Нелокальный алгоритм сглаживания изображений с помощью OpenCV

Малашонок Софья
студент группы 80-105М

20 марта 2019 г.

1 Общая информация

OpenCV (Open Source Computer Vision Library, библиотека компьютерного зрения с открытым исходным кодом) – библиотека алгоритмов компьютерного зрения, обработки рисунков и численных алгоритмов общего назначения с открытым кодом. Реализована на C/C++, также разрабатывается для Python, Java, Ruby, Matlab, Lua. Может использоваться в академических и коммерческих целях – распространяется в условиях лицензии BSD. OpenCV – это набор типов данных, функций и классов для обработки изображений алгоритмами компьютерного зрения.

2 Основные модули библиотеки

1. cxcore – ядро содержит базовые структуры данных и алгоритмы:
• базовые операции над многомерными числовыми массивами;
• матричная алгебра, функции, генераторы случайных чисел;
• запись/восстановление структур данных в/из XML;
• базовые функции 2D графики.
2. CV – модуль обработки изображений и компьютерного зрения:
• базовые операции над изображениями (фильтрация, геометрические преобразования, преобразование цветовых пространств и т. д.);
• анализ изображений (выбор отличительных признаков, морфология, поиск контуров, гистограммы);
• анализ движения, слежение за объектами;
• обнаружение объектов, в частности лиц;
• калибровка камер, восстановление пространственной структуры.

3. Highgui – модуль для ввода/вывода изображений и видео, создания пользовательского интерфейса:

• захват видео с камер и из видео файлов, чтение/запись статических изображений;

• функции для организации простого UI (демо приложения используют HighGUI).

4. Cvaux – экспериментальные и устаревшие функции:

• пространств. зрение: стерео калибрация, самокалибрация;

• поиск стерео-соответствия, клики в графах;

• нахождение и описание черт лица.

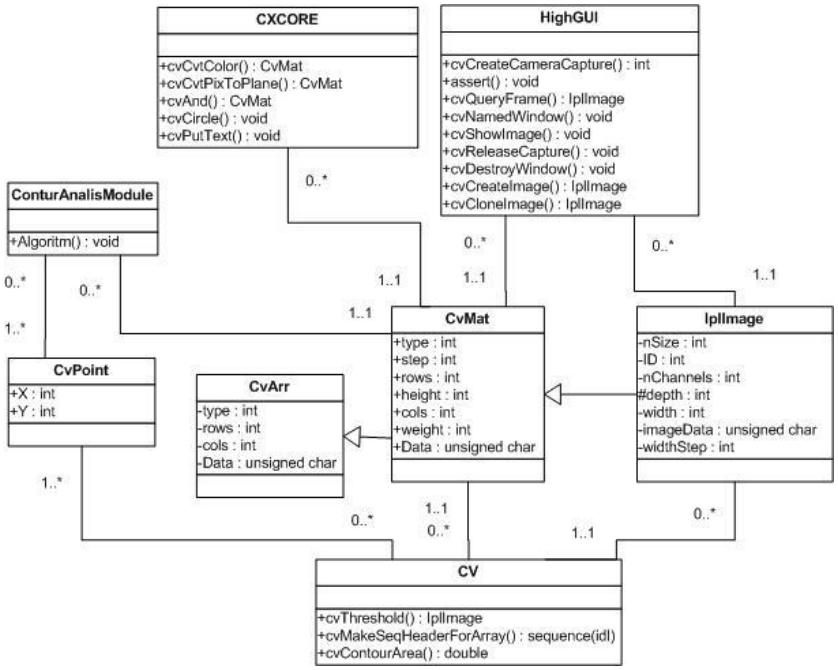


Рис. 1: Диаграмма классов OpenCV

Для использования OpenCV на платформе .Net – используется кросс-платформенная обертка EmguCV. Реализация функции OpenCV для вызова из .NET совместимых языках, таких как C Sharp, VB, VC ++, IronPython и т.д. обертки может быть скомпилирована и работать на Windows, Linux, Mac OS X, iPhone, IPAD и устройств Android.

3 Структуры данных

• Image <Bgr, byte> – переменая изображения, хранимого в трех цветовых каналах RGB палитры в значениях типа byte;
• Image <Bgr, float> ProcessedIMG – переменая изображения, хранимого в трех цветовых каналах RGB палитры в значениях типа float;
• Image <Gray, byte> GreyIMG – переменая изображения, хранимого в одном цветовом канале, показывающем яркость пикселя, в значениях типа byte.

4 Описание функций и методов

№	Название функции	Набор методов	Описание методов
1	Перевод цветного изображения в полутоновое	ConvertToGray(Image<Bgr, byte> Img)	Преобразование изображения в оттенки серого. Параметры Bgr и Byte – цвет и глубина входящего изображения.
2	Поворот изображения направо.	RotateRight(Image<Bgr, byte> Img)	Поворот изображения налево на 90 градусов. Аргументы: входящее изображение.
3	Поворот изображения налево.	RotateLeft(Image<Bgr, byte> Img)	Поворот изображение направо на 90 градусов. Аргументы: входящее изображение.
4	Добавление аддитивного шума.	ANoise(Image<Bgr, byte> Img)	Добавление аддитивного шума к входящему изображению со среднеквадратическим отклонением шума = 20. Аргументы: входящее изображение.
5	Добавление импульсного шума.	INoise(Image<Bgr, byte> Img)	Добавление импульсного шума к входящему изображению, зашумление 100 пикселей. Аргументы: входящее изображение.
6	Фильтр-свертка.	MaskFilter(Image<Bgr, byte> Img, float[,] k)	Наложение фильтра-свертки на изображение. Аргументы метода: входящее изображение и размеры матрицы свертки.
7	Медианный фильтр.	MedianFilter(Image<Bgr, byte> Img)	Наложение медианного фильтра на изображение. Аргументы: входящее изображение.
8	Фильтр K-ближайших соседей.	KNearestFilter(Image<Bgr, Byte> Img)	Наложение фильтра K-ближайших соседей на изображение. Аргументы: входящее изображение.

9	Взвешенно-медианный фильтр.	WeightedMedianFilter(Image<Bgr, Byte> Img)	Наложение взвешенно-медианного фильтра на изображение. Аргументы: входящее изображение.
10	Пороговая сегментация.	Segmentation(Image<Bgr, Byte> Img, int LowLevel, int HightLevel)	Пороговая сегментация входящего изображения на два значения: 0 и 255. Аргументы: входящее изображение, нижний и верхний порог (принимается за 255).
11	Обнаружение границ. Водораздел.	WaterFilter(Image<Bgr, Byte> Img)	Применение фильтра водораздела к входящему изображению. Аргументы: входящее изображение.
12	Обнаружение границ. Оператор Превита.	PrevitContur(Image<Bgr, Byte> Img)	Применение оператора Превита к входящему изображению. Аргументы: входящее изображение.
13	Обнаружение границ. Оператор Лапласа.	LaplasContur(Image<Bgr, Byte> Img)	Применение оператора Лапласа к входящему изображению. Аргументы: входящее изображение.
14	Обнаружение границ. Оператор Кирша.	KirshContur(Image<Bgr, Byte> Img)	Применение оператора Кирша к входящему изображению. Аргументы: входящее изображение.
15	Оконтуривание объекта. Алгоритм «жука».	Bug(Point Start,Image<Bgr,byte> Img)	Применение к изображению функции нахождения контура алгоритмом «жука». Аргументы: входящее изображение.

5 Алгоритм сглаживания изображений

Предварительная информация. Избавление изображения от шума – одна из фундаментальных операций компьютерного зрения. Алгоритмы сглаживания применяются почти везде: они могут быть как самостоятельной процедурой для улучшения фотографии, так и первым шагом для более сложной процедуры, например, для распознавания объектов на изображении.

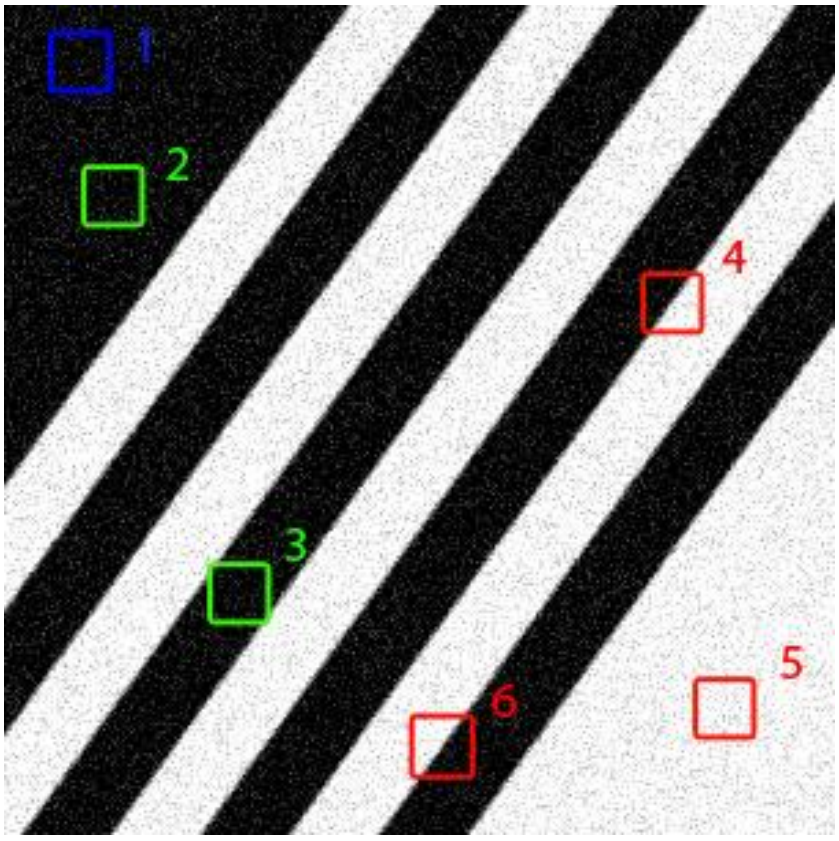
Вкратце о поставленной задаче. Предполагается, что в некотором идеальном мире мы можем получить идеальное изображение I (матрица или вектор N на M элементов), которое идеально передаёт информацию о внешнем мире. К сожалению, мы живём не в столь прекрасном месте, и поэтому по разным причинам нам приходится оперировать изображениями $I+n$, где n – матрица или вектор шума. Существует много моделей шума, но в этой статье мы будем предполагать, что n – белый гауссов шум. По имеющейся информации мы пытаемся восстановить изображение I_d, которое будет максимально близко к идеальному изображению I. К сожалению, это почти всегда невозможно, так как при зашумлении часть данных неизбежно теряется. Кроме того, сама постановка задачи предполагает бесконечное множество решений.

Как правило, обычные сглаживающие фильтры устанавливают новое значение для каждого пикселя, используя информацию о его соседях. Дешёвый и сердитый гауссов фильтр, например, сворачивает матрицу изображения с матрицей гауссианы. В результате, в новом изображении каждый пиксель является взвешенной суммой пикселя с тем же номером исходной картинки и его соседей. Немного другой подход использует медианный фильтр: он заменяет каждый пиксель медианой среди всех близлежащих пикселей. Развитием идей этих способов сглаживания является билатеральный фильтр. Эта процедура берёт взвешенную сумму вокруг, опираясь как на расстояние до выбранного пикселя, так и на близость пикселей по цвету.

Идея фильтра. Тем не менее, все вышеперечисленные фильтры используют только информацию о близлежащих пикселях. Основная же идея нелокального сглаживающего фильтра состоит в том, чтобы использовать всю информацию на изображении, а не только пиксели соседние с обрабатываемым. Как же это работает, ведь зачастую значения в отдалённых точках никак не зависят друг от друга?

Совершенно очевидно, что если у нас есть несколько изображений одного и того же объекта с разным уровнем шума, то мы можем скомпоновать их в одну картинку без шума. Если эти несколько объектов помеще-

ны в разные места одного изображения, то мы всё равно можем воспользоваться этой дополнительной информацией (дело за малым – сначала отыскать эти объекты). И даже если эти объекты выглядят немного поразному или частично перекрыты, то у нас всё равно есть избыточные данные, которыми можно воспользоваться. Нелокальный сглаживающий фильтр использует эту идею: находит похожие области изображения и применяет информацию из них для взаимного усреднения. Разумеется, что части изображения похожи, не означает, что они принадлежат одним и тем же объектам, но, как правило, приближение оказывается достаточно хорошим. Посмотрите на рисунок с линиями. Очевидно, что окна 2 и 3 похожи на окно 1, а окна 4, 5 и 6 — нет.



Предположим, что если окрестность пикселя в одном окне похожа на окрестность соответствующего пикселя в другом окне, то значения этих пикселей можно использовать для взаимного усреднения. Но как найти эти похожие окна? Как поставить им в соответствие веса, с какой силой они влияют друг на друга? Формализуем понятие «похожести» кусков изображения. Во-первых, нам понадобится функция различия двух пикселей. Здесь всё просто: для чёрно-белых изображений обычно исполь-

зуется просто модуль разности значений пикселей. Если имеется какая-то дополнительная информация об изображении, возможны варианты. Например, если известно, что чистое изображение состоит из объектов абсолютно чёрного цвета на полностью белом фоне, хорошей идеей будет использовать метрику, которая сопоставляет нулевое расстояние пикселям, не сильно отличающимся по цвету. В случае цветных изображений опять-таки возможны варианты. Можно использовать евклидову метрику в формате RGB или что-нибудь похитрее в формате HSV. «Непохожесть» окон изображения это всего лишь сумма различий их пикселей.

Теперь хорошо бы преобразовать полученную величину в более привычный формат веса $\omega \in [0, 1]$. Передадим вычисленное на предыдущем шаге различие окон в какую-нибудь убывающую (или хотя бы невозрастающую) функцию и нормируем полученный результат суммой значений этой функции на всех возможных окнах. Обычно в роли такой функции выступает знакомая всем до зубовного скрежета $e^{(-x^2/h^2)}$, где x – расстояние, а h – параметр разброса весов. Чем больше h, тем меньше влияет различие между окнами на сглаживание. При h /rightarrow /infty, все окна вне зависимости от различия между ними в равной мере вносят вклад в каждый пиксель, и изображение получится идеально серым, при h /rightarrow 0, значимый вес будет только у окна, соответствующего самому себе, и сглаживания не получится.

Таким образом, наивная математическая модель нелокального усредняющего фильтра выглядит так:

$$I_d(j) = \sum_{j \in I} \omega(i, j) I(j)$$

i-тый пиксель результирующего изображения равен сумме всех пикселей исходного изображения, взятых с весами ω, где вес это

$$\omega(i, j) = \frac{1}{Z(i)} e^{\frac{\left\| I(N_i) - I(N_j) \right\|_2^2}{h^2}}$$

а нормирующий делитель

$$Z(i) = \sum_{j \in I} e^{\frac{\left\| I(N_i) - I(N_j) \right\|_2^2}{h^2}}$$

Альтернативная метрика, предложенная выше:

$$\rho(x, y, t) = \begin{cases} 0, & \text{if } |x - y| < t; \\ |x - y| - t, & \text{otherwise} \end{cases}$$

Во всех случаях $\left\|I(N_i) - I(N_j)\right\|$ обозначает поэлементную разницу между окнами, как описано выше. Если внимательно присмотреться, то итоговая формула получается почти такой же, как и у билатерального фильтра, только в экспоненте вместо геометрического расстояния между пикселями и цветовой разницы находится разница между лоскутами картинки, а также суммирование проводится по всем пикселям изображения.

Рассуждения об эффективности реализации. Очевидно, что сложность предложенного алгоритма $O(n^2(2r+1))$, где r – радиус окна, по которому вычисляется похожесть частей изображения, а n – полное количество пикселей, ведь мы для каждого пикселя сравниваем его окрестность размера $2r+1$ с окрестностью каждого другого пикселя. Это не слишком хорошо, потому что наивная реализация алгоритма достаточно медленно работает даже на изображениях 300x300. Обычно предлагаются следующие улучшения:

• Очевидно, что необязательно пересчитывать все веса всё время, ведь вес пикселя i для пикселя j и наоборот – равны. Если сохранять вычисленные веса, время выполнения сокращается вдвое, но требуется $O(n^2)$ памяти для хранения весов.

• Для большинства реальных изображений веса между большей частью пикселей будут равны нулю. Так почему бы не обнулить их вовсе? Можно использовать эвристику на основе сегментации изображения, которая будет подсказывать, где считать вес попиксельно бессмысленно.

• Экспоненту при вычислении весов можно заменить на более дешёвую для вычисления функцию. Хотя бы кусочную интерполяцию той же экспоненты.

Обычный вопрос, который возникает при рассмотрении этого алгоритма: почему мы используем только значения центрального пикселя в области? Вообще говоря, ничего не мешает поставить в основную формулу суммирование по всей области, как в билатеральном сглаживании, и это даже несильно повлияет на сложность, но результат изменённого алгоритма, скорее всего, получится слишком размытым. Не забывайте, что в реальных изображениях даже одинаковые объекты редко бывают абсолютно идентичными, а такая модификация алгоритма усреднит их. Чтобы воспользоваться информацией с соседних элементов изображения, можно предварительно выполнить обычное билатеральное сглаживание. Хотя если известно, что исходные объекты были или должны в итоге получиться абсолютно одинаковыми (сглаживание текста), то подобное изменение как раз пойдёт на пользу.

Наивная реализация алгоритма (C + OpenCV 2.4.11):

```
#include "targetver.h"
#include <stdio.h>
#include <math.h>
#include <opencv2/opencv.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/contrib/contrib.hpp"
#include <opencv2/legacy/legacy.hpp>
#include <stdlib.h>
#include <stdio.h>
    using namespace cv;
    /*
Returns measure of diversity between two pixels. The more pixels are different
the bigger output number.
Usually it's a good idea to use Euclidean distance but there are variations.
*/
double distance_squared(unsigned char x, unsigned char y)

unsigned char zmax = max(x, y);
unsigned char zmin = min(x, y);
return (zmax - zmin)*(zmax - zmin);

/*
Returns weight of interaction between regions: the more dissimilar are regions
the less resulting influence.
Usually decaying exponent is used here. If distance is Euclidean, being combined,
they form a typical Gaussian
function.
/
double decay_function(double x, double dispersion)

return exp(-x/dispersion);


int conform_8bit(double x)
```

```
if (x < 0)
return 0;
else if (x > 255)
return 255;
else
return static_cast<int>(x);

    double distance_over_neighbourhood(unsigned char* data, int x00, int
y00, int x01, int y01, int radius, int step)

double dispersion = 15000.0; //should be manually adjusted
double accumulator = 0;
for(int x = -radius; x < radius + 1; ++x)

for(int y = -radius; y < radius + 1; ++y)

accumulator += distance_squared(static_cast<unsigned char>(data[(y00
+ y)*step + x00 + x]),
static_cast<unsigned char>(data[(y01 + y)*step + x01 + x]));


return decay_function(accumulator, dispersion);
int main(int argc, char* argv[])

int similarity_window_radius = 3; //may vary; 2 is enough for text filtering
char* imageName = "text_noised_30.png";
IplImage* image = 0;
image = cvLoadImage(imageName, CV_LOAD_IMAGE_GRAYSCALE);
if (image == NULL)
printf("Can not load image
n");
getchar();
exit(-1);

CvSize currentSize = cvGetSize(image);
int width = currentSize.width;
int height = currentSize.height;
int step = image->widthStep;
unsigned char *data = reinterpret_cast<unsigned char *>(image->imageData);
vector<double> processed_data(width*height, 0);
//External cycle
```

```
for(int x = similarity_window_radius + 1; x < width - similarity_window_radius
- 1; ++x)

printf("x:
for(int y = similarity_window_radius + 1; y < height - similarity_window_radius
- 1; ++y)

//Inner cycle: computing weight map
vector<double> weight_map(width*height, 0);
double* weight_data = &weight_map[0];
double norm = 0;
for(int xx = similarity_window_radius + 1; xx < width - similarity_window_radius
- 1; ++xx)
for(int yy = similarity_window_radius + 1; yy < height - similarity_window_radius
- 1; ++yy)

double weight = distance_over_neighbourhood(data, x, y, xx, yy, similarity_window_radius,
step);
norm += weight;
weight_map[yy*step + xx] = weight;

//After all weights are known, one can compute new value in pixel
for(int xx = similarity_window_radius + 1; xx < width - similarity_window_radius
- 1; ++xx)
for(int yy = similarity_window_radius + 1; yy < height - similarity_window_radius
- 1; ++yy)
processed_data[y*step + x] += data[yy*step + xx]*weight_map[yy*step +
xx]/norm;


//Transferring data from buffer to original image
for(int x = similarity_window_radius + 1; x < width - similarity_window_radius
- 1; ++x)
for(int y = similarity_window_radius + 1; y < height - similarity_window_radius
- 1; ++y)
data[y*step + x] = conform_8bit(processed_data[y*step + x]);
cvSaveImage("gray_denoised.png image);

cvReleaseImage(&image);
return 0;
```

Оценка алгоритма. К сожалению, высокая вычислительная сложность алгоритма ограничивает его применение на практике, тем не менее, он показывает хорошие результаты на задачах сглаживания изображений с повторяющимися элементами. Главная проблема этого алгоритма кроется в его основной идее вычисления весов. Приходится для каждого пикселя проходить по окрестностям всех других пикселей. Даже если предположить, что двукратный просмотр всех пикселей – это неизбежное зло, то просмотр его соседей кажется лишним, ведь даже для радиуса окна 3 он увеличивает время подсчёта весов в 49 раз. Вернёмся немного назад к первоначальной идее. Мы хотим найти похожие места на изображении, чтобы использовать их для сглаживания друг друга. Но нам необязательно попиксельно сравнивать все возможные окна на картинке, чтобы найти похожие места. У нас уже есть хороший способ для обозначения и сравнивания интересных элементов изображения! Разумеется, я говорю о разнообразных дескрипторах особенностей. Обычно под этим подразумевается SIFT, но в нашем случае лучше использовать что-нибудь менее точное, ведь целью на этом этапе является найти много достаточно похожих областей, а не несколько точно таких же.
В случае применения дескрипторов шаг подсчёта весов выглядит так:
• Предварительно считаем дескрипторы в каждой точке изображения
• Проходим по изображения и для каждого пикселя
• Проходим по каждому другому пикселю
• Сравниваем дескрипторы
• Если дескрипторы похожие, то сравниваем окна попиксельно и считаем вес
• Если разные, обнуляем вес
• Чтобы не проходить по изображению второй раз, посчитанные веса и координаты пикселей заносятся в отдельный список. Когда в списке накопилось достаточное число элементов, можно прекращать поиск.
• Если не получилось найти заданное количество похожих точек (пиксель ничем не примечателен), просто проходим по изображению и считаем веса как обычно

Достоинства такого подхода:
• Улучшается сглаживание в особых точках, которые нам обычно больше всего и интересны
• Если на изображении много похожих особых точек, заметно повышение скорости

Недостатки:

• Не факт, что будет достаточно много похожих точек, чтобы окупить подсчёт дескрипторов

• Результат и ускорение алгоритма сильно зависят от выбора дескриптора

Выводы:

• Алгоритм хорошо работает на периодических текстурах

• Алгоритм хорошо сглаживает однородные области

• Чем больше изображение, тем больше похожих областей и тем лучше работает алгоритм. Но безбожно долго.

• Области, для которых не удалось найти подобных, сглаживаются плохо (решается предварительным применением билатерального фильтра или рассмотрением поворотов окон)

• Как и для многих других алгоритмов, приходится подгонять значение сглаживающего параметра

Пример Гауссовского фильтра. Мы рассмотрим один из наиболее часто используемых фильтров для смазывания изображения, Гауссовский фильтр с использованием функции библиотеки OpenCV GaussianBlur() . Этот фильтр разработан специально для снятия высокочастотного шума с изображений.

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;
using namespace cv;
int main(int argc, char** argv)
Mat image , blurredImage;

// Load the image file
image = imread(argv[1], CV_LOAD_IMAGE_COLOR);

// Report error if image could not be loaded
if(!image.data)
cout«"Error loading image»>;
return -1;

// Apply the Gaussian Blur filter.
// The Size object determines the size of the filter (the "range"of the blur)
GaussianBlur( image, blurredImage, Size( 9, 9 ), 1.0);
// Show the blurred image in a named window
imshow("Blurred Image blurredImage);
// Wait indefinitely untill the user presses a key
waitKey(0);

    return 0;
```

Ещё одна реализация механизма сглаживания. Рассмотрим простой тестовый пример, который просто выводит картинку, имя которой передано в виде первого параметра программы (если параметров нет — будет пытаться открыть файл Image0.jpg).

```
    #include <cv.h> #include <highgui.h>
#include <stdlib.h>
#include <stdio.h>
IplImage* image = 0;
IplImage* src = 0;
int main(int argc, char* argv[])

// имя картинки задаётся первым параметром
char* filename = argc == 2 ? argv[1] : "Image0.jpg";
// получаем картинку
image = cvLoadImage(filename,1);
// клонируем картинку
src = cvCloneImage(image);
printf("[i] image:
assert( src != 0 );

// окно для отображения картинки
cvNamedWindow("original CV_WINDOW_AUTOSIZE);

// показываем картинку
cvShowImage("original image);
// выводим в консоль информацию о картинке
printf( "[i] channels:
printf( "[i] pixel depth:
printf( "[i] width:
printf( "[i] height:
printf( "[i] image size:
printf( "[i] width step: // ждём нажатия клавиши
cvWaitKey(0);
// освобождаем ресурсы
cvReleaseImage(& image);
cvReleaseImage(&src);
// удаляем окно
cvDestroyWindow("original");
return 0;
```

рассмотрим новые функции, которые использовались в данном примере:
IplImage* cvLoadImage(const char* filename, int iscolor=CV_LOAD_IMAGE_COLOR);
— загружает картинку из файла.
filename — имя файла
iscolor — определяет как представить картинку
iscolor > 0 — цветная картинка с 3-мя каналами
iscolor == 0 — картинка будет загружена в формате GRAYSCALE (градации серого)
iscolor < 0 — картинка будет загружена как есть:
/* 8bit, color or not */
#define CV_LOAD_IMAGE_UNCHANGED -1
/* 8bit, gray */
#define CV_LOAD_IMAGE_GRAYSCALE 0
/* ?, color */
#define CV_LOAD_IMAGE_COLOR 1
/* any depth, ? */
#define CV_LOAD_IMAGE_ANYDEPTH 2
/* ?, any color */
#define CV_LOAD_IMAGE_ANYCOLOR 4
функция поддерживает следующие форматы изображений:
- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF
IplImage* cvCloneImage(const IplImage* image);
— делает полную копию изображения image, включая заголовок, данные и ROI (Region Of Interest — Обе функции возвращают указатель на картинку IplImage.
Картинка, в OpenCV, представлена структурой вида:
// OpenCV2.0
include
opencv

cxtypes.h
typedef struct _IplImage

int nSize; /* sizeof(IplImage) */
int ID; /* version (=0)*/
int nChannels; /* Most of OpenCV functions support 1,2,3 or 4 channels */
int alphaChannel; /* Ignored by OpenCV */
int depth; /* Pixel depth in bits: IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16S,
IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F are supported.
*/
char colorModel[4]; /* Ignored by OpenCV */
char channelSeq[4]; /* ditto */
int dataOrder; /* 0 - interleaved color channels, 1 - separate color channels.
cvCreateImage can only create interleaved images */
int origin; /* 0 - top-left origin,
1 - bottom-left origin (Windows bitmaps style). */
int align; /* Alignment of image rows (4 or 8).
OpenCV ignores it and uses widthStep instead. */
int width; /* Image width in pixels. */
int height; /* Image height in pixels. */
struct _IplROI *roi; /* Image ROI. If NULL, the whole image is selected.
*/
struct _IplImage *maskROI; /* Must be NULL. */
void *imageId; /* */
struct _IplTileInfo *tileInfo; /* */
int imageSize; /* Image data size in bytes
(==image->height*image->widthStep
in case of interleaved data)*/
char *imageData; /* Pointer to aligned image data. */
int widthStep; /* Size of aligned image row in bytes. */
int BorderMode[4]; /* Ignored by OpenCV. */
int BorderConst[4]; /* Ditto. */
char *imageDataOrigin; /* Pointer to very origin of image data
(not necessarily aligned) -
needed for correct deallocation */ }
IplImage;