Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

МЕТОДЫ КОМПЬЮТЕРНОГО ЗРЕНИЯ

**Учебно-методическое пособие для студентов специальностей
1-40 01 01 «Программное обеспечение
информационных технологий»,
1-98 01 03 «Программное обеспечение
информационной безопасности мобильных систем»**

Минск 2021

УДК 004.93(075.8)(0.034)
ББК 32.973-01я73
М

Рассмотрено и рекомендовано к изданию редакционно-издательским советом Белорусского государственного технологического университета.

Составитель:
Сухорукова И. Г.

Рецензенты:
директор ООО «Ведекстим» *И. Ф. Саганович*;
кандидат технических наук, доцент кафедры информатики и веб-дизайна Белорусского государственного технологического университет» *О. А. Новосельская*

М

Методы компьютерного зрения: учеб.-метод. пособие для студентов специальностей 1-40 01 01 «Программное обеспечение информационных технологий», 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем» / сост. И. Г. Сухорукова. – Минск : БГТУ, 2021. – 71с.

В учебно-методическом пособии представлены теоретические аспекты базовых методов и алгоритмов обработки и анализа изображений. Рассмотрена реализация данных методов для решения основных задач компьютерного зрения средствами библиотеки алгоритмов компьютерного зрения OpenCV. Пособие предназначено для проведения лабораторных занятий по компьютерному зрению, а также для самостоятельной работы студентов ИТ-специальностей.

УДК 004.93(075.8)(0.034)
ББК 32.973-01я73

ВВЕДЕНИЕ

Ежедневно нас окружают удивительные вещи, к которым мы привыкли. Например, считывание штрих-кода при покупке, автоматическое определение лиц при фотографировании, разблокировка мобильного телефона по лицу, фильтры и маски при съемке мобильным телефоном – это лишь небольшой список повседневных приложений, при реализации которых решаются задачи компьютерного зрения. А есть еще такие отрасли, как промышленность, медицина, производство охранных систем, роботов, умных автомобилей и домов, в которых умение получать информацию из изображения является важной (если не главной) функцией.

Область компьютерного зрения стала стремительно развиваться еще с конца 1970-х г.г., когда компьютеры смогли управлять обработкой больших наборов данных, а математические основы были заложены еще раньше. Однако задачи по анализу изображений решались под нужды конкретных областей науки и техники, поэтому не всегда есть однозначный ответ, как решить ту или иную задачу. Вместо этого существует масса математических методов для решения различных строго определённых задач компьютерного зрения, где методы часто зависят от конкретных задач и редко могут быть обобщены для широкого круга использования.

В 1999 г. на конференции по компьютерному зрению и распознаванию образов корпорацией Intel был заложен фундамент открытой библиотеки алгоритмов компьютерного зрения *OpenCV*, целью которой было стимулирование исследований в области компьютерного зрения и его практическое применение в обществе. Создание библиотеки позволило собрать и реализовать несколько сотен алгоритмов компьютерного зрения, что позволило многим разработчикам без особого труда использовать математически сложные методы в своих приложениях.

Необходимо отметить, что данное пособие не рассматривает всего множества методов компьютерного зрения в силу их большого количества и специфики применения к предметным областям. Однако освоение базовых алгоритмов и методов анализа изображений позволит по-новому взглянуть на технические возможности разрабатываемых приложений, а использование функционала библиотеки *OpenCV* поможет реализовать намеченные цели.

1. БИБЛИОТЕКА АЛГОРИТМОВ КОМПЬЮТЕРНОГО ЗРЕНИЯ OPENCV

OpenCV (Open Source Computer Vision Library) является одной из наиболее популярных библиотек компьютерного зрения с открытыми исходными кодами, в состав которой входит большое количество функций обработки изображений и видео в реальном времени.

Библиотека реализована на языках *C/C++*, имеются обертки для вызова функций из языка *Python*, также существует кроссплатформенные *.NET*-обертки в составе *EmguCV*, позволяющие работать с *OpenCV* из *C#, VB, VC++, IronPython* и других языков платформы *.NET*. В настоящее время активно развивается *Java*-интерфейс в связи с портированием библиотеки на мобильные платформы. Обеспечена стабильная работа на базе операционных систем семейства Windows, Linux, MacOS, Android и iOS.

Характерной особенностью *OpenCV* является модульность архитектуры, которая предполагает наличие нескольких статических или динамических библиотек. На данный момент доступно около двух десятков модулей. В частности, наиболее используемыми являются [1]:

– *core* – модуль, содержащий объявление всех структур данных, включая базовую структуру для представления многомерного массива *Mat* и функции работы с ней;

– *imgproc* – модуль обработки изображений, который включает в себя линейную и нелинейную фильтрацию, геометрические преобразования, преобразования цветовых пространств и т. д.;

– *highgui* – модуль, позволяющий отображать изображения, проигрывать видео и создавать простые интерфейсы управления;

– *ml* – модуль, содержащий реализацию некоторых алгоритмов машинного обучения;

– *objdetect* – модуль детектирования объектов;

– *gpu* – модуль gpu-реализации некоторых алгоритмов, которые реализованы на центральном процессоре в других модулях;

– *video* – модуль анализа видео, включающий функции оценивания движения на видео, вычитания фона и слежения за объектами на последовательности кадров видеопотока;

– *features2d* – модуль выделения и сопоставления особых точек на изображениях.

1.1. Установка библиотеки OpenCV под Windows

Выделяется два основных способа установки библиотеки *OpenCV* под Windows: посредством установочного файла и из исходных кодов. Первый способ является достаточно простым, интуитивно понятным и не требует значительных усилий и сводится к распаковке архива, содержащего заголовочные файлы, исходные коды и бинарные файлы. Второй способ более предпочтителен, если интересно разобраться в том, как устроена библиотека в целом, чтобы перенять опыт распределенной разработки. При установке библиотеки из исходных кодов используется широко известная открытая утилита *CMake*. Указания по установке библиотеки из исходных кодов можно без труда найти в открытых источниках.

Рассмотрим установку библиотеки *OpenCV* посредством установочного файла. Для этого необходимо загрузить установочный файл с официальной страницы проекта, расположенной по адресу https://sourceforge.net/projects/opencvlibrary/, и запустить его, указав путь для извлечения файлов библиотеки. По окончании распаковки указанная при установке директория будет содержать необходимые заголовочные файлы (директория *build/include/opencv2*) и бинарные файлы библиотеки (*build*), а также исходные коды библиотеки (*modules*), набор примеров использования библиотечных функций (*samples*) и ряд других вложенных директорий, которые будут рассмотрены в работе по мере необходимости.

Остановимся более подробно на составе директории *build*. Она содержит несколько вложенных директорий, в частности, ×86 и ×64, в которых находятся сборки библиотеки для соответствующих архитектур. Файлы с расширением *.dll* расположены в папке *bin*. Файлы с расширением *.lib* находятся в директории с одноименным названием. Бинарные файлы, имеющие приставку d в конце названия, соответствуют *Debug*-сборке, остальные – сборке в режиме *Release*. Наряду с этим названия *lib*-файлов содержат набор цифр, которые определяют версию сборки библиотеки.

После установки библиотеки необходимо изменить переменную среды *Path*: ***Панель управления*** → ***Система*** → ***Дополнительные параметры системы*** → ***Переменные среды*** → ***Path*** → ***Изменить***. В этом окне нужно создать переменную *C:\opencv\build\x64\vc14\bin* и перезагрузить Windows.

1.2. Настройка проекта в Microsoft VS для работы с OpenCV

Необходимо создать пустое консольное приложение *Win32*. Затем следует добавить в исходные файлы пустой файл *.cpp* и настроить проект для работы с библиотекой *OpenCV*, руководствуясь следующим алгоритмом.

1. Установка путей до заголовочных файлов библиотеки OpenCV. Необходимо выполнить команду контекстного меню ***Properties***, чтобы получить доступ к настройкам проекта. На вкладке ***Configuration Properties → C/C++ → General*** (рис. 1.1) следует выбрать значение *All Configurations*, чтобы установить свойство для всех режимов компиляции (*Debug* и *Release*). В поле *Additional Include Directories* нужно указать пути до заголовочных файлов библиотеки *OpenCV*: *C:\opencv\build\include*.

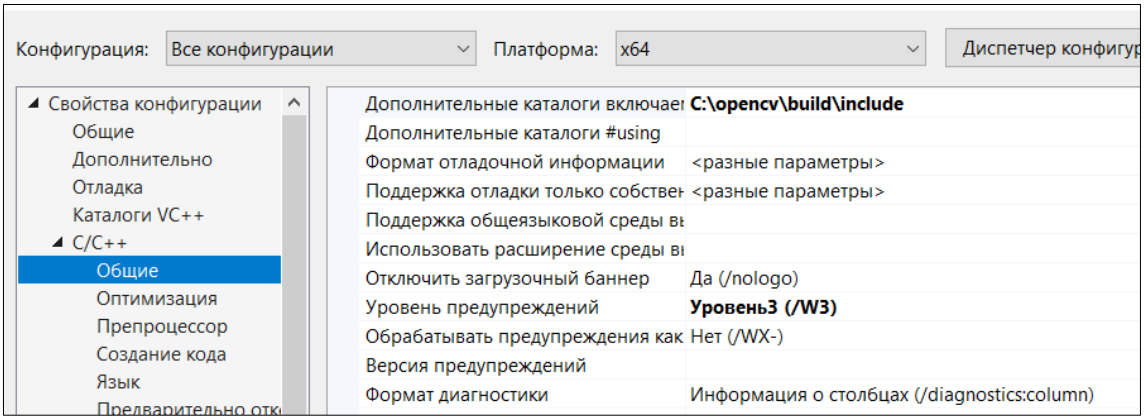


Рис. 1.1. Окно свойств проекта для установки путей до заголовочных файлов

2. **Установка путей до подключаемых библиотек (*lib*-файлов).**

Необходимо открыть вкладку ***Configuration Properties → Linker → General*** (рис.1.2) и в *поле Additional Library Directories* указать путь до *lib*-файлов *C:\opencv\build\x64\vc14\lib*.

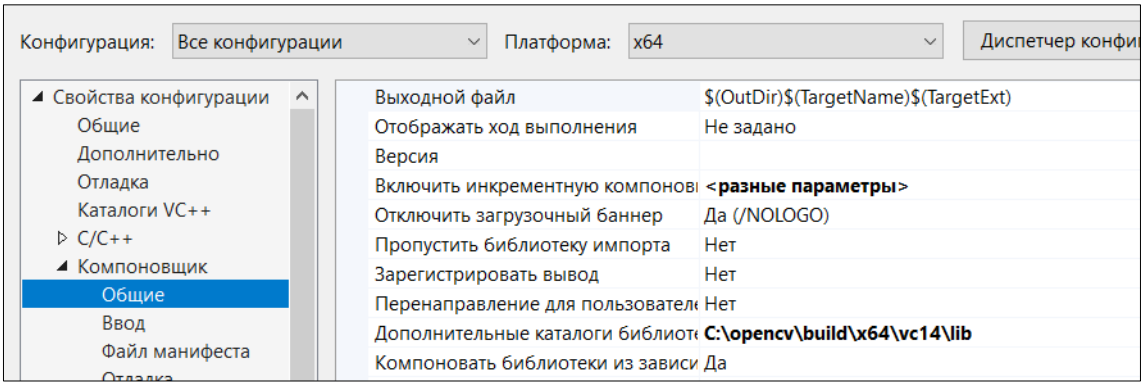


Рис. 1.2. Окно свойств проекта для установки путей до подключаемых библиотек

3. **Указание списка подключаемых программных библиотек.** Необходимо открыть вкладку *Configuration Properties→Linker→Input* (рис.3) и установить в поле *Additional Dependencies* список *lib*-файлов: *opencv_world348.lib; opencv_world348d.lib* Схема именования *lib*-файлов следующая: *opencv_<модуль><версия>d.lib*, где *<модуль>* – название подключаемого модуля, *<версия>* – версия библиотеки (например, *opencv_core348d.lib*).

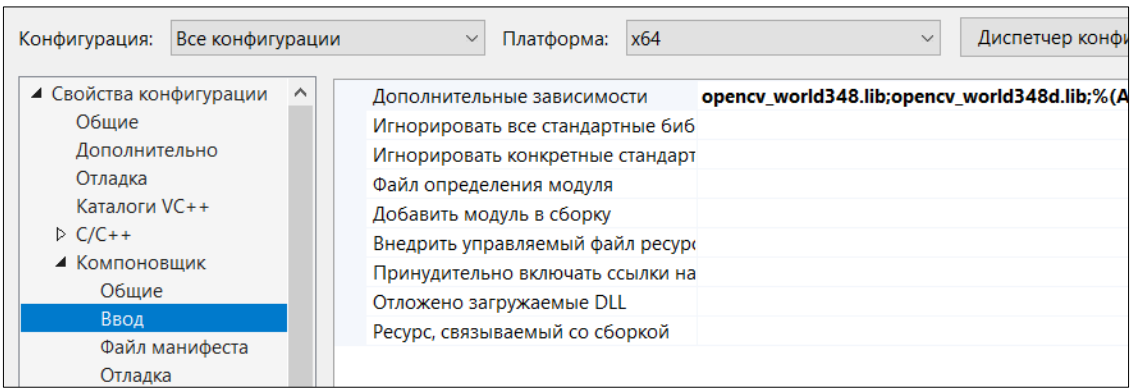


Рис. 1.3. Окно свойств проекта для установки списка подключаемых библиотек

4. **Подключение заголовочных файлов в исходном коде приложения**. Чтобы использовать функции библиотеки *OpenCV* при разработке собственных приложений, достаточно подключить заголовочный файл *opencv.hpp*, содержащий подключение большинства установленных модулей библиотеки, и пространство имен *cv*, в которое заключены все функции библиотеки:

```
#include <opencv2\opencv.hpp>
using namespace cv;
```

Если заранее известно, что в процессе разработки будет использован функционал конкретного набора модулей, то можно подключить только заголовочные файлы соответствующих модулей, например:

```
#include <opencv2\core.hpp>
#include <opencv2\objdetect.hpp>
using namespace cv;
```

2. БАЗОВЫЕ ОПЕРАЦИИ РАБОТЫ С ИЗОБРАЖЕНИЯМИ

2.1. Представление изображения

В *OpenCV* любое изображение представляет собой двумерную матрицу интенсивностей, а именно объект класса *Mat*.

Класс *Mat* имеет набор полей и методов, часть из которых представляет интерес с точки зрения работы с изображением:

– *uchar* data* – поле, содержащее значение интенсивностей для каждого пикселя изображения. Если данное поле после загрузки изображения имеет значение *NULL*, значит, произошла ошибка в процессе загрузки изображения;

– *int rows, cols* – количество строк и столбцов в матрице;

– *int channels() const* – метод, который возвращает количество каналов в изображении;

– *Size size() const* – метод для получения размера изображения (*width, height* – поля класса *Size*). При этом *rows* и *cols* совпадают с *width, height* соответственно.

OpenCV предоставляет функции для чтения файлов в самых разных графических форматах, а также изображений, снятых фото- или видеокамерой. Все они входят в состав комплекта инструментов *HighGUI*, включенного в состав *OpenCV*.

2.2. Загрузка изображения

В *OpenCV* за загрузку изображений отвечает функция ***imread***(). Ниже приведен её прототип:

```
Mat imread(
  const string& filename,      //имя входного файла
  int flags=1                  //определяет правило загрузки
);
```

При открытии файла функция ***imread***() не смотрит на расширение имени, а анализирует первые несколько байтов (так называемую сигнатуру, или «магическое число») и так определяет, какой кодек использовать [2]. Эта функция умеет читать файлы в различных

форматах, включая *BMP, DIB, JPEG, JPE, PNG, PBM, PGM, PPM, SR, RAS* и *TIFF*.

Функция ***imread***() возвращает объект типа *Mat*, а в качестве входных параметров принимает название файла *filename* и флаг *flags*, определяющий правило загрузки. Аргумент *flags* может принимать следующие значения:

– *IMREAD_COLOR* – всегда загружать в трехканальный массив, даже если в файле хранится полутоновое изображение, все равно в памяти под него будут отведены три канала, только все они будут содержать одну и ту же информацию, значение по умолчанию;

– *IMREAD_GRAYSCALE* – всегда загружать в одноканальный массив, вне зависимости от числа каналов в файле;

– *IMREAD_ANYCOLOR* – столько каналов, сколько указано в файле (но не более трех), изображение загружается «как есть»;

– *IMREAD_ANYDEPTH* – разрешить каналы глубиной больше 8, (т. е. массив в памяти будет того типа, какой указан в файле).

Помимо цветового отображения флагом можно масштабировать загружаемое изображение (подробнее об этом можно посмотреть в документации на официальном сайте *docs.opencv.org*).

2.3. Сохранение изображения

Сохранение изображения в файл осуществляется посредством вызова функции ***imwrite***():

```
bool imwrite(
  const string& filename,             // имя входного файла
  const Mat& img,                     // записываемое изображение
  const vector& params=vector()       // параметры сохранения
);
```

Функция в качестве параметров принимает название файла *filename*, в который необходимо сохранить изображение, само изображение *img*, а также вектор целочисленных параметров *params*. Указанный вектор определяет параметры сохранения в файл, специфичные для выбранного формата сохранения. Формат определяется расширением файла *filename*. Вектор параметров представляет собой единую последовательность пар <идентификатор_параметра> и <значение_параметра>. На данный момент доступны следующие идентификаторы:

– *CV_IMWRITE_JPEG_QUALITY* – качество сохранения изображения в формате *JPEG*, принимает значения от 0 до 100%, по умолчанию значение равно 95%;

– *CV_IMWRITE_PNG_COMPRESSION* – уровень сжатия изображения при сохранении в формате *PNG*, принимает значения от 0 до 9 (чем больше уровень, тем мельче результирующее изображение), по умолчанию используется уровень сжатия, равный 3;

– *CV_IMWRITE_PXM_BINARY* – флаг, который определяет тип хранения (бинарный или нет) изображения в форматах *PPM, PGM, PBM*, принимает значение 0 или 1, по умолчанию используется 1.

Функция возвращает значение *true*, если сохранение завершилось успешно, в противном случае должно быть возвращено *false.*

2.4. Отображение изображения

Для корректного отображения изображения необходимо последовательно выполнить три действия:

1. Создать окно для отображения с помощью функции ***namedWindow***(). Функция принимает на вход название окна, которое служит идентификатором окна, и флаги. Если окно с аналогичным идентификатором уже существует, то функция не выполняет никаких действий:

```
void namedWindow(
  const string& winname,        //идентификатор окна
  int flags                     //параметры размера окна
);
```

Параметр ***flags*** может принимать следующие значения или их комбинации, полученные в результате применения оператора *OR*:

– *CV_WINDOW_NORMAL* или *CV_WINDOW_AUTOSIZE* – первое значение позволяет пользователю вручную изменять размеры окна в процессе работы приложения, в то время как второе автоматически подгоняет размеры окна под размеры изображения, запрещая пользователю изменять размер вручную;

– *CV_WINDOW_FREERATIO* или *CV_WINDOW_KEEPRATIO* – первое значение подгоняет изображение под окно без сохранения пропорций, второе значение позволяет сохранять пропорции;

– *CV_GUI_NORMAL* или *CV_GUI_EXPANDED* – первое значение обеспечивает отрисовку окна без дополнительных компонент, таких,

например, как строка состояний и панель инструментов, второе предоставляет более новые возможности по созданию графических интерфейсов пользователя наряду с отображением изображений.

По умолчанию для окна устанавливаются свойства *CV_WINDOW_AUTOSIZE, CV_WINDOW_KEEPRATIO* и *CV_GUI_EXPANDED*.

2. Отобразить изображение посредством вызова функции ***imshow***(). Для отображения необходим идентификатор окна *winname*, в которое будет помещено изображение, а также само изображение *image*.

```
void imshow(
  const string& winname,   // идентификатор окна
  const Mat& image         // изображение, показываемое в окне
);
```

3. Дождаться нажатия какой-либо клавиши, чтобы закрыть окно. Ожидание реализуется с помощью вызова функции *waitKey*, которая принимает на вход время ожидания *delay* в миллисекундах. По умолчанию *delay*=0, что означает ожидание в течение бесконечного промежутка времени, т. е. исполнение приложения будет продолжено после нажатия какой-либо клавиши. Отметим, что отсутствие вызова указанной функции приведет к тому, что по завершении программы окно будет автоматически закрыто:

```
int waitKey(int delay=0)  // ожидание в миллисекундах (0 =
                          бесконечно)
```

Ставшие ненужными окна необходимо удалять. Для этого предназначена функция ***destroyWindow***(), принимающая один аргумент – имя, данное окну при его создании:

```
int destroyWindow(const string& name);
```

2.5. Копирование изображения

В некоторых случаях при разработке приложений необходимо работать не с самим изображением, а с его полной копией, чтобы повторно использовать исходное изображение. Напомним, что изображение представляется объектом класса *Mat*. В состав методов

данного класса входит метод ***clone***(), который решает указанную задачу:

```
Mat clone() const;
```

Есть еще пара полезных методов копирования *copyTo*: первый (см. прототип ниже) выделяет память для хранения полностью аналогичной матрицы m и копирует в нее содержимое текущей матрицы, второй метод выполняет копирование в соответствии с передаваемой маской mask (копируются только элементы, которым соответствуют ненулевые элементы маски):

```
void copyTo(Mat& m) const;
void copyTo(Mat& m, const Mat& mask) const;
```

2.6. Конвертирование изображения в другое цветовое пространство

Конвертирование изображения из одного цветового пространства в другое – это очень важная операция, так как, например, многие алгоритмы компьютерного зрения работают на изображениях в оттенках серого, в то время как исходное изображение с камеры цветное. Функция ***cvtColor***() позволяет конвертировать изображение в другое цветовое пространство:

```
void cvtColor(
  const Mat& src,     // входной массив
  Mat& dst,           // выходной массив
  int code,           // код преобразования цветов
  int dstCn=0         // число выходных каналов (0 автоматически)
);
```

Входными параметрами данной функции являются исходное изображение *src*, которое необходимо конвертировать в другое цветовое пространство, конвертированное изображение *dst*, код операции конвертирования *code* (из какого в какое пространство) и количество каналов в результирующем изображении *dstCn*. По умолчанию параметр *dstCn* принимает значение 0, это означает, что количество каналов результирующего изображения будет определено автоматически. Отметим, что в случае, когда исходное изображение

цветное, явно указывается порядок сохранения каналов (*RGB* или *BGR*). Рассмотрим лишь некоторые возможные значения, которые принимает параметр *code* (подробнее об этом можно посмотреть в документации на официальном сайте *docs.opencv.org*):

– *CV_RGB2GRAY*, *CV_GRAY2RGB* – конвертирование из *RGB*-пространства в оттенки серого (значение интенсивности вычисляется как взвешенная линейная свертка интенсивностей по всем трем каналам) и наоборот (дублирование интенсивности по трем каналам);

– *CV_BGR2XYZ*, *CV_RGB2XYZ*, *CV_XYZ2BGR*, *CV_XYZ2RGB* – преобразование из *RGB/BGR*-пространств в линейное трехкомпонентное пространство *CIE XYZ*, основанное на результатах измерения характеристик человеческого глаза, и обратно;

– *CV_BGR2HSV*, *CV_RGB2HSV*, *CV_HSV2BGR*, *CV_HSV2RGB* – коды операций преобразования из *RGB/BGR* пространств в *HSV* и наоборот;

– *CV_BGR2HLS*, *CV_RGB2HLS*, *CV_HLS2BGR*, *CV_HLS2RGB* – коды операций конвертирования из *RGB/BGR*-пространств в *HLS* и наоборот.

2.7. Выделение подобласти изображения

Существует значительный класс алгоритмов, которые работают не на всем изображении, а только на некоторой его подобласти – области интереса (*region of interests*). Поэтому на данном этапе рассмотрим функции для выделения такой подобласти. Поскольку изображение представляет собой матрицу, то для получения части изображения необходима операция выделения подматрицы в матрице *Mat*.

Класс *Mat* имеет два перегруженных оператора круглые скобки, которые и позволяют получить подобласть изображения:

```
Mat operator()( Range rowRange, Range colRange ) const;
Mat operator()( const Rect& roi ) const;
```

Разница между указанными методами лишь в том, что в первом случае на вход отдаются интервалы изменения индексов по горизонтали *rowRange* и по вертикали *colRange*, а во втором – прямоугольник *roi*, который необходимо выделить из изображения.

2.8. Бинаризация изображения

Остановимся на рассмотрении функции отсечения и бинаризации изображения. Операция отсечения предполагает, что имеется одноканальное изображение (в оттенках серого). В случае бинаризации на выходе формируется черно-белое изображение согласно правилу: если в текущем пикселе исходного изображения интенсивность меньше некоторого порога, то в результирующем изображении данный пиксель имеет черный цвет, в противном случае, белый или наоборот. Операция отсечения является более общей, и на выходе может формироваться изображение в оттенках серого, но в основе по-прежнему лежит принцип сравнения с заданным пороговым значением интенсивности. Функция ***threshold***() реализует указанную операцию отсечения для изображения *src* по порогу бинаризации *thresh* и записывает результат в матрицу *dst*:

```
double threshold(
  const Mat& src,        // входное изображение
  Mat& dst,              // выходное изображение
  double thresh,         // пороговое значение
  double maxVal,         // максимальное значение
  int thresholdType      // определяет правило отсечения
);
```

Рассмотрим возможные значения параметра *thresholdType*, который определяет правило отсечения:

THRESH_BINARY	$dst(x,y) = \begin{cases} maxVal, & \text{if } src(x,y) > thresh \\ 0, & \text{otherwise} \end{cases}$
THRESH_BINARY_INV	$dst(x,y) = \begin{cases} 0, & \text{if } src(x,y) > thresh \\ maxVal, & \text{otherwise} \end{cases}$
THRESH_TRUNC	$dst(x,y) = \begin{cases} thres, & \text{if } src(x,y) > thresh \\ src(x,y) & \text{otherwise} \end{cases}$
THRESH_TOZERO	$dst(x,y) = \begin{cases} src(x,y), & \text{if } src(x,y) > thresh \\ 0, & \text{otherwise} \end{cases}$
THRESH_TOZERO_INV	$dst(x,y) = \begin{cases} 0, & \text{if } src(x,y) > thresh \\ src(x,y), & \text{otherwise} \end{cases}$

Используя рассмотренные выше функции напишем код для вывода исходного и бинарного изображений:

```
#include <opencv2/opencv.hpp>
using namespace cv;
int main()
{
    Mat gray, grayThresh;
    const char* srcWinName = "src",
        * binaryWinName = "binary";
    // загрузка исходного изображения
    Mat src = imread("plates.jpg", 1);
    if (src.data == 0)
    {
        printf("Incorrect image name or format.\n");
        return 1;    }
    // создание копии исходного изображения
    Mat copy = src.clone();
    // конвертирование исходного изображения в оттенки серого
    cvtColor(src, gray, CV_BGR2GRAY);
    // операция отсечения с порогом 120,
    //  в  результате  которой  получаем  бинарное  изображение
grayThresh
    threshold(gray, grayThresh, 120, 255, CV_THRESH_BINARY);
    // создание окон для отображения
    namedWindow(srcWinName, 1);
    namedWindow(binaryWinName, 1);
    // отображение изображений
    imshow(srcWinName, src);
    imshow(binaryWinName, grayThresh);
    // ожидание нажатия какой-либо клавиши
    waitKey(0);
    // освобождение ресурсов
    gray.release();
    grayThresh.release();
    copy.release();
    src.release();
    return 0;
}
```

2.9. Работа с видео

Работа с видео в *OpenCV* организована при помощи класса ***VideoCapture***(). Чтобы открыть видеофайл или поток видео с камеры, достаточно создать объект указанного класса, вызвав конструктор по умолчанию и метод *open*(), которому на вход передать название файла или идентификатор устройства соответственно.

```
VideoCapture();
virtual bool open(const string& filename);
virtual bool open(int device);
```

Альтернативный вариант – в конструктор класса передать название видеофайла или идентификатор устройства. Если в системе всего одна камера, то аргумент *device* равен 0:

```
VideoCapture(const string& filename);
VideoCapture(int device);
```

Для проверки корректности открытия видеопотока, необходимо вызвать метод *isOpened*(). Если открытие завершилось успешно, то метод *isOpened*() вернет *true*.

После обработки видеопотока необходимо освободить занятые ресурсы посредством вызова метода *release*().

Ниже приведен пример кода для вывода видео с камеры:

```
using namespace cv;
using namespace std;
int main(int argc, char* argv[])
{
// видео с камеры
VideoCapture capture(0);
// проверка корректности открытия
if (!capture.isOpened()) {
cerr << "Unable to open: "  << endl;
return 0;
}
Mat frame;
while (true) {
// получение кадра видеопотока
capture >> frame;
// вывод кадра
imshow("Frame", frame);
// условие для выхода из цикла
```

```
int keyboard = waitKey(30);
if (keyboard == 'q' || keyboard == 27)
break;
}
return 0;
}
```

Следует отметить, что кадр *frame* представляет собой изображение типа *Mat* и к нему применимы все функции для работы с простыми изображениями, рассмотренные в подразделах 2.5-2.9.

Вопросы для самоконтроля

1. Объектом какого типа является изображение в *OpenCV*?
2. С какими типами файлов работает *OpenCV*?
3. Как конвертировать изображение в другие цветовые форматы?
4. Какие цветовые форматы поддерживает *OpenCV*?
5. Как получить бинарное изображение? Что такое порог бинаризации?
6. Какие изображения необходимо использовать в качестве входного в функции бинаризации?
7. Каких эффектов можно добиться, изменяя правило отсечения?

3. ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ИЗОБРАЖЕНИЙ

Основная задача компьютерного зрения – это извлечение информации из изображения, например, описание сцены, классификация объектов и их свойств. На сегодняшний день разработано сотни алгоритмов, позволяющих решить эти задачи, однако для получения наилучшего результата перед применением алгоритмов необходимо улучшить качество изображения, убрать шумы, усилить или подавить некоторые детали изображения, улучшить освещенность и контрастность. Рассмотрим наиболее распространённые методы предварительной обработки.

3.1. Свертка и линейные фильтры

Линейные фильтры представляют собой семейство фильтров, имеющих очень простое математическое описание. Вместе с тем они позволяют добиться самых разнообразных эффектов.

Будем считать, что задано исходное полутоновое изображение I, и обозначим интенсивности его пикселей I (x, y). Линейный фильтр определяется вещественнозначной функцией F, заданной на растре. Данная функция называется ядром фильтра, а сама фильтрация производится при помощи операции дискретной свертки (взвешенного суммирования) по следующей формуле [3]:

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

Иными словами, ядро фильтра (матрица с заданными коэффициентами) «умножается» на значение пикселей изображения и в результате вычисляется новое значение пикселя, зависящее от значений окружающих его пикселей. В процессе вычисления свертки выполняется проход по пикселям всего изображения и пересчет каждого пикселя. На рис. 3.1 наглядно представлена операция свертки, результатом которой является пересчет значения центрального пикселя фрагмента изображения со значением 84. Если проанализировать значения соседних пикселей, то можно сделать вывод, что на начальном изображении значение данного пикселя намного отличалось от значений соседних пикселей, а после применения операции свертки новое значение пикселя, равное 32, уже не так сильно отличается от

своих соседей, т. е. при данных значениях коэффициентов фильтра произошло сглаживание изображения.

Коэффициент нормирования необходим для того, чтобы средняя интенсивность оставалась неизменной. В примере на рис. 3.1 сумма коэффициентов ядра фильтра равна 6, поэтому коэффициент нормирования тоже равен 6. Если сумма коэффициентов ядра фильтра равна 1, то коэффициент нормирования тоже равен 1.

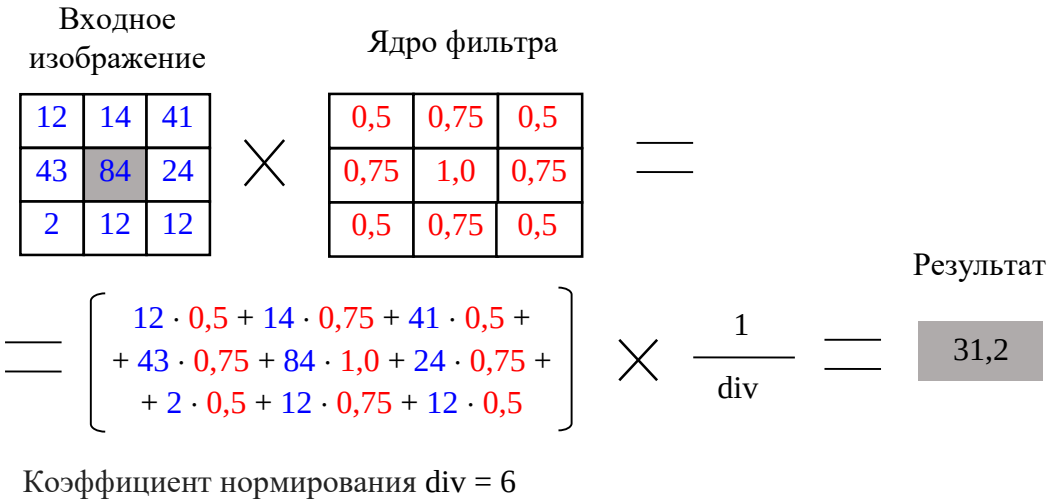


Рис. 3.1. Вычисление нового значения пикселя с помощью операции свертки

Ядро фильтра применяется к некоторой окрестности точки, размеры которой определяются размерами самого ядра фильтра. На рис. 3.1 размер фильтра равен 3×3, однако он может быть и больше. Размер фильтра, как правило, влияет на степень размытия конечного изображения.

Стоит упомянуть о граничных условиях (эта проблема актуальна для всех матричных фильтров) [4]. У верхнего левого пикселя (рис. 3.2) не существует «соседа» справа от него, следовательно, не на что умножать коэффициент матрицы.

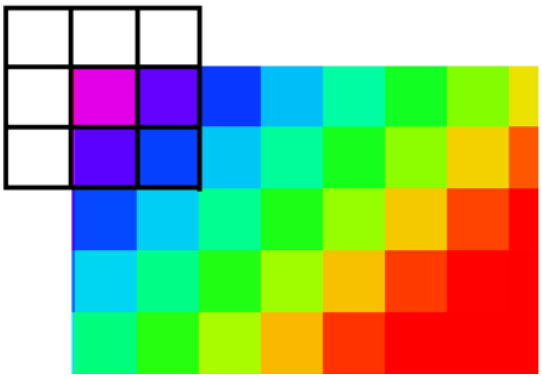
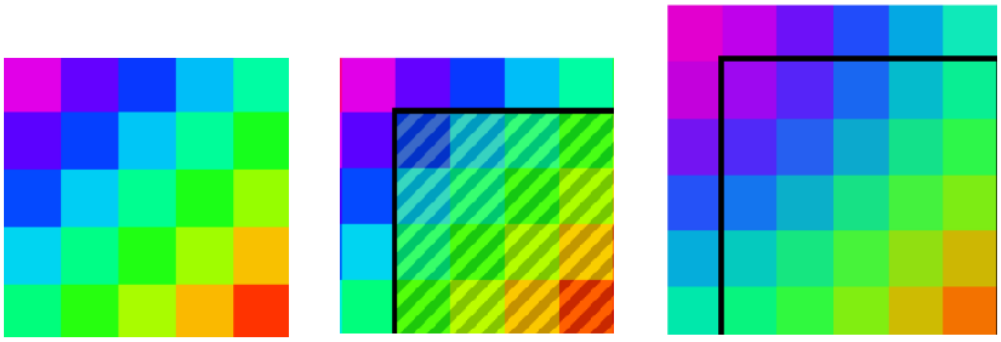


Рис. 3.2. Применение матричного фильтра на границе изображения

Эту проблему можно решить различными способами. Рассмотрим 2 наиболее распространенных способа (рис. 3.3):

- обрезать края, т. е. не проводить фильтрацию для всех граничных пикселей, на которые невозможно наложить шаблон без выхода за пределы изображения (рис. 3.3, *б*);

– доопределить окрестности граничных пикселей посредством экстраполяции (например, простым дублированием граничных пикселей) (рис. 3.3, *в*).



а *б* *в*

Рис. 3.3. Граничные условия: *а* – исходное изображение; *б* – обрезка граничных пикселей; *в* – дублирование граничных пикселей

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;

– *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*.

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

Матрицы свертки используются в различных задачах, в зависимости от коэффициентов ядра свертки можно получить эффекты сглаживания, повышение или уменьшение контрастности. Например, для повышения контраста (рис. 3.4) можно воспользоваться следующими коэффициентами ядра свертки:

$$\text{kernel} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

а *б*

Рис. 3.4. Оригинальное изображение (а) и изображение после применения операции свертки для повышения контраста (б)

Ниже приведен пример фрагмента программы с использованием функции ***filter2D***(). Представленный фрагмент программы демонстрирует применение линейного фильтра с вещественным ядром, заданным константой *kernel*:

```
const float kernelData[] = { -1.0f, -1.0f, -1.0f,
                             -1.0f, 9.0f, -1.0f,
                             -1.0f, -1.0f, -1.0f };
const Mat kernel(3, 3, CV_32FC1, (float*)kernelData);
filter2D(src, dst, -1, kernel);
```

3.2. Сглаживание изображений

Сглаживание или размытие – это одна из самых простых и часто используемых операций обработки изображений. Как правило, размытие применяется, чтобы уменьшить шумы или артефакты, которые обусловлены выбором камеры.

В библиотеке *OpenCV* реализовано несколько функций размытия изображения. Рассмотрим некоторые из них, а именно функции ***blur***(), ***boxFilter***(), ***GaussianBlur***() *и* ***medianBlur***():

```
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)
```

Изучим подробнее параметры 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 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*:

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

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

$$K = \alpha \cdot \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix},$$

где $\alpha = 1$, если ядро не нормализовано, а в случае нормализованного ядра функция ***boxFilter***() перерождается в функцию ***blur***().

Функция ***GaussianBlur*()** осуществляет размытие с помощью вычисления свертки изображения с дискретным ядром Гаусса со стандартными отклонениями, равными *sigmaX* и *sigmaY* по осям *Ox* и *Oy* соответственно. При вызове данной функции накладывается ограничение на параметр *kSize*. Ширина и высота ядра должны быть нечетными.

Следует отметить, что параметр *kSize* в функциях ***blur***(), ***box***(), ***GaussianBlur***() влияет на степень размытия изображения.

Функция ***medianBlur*()** обеспечивает размытие посредством применения медианного фильтра. Фильтр работает с матрицами различного размера, но в отличие от матрицы свёртки, размер матрицы влияет только на количество рассматриваемых пикселей.

На рис. 3.5 представлена работа медианного фильтра для размера ядра 3×3. Пиксели, которые «попадают» в матрицу вокруг текущего пикселя, сортируются, и выбирается серединное значение из отсортированного массива. По сути, определяется медиана в отсортированном наборе данных. Это значение и является выходным для текущего пикселя.

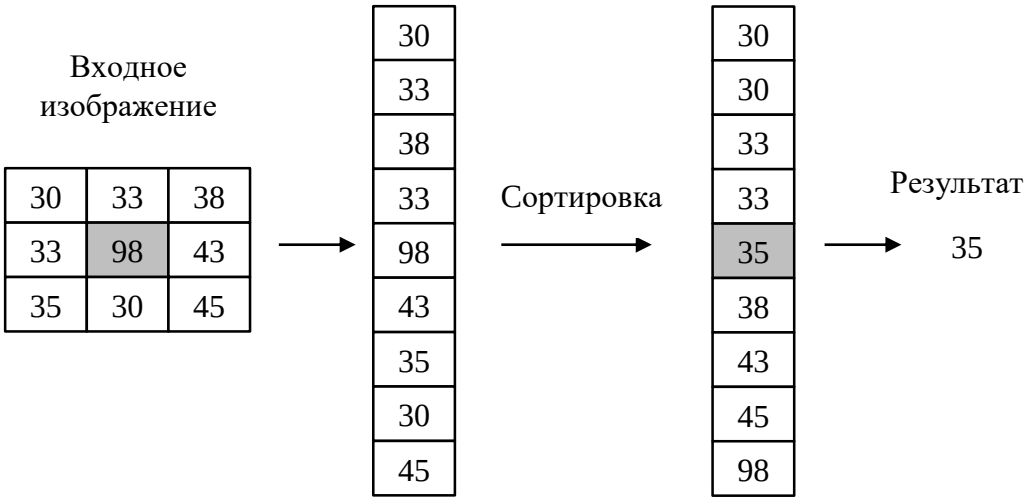


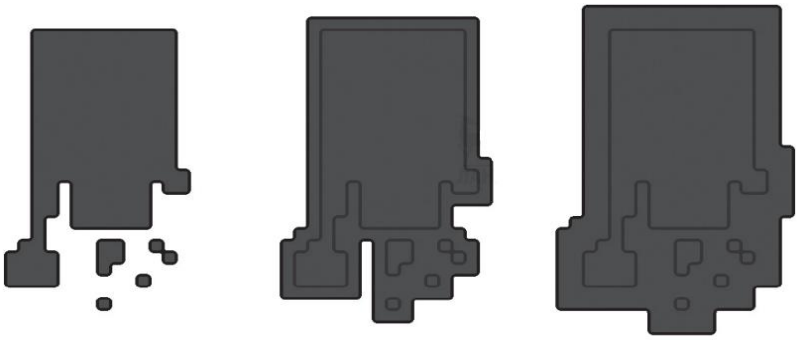
Рис. 3.5. Алгоритм работы медианного фильтра

Медианная фильтрация способна эффективно справляться с помехами, которые независимо воздействуют на отдельные пиксели. Например, с шумом «соль и перец». Этот шум представляет собой случайно возникающие черные и белые пиксели. Преимущество медианной фильтрации перед фильтрами размытия (***blur***(), ***box***(), ***GaussianBlur***()) заключается в том, что «битый» пиксель на темном фоне будет заменен на темный, а не «размазан» по окрестности.

3.3. Морфологические преобразования

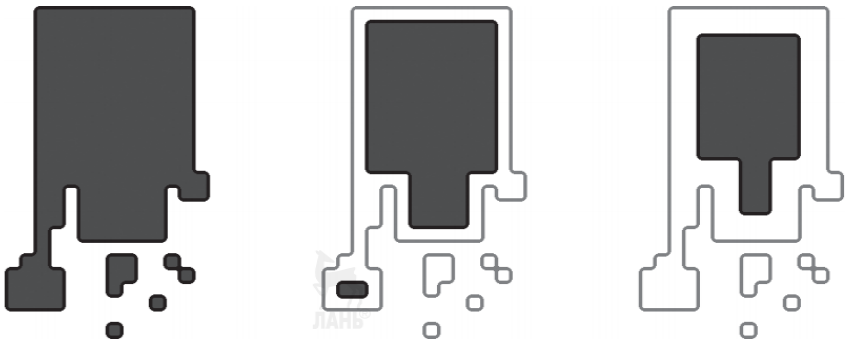
Базовые морфологические операции – дилатация (наращивание) и эрозия (сужение) – встречаются в таких разных контекстах, как устранение шумов, выделение отдельных элементов и объединение разрозненных элементов в одно изображение.

Наращивание – это свертка изображения с ядром, в котором пиксель заменяется локальным максимумом всех пикселей в области ядра. Чаще всего для наращивания используют «сплошное» квадратное ядро, а иногда круг с якорной точкой в центре. Результатом наращивания является увеличение залитых областей изображения, как показано на рис. 3.6. Эрозия – это противоположная операция, при которой вычисляется локальный минимум в области ядра (рис. 3.7).



а *б* *в*

Рис. 3.6. Результат применения дилатации: а – оригинал; б – дилатация; в – после второго применения



а *б* *в*

Рис. 3.7. Результат применения эрозии: а – оригинал; б – эрозия; в – после второго применения

Если наращивание расширяет яркую область, то эрозия, наоборот, делает ее меньше. Кроме того, наращивание заполняет вогнутости, а эрозия устраняет выступы.

В *OpenCV* эти преобразования реализуются функциями ***erode***() и ***dilate***():

```
void erode(
  const Mat& src,               // входное изображение
  const Mat& dst,               // выходное изображение
  const Mat& element,           // ядро, объект типа Mat()
  Point anchor = Point(-1,-1),  // позиция якоря
  int iterations = 1,           // сколько раз применять
  int borderType = BORDER_CONSTANT,  // экстраполяция границы
  const Scalar& borderValue = morphologyDefaultBorderValue()
  );
```

```
void dilate(
  const Mat& src,                    // входное изображение
  const Mat& dst,                    // выходное изображение
  const Mat& element,                // ядро, объект типа Mat()
  Point anchor = Point(-1,-1),       // позиция якоря
  int iterations = 1,                // сколько раз применять
  int borderType = BORDER_CONSTANT,  // экстраполяция границы
  const Scalar& borderValue = morphologyDefaultBorderValue()
  );
```

Обе функции принимают входное и выходное изображения и поддерживают обновление на месте (когда входное изображение совпадает с выходным). В качестве третьего аргумента – ядра – можно передать неинициализированный массив, тогда по умолчанию будет использовано ядро 3×3 с якорем в центре (можно создать свое ядро). Четвертый аргумент – число итераций. Если он задан и не равен 1, то в одном вызове функции операция будет применена несколько раз. Аргумент *borderType* интерпретируется как обычно, а *borderValue* – значение, которое будет использоваться для пикселей за границей изображения, если *borderType* равно *BORDER_CONSTANT*.

Запишем фрагмент кода, применяющий операции эрозии и дилатации к изображению *img*:

```
element = Mat();                   // ядро по умолчанию
erode(img, erodeImg, element);     // вычисление эрозии
dilate(img, dilateImg, element);   // вычисление дилатации
```

Простые комбинации операторов эрозии и наращивания образуют операции размыкания и замыкания. В случае размыкания сначала выполняется эрозия, затем наращивание. Размыкание часто применяется для подсчета областей в бинарном изображении. Например, мы можем сначала подвергнуть бинаризации изображение клеток на снимке с микроскопа, а затем с помощью размыкания отделить находящиеся рядом клетки и подсчитать число областей. В случае замыкания сначала выполняется наращивание, затем эрозия. Эта операция используется в более сложных алгоритмах, относящихся к связным компонентам, чтобы убрать нежелательные или обусловленные шумом сегменты. Обычно для связных компонент сначала выполняется эрозия или размыкание, чтобы устранить элементы, привнесенные шумом, а затем – замыкание, чтобы соединить расположенные рядом крупные области. И хотя конечный результат размыкания (замыкания) похож на результат применения

эрозии (дилатации), эти операции более точно сохраняют площадь связных областей. Операции размыкания и замыкания можно реализовать с помощью многоцелевой функции ***morphologyEx***(), параметры которой аналогичны параметрам функций ***erode***() и ***dilate***()

```
void morphologyEx(
   const Mat& src,              // входное изображение
   const Mat& dst,              // выходное изображение
   int op,                      // оператор
   const Mat& element,          // ядро, объект типа Mat()
   Point anchor = Point(-1,-1),      // позиция якоря
   int iterations = 1,               // сколько раз применять
   int borderType = BORDER_CONSTANT, // экстраполяция границы
   const Scalar& borderValue = morphologyDefaultBorderValue()
   );
```

Оператор *op* определяет выбранную операцию: *MOP_OPEN* – размыкание; *MOP_CLOSE* – замыкание; *MOP_GRADIENT* – морфологический градиент; *MOP_TOPHAT* – «Верх шляпы»; *MOP_BLACKHAT* – «Черная шляпа».

Ниже приведен пример кода:

```
// ядро по умолчанию
element = Mat();
// размыкание
morphologyEx(img, morphologyOpenImg, MORPH_OPEN, element);
```

Вопросы для самоконтроля

1. Что такое свертка изображения?
2. Чем результат работы медианного фильтра отличается от результата работы фильтра Гаусса?
3. Какой эффект наблюдается в результате применения операций эрозии и дилатации к бинарному изображению?
4. В каких ситуациях имеет смысл применять операции замыкания и размыкания?

4. ПРЕОБРАЗОВАНИЯ ИЗОБРАЖЕНИЙ

Функции растяжения, сжатия, деформирования и поворота изображений называются геометрическими преобразованиями. На плоскости используются два вида геометрических преобразований (рис. 4.1): с матрицей 2×3 (аффинные преобразования) и с матрицей 3×3 (перспективные преобразования, или гомографии). Можно считать, что последние описывают, как видит плоскость наблюдатель, находящийся в трехмерном пространстве, который может смотреть на нее под углом.

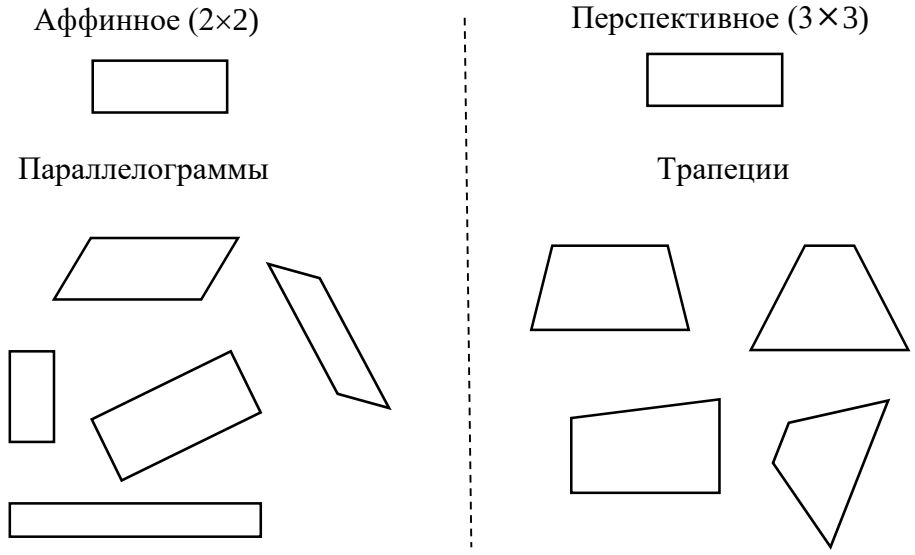


Рис. 4.1. Аффинные и перспективные преобразования

4.1. Аффинные преобразования

В *OpenCV* аффинные преобразования осуществляются функцией *warpAffine*():

```
void warpAffine(
    InputArray src,             // входное изображение
    OutputArray dst,            // выходное изображение
    InputArray M,               // матрица преобразования 2×3
    Size dsize,                 // размер выходного изображения
```

```
    int flags = INTER_LINEAR,         // интерполяция, обращение
    int borderMode = BORDER_CONSTANT, // экстраполяция пикселей
    const Scalar& borderValue = Scalar() // для константных рамок
    );
```

Аргумент *M* – матрица 2×3, описывающая преобразование. Элемент выходного массива вычисляется по элементу входного следующим образом:

$$dst(x, y) = src(M_{00}x + M_{01}y + M_{02}, M_{10}x + M_{11}y + M_{12}).$$

В общем случае координаты пикселя в правой части необязательно получаются целыми. Тогда необходимо будет интерполировать значение *dst*(*x, y*). Аргумент *flags* задает метод интерполяции (по умолчанию линейная) плюс дополнительный флаг *WARP_INVERSE_MAP* (который можно присоединить с помощью оператора *OR*). Этот флаг говорит, что нужно деформировать не *src* в *dst*, а *dst* в *src*. Последние два аргумента описывают экстраполяцию на границе и интерпретируются так же, как при свертке.

В *OpenCV* есть две функции для вычисления матрицы преобразования *M*: ***getAffineTransform***() или ***getRotationMatrix2D***().

Функция ***getAffineTransform***() используется, когда уже имеются два изображения, заведомо связанные аффинным преобразованием, и нам нужно найти его аппроксимацию:

```
Mat getAffineTransform(          // возвращает матрицу 2×3
    const Point2f* src,          // координаты трех вершин
    const Point2f* dst);         // координаты трех
                                соответствующих вершин
```

Здесь *src* и *dst* – массивы, содержащие точки (*x, y*) на плоскости. Возвращается массив, содержащий вычисленную матрицу аффинного преобразования одного массива в другой. Точки из *src* отображаются на точки из *dst* функцией ***warpAffine***() с вычисленной матрицей *M*, а все остальные точки тянутся за ними, т. е. отображение трех вершин полностью определяет отображение всех точек.

Другой способ вычисления матрицы преобразования *M* предлагает функция ***getRotationMatrix2D***(), которая вычисляет матрицу поворота вокруг произвольной точки с последующим необязательным масштабированием. Это лишь частный, хотя и важный случай аффинного преобразования, но он дает интуитивно более понятное представление:

```
Mat getRotationMatrix2D(        // возвращает матрицу 2×3
    Point2f center              // центр поворота
    double angle,               // угол поворота
    double scale );             // коэффициент масштабирования
```

Аргумент *center* задает центр поворота, а следующие два – угол поворота и коэффициент масштабирования после поворота. Функция возвращает матрицу *M* размера 2×3, содержащую числа с плавающей точкой.

4.2. Перспективные преобразования

В *OpenCV* аффинные преобразования осуществляются функцией ***warpPerspective***():

```
void warpPerspective(
    InputArray src,             // входное изображение
    OutputArray dst,            // выходное изображение
    InputArray M,               // матрица преобразования 3×3
    Size dsize,                 // размер выходного изображения
    int flags = INTER_LINEAR,         // интерполяция, обращение
    int borderMode = BORDER_CONSTANT, // экстраполяция пикселей
    const Scalar& borderValue = Scalar() // для константных рамок
    );
```

Она принимает те же аргументы, что ***warpAffine***(), с одним важным отличием: матрица *M* теперь имеет размер 3×3.

Как и в аффинном случае, имеется функция, вычисляющая матрицу преобразования по спискам соответствующих точек:

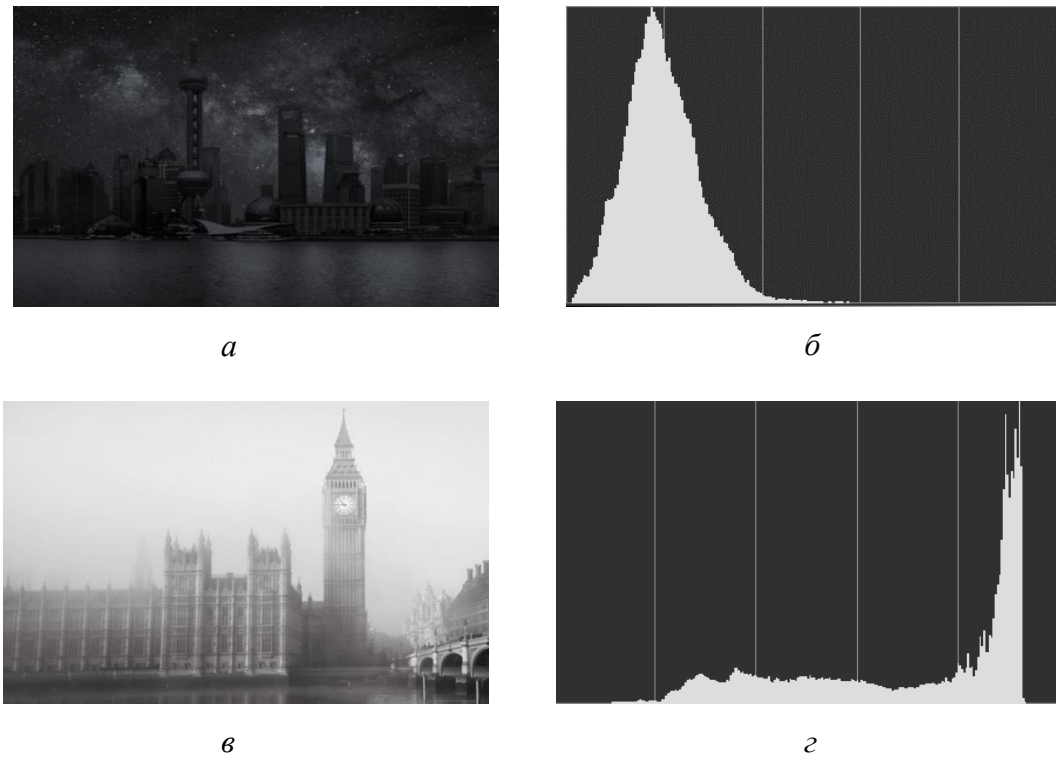
```
Mat getPerspectiveTransform(    // возвращает матрицу 3×3
    const Point2f* src,         // координаты четырех вершин
    const Point2f* dst          // координаты четырех
                               соответствующих вершин
    );
```

Массивы *src* и *dst* теперь содержат четыре, а не три точки, так что мы можем независимо задавать, в какие точки переходят все четыре угла прямоугольника (как правило) *src*. Преобразование однозначно определено образами четырех точек. Как уже было сказано, для перспективного преобразования возвращается массив 3×3. Если не считать изменение размера выходной матрицы и переход от трех к четырем управляющим точкам, то перспективное преобразование в точности аналогично аффинному.

Пример реализации перспективного преобразования можно найти в установочном каталоге *samples/cpp/warpPerspective_demo.cpp*.

4.3. Гистограмма изображения

Допустим, что имеется изображение в оттенках серого, интенсивность пикселей которого изменяется в пределах значений от 0 до 255. Для изображения можно построить гистограмму со столбцами, отвечающими количеству пикселей определенной интенсивности. Такого рода гистограмма позволяет представить распределение оттенков на изображении. Например, если в изображении преобладают темные тона, то подавляющее большинство пикселей будет сосредоточено в начале гистограммы (ближе к нулевому значению) (рис. 4.2, *а*, *б*), и наоборот, гистограмма светлого изображения смещена влево к максимальным значениям интенсивности. (см. рис. 4.2, *в*, *г*).



а *б*

в *г*

Рис. 4.2. Затемненное изображение (*а*) и соответствующая ему гистограмма (*б*); засвеченное изображение (*в*) и соответствующая ему гистограмма (*г*).

4.3.1. Вычисление гистограммы

Функция ***calcHist***() из библиотеки *OpenCV* вычисляет значения интервалов гистограммы по одному или нескольким массивам данных:

```
void calcHist(
    const Mat* images,      // массивы изображений типа 8U или 32F
    int nimages,            // число изображений в массиве
    images const int* channels, // C-массив номеров каналов
    InputArray mask,        // используемые элементы массивов
    OutputArray hist,       // выходной массив гистограммы
    int dims,         //размерность гистограмм < MAX_DIMS (32)
    const int* histSize,    // C-массив размеров по каждому
                            измерению
    const float** ranges,   // C-массив, dims-пар, описывающих
                            диапазоны интервалов
    bool uniform = true,          // true, если интервалы
                                  одинаковой ширины
    bool accumulate = false    // true - прибавлять к 'hist',
                               false - заменять );
```

Подробно с описанием многочисленных параметров функции можно ознакомиться в официальной документации, а для простоты восприятия рассмотрим фрагмент реализации данной функции с инициализированными параметрами:

```
Mat img, gray, gHist;

// количество бинов гистограммы
int kBins = 256;

// 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);

// загрузка изображения
img = imread("photo.jpg", IMREAD_COLOR);
```

```
// конвертация изображения в одноканальное
cvtColor(img, gray, COLOR_BGR2GRAY);

// вычисление гистограммы
calcHist( &gray, 1, 0, Mat(), gHist, 1, &kBins,
        &histRange, uniform, accumulate
        );
```

Для отрисовки гистограммы можно воспользоваться функциями ***line***() или ***rectangle***().

Функция ***line***() рисует линию между двух точек:

```
void line(
  InputOutputArray img,          // изображение
  Point p1,                      // начальная точка
  Point p2,                      // конечная точка
  const Scalar & color,          // цвет линии
  int thickness = 1,             // толщина линии
  int lineType = LINE_8,         // тип линии
  int shift = 0                  // количество дробных битов в
                          координатах точки
  );
```

Пример отрисовки гистограммы функцией ***line***() можно посмотреть по ссылке https://docs.opencv.org/3.4.8/d8/dbc/tutorial_histogram_calculation.html.

Функция ***rectangle*** () имеет такие же параметры, как и ***line***(), только рисует прямоугольник, в котором *p1* и *p2* – это два противоположных угла:

```
void rectangle (
  InputOutputArray img,          // изображение
  Point p1,                      // начальная точка
  Point p2,                      // конечная точка
  const Scalar & color,          // цвет линии
  int thickness = 1,             // толщина линии
  int lineType = LINE_8,         // тип линии
  int shift = 0                  // количество дробных битов в
                          координатах точки
  );
```

Пример реализации рисования гистограммы функцией ***rectangle*** () можно найти в установочном каталоге *sources\samples\cpp\ demhist.cpp*

4.3.2. Выравнивание гистограммы

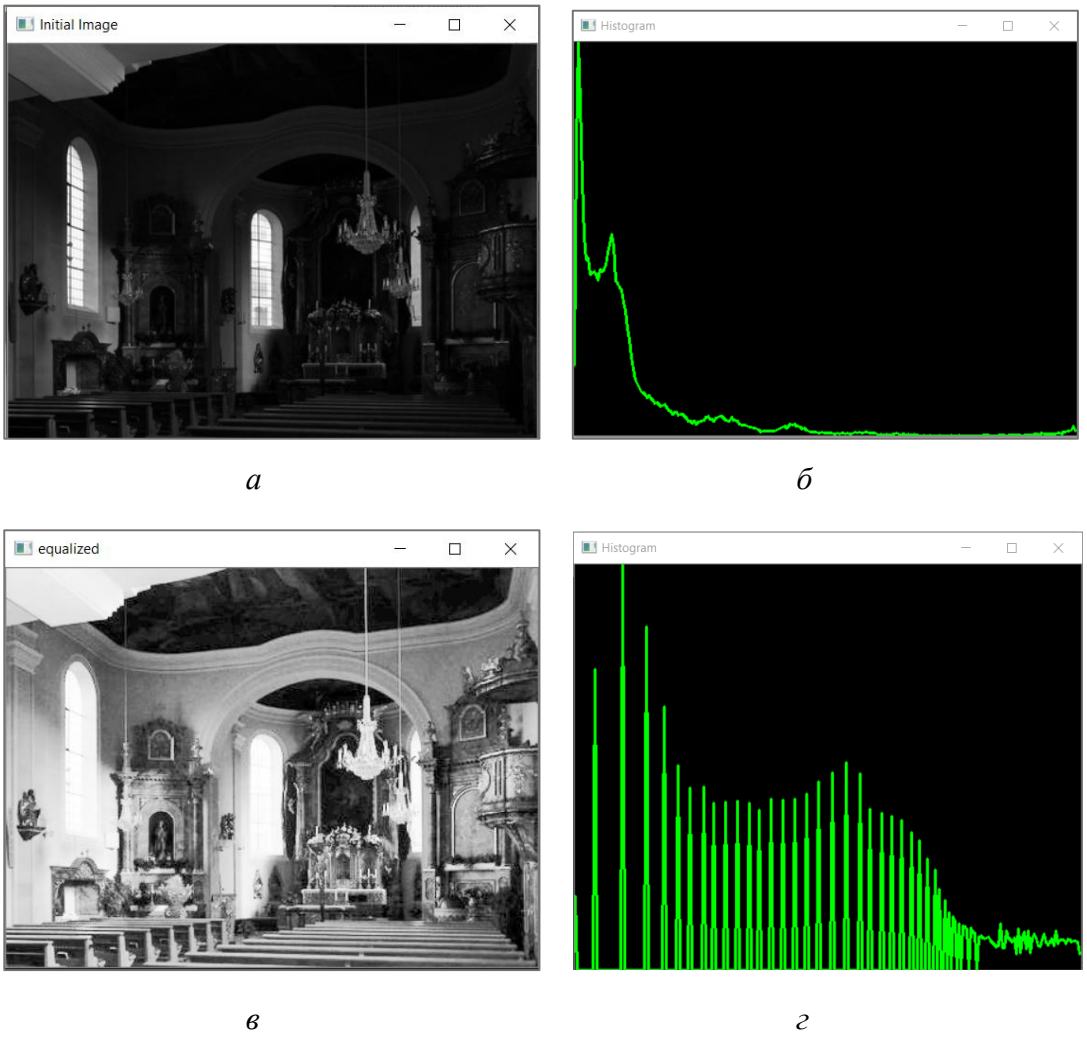
Выравнивание гистограммы применяют для увеличения контраста изображения. На рис. 4.3, *б* видно, что значения пикселей на гистограмме сосредоточены в начале диапазона, т. е. в изображении преобладают темные тона (рис. 4.3, *а*). Выравнивание гистограммы позволяет растянуть диапазон, и тем самым увеличить диапазон значений интенсивности (рис. 4.3, *в*, *г*).

Процедуру выравнивания гистограммы реализует функция ***equalizeHist***().

```
void equalizeHist(
   const InputArray src,   // входное одноканальное изображение
   OutputArray dst         // выходное одноканальное изображение
  );
```

Для цветных изображений следует разделить каналы и обрабатывать их по отдельности. Пример с разделением каналов цветного изображения можно найти на официальном сайте *OpenCV*: https://docs.opencv.org/3.4.8/d8/dbc/tutorial_histogram_calculation.html

Гистограммы применяются во многих приложениях компьютерного зрения. Например, заметив существенное изменение статистики границ и цветов между соседними кадрами, можно сделать вывод, что в видео сменилась сцена. Гистограммы границ, цветов, углов и т. д. образуют общий тип признака, который передается классификатору для распознавания объектов. Последовательности гистограмм цветов или границы применяются для поиска копий изображений.



а *б*

в *г*

Рис. 4.3. Исходное изображение (*а*) и соответствующая ему гистограмма (*б*); изображение после нормализации гистограммы (*в*) и соответствующая ему гистограмма (*г*).

Вопросы для самоконтроля

1. Чем аффинные преобразования отличаются от перспективных?
2. Что такое матрица преобразований?
3. Дайте определение гистограммы изображения.
4. Объясните назначение параметров функции ***calcHist***().
5. Для чего применяется выравнивание гистограммы?
6. Какие возможности предоставляет применение гистограмм при решении задач компьютерного зрения?

5. ВЫДЕЛЕНИЕ ГРАНИЦ (РЕБЕР) НА ИЗОБРАЖЕНИИ

В предыдущей работе было рассмотрено понятие свертки. Одно из наиболее важных свойств сверток – это вычисление производных изображения (или приближения к ним). Почему вычисление производных может быть важно для изображения? Представим, что мы хотим обнаружить края изображения. Легко заметить, что на границах яркость пикселей резко меняется. Хороший способ представить изменения – использовать производные. Большое изменение градиента указывает на значительное изменение интенсивности изображения.

5.1. Оператор Собеля

Допустим у нас есть одномерное изображение. Граница показана «скачком» интенсивности на графике (рис. 5.1).

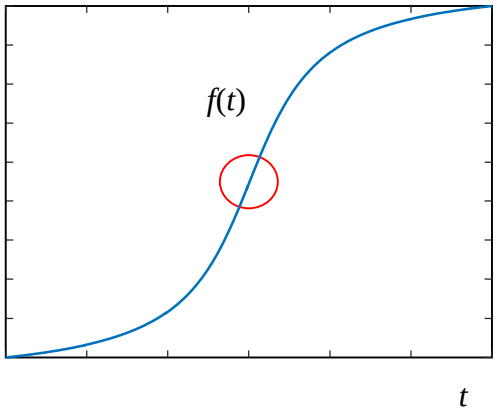


Рис. 5.1. Функция интенсивности на границе объекта

«Скачок» границы будет легче увидеть, если взять первую производную (фактически здесь появляется максимум) (рис. 5. 2).

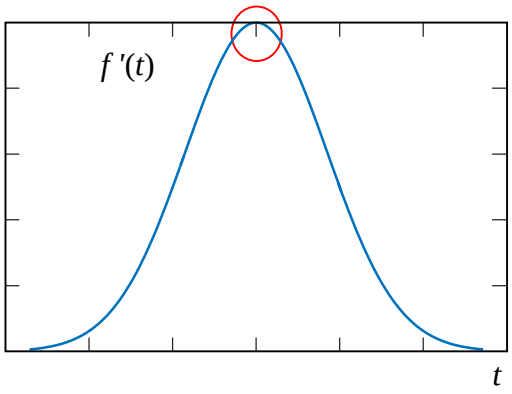


Рис. 5.2. Первая производная функции интенсивности на границе объекта

Из приведенного выше объяснения можно сделать вывод, что метод обнаружения границ в изображении может быть выполнен путем определения пикселей, в которых градиент выше, чем у соседей (или, если обобщить, выше порогового значения).

Для дифференцирования изображения используется оператор Собеля. Он вычисляет приближение градиента функции интенсивности изображения. Оператор Собеля объединяет Гауссовское сглаживание и дифференцирование.

Оператор Собеля основан на свертке изображения небольшими целочисленными фильтрами в вертикальном и горизонтальном направлениях, поэтому его относительно легко вычислять. Оператор использует ядра 3×3, с которыми свертывают исходное изображение для вычисления приближенных значений производных по горизонтали и по вертикали:

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

Применение оператора G_x позволяет определить приближенное значение первой частной производной изменения интенсивности в горизонтальном направлении, а G_y – в вертикальном. На основании данной информации можно вычислить магнитуду градиента для пикселя с координатами (i, j) согласно формуле

$$\left|G^{ij}\right| = \sqrt{\left(G_x^{\,ij}\right)^2 + \left(G_y^{\,ij}\right)^2}.$$

Также используя полученные данные, можно определить направление градиента как $\theta^{ij} = \arctan\left(\frac{G_y}{G_x}\right)$.

В *OpenCV* оператор Собеля реализуется функцией ***Sobel***():

```
void Sobel(
    InputArray src,         // входное изображение
    OutputArray dst,        // выходное изображение
    int ddepth,             // глубина выходного изображения
    int xorder,             // порядок частной производной по x
    int yorder,             // порядок частной производной по y
    Size ksize = 3,         // размер ядра
    double scale = 1,       // коэффициент масштабирования
    double delta = 0,                   // смещение
    int borderType = BORDER_DEFAULT   // экстраполяция границы
    );
```

Здесь *src* и *dst* – соответственно входное и выходное изображения. Аргумент *ddepth* задает глубину выходного изображения. Аргументы *xorder* и *yorder* присваивают порядки частных производных. Обычно задается значение 0, 1 или, самое большее, 2; 0 означает, что по этому направлению дифференцирование не производится. Аргумент *ksize* должен быть нечетным, он задает ширину и высоту фильтра. В настоящее время поддерживаются размеры до 31 включительно. Для всех случаев, кроме одного, для вычисления производной используется сепарабельное ядро размера *ksize*×*ksize*. Если *ksize* = 1, то применяется ядро 3×1 или 1×3 (по существу применяется фильтр Гаусса). Значение *ksize* = 1 можно использовать только для первой или второй производной по x или по y. Коэффициент *scale* задает коэффициент масштабирования для вычисляемых значений производных. По умолчанию масштабирование не применяется. Смещение интенсивности *delta* добавляется перед сохранением результата в матрицу *dst*.

Рассмотрим пример фрагмента программы для выделения краев на изображении с использованием оператора Собеля. В данном коде реализованы горизонтальный и вертикальный операторы Собеля и усреднения полученных градиентов по направлениям. Предварительно для удаления шумов на исходном изображении применяется фильтр Гаусса (*GaussianBlur*) 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 осуществляется преобразование указанных матриц в 8-битные целочисленные (*convertScaleAbs*):

```
// сглаживание исходного изображения img с помощью фильтра Гаусса
GaussianBlur(img, img, Size(3, 3), 0, 0, BORDER_DEFAULT);
// преобразование сглаженного изображения в оттенки серого
cvtColor(img, img, CV_RGB2GRAY);
// вычисление производных по двум направлениям
Sobel(img, xGrad, ddepth, 1, 0); // по Ox
Sobel(img, 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);
```

На рис. 5.3 показан результат выполнения оператора Собеля на тестовом изображении в горизонтальном и вертикальном направлениях, а также результирующее значение.

а *б* *с*

Рис. 5.3. Результат применения оператора Собеля в горизонтальном (*а*) и вертикальном (*б*) направлениях, усредненное значение проекций (*в*)

5.2. Оператор Лапласа

Из предыдущей темы известно, что граница на изображении характеризуется максимумом первой производной (см. рис. 5.2 на стр. 33). А что может охарактеризовать вторая производная? На рис.5.4 можно заметить, что в точке «скачка» интенсивности вторая производная равна нулю.

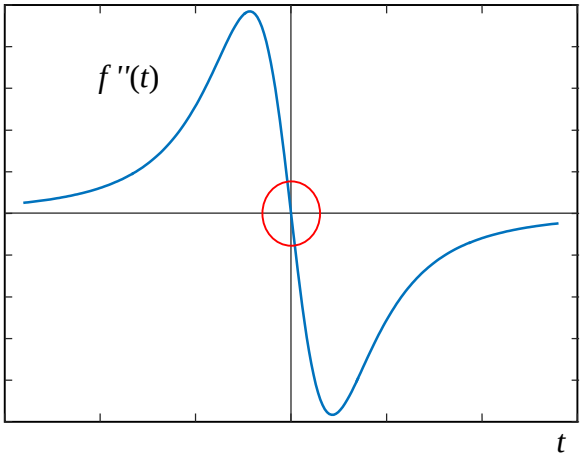


Рис. 5.4. Вторая производная в точке «скачка» интенсивности

Таким образом, вторая производная может также использоваться для обнаружения краев. Для этих целей служит оператор Лапласа, который позволяет вычислить так называемый лапласиан изображения – суммирование производных второго порядка:

$$\text{Laplase}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

Оператор Лапласа реализован в *OpenCV* функцией *Laplacian*().

Поскольку оператор Лапласа определяется в терминах вторых производных, можно предположить, что дискретная реализация работает примерно так же, как производная Собеля второго порядка. В реализации оператора Лапласа в *OpenCV* напрямую используются операторы Собеля:

```
void Laplacian(
    InputArray src,         // входное изображение
    OutputArray dst,        // выходное изображение
    int ddepth,             //глубина выходного изображения
    int ksize = 3,          // размер ядра
    double scale = 1,       // масштабный коэффициент
    double delta = 0,       // смещение
    int borderType = BORDER_DEFAULT   // экстраполяция границы
    );
```

Функция ***Laplacian***() принимает такие же аргументы, как ***Sobel***() (см. подраздел 5.1), только отсутствуют порядки производных – за ненадобностью.

5.3. Детектор границ Кэнни

Детектор ребер Кэнни также предназначен для поиска границ объектов на изображении. Детектор строится на основе оператора Собеля и включает несколько этапов:

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

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

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. подраздел 5.2). Направления градиентов округляются до одного из возможных значений 00, 450, 900, 1350.

3. Отбор пикселей, которые потенциально принадлежат ребру с использованием процедуры *non-maximum suppression*. Пиксели, которым соответствуют векторы 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.

Этот алгоритм детектирования границ Кэнни реализован в библиотеке *OpenCV* в виде отдельной функции ***Canny***(). Реализация алгоритма преобразует исходное изображение в «изображение границ»:

```
void Canny(
    InputArray image,       // входное одноканальное изображение
    OutputArray edges,      // выходное изображение границ
```

```
    double threshold1,      // нижний порог
    double threshold2,      // верхний порог
    int apertureSize = 3,  // размер ядра Собеля
    bool L2gradient = false // true = L2 - норма (более точная)
    );
```

Функция ***Canny***() получает входное изображение, обязательно одноканальное, и выходное изображение, тоже полутоновое (но в действительности оно будет бинарным). Далее задаются оба порога. Аргумент *apertureSize* задает размер ядра операторов Собеля для вычисления производных, которые вызываются из ***Canny***(). Флаг *L2gradient* указывает, по какой норме будет вычисляться магнитуда градиента. Принимает значение *true*, если используется норма L_2 (корень квадратный из суммы квадратов частных производных), в противном случае L_1 (сумма модулей частных производных). Как правило, нормы достаточно, и вычисляется она быстрее.

Ниже приведен фрагмент кода, реализующего детектор Кэнни. Перед непосредственным применением детектора выполняется размытие изображения (*blur*) и преобразование в оттенки серого (*cvtColor*):

```
    Mat img, grayImg, edgesImg;
    double lowThreshold = 70, uppThreshold = 260;
    . . .
    . . .

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

Вопросы для самоконтроля

1. Чем характеризуются границы на изображении?
2. Что означают первая и вторая производные от изображения?
3. Какие функции осуществляют поиск границ на изображении?
4. Расскажите алгоритм метода Кэнни.
5. Каким образом выделить «сильные» и «слабые» ребра в алгоритме Кэнни?
6. Как на ваш взгляд, можно использовать полученные границы при решении задач компьютерного зрения?

6. ВЫДЕЛЕНИЕ КОНТУРОВ НА ИЗОБРАЖЕНИИ

6.1. Поиск контуров

Рассмотренные ранее методы выделения границ можно использовать для нахождения пикселей, расположенных на границе между различными участками изображения, но они ничего не говорят о границах как самостоятельных сущностях. А ведь выделение контуров объектов позволит решить некоторые задачи, например, найти на изображении заданные объекты, зная их геометрию.

Контур – это список точек, представляющий кривую в изображении. Существует много способов представить кривую, все зависит от обстоятельств. В *OpenCV* контуры представляются *STL*-вектором *vector*< >, каждый элемент которого содержит информацию об одной точке на кривой. Например, в цепочке Фримена каждая точка представлена «шагом» в направлении от предыдущей точки (рис. 6.1).

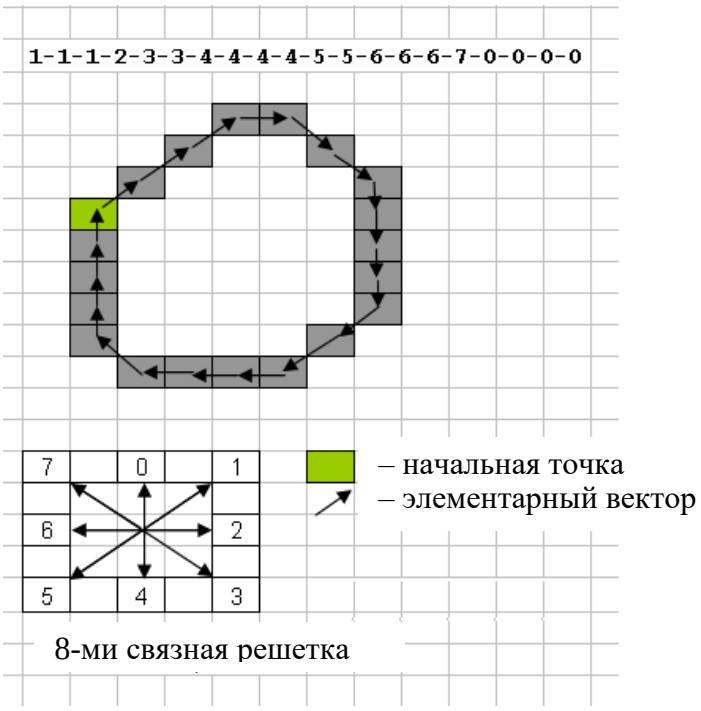


Рис. 6.1. Цепной код Фримена

В *OpenCV* есть функция ***findContours***(), которая вычисляет контуры в бинарном изображении. Она может получать на входе изображение, созданное функцией ***Canny***(), где представлены

граничные пиксели, или изображение, созданное функциями типа ***threshold***() или ***adaptiveThreshold***(), в которых границы неявно представлены как линии, разделяющие положительные и отрицательные области.

Рассмотрим функцию ***findContours***() для определения контуров объектов на бинарном (черно-белом) изображении:

```
void findContours(
const Mat& image image,    // бинарное одноканальное изображение
OutputArrayOfArrays contours,   // вектор векторов точек
OutputArray hierarchy, // факультативная информация о топологии
int mode,                  // режим возврата контуров
int method,                // метод аппроксимации
Point offset = Point()     // сдвиг точек
);

void findContours(
const Mat& image image,    // бинарное одноканальное изображение
OutputArrayOfArrays contours,   // вектор векторов точек
int mode,                       // режим возврата контуров
int method,                     // метод аппроксимации
Point offset = Point()          // сдвиг точек
);
```

Первый аргумент *image* – входное изображение, оно должно быть 8-разрядным одноканальным и интерпретируется как бинарное. Во время выполнения ***findContours***() использует этот массив как рабочую область, поэтому если изображение нужно еще для чего-то, необходимо сделать предварительно копию.

Второй аргумент *contours* – массив массивов, а на практике обычно *STL*-вектор *STL*-векторов. В него будут помещены найденные контуры. Опционально функция возвращает иерархию отношений между контурами *hierarchy*, для каждого контура *contours[i]* в *hierarchy[i][0]*, *hierarchy[i][1]*, *hierarchy[i][2]*, *hierarchy[i][3]* хранятся индексы следующего и предыдущего контуров того же уровня иерархии, индексы первого дочернего и родительского контуров соответственно. Если какого-либо элемента приведенной последовательности нет, то в соответствующей ячейке хранится отрицательное значение.

Аргумент *hierarchy* может отсутствовать (как во втором перегруженном варианте). Если он задан, то функция поместит в него массив (обычно *STL*-вектор), содержащий по одному элементу для каждого контура в *contours*.

Аргумент *mode* задает режим поиска и возврата контуров и может принимать следующие значения:

– *CV_RETR_EXTERNAL* обеспечивает восстановление только внешних контуров;

– *CV_RETR_LIST* позволяет восстанавливать все контуры без установления иерархии;

– *CV_RETR_CCOMP* обеспечивает восстановление всех контуров (внешних и внутренних) и собирает их в двухуровневую иерархию;

– *CV_RETR_TREE* предоставляет возможность получения полной иерархии контуров.

Перечислим возможные значения параметра *method*:

– *RETR_EXTERNAL* возвращает только самые внешние контуры;

– *RETR_LIST* находит все контуры и возвращает их в виде списка;

– *RETR_CCOMP* находит все контуры и организует их в виде двухуровневой иерархии, на верхнем уровне которой расположены внешние границы компонент, а на втором уровне – границы дырок;

– *RETR_TREE* находит все контуры и реконструирует полную иерархию вложенности.

Осталось сказать несколько слов о последнем параметре *offset* функции ***findContours***(). Данный параметр задает постоянное смещение каждой точки контура. Как правило, используется в случае, если поиск контура выполняется в некоторой области интереса, а отображение контуров необходимо выполнить на полном изображении.

6.2. Отображение контуров на изображении

После нахождения контуры необходимо отобразить. Такой функционал сосредоточен в функции ***drawContours***(), которая позволяет отобразить на изображении *image* контур *contours[contourIdx]* посредством отрисовки линии, имеющей цвет *color* и толщину *thickness*. Можно указать тип связности линии *lineType* (4, 8, CV_AA). Опционально также можно передать иерархию контуров *hierarchy*, максимальное количество уровней иерархии для отображения *maxLevel* и сдвиг каждой точки контура *offset*:

```
void drawContours(
Mat& image,                 // рисуются на входном изображении
InputArrayOfArrays contours,    // ветор векторов точек
int contourIdx,             // какой контур рисовать (-1 – все)
```

```
const Scalar& color,                    // цвет контуров
int thickness = 1,                      // толщина линий
int lineType = 8,                       // связность ('4' или '8')
InputArray hierarchy = noArray(),       // от findContours
int maxLevel = INT_MAX,             // максимальный уровень иерархии
Point offset = Point()              // смещение всех точек
);
```

Ниже приведен пример кода поиска контуров, из официальной документации *OpenCV*. В данной реализации с помощью ползунка задается порог бинаризации, после чего в бинаризованном изображении находятся и рисуются контуры. При изменении положения ползунка изображение обновляется:

```
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;
Mat src_gray;
int thresh = 100;
RNG rng(12345);
void thresh_callback(int, void*);
int main(int argc, char** argv)
  {
  Mat src = imread("lozhki.jpg", 1);
  if (src.empty())
    {
    cout << "Could not open or find the image!\n" << endl;
    cout << "Usage: " << argv[0] << " <Input image>" << endl;
    return -1;
    }
cvtColor(src, src_gray, COLOR_BGR2GRAY);
blur(src_gray, src_gray, Size(3, 3));
const char* source_window = "Source";
namedWindow(source_window);
imshow(source_window, src);
const int max_thresh = 255;

createTrackbar("Canny thresh:", source_window, &thresh,
max_thresh, thresh_callback);
thresh_callback(0, 0);
waitKey();
return 0;
```

```
}
void thresh_callback(int, void*)
{
Mat canny_output;
Canny(src_gray, canny_output, thresh, thresh * 2);
vector<vector<Point> > contours;
vector<Vec4i> hierarchy;
findContours(canny_output, contours, hierarchy, RETR_TREE,
CHAIN_APPROX_SIMPLE);
Mat drawing = Mat::zeros(canny_output.size(), CV_8UC3);
for (size_t i = 0; i < contours.size(); i++)
  {
  Scalar color = Scalar(rng.uniform(0, 256), rng.uniform(0,
  256), rng.uniform(0, 256));
  drawContours(drawing, contours,(int)i, color,
  2,LINE_8,hierarchy, 0);
  }
imshow("Contours", drawing);
}
```

Результат работы программы представлен на рис. 6.2.

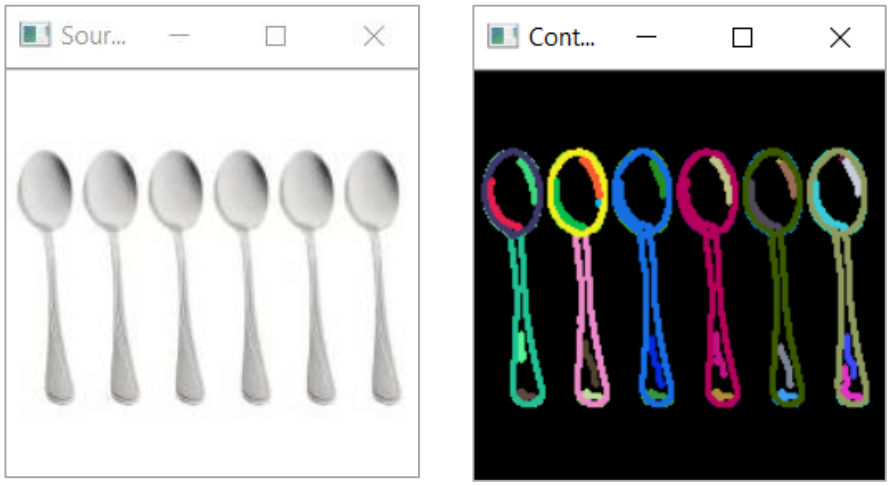


Рис. 6.2. Контуры, найденные на одноканальном изображении

Как видно из рис. 6.2 на изображении нашлись не только внешние контуры предметов, но и мелкие внутренние контуры, обусловленные игрой света и тени (в данном конкретном случае). И если задача состоит, например, в подсчете предметов по контурам, то эти внутренние контуры приведут к неверному результат. Эту задачу можно решить различными способами. Например, можно в параметре *method* указать *RETR_EXTERNAL* (возвращает только внешние

контуры) или перед поиском контуров преобразовать изображение в бинарное, чтобы исключить влияние теней или текстуры предметов. Если в рассмотренном выше коде в качестве входного изображения функции ***findContours***() использовать бинарное изображение, то результат будет гораздо лучше (рис. 6.3).

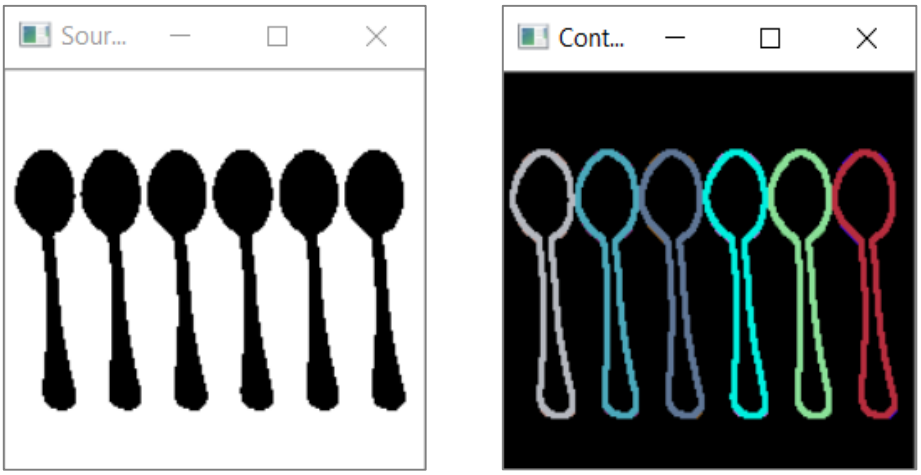


Рис. 6.3. Контуры, найденные на бинарном изображении

6.3. Преобразования Хафа для поиска линий

В нашем рукотворном мире городов и квартир преобладают прямые линии и другие простые геометрические формы (квадрат, прямоугольник, треугольник, круг). Поэтому одной из задач компьютерного зрения может быть детектирование этих линий и фигур (например, дверного проема или круглой розетки для робота, дорожных знаков для систем автомобиля). Эта задача весьма неплохо решается с помощью преобразований Хафа.

Преобразование Хафа (*Hough Transform*) –это метод для поиска линий, кругов и других простых форм на изображении.

Преобразование Хафа основывается на представлении искомого объекта в виде ***параметрического уравнения***. Параметры этого уравнения представляют фазовое пространство (так называемый аккумуляторный массив/пространство, пространство Хафа).

Затем берется двоичное изображение (например, результат работы детектора границ Кэнни). Перебираются все точки границ и делается предположение, что точка принадлежит линии искомого объекта. Таким образом, для каждой точки изображения рассчитывается нужное уравнение и получаются необходимые параметры, которые сохраняются в пространстве Хафа.

Финальным шагом является обход пространства Хафа и выбор максимальных значений, за которые «проголосовало» больше всего пикселей картинки, что и дает нам параметры для уравнений искомого объекта.

В основе теории преобразования Хафа лежит утверждение, что любая точка двоичного изображения может быть частью некоторого набора возможных прямых. Уравнение прямой можно записать:

– в декартовых координатах

$$y = a \cdot x + b;$$

– в полярных координатах

$$p = x \cdot \cos(f) + y \cdot \sin(f).$$

Прямую на плоскости можно представить:

$$r = x \cdot \cos(\theta) + y \cdot \sin(\theta),$$

где ρ (*ро*) – длина перпендикуляра, опущенного на прямую из начала координат, θ (*тета*) – угол между перпендикуляром к прямой и осью *Ox*.

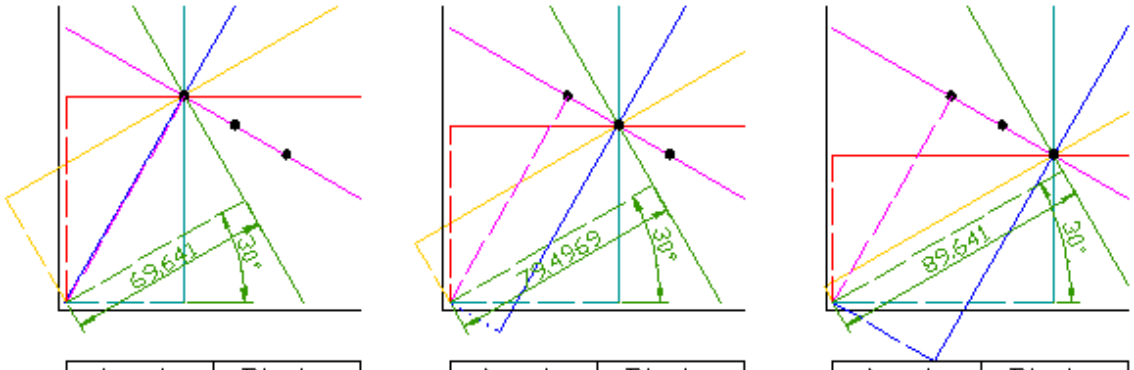
Плоскость (ρ, θ) иногда называется пространством Хафа для множества прямых в 2-мерном случае.

Через каждую точку плоскости может проходить бесконечно много прямых. Если эта точка имеет координаты (x_0, y_0), то все прямые, проходящие через неё, соответствуют уравнению

$$\rho(\theta) = x_0 \cdot \cos(\theta) + y_0 \cdot \sin(\theta).$$

На примере (рис. 6.4) исходное изображение содержит три точки. Проверим, расположены ли точки на прямой линии. Для этого через каждую точку проведено (для наглядности) только по шесть прямых, имеющих разный угол. К каждой прямой из начала координат построен перпендикуляр. Для всех прямых длина соответствующего перпендикуляра и его угол с осью абсцисс сведены в таблицу.

Данные таблицы являются результатом преобразования Хафа и могут служить основой для графического представления в пространстве Хафа.



Angle	Dist.
0	40
30	69.6
60	81.2
90	70
120	40.6
150	0.4

Angle	Dist.
0	57.1
30	79.5
60	80.5
90	60
120	23.4
150	-19.5

Angle	Dist.
0	74.6
30	89.6
60	80.6
90	50
120	6.0
150	-39.6

Рис. 6.4. Значения ρ и θ для трех точек

Уравнение прямой $\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta)$, соответствует синусоиде в пространстве Хафа (ρ, θ), которая, в свою очередь, уникальна для данной точки и однозначно ее определяет. Если эти линии (кривые), соответствующие двум точкам, накладываются друг на друга, то точка (в пространстве Хафа), где они пересекаются, соответствует прямым (в оригинальном месте изображения), которые проходят через обе точки. В общем случае ряд точек, которые формируют прямую линию, определяют синусоиды, которые пересекаются в точке параметров для этой линии. Таким образом, проблема обнаружения прямых на изображении может быть сведена к проблеме обнаружения пересекающихся кривых в пространстве Хафа. На рис. 6.5 наглядно продемонстрированы две точки пересечения в пространстве Хафа, что соответствует двум прямым в начальном изображении. Координаты точек пересечения являются параметрами уравнения прямых на искомом изображении.

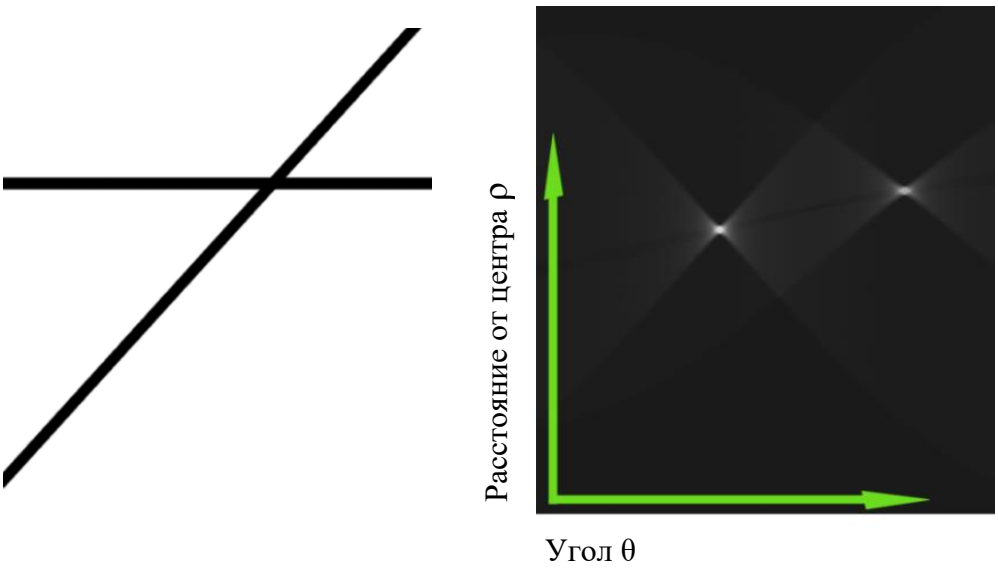


Рис. 6.5. Отображение в пространстве Хафа двух прямых.

В *OpenCV* преобразование Хафа реализовано функцией ***HoughLines***():

```
void HoughLines(
Mat& image,           // входное одноканальное изображение
OutputArray lines,    // двухканальный массив Nx1
double rho,     // разрешающая способность по rho (в пикселях)
double theta,   // разрешающая способность по theta (в радианах)
int threshold,        // ненормированный порог аккумулятора
double srn = 0,       // уточнение rho (для MHT)
double stn = 0        // уточнение theta (для MHT)
);
```

Входное изображение *image* должно быть 8-разрядным, но рассматривается оно как бинарное. Второй аргумент *lines* – массив для размещения найденных прямых – будет создан как двухканальный массив *N*×1 с плавающей точкой (число столбцов *N* будет равно числу возвращенных прямых). Каналы будут содержать значения ρ и θ для каждой найденной прямой. Аргументы *rho* и *theta* задают требуемую разрешающую способность для прямых. Аргумент *threshold* определяет порог, при котором алгоритм сообщает о найденной прямой. Другими словами, этот аргумент указывает, сколько точек (на границе внутри изображения) должно принадлежать прямой, чтобы алгоритм вернул эту прямую. Аргументы *srn* и *stn* в алгоритме стандартного преобразования Хафа (SHT) не используются, они управляют поведением его обобщения – многомасштабного

преобразования Хафа (МНТ). В МНТ они задают более высокое разрешение, с которым следует вычислять параметры прямых.

6.4. Преобразования Хафа для поиска окружностей

Преобразования Хафа используют и для нахождения окружностей, однако в случае с окружностями нужно задавать уже 3 параметра — координаты центра окружности и её радиус. Это приводит к увеличению пространства Хафа на целое измерение, что в итоге сказывается на скорости работы. Поэтому для поиска окружностей применяется т.н. градиентный метод Хафа (Hough gradient method).

В OpenCV поиск окружностей градиентным методом Хафа реализован функцией ***HoughCircles***():

```
void HoughCircles(
Mat& image,           // входное одноканальное изображение
OutputArray circles,  // трехканальный массив Nx1 или вектор Vec3f
int method,           // всегда HOUGH_GRADIENT
double dp,            // разрешающая способность аккумулятора
double minDist,       // требуемый промежуток между окружностями
double param1 = 100,       // верхний порог детектора Кэнни
double param2 = 100,       // ненормированный порог аккумулятора
int minRadius = 0,         // минимально допустимый радиус
int maxRadius = 0          // максимально допустимый радиус
);
```

Входное изображение *image* (8-битное, одноканальное, градации серого). Выходной массив *circles* хранит найденные круги, он может будет матрицей или вектором. Если используется матрица, то *circles* будет одномерным массивом типа F32C3, а его три канала служат для кодирования координат центра и радиуса окружности. Аргумент *method* всегда должен быть равен HOUGH_GRADIENT.

Аргумент *dp* – разрешающая способность аккумуляторного изображения. Он позволяет создать аккумулятор с меньшим разрешением, чем у входного изображения. Если *dp* равен 1, то разрешения совпадают; если *dp* больше 1 (например, 2), то разрешение аккумулятора будет во столько раз меньше (в 2 раза). Аргумент *minDist* задает минимальное расстояние между окружностями, при котором алгоритм будет считать их различными. Аргументы *param1* и *param2* – соответственно порог детектора Кэнни и порог аккумулятора. При вызове ***Canny***() из ***HoughLines***() первому (верхнему) порогу

присваивается значение *param1*, а второму (нижнему) – в 2 раза меньшее. Аргумент *param2* в точности аналогичен аргументу *threshold* функции ***HoughLines***(). Параметры *minRadius* и *maxRadius* – соответственно минимальный и максимальный радиусы распознаваемых окружностей.

Ниже приведен код, реализующий метод Хафа для поиска и отображения окружностей на изображении:

```
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
using namespace cv;
using namespace std;
int main(int argc, char** argv)
{
const char* filename = argc >= 2 ? argv[1] : "bike.png";
// Loads an image
Mat src = imread(filename, IMREAD_COLOR);
// Check if image is loaded fine
if (src.empty()) {
printf(" Error opening image\n");
printf(" Program Arguments: [image_name -- default %s] \n",
filename);
return EXIT_FAILURE;
}
Mat gray;
cvtColor(src, gray, COLOR_BGR2GRAY);
medianBlur(gray, gray, 5);
vector<Vec3f> circles;
HoughCircles(gray, circles, HOUGH_GRADIENT, 1,
//gray.rows / 16,  // change this value to detect circles with
different distances to each other
50, 100, 30, 30, 100   // change the last two parameters
);         // (min_radius & max_radius) to detect larger circles
for (size_t i = 0; i < circles.size(); i++)
{
Vec3i c = circles[i];
Point center = Point(c[0], c[1]);
// circle center
circle(src, center, 1, Scalar(0, 100, 100), 3, LINE_AA);
// circle outline
int radius = c[2];
```

```
circle(src, center, radius, Scalar(255, 0, 255), 3, LINE_AA);
}
const char* detected_circles = "detected circles";
namedWindow(detected_circles);
imshow(detected_circles, src);
waitKey();
return EXIT_SUCCESS;
}
```

Вопросы для самоконтроля

1. Какими признаками характеризуются на изображении границы объектов?
2. В каком виде возвращает контуры функция ***findContours***()?
3. Стоит задача найти только внешние контуры объектов. Как можно решить эту задачу?
4. Для каких задач предназначен метод Хафа?
5. Что такое пространство Хафа?
6. Что может быть на исходном изображении, если его пространство Хафа имеет следующий вид (рис. 6.6)?

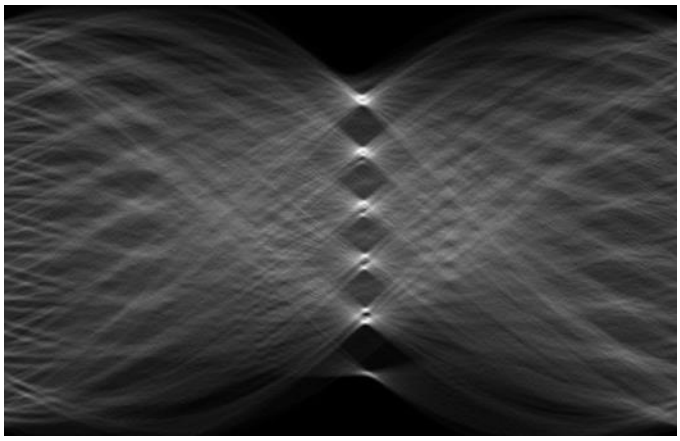


Рис. 6.6. Пространство Хафа

7. Как можно по точкам в пространстве Хафа определить наличие параллельных линий на изображении?

7. ЛОКАЛЬНЫЕ ОСОБЕННОСТИ

Одной из задач компьютерного зрения является нахождение заданного объекта на разных изображениях. Из этой задачи вытекает требование описать объект такими признаками, которые будут независимы (инвариантны) от внешних условий (освещенности, поворота, перекрытия и т. д.) Такими признаками могут служить некоторые локальные особенности. Локальные особенности представляют собой небольшие участки изображения, которые содержат много локальной информации и потому с большой вероятностью могут быть распознаны в другом изображении.

Локальные особенности должны быть:

– *повторяемыми* – должны находиться в одном и том же месте сцены независимо от ракурса и освещения;

– *локальными* (как родинки у человека) – должны быть расположены в малой области и не страдать от перекрытий;

– *компактными* – по сравнению с общим числом пикселей изображения их должно быть в разы меньше;

– *значимыми* (уникальными) – каждая особенность должна иметь свое собственное описание (представленное в виде дескриптора), по которому эту особенность можно будет распознать в другом изображении.

Исходя из этих требований можно предположить, что уникальностью обладают:

– края;

– углы (особенные точки или точки интереса);

– блобы (регионы интереса).

Существует множество методов поиска названных выше особенностей:

– детектор Харриса (Forstner);

– детектор Харриса-Лапласа ;

– детектор на основе LoG (Laplacian of Gaussian);

– детектор на основе DoG (Difference of Gaussian).

В рамках данного пособия остановимся на угловых точках и рассмотрим подробнее реализацию методов Харриса (Harris) и Ши-Томаси (Shi-Tomasi) в библиотеке *OpenCV*.

7.1 Поиск угловых точек

Углы (corners) – особые точки, которые формируются из двух или более граней. По-другому можно сказать, что угол – это точка, у которой в окрестности интенсивность изменяется относительно центра (x, y). Углы определяются по координатам и изменениям яркости окрестных точек изображения. Главное свойство таких точек заключается в том, что в области вокруг угла у градиента изображения преобладают два доминирующих направления, что делает их различимыми.

В зависимости от количества пересекаемых граней существуют разные виды уголков: L-, Y- (или T-), X-связные и стреловидно связные углы (рис. 7.1). Различные детекторы углов по-разному реагируют на каждый из таких видов уголков.

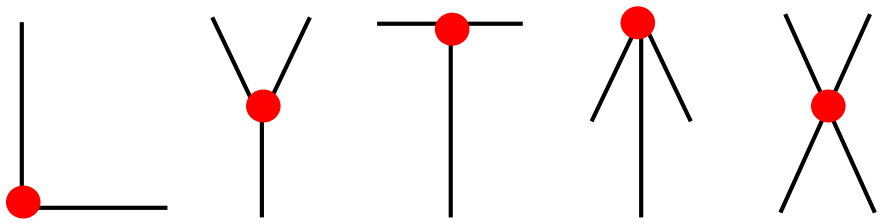


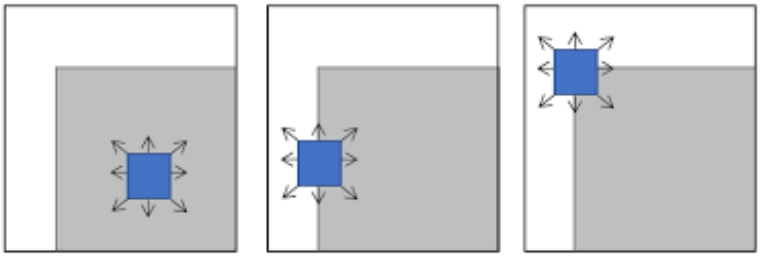
Рис. 7.1. Виды уголков

Существуют различные подходы к определению особых точек. Это могут быть методы, основанные на интенсивности изображения (особые точки вычисляются напрямую из значений интенсивности пикселей изображения), или методы, использующие контуры изображения (ищут места с максимальным значением кривизны контура или делают полигональную аппроксимацию контуров и определяют пересечение) и т. д. Однако на практике для широкого применения наиболее распространены методы, основанные на интенсивности изображения.

7.2. Детектор Харриса

Работа в исследовании особых точек началась с детектора Моравеца (Moravec, 1977). Детектор Моравеца – самый простой из существующих. Автор рассматривает изменение яркости квадратного окна W (обычно размера 3×3, 5×5, 7×7 пикселей) относительно

интересующей точки при сдвиге окна *W* на 1 пиксель в восьми направлениях (рис. 7.2)



а *б* *в*

Рис. 7.2. Определение угла при изменение яркости квадратного окна: а – фон; б – граница; в – угол.

Позднее Харрис и Стефенс улучшили детектор Моравеца, введя анизотропию по всем направлениям, т.е. рассматриваются производные яркости изображения для исследования изменений яркости по множеству направлений. Как показал опыт использования детектор Харриса является оптимальным детектором L-связных углов.

Детектор Харриса рассчитывает для каждого пикселя (x,y) ковариантную матрицу M(x,°y) размером 2×2 в области $blockSize \times blockSize$. Затем он вычисляет следующую характеристику:

$$dst(x, y) = \det M^{(x,y)} - k \cdot (trM^{(x,y)})^2 .$$

Углы изображения можно найти как локальные максимумы функции *dst*(*x*,*y*).

Детектор Харриса инвариантен к поворотам, частично инвариантен к аффинным изменениям интенсивности. К недостаткам стоит отнести чувствительность к шуму и зависимость детектора от масштаба изображения.

Функция ***cornerHarris***() реализует детектор Харриса:

```
void cornerHarris     (
    InputArray src,   // входное одноканальное изображение
    OutputArray dst,  // выходной вектор углов
    int blockSize,    // размер окрестности
    int ksize,        // параметр апертуры для оператора Собеля
    double k,         // свободный параметр детектора Харриса
    int borderType = BORDER_DEFAULT   //метод экстраполяции
);
```

Параметр *k* (см.формулу) – свободный параметр детектора Харриса, рекомендуют устанавливать равным 0,04.

Ниже приведен код, реализующий метода Харриса. В результате выводятся только те точки, интенсивность которых больше заданного порога, который можно регулировать при помощи ползунка (элемент *Trackbar*):

```
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;
Mat src, src_gray;
int thresh = 200;
int max_thresh = 255;
const char* source_window = "Source image";
const char* corners_window = "Corners detected";
void cornerHarris_demo(int, void*);
int main(int argc, char** argv)
{
src = imread("bike.jpg");
if (src.empty())
{
cout << "Could not open or find the image!\n" << endl;
cout << "Usage: " << argv[0] << " <Input image>" << endl;
return -1;
}
cvtColor(src, src_gray, COLOR_BGR2GRAY);
namedWindow(source_window);
createTrackbar("Threshold: ", source_window, &thresh,
max_thresh, cornerHarris_demo);
imshow(source_window, src);
cornerHarris_demo(0, 0);
waitKey();
return 0;
}
void cornerHarris_demo(int, void*)
{
int blockSize = 2;
int apertureSize = 3;
double k = 0.04;
// определение углов
Mat dst = Mat::zeros(src.size(), CV_32FC1);
cornerHarris(src_gray, dst, blockSize, apertureSize, k);
// нормализация выходного вектора углов
Mat dst_norm, dst_norm_scaled;
normalize(dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat());
```

```
convertScaleAbs(dst_norm, dst_norm_scaled);
// рисование кругов вокруг углов
// параметр thresh определяет порог отсечения (0-255)
// чем он меньше, тем больше точек будет отрисовано
for (int i = 0; i < dst_norm.rows; i++)
{
     for (int j = 0; j < dst_norm.cols; j++)
     {
          if ((int)dst_norm.at<float>(i, j) > thresh)
          {
circle(dst_norm_scaled, Point(j, i), 5, Scalar(0), 2, 8, 0);
          }
     }
}
namedWindow(corners_window);
imshow(corners_window, dst_norm_scaled);
}
```

7.3. Детектор Ши – Томаси

Функция ***goodFeaturesToTrack***() реализует метод Харриса с усовершенствованиями Ши-Томаси. Она вычисляет необходимые операторы производных, анализирует их и возвращает список особых точек:

```
void goodFeaturesToTrack(
Mat& image,                 // входное изображение
OutputArray corners,        // выходной вектор углов
int maxCorners,             // сколько углов сохранять
double qualityLevel,        // отношение к наилучшему углу
double minDistance,         // минимальное расстояние между углами
InputArray mask = noArray(), // игнорировать углы в точка, где
mask=0
int blockSize = 3,               // размер окрестности
bool useHarrisDetector = false, // false°=°'метрика Ши-Томаси'
double k = 0.04             // используется для метрики Харриса
);
```

Входное изображение *image* должно быть одноканальным 8- или 32-разрядным (типа 8U или 32F). Выходной аргумент *corners* – вектор или массив (в зависимости от того, что передано) – будет содержать все найденные углы. Количество возвращенных углов можно ограничить, задав аргумент *maxCorners*, их качество – с помощью аргумента

qualityLevel (обычно от 0,01 до 0,10, но не больше 1,0), а минимальное расстояние между соседними углами – с помощью *minDistance*. Если задан аргумент *mask*, то его размер должен быть такой же, как у *image*; в тех точках изображения, где *mask* содержит 0, функция не будет искать углы. Аргумент *blockSize* задает размер окрестности при вычислении угла; типичное значение равно 3, но для изображений с высоким разрешением его можно немного увеличить. Если аргумент *useHarrisDetector* равен *true*, то функция будет использовать формулу из оригинального алгоритма Харриса, в противном случае – метод Ши-Томаси. Параметр *k* применяется только в алгоритме Харриса, лучше не менять его значение по умолчанию.

С точки зрения конечного пользователя функция *goodFeaturesToTrack*() удобнее чем функция *cornerHarris*(), так как предоставляет больше параметров для выбора сильных углов.

Ниже приведен пример реализации метода *goodFeaturesToTrack*(). Данный код находит особые точки на изображении и отрисовывает их разноцветными кругами на начальном изображении:

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

using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
Mat src, src_gray;
RNG rng(12345);
const char* source_window = "Image";

// загружаем изображение и переводим его в одноканальное
src = imread("bike.jpg", 1);
cvtColor(src, src_gray, COLOR_BGR2GRAY);

// создаем окно для вывода результата
namedWindow(source_window);
imshow(source_window, src);

// находим особые точки
goodFeaturesToTrack( src_gray, corners, 50, 0.01, 10, Mat(), 3,
false, 0.04);
```

```
// рисуем найденные точки
cout << "** Number of corners detected: " << corners.size() <<
endl;
int radius = 4;
for (size_t i = 0; i < corners.size(); i++)
{
circle(copy, corners[i], radius, Scalar(rng.uniform(0, 255),
rng.uniform(0, 256), rng.uniform(0, 256)), FILLED);
}
namedWindow(source_window);
imshow(source_window, copy);
waitKey();
return 0;
}
```

Метод Ши–Томаси используется при отслеживании объектов на видео в методе Лукаса–Канаде, который будет рассмотрен позже.

7.4. Дескрипторы

Исходя из требования значимости (уникальности) каждая особая точка должна быть описана вектором признаков, вычисляемых на основе интенсивности, градиентов или других характеристик точек окрестности. Эта описательная информация об особой точке представляется в виде ее дескриптора.

Построить дескриптор признака можно как угодно – например, составить вектор из значений яркости в квадрате 3×3 вокруг особой точки. Проблема в том, что такой дескриптор примет другие значения, если посмотреть на особую точку под другим углом. В общем случае инвариантность относительно вращения – желательное свойство дескриптора. Конечно, только от конкретной задачи зависит, так ли нужна инвариантность относительно вращения. Например, при обнаружении и сопровождении людей на кадрах видеопотока голова человека обычно находится вверху, а ноги – внизу. В таких приложениях отсутствие у дескриптора вращательной симметрии – не обязательно проблема. Напротив, при аэрофотосъемке земля «поворачивается», когда самолет меняет направление, поэтому снимок одной и той же местности может представать в разной ориентации.

В общем случае дескрипторы должны обеспечить инвариантность относительно смещения, поворота, масштаба, изменения яркости и поворота. Этим условиям в некоторой мере обладает дескриптор

методом SIFT (Scale Invariant Feature Transform). Однако следует отметить, что этот метод запатентован и не доступен в свободной версии *OpenCV*.

Вопросы для самоконтроля

1. Что означает термин «локальные особенности»?
2. Какими свойствами должны обладать локальные особенности?
3. Какой принцип работы детектора Моравеца? Какие локальные особенности он способен выделить?
4. Перечислите функции, которые реализуют поиск особенных точек в *OpenCV*.
5. Какими параметрами можно регулировать качество и количество найденных особенных точек?
6. Как можно использовать результаты работы детекторов углов?

8. ДЕТЕКТИРОВАНИЕ ЛИЦА

Задача детектирования лица на изображении, как правило, является подзадачей при решении более сложных проблем (например, распознавания лица или определения выражения лица). Для решения этой задачи успешно применяется метод Виолы – Джонса.

8.1. Метод Виолы – Джонса

Метод Виолы – – Джонса – алгоритм, позволяющий обнаруживать объекты на изображениях в реальном времени. Его предложили Паул Виола и Майкл Джонс в 2001°г. Хотя алгоритм может распознавать объекты на изображениях, основной задачей при его создании было обнаружение лиц. Метод Виолы – Джонса является одним из лучших по соотношению распознавание: скорость работы. Благодаря скорости обработки изображения можно с легкостью обрабатывать потоковое видео. Однако у данного метода есть ограничение – он хорошо работает и распознает черты лица под небольшим углом, примерно до 30°.

Основу метода Виолы – Джонса составляют примитивы Хаара, представляющие собой разбивку заданной прямоугольной области на наборы разнотипных прямоугольных подобластей (рис. 8.1).

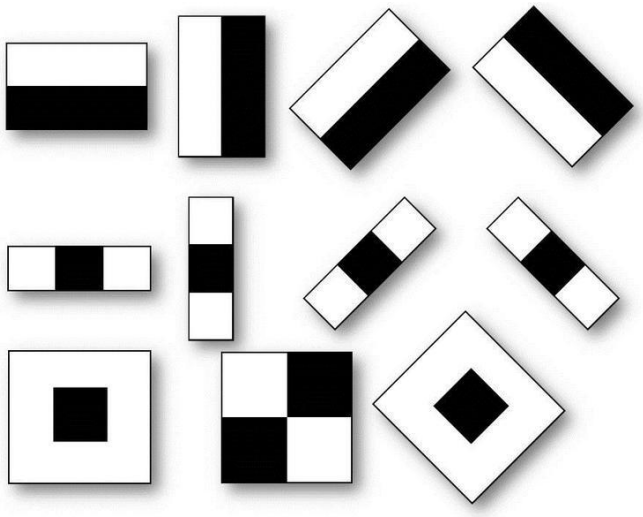


Рис. 8.1. Примитивы Хаара

Например, из двух признаков Хаара строится первый каскад системы по распознаванию лиц, который имеет вполне осмысленную интерпретацию: зона глаз должна быть темнее, чем зона носа и щек, а переносица светлее, чем глаза (рис. 8.2).

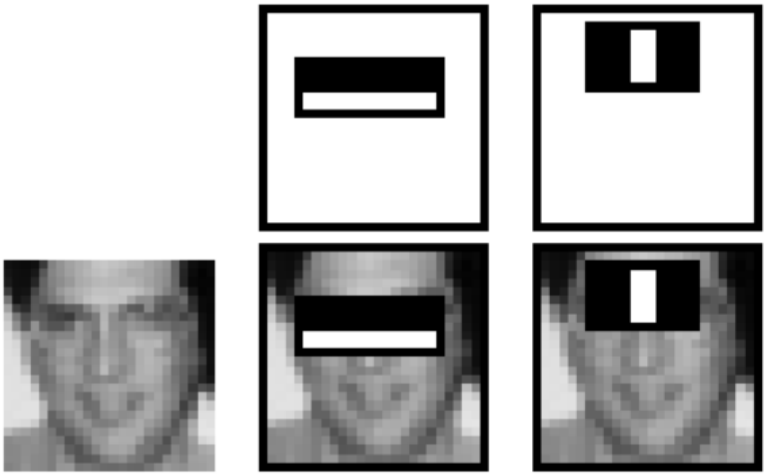


Рис. 8.2. Первый каскад из двух признаков Хаара

При наложении примитивов Хаара на изображение рассчитываются признаки, которые равны разности суммы пикселей под белым прямоугольником и суммы пикселей под черным прямоугольником. Для увеличения скорости расчетов используется интегральное представление изображения.

Интегральное представление изображения – это матрица, такого же размера, как исходное изображение. Каждый элемент интегрального изображения содержит в себе сумму значений всех пикселей левее и выше данного пикселя (рис. 8.3).

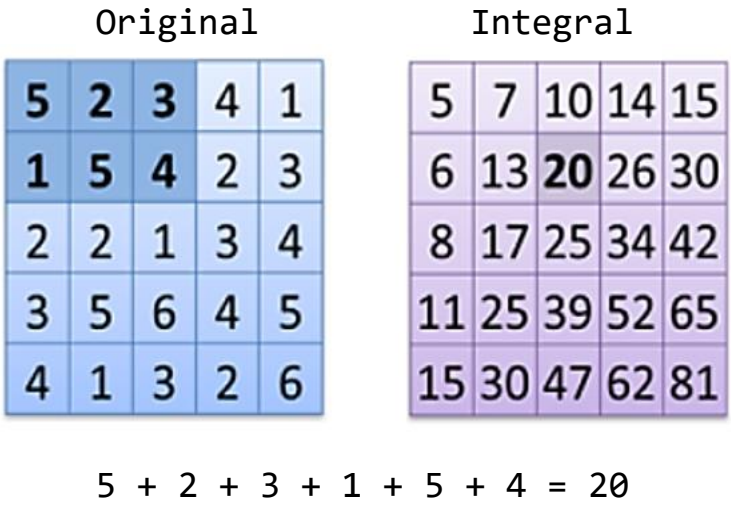


Рис. 8.3. Интегральное представление изображения

Интегральное представление позволяет быстро вычислить сумму пикселей произвольного прямоугольника с помощью всего лишь четырех ссылок на массив, независимо от размера массива (рис. 8.4).

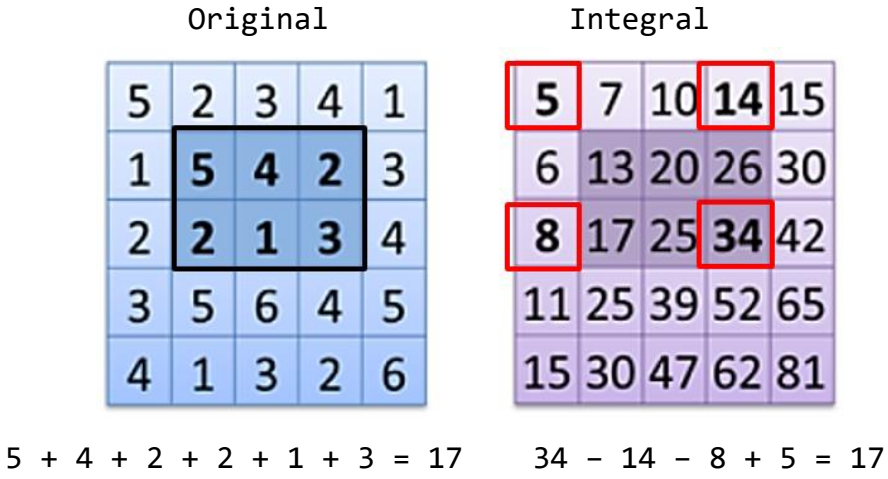


Рис. 8.4. Вычисление суммы пикселей при интегральном представлении

Именно использование при вычислениях интегрального представления изображения, которое рассчитывается один раз и в дальнейшем используется для расчета признаков, позволило создать быстрый алгоритм поиска объектов. Однако высокая скорость расчета признака не компенсирует значительное количество различных возможных признаков. К примеру, при стандартном размере сканирующего окна в 24×24 пикселя возможно 162 тысячи вариантов расположения признаков за счет изменения их положения и масштаба в окне сканирования. Расчет такого количества признаков может занять большое количество времени, следовательно, необходимо выбрать наиболее подходящие признаки для классифицируемого объекта (например, для лица), иными словами, обучить классификатор реагировать только на определенные признаки.

В алгоритме Виолы – Джонса для обучения классификатора используется технология бустинга, а точнее вариация бустинга – *AdaBoost*. Бустинг (от англ. *boosting* – повышение, усиление, улучшение) означает «усиление» «слабых» моделей. Это процедура последовательного построения композиции алгоритмов машинного обучения, когда каждый следующий алгоритм стремится компенсировать недостатки композиции всех предыдущих алгоритмов.

В результате обучения формируется xml-файл, содержащий каскад обученных классификаторов с выбранными признаками и со значениями пороговых величин для классификаторов.

Для определения принадлежности объекта к классу в каждом каскаде находится сумма значений слабых классификаторов этого каскада. Каждый слабый классификатор выдает два значения в зависимости от того, больше или меньше заданного порога значение признака, принадлежащего этому классификатору. В конце сумма значений слабых классификаторов сравнивается с порогом каскада и выносится решения, найден объект или нет данным каскадом.

8.2. Реализация метода Виолы – Джонса в OpenCV

Рассмотрим пример создания приложения детектирования лица на видео с камеры с использованием каскадов Хаара с помощью библиотеки *OpenCV*. Библиотека предоставляет уже обученные классификаторы для различных объектов, которые можно найти в папке данных в установочном каталоге *OpenCV* или в открытых интернет-источниках.

Доступ к реализации каскадного классификатора Хаара осуществляется посредством работы с объектами класса *CascadeClassifier*.

Для создания объекта классификатора можно использовать конструктор класса по умолчанию, после вызова которого необходимо загрузить обученный каскад (xml-файл) с помощью метода ***load***().

```
CascadeClassifier();
bool load(const string& filename);
```

Или воспользоваться конструктором класса, который в качестве параметра принимает имя xml - файла с обученным каскадом:

```
CascadeClassifier(const String& filename);
```

Чтобы найти лица разного масштаба на изображении, необходимо вызвать метод ***detectMultiScale***().

```
void detectMultiScale(
const Mat& image,      // входное (полутоновое) изображение
vector& objects,       // выходные прямоугольники
double scaleFactor=1.1, // коэффициент изменения масштаба
int minNeighbors=3, // минимальное число учитываемых соседей
```

```
    int flags=0,            // флаги
    Size minSize=Size()     // наименьший рассматриваемый размер
    Size maxSize=Size()     // наибольший рассматриваемый размер
    );
```

ScaleFactor представляет собой коэффициент масштабирования, который определяет, во сколько раз будет уменьшено изображение на каждом масштабе. Аргумент *minNeighbors* предотвращает ложное распознавание. Одно и то же лицо обычно обнаруживается несколько раз, потому что соседние пиксели и масштабы свидетельствуют о наличии лица. Значение по умолчанию (3) означает, что существование лица в некоторой области признается, только если оно обнаружено по меньшей мере 3 раза при перекрывающихся положениях окна. Параметр *flags* присутствует для совместимости со старым интерфейсом. Параметры *minSize* и *maxSize* задают соответственно минимальный и максимальный размеры области, в которой ищется лицо. Их задание повышает скорость вычислений, но с риском пропустить некоторые лица необычно малого или большого размера.

Функция ***detectMultiScale***() возвращает вектор окаймляющих прямоугольников. Эти прямоугольники необходимо отобразить на изображении. Чтобы нарисовать прямоугольник, достаточно вызвать функцию ***rectangle***():

```
void rectangle(
   CvArr* img,        // изображение для вывода прямоугольников
   CvPoint pt1,       //координаты левого верхнего
   CvPoint pt2,       // и правого нижнего угла прямоугольника
   CvScalar color,    // цвет линии
   int thickness=1,   // толщина линии
   int lineType=8,    // тип линии
   int shift=0        //  количество  дробных  бит  в  координатах
точек
   );
```

Ниже приведен пример кода для детектирования лица на видео, выделяющий найденные лица прямоугольником:

```
#include <opencv2/opencv.hpp>
#include <iostream>
#define DELAY 30
#define ESC_KEY 27
using namespace cv;
using namespace std;
```

```
int main()
{
// создание объект классификатора
CascadeClassifier faceCascade;
string faceCascadeName = "haarcascade_frontalface_alt.xml";
faceCascade.load(faceCascadeName);
// открытие видео с камеры
VideoCapture capture(0);
if (!capture.isOpened()) {
cerr << "Unable to open: " << endl;
return 0;
}
while (true)
{capture >> frame;
// подготовка изображения
Mat grayFrame;
cvtColor(frame, grayFrame, COLOR_BGR2GRAY);
equalizeHist(grayFrame, grayFrame);
// обнаружение объектов
vector<Rect> faces;
faceCascade.detectMultiScale(grayFrame, faces, 1.1, 2, 0 |
CASCADE_SCALE_IMAGE, Size(30, 30));
// перебор найденных объектов и отрисовка прямоугольников
for (int i = 0; i < faces.size(); i++)
{
  Point center(faces[i].x + faces[i].width/2,
            faces[i].y + faces[i].height/2);
  Mat faceROI = grayFrame(faces[i]);
  for (i = 0; i < faces.size(); i++)
    {
    rectangle(frame,
    Point(faces[i].x, faces[i].y),
    Point(faces[i].x + faces[i].width,
    faces[i].y + faces[i].height),
    CV_RGB(255, 0, 0), 2);
    }
}
imshow("video", frame);
keyboard = (char)waitKey(30);
}
waitKey();
return 0;
}
```

Вопросы для самоконтроля

1. Что представляют из себя примитивы Хаара?
2. Как при помощи примитивов Хаара можно детектировать заданный объект?
3. Что такое интегральное представление изображения?
4. Из каких задач состоит алгоритм Виола – Джонса?
5. Как получить каскад классификаторов для заданного объекта?
6. По какому принципу работает каскад классификаторов?
7. Как уменьшить ложные срабатывания детектора Виола – Джонса?
8. Какие преимущества и ограничения детектора Виола – Джонса?

9. ОТСЛЕЖИВАНИЕ ОБЪЕКТОВ НА ВИДЕО

9.1. Оптический поток

При обработке видеоизображений часто возникает задача определения перемещений объектов в трехмерном пространстве. Для ее решения рассмотрим понятие оптического потока.

Оптический поток (ОП) – изображение видимого движения, представляющее собой сдвиг каждой точки между двумя изображениями.

Суть ОП состоит в том, что для каждой точки изображения $I_1(x,y)$ находится такой сдвиг (dx, dy), чтобы исходной точке соответствовала точка на втором изображении $I_2(x + dx, y + dy)$.

Как определить соответствие точек? Для этого надо взять какую-то функцию точки, которая не изменяется в результате смещения. Обычно считается, что у точки сохраняется интенсивность (т. е. яркость или цвет для цветных изображений), но можно взять и другие характеристики.

Очевидно, сохранение интенсивности дает сбои, если меняется освещенность или угол падения света. Тем не менее, если речь идет о видеопотоке, то, скорее всего, между двумя кадрами освещение сильно не изменится, хотя бы потому, что между ними проходит малый промежуток времени. Поэтому часто используют интенсивность в качестве функции, сохраняющейся у точки.

Различают *плотный оптический поток*, при котором рассматривается смещение каждого пикселя в видеокадре, и *разреженный (редкий) оптический поток*, при котором отслеживают смещение между отдельными точками в изображении (например, особыми точками).

9.2. Алгоритм Лукаса – Канаде

Наибольшее распространение для работы с оптическим потоком получил алгоритм Лукаса – Канаде (*Lucas – Kanade*). В его основе лежат три предположения:

– *постоянство яркости*, т. е. пиксель изображения объекта на сцене не меняется при переходе от кадра к кадру. Для полутоновых

изображений (алгоритм Лукаса – Канаде применим и к цветным) это означает, что не меняется яркость прослеживаемого пикселя;

– *постоянство во времени*, или «малые сдвиги», т. е. образ участка поверхности на изображении медленно изменяется во времени. На практике это означает, что шаг по времени настолько мал относительно масштаба перемещения, что положение объекта не сильно изменяется от кадра к кадру;

– *пространственная когерентность*, т. е. соседние точки сцены принадлежат одной и той же поверхности, движутся одинаково и проецируются в близкие точки на плоскости изображения.

Рассмотрим математическую модель оптического потока, предположив, что у точки в результате смещения не изменилась интенсивность.

Пусть $I_1 = I(x, y, t_1)$ – интенсивность в некоторой точке (x, y) на первом изображении (т. е. в момент времени t). На втором изображении эта точка сдвинулась на (dx, dy), при этом прошло время dt, тогда $I_2 = I(x + dx, y + dy, t_1 + dt)$. Если функцию интенсивности разложить по Тейлору до первого члена, то получим $I_2 = I_1 + I_x dx + I_y dy + I_t dt$, здесь I_x, I_y, I_t – частные производные по координатам и времени, то есть по сути $I_t dt$ – изменение яркости в точке (x, y) между двумя кадрами. А так как мы предположили, что у точки сохранилась интенсивность, значит $I_2 = I_1$, откуда следует, что $I_x dx + I_y dy + I_t dt = 0$. Получаем одно уравнение с двумя неизвестными (dx и dy), которое не может быть однозначно разрешено.

Алгоритм Лукаса – Канаде обходит неоднозначность за счет использования информации о соседних пикселях в каждой точке. Метод основан на предположении, что в локальной окрестности каждого пикселя значение оптического потока одинаково. Таким образом, можно записать основное уравнение оптического потока для всех пикселей окрестности и решить полученную систему уравнений методом наименьших квадратов.

В *OpenCV* функция ***calcOpticalFlowPyrLK***() реализует пирамидальный алгоритм Лукаса – Канаде для вычисления оптического потока. Она использует «хорошие для прослеживания признаки» и возвращает информацию о качестве прослеживания каждой точки:

```
void calcOpticalFlowPyrLK(
InputArray prevImg,  // предыдущее изображение (t-1), CV_8UC1
InputArray nextImg,  // следующее изображение (t), CV_8UC1
InputArray prevPts,  // вектор начальных точек (CV_32F)
```

```
InputOutputArray nextPts, // результаты: конечные положения точек
OutputArray status, // для каждой точки: 1 - найдена, 0 – не
найдена
OutputArray err,      // мера погрешности для найденных точек
Size winSize = Size(15,15),      // размер окна поиска i
nt maxLevel = 3,                 // число уровней пирамиды
TermCriteria criteria = TermCriteria(      // критерий остановки
TermCriteria::COUNT | cv::TermCriteria::EPS, 30, 0.01 ),
int flags = 0,        // использовать гипотезы и/или собственные
значения
double minEigThreshold = 1e-4   // матрица     пространственного
градиента
);
```

Разберемся, как работает функция. Берем два изображения *prevImg* и *nextImg*, формируем список прослеживаемых точек в массиве *prevPts* и вызываем функцию. Когда она вернет управление, смотрим по массиву *status*, какие точки успешно прослежены, а затем из массива *nextPts* получаем координаты новых точек. Теперь обратимся к деталям. Первые два аргумента функции ***calcOpticalFlowPyrLK***() *prevImg* и *nextImg* – соответственно начальное и конечное изображения. Размеры и число каналов должны быть одинаковыми. Аргумент *prevPts* – входной список признаков в первом изображении, а *nextPts* – выходной список, в который помещаются соответствующие им точки во втором изображении. Тот и другой могут быть массивом $N\times2$ или вектором точек. В массивы *status* и *err* будет помещена информация об успешности прослеживания. Точнее, элемент массива *status* говорит, была ли вообще найдена точка, соответствующая признаку в *prevPts* (*status[i]* отличен от нуля тогда и только тогда, когда для *prevPts[i]* есть соответствие в *nextImg*). А элемент *err[i]* содержит погрешность прослеживания для точек *prevPts*, найденных в *nextImg* (если точка *i* не обнаружена, то значение *err[i]* не определено). Размер окна для вычисления когерентного перемещения задает аргумент *winSize*. Поскольку мы строим пирамиду изображений, то нужен аргумент *maxLevel*, определяющий ее глубину. Если *maxLevel* равен 0, то пирамида не используется. Аргумент *criteria* определяет критерий остановки. *TermCriteria* – структура, применяемая во многих итеративных алгоритмах *OpenCV*, определяет критерии завершения для итерационных алгоритмов.

Значения по умолчанию дают удовлетворительные результаты в большинстве ситуаций. Но если изображение очень большое, то имеет

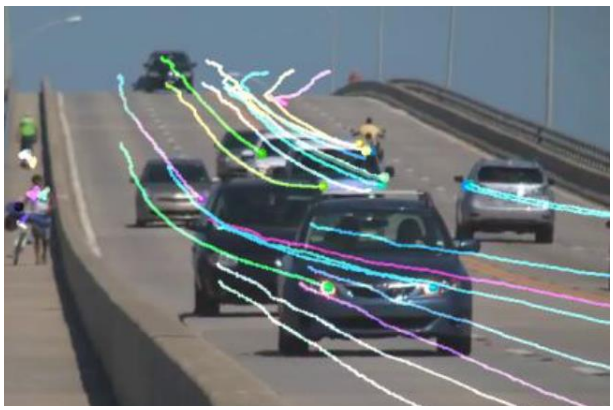
смысл немного увеличить максимальное число итераций. В аргументе *flags* могут быть подняты один или оба следующих флага:

– *OPTFLOW_LK_GET_MIN_EIGENVALS* задает более детальную меру погрешности. По умолчанию в массиве *err* для каждого пикселя сохраняется среднее изменение яркости между окнами вокруг предыдущего и нового углов. Если же этот флаг поднят, то в качестве меры погрешности берется минимальное собственное значение матрицы Харриса, ассоциированной с углом;

– *OPTFLOW_USE_INITIAL_FLOW* используется, если массив *nextPts* уже содержит начальную гипотезу о координатах признаков при вызове функции (если флаг не поднят, то в качестве начальной гипотезы берутся просто координаты точек из *prevPts*).

Последний аргумент *minEigThreshold*, используется как фильтр для удаления точек, оказавшихся не такими уж хорошими для прослеживания. Значение по умолчанию 10^{-4} вполне приемлемо, если его увеличить, будет отбрасываться больше точек.

По ссылке *OpenCV*: Optical Flow можно найти пример кода, реализующий алгоритм Лукаса-Канаде. В качестве точек отслеживания в первом кадре методом ***goodFeaturesToTrack***() определяются угловые точки. Результаты работы алгоритма представлены на рисунке.



Результат применения алгоритма Лукаса-Канаде

В состав дистрибутива *OpenCV* входит несколько программ для демонстрации алгоритмов отслеживания объектов на видео, а именно: lkdemo.cpp (разреженный оптический поток); fback.cpp (плотный оптический поток, алгоритм Фарнебэка); tvl1_optical_flow.cpp (плотный оптический поток); camshiftdemo.cpp (сопровождение цветных областей методом сдвига среднего); kalman.cpp (фильтр Калмана); 768x576.avi (видеофайл, samples/data).

9.3. Трекинг объектов

Трекинг (отслеживание) объектов – это задача обнаружения объекта в последовательных кадрах видео. Ранее при изучении детектора Виолы – Джонса было рассмотрено приложение, которое тоже «умеет» следить за лицом, однако алгоритмы детектирования и алгоритмы отслеживания работают по-разному.

Обычно алгоритмы отслеживания быстрее, чем алгоритмы обнаружения. Это объясняется тем, что при отслеживании объекта, обнаруженного в предыдущем кадре, нам уже известно, как объект выглядит, его расположение в предыдущем кадре, направление и скорость его движения, и алгоритм трекинга способен по этим данным предсказать положение объекта в следующем кадре, а детектор заново ищет объект на каждом новом кадре. И еще, если отслеживать объект на видео детектором, то при перекрытии объекта детектор скорее всего не сможет определить его местоположение, в то время как хороший алгоритм отслеживания способен справиться с некоторым уровнем окклюзии (перекрытия).

Разработано большое количество методов отслеживания (трекеров), каждый из которых в той или иной мере подходит для конкретной задачи.

В *OpenCV* версии 3.4 представлены следующие трекеры:

– *BOOTTING* отслеживает объекты в режиме реального времени на основе алгоритма *AdaBoost*;

– *MIL* генерирует классификатор в онлайн-режиме, чтобы отделить объект от фона;

– *KCF* использует свойства циркулянтной матрицы для повышения скорости обработки;

– *MEDIANFLOW* подходит для очень плавных и предсказуемых движений, когда объект виден на протяжении всей последовательности;

– *TLD* реализует отслеживание, обучение и обнаружение. Трекер следует за объектом от кадра к кадру. Детектор локализует все видимые явления, которые наблюдались до сих пор, и при необходимости корректирует трекер. Изучение оценивает ошибки детекторов и обновляет их, чтобы избежать этих ошибок в будущем;

– *MOSSE* отслеживает объекты с использованием адаптивных корреляционных фильтров;

– *GOTURN* является своего рода трекером на основе сверточных нейронных сетей (*CNN*). Текущий метод *GOTURN* не обрабатывает

окклюзии; однако он довольно устойчив к изменениям зрения, изменениям освещения и деформациям.

Сравнительный анализ и рекомендации по их использованию подробно изложены в статье [8].

ЛИТЕРАТУРА

1. Кустикова, В. Д. Разработка мультимедийных приложений с использованием библиотек OpenCV и IPP: учеб. курс/ В.Д.Кустикова.– Н. Новгород: НГУ им. Н.И. Лобачевского, 2012 – 49 с.

2. Кэлер, А., Изучаем OpenCV 3 / А.Кэлер, Г. Брэдски, пер. с анг. А. А. Слинкина. – М.: ДМК Пресс, 2017. – 826 с.

3. Обработка и анализ изображений в задачах машинного зрения: курс лекций и практических занятий / Ю.В. Визильтер [и др.]. – М.: Физматкнига, 2010. – 672 с.

4. Матричные фильтры обработки изображений [Электронный ресурс]// Habr. – Режим доступа: https://habr.com/ru/ post/142818/. – Дата доступа: 05.02.2021.

5. OpenCV Tutorials [Electronic resourse]// Docs.opencv. – Mode of access: https://docs.opencv.org/3.4/d9/df8/tutorial_root.html// – Date of access: 27.01.2021.

6. David G. Lowe, Distinctive Image Features from Scale-Invariant Keypoints [Electronic resourse]// citeseerx. – Mode of access: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.3843&rep=rep1&type=pdf /. – Date of access: 12.02.2021.

7. OpenCV шаг за шагом [Электронный ресурс]// Robocraft. – Режим доступа: http://robocraft.ru/page/opencv/. – Дата доступа: 06.02.2021.

8. Object Tracking with OpenCV [Электронный ресурс]// Ehsangazar //. – Mode of access: https://ehsangazar.com/object-tracking-with-opencv-fd18ccdd7369. – Date of access: 11.02.2021.

СОДЕРЖАНИЕ