имеет разрешение $N \times M$ пикселей. Количество уровней квантования яркости пикселей (число бинов) составляет J. Тогда в среднем на каждый уровень яркости должно выпадать $n_{aver} = \frac{N \cdot M}{J}$ пикселей. Базовая математика лежит в сопоставлении двух распределений. Пусть x, y – случайные величины, описывающие изменение интенсивности пикселей на изображениях, $w_x(x)$ – плотность распределения интенсивности на исходном изображении, $w_y(y)$ – желаемая плотность распределения. Необходимо найти преобразование плотностей распределения $y = f(x)$, которое позволило бы получить желаемую плотность:

$$w_y(y) = \begin{cases} \dfrac{1}{y_{max} - y_{min}}, y_{min} \le y \le y_{max} \\ 0, \quad \text{в противном случае} \end{cases}$$

Обозначим через $F_x(x)$ и $F_y(y)$ интегральные законы распределения случайных величин x и y. Из условия вероятностной эквивалентности следует, что $F_x(x) = F_y(y)$. Распишем интегральный закон распределения по определению:

$$F_x(x) = F_y(y) = \int_{y_{min}}^{y} w_y(y)dy = \frac{y - y_{min}}{y_{max} - y_{min}}$$

Отсюда получаем, что

$$y = (y_{max} - y_{min})F_x(x) + y_{min}$$

Осталось выяснить, как оценить интегральный закон распределения $F_x(x)$. Для этого необходимо сначала построить гистограмму исходного изображения, затем нормализовать полученную гистограмму, разделив

величину каждого бина на общее количество пикселей $N \cdot M$. Значения бинов можно рассматривать как приближенное значение функции плотности распределения $w_x^*(x), 0 \le x \le 255$. Таким образом, значение интегральной функции распределения можно представить как сумму следующего вида:

$$F_x^*(x) = \sum_{j=0}^{x} w_x^*(j)$$

Построенную оценку можно использовать для вычисления новых значений интенсивности. Заметим, что перечисленные преобразования гистограмм можно применять не только ко всему изображению, но и к отдельным его частям.

В библиотеке OpenCV реализована функция **`equalizeHist`**, которая обеспечивает повышение контрастности изображения посредством выравнивания гистограммы [1, 7]. Прототип функции показан ниже.

```
void equalizeHist(const Mat& src, Mat& dst)
```

Функция работает в четыре этапа:

1. Вычисление гистограммы H исходного изображения **`src`**. Отметим, что **`src`** – 8-битное одноканальное изображение.
2. Нормализация гистограммы. Нормализация посредством деления величины каждого бина гистограммы на общее количество пикселей.
3. Построение интегральной гистограммы $H_i' = \sum_{0 \le j < i} H_j$.
4. Определение нового значения интенсивности пикселя **`dst(x,y) = H'(src(x,y))`**.

Далее приведем пример программы, обеспечивающей выравнивание гистограммы. Приложение принимает в качестве аргумента командной строки название исходного изображения. После выполнения операции выравнивания гистограммы выполняется отображение исходного изображения [1], переведенного в оттенки серого (рис. 11, слева), и изображения с выровненной гистограммой (рис. 11, справа).

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

using namespace cv;

const char helper[] =
    "Sample_equalizeHist.exe  <img_file>\n\
    \t<img_file> - image file name\n";
```

[1] Использовано изображение, входящее в состав базы PASACL VOC 2007.

```
int main(int argc, char* argv[])
{
    const char *initialWinName = "Initial Image",
               *equalizedWinName = "Equalized Image";
    Mat img, grayImg, equalizedImg;
    if (argc < 2)
    {
        printf("%s", helper);
        return 1;
    }
    // загрузка изображения
    img = imread(argv[1], 1);
    // преобразование в оттенки серого
    cvtColor(img, grayImg, CV_RGB2GRAY);
    // выравнивание гистограммы
    equalizeHist(grayImg, equalizedImg);

    // отображение исходного изображения и гистограмм
    namedWindow(initialWinName, CV_WINDOW_AUTOSIZE);
    namedWindow(equalizedWinName, CV_WINDOW_AUTOSIZE);
    imshow(initialWinName, grayImg);
    imshow(equalizedWinName, equalizedImg);
    waitKey();

    // закрытие окон
    destroyAllWindows();
    // осовобождение памяти
    img.release();
    grayImg.release();
    equalizedImg.release();
    return 0;
}
```

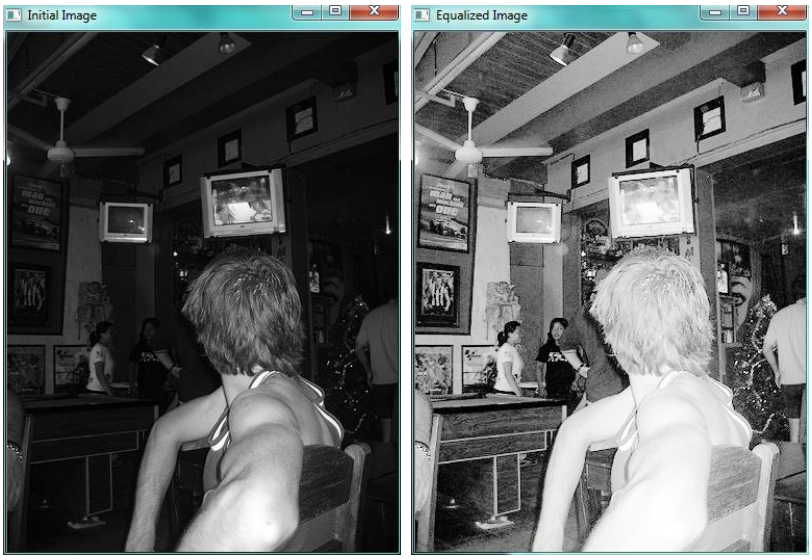


Рис. 11. Результат выравнивания гистограммы

3. Программная реализация

3.1. Разработка консольного редактора изображений

3.1.1. Требования к приложению

На данном этапе предлагается разработать простейший консольный редактор изображений, предусматривающий возможность применения всех операций, перечисленных в разделе 2. К приложению предъявляются следующие требования:

1. Организация диалога с пользователем. Предполагается вывод перечня пунктов меню (загрузка изображения и набор операций) и возможность выбора определенной операции. Отметим, что программа должна обеспечивать многократное выполнение различных операций.
2. Отображение исходного изображения.
3. Отображение результата применения операции.
4. Контроль корректности вводимых пользователем данных.

3.1.2. Структура консольного приложения

Сначала необходимо создать и настроить проект в среде Mcrosoft Visual Studio. Процедура подробно была описана ранее в работе «Сборка и установка библиотеки OpenCV. Использование библиотеки в среде Microsoft Visual Studio», поэтому в данном разделе остановимся на разработке структуры приложения.

Консольное приложение будет состоять из модуля, содержащего вызов функций OpenCV для обработки изображений, и файла исходного кода основной функции.

Для упрощения разработки введем несколько констант:

- **kMenuTabs** – количество пунктов меню (фактически число допустимых операций);

```
const int kMenuTabs = 12;
```

- **menu** – массив строк, содержащий описание пунктов меню;

```
const char* menu[] =
{
    "0 - Read image",
    "1 - Apply linear filter",
    "2 - Apply blur(...)",
    "3 - Apply medianBlur(...)",
    "4 - Apply GaussianBlur(...)",
    "5 - Apply erode(...)",
    "6 - Apply dilate(...)",
    "7 - Apply Sobel(...)",
    "8 - Apply Laplacian(...)",
    "9 - Apply Canny(...)",
    "10 - Apply calcHist(...)",
    "11 - Apply equalizeHist(...)"
};
```

- **winNames** – массив строк, содержащий названия окон, которые будут использованы при отображении результата выполнения той или иной операции;

```
const char* winNames[] =
{
    "Initial image",
    "filter2d",
    "blur",
    "medianBlur",
    "GaussianBlur",
    "erode",
    "dilate",
    "Sobel",
    "Laplacian",
    "Canny",
    "calcHist",
    "equalizeHist"
};
```

- **maxFileNameLen** – максимальная длина строки, содержащей название файла с изображением;

```
const int maxFileNameLen = 1000;
```

- **escCode** – код символа ESC для организации выхода из приложения.

```
const int escCode = 27;
```

Рассмотрим структуру основной функции. По сути, данная функция содержит цикл опроса пользователя до момента необходимости выхода из программы. Выход из программы организован посредством нажатия клавиши ESC. Цикл включает в себя выполнение нескольких операций:

1. Выбор пункта меню. Отметим, что выбор операции обработки изображения не должен быть доступен до тех пор, пока оно не будет загружено. Реализуется посредством функции **chooseMenuTab**, которая обеспечивает печать пунктов меню с использованием функции **printMenu**, контроль ввода данных и загрузку изображения с помощью функции библиотеки OpenCV **loadImage**.
2. Применение операции обработки изображения, отображение исходного и результирующего изображений. Выполняется при вызове функции **applyOperation**.
3. Ожидание нажатия произвольной клавиши для продолжения работы приложения, либо нажатия ESC для выхода из приложения. Для реализации ожидания используется встроенная функция OpenCV **waitKey**.

```
void printMenu();
void chooseMenuTab(int &activeMenuTab, Mat &srcImg);
void loadImage(Mat &srcImg);

int main(int argc, char** argv)
{
    Mat srcImg; // исходное изображение
    char ans;
    int activeMenuTab = -1;
    do
    {
        // вызов функции выбора пункта меню
        chooseMenuTab(activeMenuTab, srcImg);
        // применение операций
        applyOperation(srcImg, activeMenuTab);
        // вопрос о необходимости продолжения
        printf("Do you want to continue? ESC - exit\n");
        // ожидание нажатия клавиши
        ans = waitKey();
    }
    while (ans != escCode);
    destroyAllWindows(); // закрытие всех окон
    srcImg.release(); // освобожение памяти
    return 0;
```

```
}
```

Перейдем к реализации необходимых функций.

1. **printMenu** – функция вывода пунктов меню на консоль. Реализуется в виде прохода по элементам константного массива, содержащего описания пунктов меню.

```
void printMenu()
{
    int i = 0;
    printf("Menu items:\n");
    for (i; i < kMenuTabs; i++)
    {
        printf("\t%s\n", menu[i]);
    }
    printf("\n");
}
```

2. **chooseMenuTab** – функция выбора пункта меню. Функция выполняет запрос ввода идентификатора пункта меню до тех пор, пока не будет введено корректное значение. Заметим, что операции обработки изображения не могут быть выбраны, пока не загружено изображение. Функция возвращает индекс выбранного меню **activeMenuTab** и загруженное изображение **srcImg**.

```
void chooseMenuTab(int &activeMenuTab, Mat &srcImg)
{
    int tabIdx;
    while (true)
    {
        // print menu items
        printMenu();
        // get menu item identifier to apply operation
        printf("Input item identifier to apply \
                operation: ");
        scanf("%d", &tabIdx);
        if (tabIdx == 0)
        {
            // read image
            loadImage(srcImg);
        }
        else if (tabIdx >=1 && tabIdx < kMenuTabs &&
                 srcImg.data == 0)
        {
            // read image
            printf("The image should be read to apply\
                    operation!\n");
            loadImage(srcImg);
        }
        else if (tabIdx >=1 && tabIdx < kMenuTabs)
```

```
        {
            activeMenuTab = tabIdx;
            break;
        }
    }
}
```

3. **loadImage** – функция загрузки изображения. Функция реализована в виде цикла, в котором выполняется запрос на ввод полного имени изображения и контроль корректности операции чтения изображения. Возвращает загруженное изображение **srcImg**.

```
void loadImage(Mat &srcImg)
{
    char fileName[maxFileNameLen];
    do
    {
        printf("Input full file name: ");
        scanf("%s", &fileName);
        srcImg = imread(fileName, 1);
    }
    while (srcImg.data == 0);
    printf("The image was succesfully read\n\n");
}
```

4. **applyOperation** – функция применения операции обработки изображения. Принимает на вход исходное изображение **src** и индекс операции **operationIdx**. Функция содержит оператор множественного выбора, предполагается, что в зависимости от выбранной операции выполняется вызов той или иной функции обработки с предварительным вводом входных параметров этих функций. Далее осуществляется отображение исходного и результирующего изображений. Объявление и реализацию данной функции поместим в отдельный модуль.

```
int applyOperation(const Mat &src, const int operationIdx)
{
    char key = -1;
    Mat dst;
    switch (operationIdx)
    {
    case 1:
        {
            // TODO: "1 - Apply linear filter"
            break;
        }
    // TODO: Apply another operations (from 2 to 11)
    }
    // show initial image
    namedWindow(winNames[0], 1);
```

```
    imshow(winNames[0], src);

    // show processed image
    namedWindow(winNames[operationIdx]);
    imshow(winNames[operationIdx], dst);

    return 0;
}
```

Реализацию функций операций обработки изображений с организацией ввода необходимых параметров предоставляем читателю в качестве самостоятельной работы. Пример организации диалога с пользователем показан ниже (рис. 12).

Рис. 12. Диалог с пользователем

3.2. *Разработка редактора изображений с графическим интерфейсом с использованием функций OpenCV, реализованных на базе Qt

Разработка редактора изображений с графическим интерфейсом на базе Qt требует выполнения ряда предварительных действий:

1. Установка библиотек Qt. Для установки необходимо загрузить Qt libraries 4.x for Windows (VS 2010) с официальной страницы проекта [11] и далее следовать подсказкам инсталлятора.
2. Сборка исходных кодов библиотеки OpenCV с опцией (**`-D WITH_QT=YES`**), предусматривающей возможность последующего использования интерфейсных компонент на базе Qt. Подробнее процедура сборки OpenCV описана в лабораторной работе «Сборка и установка библиотеки OpenCV. Использование библиотеки в среде Microsoft Visual Studio». Отметим, что установочный файл содержит сборку исходных кодов, не предусматривающую возможность использования Qt-компонент.
3. Создание консольного приложения в среде Microsoft Visual Studio и установка свойств проекта, обеспечивающих использование функционала библиотеки OpenCV.

После предварительной подготовки инфраструктуры необходимо выполнить создание и настройку проекта в Visual Studio по схеме, описанной ранее. Далее можно использовать все доступные функции библиотеки OpenCV, реализованные на базе Qt [12]. Количество доступных функций ограничено, тем не менее, для создания минимального графического интерфейса предоставляемого набора вполне достаточно: создание окон с базовым набором операций просмотра (сохранение, масштабирование изображения и т.п., рис. 13), создание кнопочных элементов, полос прокрутки и др.

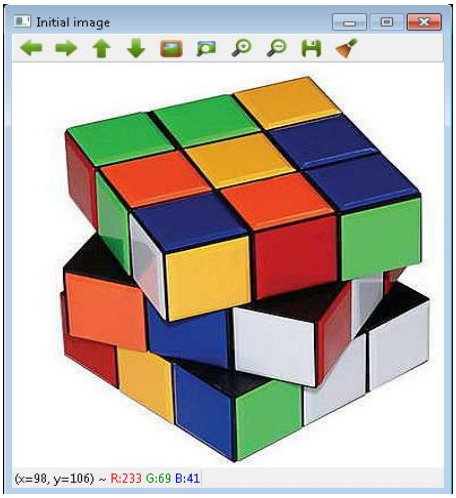


Рис. 13. Создание окна просмотра изображения с использованием функции **`namedWindow`**, реализованной на базе Qt

Читателю предлагается самостоятельно продумать структуру приложения и разработать его программную реализацию.

4. Контрольные вопросы

1. Какой эффект наблюдается в результате применения операции эрозии к бинарному изображению?
2. Какой эффект наблюдается в результате применения операции дилатации к бинарному изображению?
3. В каких ситуациях имеет смысл применять операции замыкания и размыкания?
4. Какую информацию позволяет получить гистограмма изображения?

5. Что позволяет получить применение оператора Собеля с ядром $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$?
6. Что позволяет получить применение оператора Собеля с ядром $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$?
7. Перечислите основные способы повышения контраста изображений. Приведите принципиальные отличия в вычислительных схемах.
8. Перечислите некоторые способы выделения ребер на изображении, предложенные в настоящей работе. Приведите последовательность операций, которые необходимо выполнить для реализации каждого из рассмотренных способов.

5. Дополнительные задания

1. Добавьте в разработанную структуру консольного графического редактора поддержку операций, описанных в данной лабораторной работе.
2. Добавьте в разработанный консольный графический редактор возможность сохранения изображений, которые получены в результате применения различных операций.
3. Добавьте возможность рисования геометрических примитивов в разработанный консольный редактор. Предусмотрите возможность удаления отрисованных примитивов.
4. Разработайте редактор изображений с использованием графических компонент библиотеки OpenCV, реализованных на базе Qt.

6. Литература

6.1. Основная литература

1. Bradski G., Kaehler A. Learning OpenCV Computer Vision with OpenCV Library. O' Reilly Media Publishers, 2008. 571p.
2. Шапиро Л., Стокман Дж. Компьютерное зрение. М.: Бином. Лаборатория знаний, 2006. 752с.
3. Форсайт Д., Понс Ж. Компьютерное зрение. Современный подход. М.: Изд. д. Вильямс, 2004. 465с.

4. Canny J. A computational approach to edge detection // IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol. PAMI-8. No.6. 1986. P.679-698.

6.2. Ресурсы сети Интернет

5. Официальная страница библиотеки OpenCV [http://opencv.org].
6. Лекция Интернет Университета «Фильтрация изображений» [http://www.intuit.ru/department/graphics/rastrgraph/8/].
7. Документация модуля imgproc библиотеки OpenCV [http://opencv.willowgarage.com/documentation/cpp/imgproc__image_processing.html].
8. Схема работы детектора Канни, представленная в документации OpenCV [http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html].
9. Canny Edge Detection [http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf].
10. Документация OpenCV с описанием базовых структур данных [http://opencv.willowgarage.com/documentation/cpp/basic_structures.html].
11. Страница загрузки Qt libraries [http://qt-project.org/downloads].
12. Описание функций библиотеки OpenCV, реализованных на базе Qt [http://opencv.willowgarage.com/documentation/cpp/highgui_qt_new_functions.html].

7. Приложения

7.1. Приложение А. Исходный код основной функции консольного редактора изображений

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

using namespace cv;

const int kMenuTabs = 12;
const char* menu[] =
{
    "0 - Read image",
    "1 - Apply linear filter",
    "2 - Apply blur(...)",
```

```
    "3 - Apply medianBlur(...)",
    "4 - Apply GaussianBlur(...)",
    "5 - Apply erode(...)",
    "6 - Apply dilate(...)",
    "7 - Apply Sobel(...)",
    "8 - Apply Laplacian(...)",
    "9 - Apply Canny(...)",
    "10 - Apply calcHist(...)",
    "11 - Apply equalizeHist(...)"
};
const char* winNames[] =
{
    "Initial image",
    "filter2d",
    "blur",
    "medianBlur",
    "GaussianBlur",
    "erode",
    "dilate",
    "Sobel",
    "Laplacian",
    "Canny",
    "calcHist",
    "equalizeHist"
};
const int maxFileNameLen = 1000;
const int escCode = 27;

void printMenu();
void chooseMenuTab(int &activeMenuTab, Mat &srcImg);
void loadImage(Mat &srcImg);

int main(int argc, char** argv)
{
    Mat srcImg; // исходное изображение
    char ans;
    int activeMenuTab = -1;
    do
    {
        // choose menu item
        chooseMenuTab(activeMenuTab, srcImg);
        // apply operation
        applyOperation(srcImg, activeMenuTab);
        // ask a question
        printf("Do you want to continue? ESC - exit\n");
        // waiting for key will be pressed
        ans = waitKey();
    }
    while (ans != escCode);
    destroyAllWindows(); // destroy all windows
```

```
    // release memory allocated for storing image
    srcImg.release();
    return 0;
}

void printMenu()
{
    int i = 0;
    printf("Menu items:\n");
    for (i; i < kMenuTabs; i++)
    {
        printf("\t%s\n", menu[i]);
    }
    printf("\n");
}

void loadImage(Mat &srcImg)
{
    char fileName[maxFileNameLen];
    do
    {
        printf("Input full file name: ");
        scanf("%s", &fileName);
        srcImg = imread(fileName, 1);
    }
    while (srcImg.data == 0);
    printf("The image was succesfully read\n\n");
}

void chooseMenuTab(int &activeMenuTab, Mat &srcImg)
{
    int tabIdx;
    while (true)
    {
        // print menu items
        printMenu();
        // get menu item identifier to apply operation
        printf(
           "Input item identifier to apply operation: ");
        scanf("%d", &tabIdx);
        if (tabIdx == 0)
        {
            // read image
            loadImage(srcImg);
        }
        else if (tabIdx >=1 && tabIdx < kMenuTabs &&
                 srcImg.data == 0)
        {
            // read image
            printf("The image should be read\
```

```
                to apply operation!\n");
            loadImage(srcImg);
        }
        else if (tabIdx >=1 && tabIdx < kMenuTabs)
        {
            activeMenuTab = tabIdx;
            break;
        }
    }
}
```

7.2. Приложение Б. Исходный код заголовочного файла функционального модуля консольного редактора изображений

```
#ifndef __FUNCTIONS_H__
#define __FUNCTIONS_H__

#include <opencv2/opencv.hpp>

using namespace cv;

extern const int kMenuTabs;
extern const char* winNames[];

// TODO: добавить прототипы функций, реализующих
//       другие операции
int applyMedianBlur(const Mat &src, Mat &dst);

int applyOperation(const Mat &src, const int operationIdx);

#endif
```

7.3. Приложение В. Исходный код функционального модуля консольного редактора изображений

```
#include "functions.h"

int applyMedianBlur(const Mat &src, Mat &dst)
{
    int kSize = 3, minDim = -1;
    minDim = min(src.size().height, src.size().width);
    do
    {
        printf("Set kernel size (odd, > 3, \
               < min(img.width, img.height)): ");
        scanf("%d", &kSize);
    }
    while (kSize < 3 && kSize > minDim && kSize %2 == 0);
```

```
    medianBlur(src, dst, kSize);
    return 0;
}

int applyOperation(const Mat &src, const int operationIdx)
{
    char key = -1;
    Mat dst;
    switch (operationIdx)
    {
    case 1:
        {
            // TODO: "1 - Apply linear filter"
            break;
        }
    case 2:
        {
            // TODO: "2 - Apply blur(...)"
            break;
        }
    case 3:
        {
            // "3 - Apply medianBlur(...)"
            applyMedianBlur(src, dst);
            break;
        }
    case 4:
        {
            // TODO: "4 - Apply GaussianBlur(...)"
            break;
        }
    case 5:
        {
            // TODO: "5 - Apply erode(...)"
            break;
        }
    case 6:
        {
            // TODO: "6 - Apply dilate(...)"
            break;
        }
    case 7:
        {
            // TODO: "7 - Apply Sobel(...)"
            break;
        }
    case 8:
        {
            // TODO: "8 - Apply Laplacian(...)"
            break;
```

```
        }
    case 9:
        {
            // TODO: "9 - Apply Canny(...)"
            break;
        }
    case 10:
        {
            // TODO: "10 - Apply calcHist(...)"
            break;
        }
    case 11:
        {
            // TODO: "11 - Apply equalizeHist(...)"
            break;
        }
    }
    // show initial image
    namedWindow(winNames[0], 1);
    imshow(winNames[0], src);

    // show processed image
    namedWindow(winNames[operationIdx]);
    imshow(winNames[operationIdx], dst);
    return 0;
}
```