目录

[在VS2012中配置Opencv3.0 3](#_Toc460953891)

[1 下载和安装Opencv SDK（software development kit 软件开发工具包） 3](#_Toc460953892)

[2 配置环境变量 3](#_Toc460953893)

[3 VS2012中具体工程的配置 4](#_Toc460953894)

[4 小结 5](#_Toc460953895)

[ROI和初级图像混合-线性混合操作 7](#_Toc460953896)

[1 定义ROI的两种方式 7](#_Toc460953897)

[2 线性图像混合操作 7](#_Toc460953898)

[3 应用实例--将一个logo加入到原图中选定的ROI 8](#_Toc460953899)

[矩阵（图像）的掩码操作 11](#_Toc460953900)

[1 图像进行掩码操作的目的和原理 11](#_Toc460953901)

[2 opencv中实现掩码操作的两种方法 11](#_Toc460953902)

[改变图像的对比度和亮度 15](#_Toc460953903)

[（changing the contrast and brightness of an image） 15](#_Toc460953904)

[1 学习到的内容 15](#_Toc460953905)

[2 对比度和亮度变换的原理 15](#_Toc460953906)

[3 实现代码 15](#_Toc460953907)

[C/C++语言知识整理 18](#_Toc460953908)

[1 标识符const在程序代码中的作用 18](#_Toc460953909)

[2 字符’#’在程序代码中的作用 18](#_Toc460953910)

[2.1 宏定义 18](#_Toc460953911)

[2.2 文件包含 19](#_Toc460953912)

[2.3 条件编译 19](#_Toc460953913)

[3 关于名字空间（namespace） 20](#_Toc460953914)

[3.1 为什么需要使用名字空间 20](#_Toc460953915)

[3.2 什么是名字空间 20](#_Toc460953916)

[3.3 如何使用名字空间 20](#_Toc460953917)

[4 用typedef声明新类型名 21](#_Toc460953918)

[为程序窗口添加滑动条 23](#_Toc460953919)

[（函数createTrackbar()的使用） 23](#_Toc460953920)

[1 滑动条函数createTrackbar()简介 23](#_Toc460953921)

[2 滑动条函数createTrackbar()参数说明 23](#_Toc460953922)

[3 createTrackbar()函数应用举例 23](#_Toc460953923)

[Opencv小常识 26](#_Toc460953924)

[1 waitKey();的使用 26](#_Toc460953925)

[Opencv中基本图形的绘制 28](#_Toc460953926)

[1 Point和Scalar两个结构体的介绍 28](#_Toc460953927)

[2 直线段的绘制 28](#_Toc460953928)

[3 椭圆/椭圆弧的绘制 28](#_Toc460953929)

[4 圆的绘制 29](#_Toc460953930)

[5 矩形的绘制 29](#_Toc460953931)

[6 绘制填充多边形 30](#_Toc460953932)

[7 综合实例（来自opencv\sources\samples\cpp\tutorial\_code\core\Matrix/draw1.cpp） 30](#_Toc460953933)

[霍夫线变换、霍夫圆变换 35](#_Toc460953934)

[1 霍夫线变换 35](#_Toc460953935)

[1.1 霍夫线变换原理概述 35](#_Toc460953936)

[1.2 标准霍夫线变换HoughLines()函数定义： 35](#_Toc460953937)

[1.3 统计概率霍夫线变换HoughLinesP()函数定义： 35](#_Toc460953938)

[1.3 Sample（取自opencv tutorials） 36](#_Toc460953939)

[2 霍夫圆变换 38](#_Toc460953940)

[2.1 霍夫圆变换原理概述 38](#_Toc460953941)

[2.2 霍夫圆变换函数HoughCircles()定义 38](#_Toc460953942)

[2.3 sample（取自opencv tutorials） 38](#_Toc460953943)

[Opencv中调用鼠标事件 40](#_Toc460953944)

[1 设置回调函数cvSetMouseCallback() 40](#_Toc460953945)

[2 专门的鼠标操作回调函数CvMouseCallback() 40](#_Toc460953946)

[3 实例程序 41](#_Toc460953947)

[寻找图像轮廓 43](#_Toc460953948)

[Mat类和IplImage和CvMat之间的相互转换 46](#_Toc460953949)

[1 Mat->IplImage 46](#_Toc460953950)

[2 Mat->CvMat 46](#_Toc460953951)

[3 IplImage-> Mat 46](#_Toc460953952)

[4 CvMat->Mat 46](#_Toc460953953)

在VS2012中配置Opencv3.0.0

1 下载和安装Opencv SDK（software development kit 软件开发工具包）

Opencv各个版本都可以在其官网<http://opencv.org/>找到，下载完后得到文件OpenCV 3.0.0，双击后会提示解压到某个地方，比如D:\Program Files，（因为OpenCV项目文件打包的时候，根目录就是opencv，所以我们不需要额外的新建一个名为opencv的文件夹，然后再解压，那是多此一举的事情）然后点击Extract按钮。

解压之后，里面主要有“build”和“sources”两个文件夹。其中，build里面是使用OpenCV相关的文件，我们如果只是使用OpenCV的话呢，就只用管build里面的内容。官方示例集，也就是samples文件夹里面的示例程序，在sources文件夹里面。

2 配置环境变量

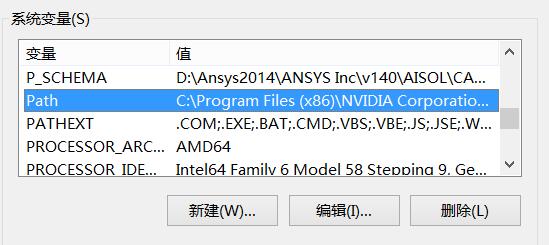
这步的配置方法如下:

【计算机】->【（右键）属性】->【高级系统设置】->【高级（标签）】->【环境变量】。然

后在“系统变量”中找到“Path”变量，点击“编辑”，根据VS程序里调试程序时使用的编译器类型：

（1）如果使用的是 “Win32”编译器，需在“Path”中添加“F:\opencv\build\x86\vc11\bin”这条路径；

（2）如果使用的是 “X64”编译器，需在“Path”中添加“F:\opencv\build\x64\vc11\bin”这条路径；注意不要删掉里面原有的路径，并用“；”和原有路径隔开：（建议将以上两个路径都添加上去）



注：变量值实际为bin文件夹的路径；若编译环境为Microsoft Visual Studio 2012，则选择vc11; 若编译环境为Microsoft Visual Studio 2013，则是vc12;变量添加完成后最好注销系统，才会生效。（注意：实际配置时应该根据自己的opencv文件在电脑中的实际路径进行配置）

3 VS2012中的配置

这里需要在VS2012中新建一个工程，比如Win32控制台应用程序（注意建立过程中要勾选空项目）；空项目建好之后，右击“源文件”，添加一个.CPP文件；之后便是配置“包含目录”、“库目录”和“附加依赖项”了。网上的教程一般可以找到两种配置方式：具体工程的配置和一次性永久配置。

（1）具体工程的配置

这种配置方式比较麻烦，每次新建一个工程都得重新配置一次。步骤如下：

（1.1）选择所建立的工程，右击选择“属性”，在“通用属性”-》“VC++目录”中，找到“包含目录”和“库目录”，其中“包含目录”中添加路径“F:\opencv\build\include”；然后在“库目录”中添加路径“F:\opencv\build\x86\vc11\lib”（如果使用win32编译器）或“F:\opencv\build\x64\vc11\lib”（如果使用x64编译器）。

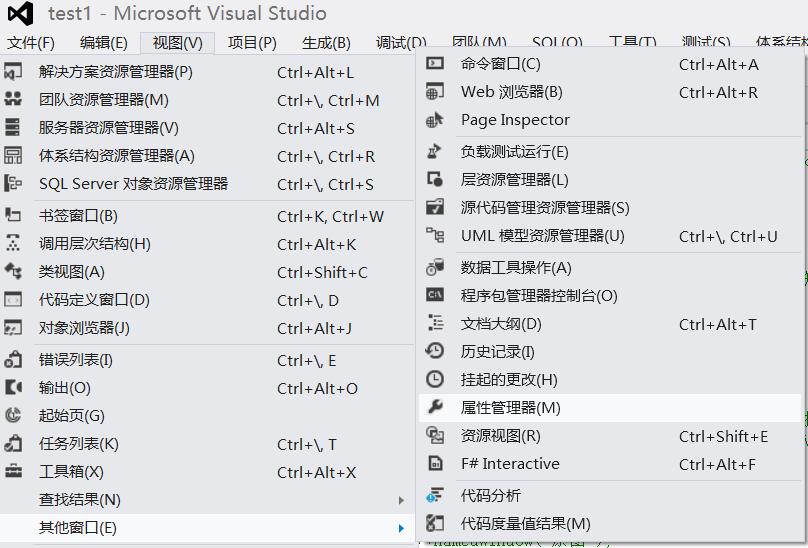
（1.2）最后配置“附加依赖项”，点击“属性”-》“通用属性”-》“链接器”-》“输入”。在其“附加依赖项”中添加以下路径（在网上了解到，数字后带“d”的是Debug版本的.lib文件，同时还有相同数量的不带“d”的是Release版本的.lib文件，所以这里只添加了Debug版本的.lib文件，估计配置用Release版本时会出错）：

opencv\_ts300d.lib

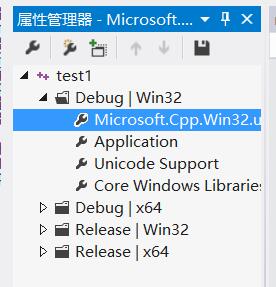
opencv\_world300d.lib

（2）一次性永久配置

同样先建立一个具体工程，然后点击“视图”->点击“其他窗口”->找到并点击“属性管理器”。如下图所示：



打开“属性管理器”之后，根据自己选择的编译器类型是“win32”或“x64”，“Debug模式”或“Release模式”，有4个选择，根据自己的具体情况选择。比如我常用的是“win32”编译器、“Debug模式”。如下图所示：



点击图中蓝色部分，点击右键“属性”，接下来的方法就和“（1）”中方法相同了，找到并配置“包含目录”、“库目录”和“附加依赖项”。

这样，以后新建工程时就不用重新配置了。

为了配置简单，建议在所有涉及选择“win32”还是“x64”时都选择“win32”，也就是添加各种路径时都选择“x86”路径，不选择“x64”。

4 小结

通过配置过程和简单的学习之后，搞清楚了前面配置的“包含目录”中为什么要添加路径“F:\opencv\build\include”了，因为我们在程序中要用到的关于计算机视觉的所有函数，其声明和实现的头文件（.hpp文件）源代码都在“include”文件夹中，该文件夹里面有“opencv”文件夹和“opencv2”文件夹，其中“opencv”文件夹中存放的的 最早的OpenCV1.0时代的所有组件的.hpp文件；而“opencv2”文件夹中存放的是OpenCV2.0版本之后新增加的很多组件,功能更加强大。

举个例子，我们在图像处理时需要定义一个Mat类型的变量，怎么实现？

其实，Mat类是在OpenCV的core.hpp这个头文件中命名的，而core.hpp这个源文件是放在“F:\opencv\build\include\opencv2\core”这个文件夹中。所以首先肯定是需要在该程序的工程属性的“包含目录”中添加“F:\opencv\build\include”这个路径，然后在程序的开头再进一步包含core.hpp头文件：#include<opencv2/core/core.hpp>（这里选择的是opencv2文件夹，说明我们引用的core.hpp头文件是opencv2.0版本以后的新的组件，这里包含的头文件还不能是：#include<opencv/core.hpp>，因为选择这条路径说明我们引用的是早期opencv1.0版本中的core.hpp头文件，而opencv1.0版本中还没有Mat类）。

下图是在网上找到的关于在VS中配置opencv的原理说明：

（该文章网址：<http://blog.csdn.net/lu597203933/article/details/17717131>，

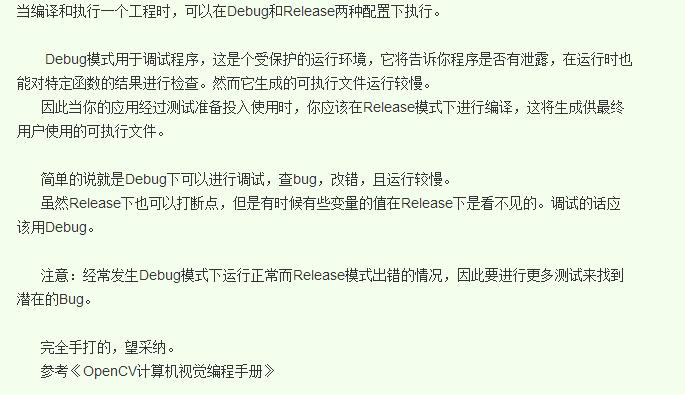
另外介绍lib文件和.dll文件的区别的文章网址：<http://www.cnblogs.com/devilm> sg/articles/1266336.html）



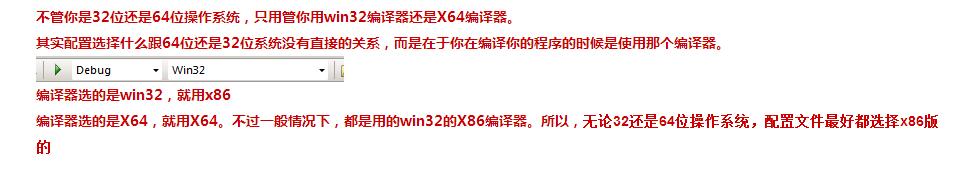


注：

1、关于VS2012中Debug和Release两种模式的选择请参考以下说法（来自“百度知道”）：



2、关于编译平台选择Win32还是X64请参考以下说法（来自“浅墨”的博文-OpenCV入门教程之一）：



笔记中绝大部分内容参考的是毛星云（浅墨）写的博文OpenCV入门教程之一,网址：<http://blog.csdn.net/poem_qianmo/article/details/19809337>。

ROI和初级图像混合-线性混合操作

ROI（region of interest）的意思是选择感兴趣区域，也就是可以在一张图片当中选择任意一片感兴趣的区域，从而对该片区域进行一系列操作，比如进行图像的线性混合操作。

1 定义ROI的两种方式

第一种是使用cv:Rect.顾名思义，cv::Rect表示一个矩形区域。指定矩形的左上角坐标（构造函数的前两个参数）和矩形的长宽（构造函数的后两个参数）就可以定义一个矩形区域。比如看如下代码：

Mat image= imread("F:\\2016\_Summer\\OPENCV3.0\_Test\\test1\\dota.jpg",199);

Mat logo= imread("F:\\2016\_Summer\\OPENCV3.0\_Test\\test1\\logo.jpg");

Mat imageROI;

imageROI=image(Rect(0,0,logo.cols,logo.rows));

目的是选择图像image中的某块ROI（感兴趣区域），这里用的是Rect()来实现，该函数声明如下：

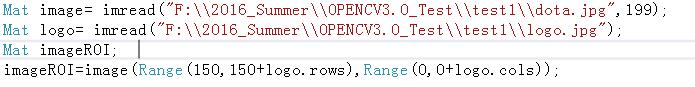


其中\_x、\_y表示ROI区域的左上角顶点距离图像image左上角顶点的横向和纵向距离。也就确定了ROI区域的一个顶点（左上角），然后，\_width和\_height两个参数分别确定ROI区域的宽和高（也就是矩形的长和宽），至此，原始图像image中的ROI矩形区域就选定好了。

第二种定义ROI的方式是指定感兴趣行或列的范围（Range）。Range是指从起始索引到终止索引（不包括终止索引）的一连段连续序列。cv::Range可以用来定义Range。Range（）的声明如下：



对应的代码例子如下：



其中第一个Range（150,150+logo.rows）确定了ROI的高度，Range（0,0+logo.cols）确定了ROI的长度，而ROI的左上角顶点距离原始图像image的左上角顶点的纵向、横向距离分别为150和0。所以：

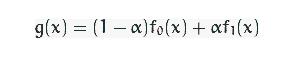
imageROI=image(Rect(0,150,logo.cols,logo.rows));

等效于

imageROI=image(Range(150,150+logo.rows),Range(0,0+logo.cols));

2 线性图像混合操作

线性混合操作是一种典型的二元（两个输入）的像素操作，它的理论公式是这样的：



我们通过在范围0到1之间改变alpha值，来对两幅图像（f0（x）和f1（x））或两段视频（同样为（f0（x）和f1（x））产生时间上的画面叠化（cross-dissolve）效果，就像幻灯片放映和电影制作中的那样。

实现方面，我们主要运用了OpenCV中addWeighted（）（加权）函数，其函数原型如下：



* 第一个参数，InputArray类型的src1，表示需要加权的第一个数组，常常填一个Mat。
* 第二个参数，alpha，表示第一个数组的权重
* 第三个参数，src2，表示第二个数组，它需要和第一个数组拥有相同的尺寸和通道数。
* 第四个参数，beta，表示第二个数组的权重值。
* 第五个参数，dst，输出的数组，它和输入的两个数组拥有相同的尺寸和通道数。
* 第六个参数，gamma，一个加到权重总和上的标量值。看下面的式子自然会理解。

第七个参数，dtype，输出阵列的可选深度，有默认值-1。;当两个输入数组具有相同的深度时，这个参数设置为-1（默认值），即等同于src1.depth（）。

所以，对两张图像进行线性混合的前提是两个Mat变量必须具有相同的尺寸和通道数。如果用数学公式来表达，addWeighted函数计算如下两个数组（src1和src2）的加权和，得到结果输出给第四个参数。即addWeighted函数的作用可以被表示为为如下的矩阵表达式为：



其中的I，是多维数组元素的索引值。而且，在遇到多通道数组的时候，每个通道都需要独立地进行处理。另外需要注意的是，当输出数组的深度为CV\_32S时，这个函数就不适用了，这时候就会内存溢出或者算出的结果压根不对。

3 应用实例--将一个logo加入到原图中选定的ROI

相关代码如下：

Mat image= imread("F:\\2016\_Summer\\OPENCV3.0\_Test\\test1\\dota.jpg",199);//载入原图

Mat logo= imread("F:\\2016\_Summer\\OPENCV3.0\_Test\\test1\\logo.jpg");//载入logo

namedWindow("原画");

imshow("原画",image);

namedWindow("logo图");

imshow("logo图",logo);

Mat imageROI;//可用两种方法在原图中选定与logo相同尺寸的ROI区域

//方法一

//imageROI=image(Rect(50,150,logo.cols,logo.rows));

//方法二

imageROI=image(Range(150,150+logo.rows),Range(50,50+logo.cols));

//将logo加到原图上 （logo和原图选定的ROI进行加权混合）

addWeighted(imageROI,0.2,logo,0.8,0.0,imageROI);

//显示结果

namedWindow("【4】原画+logo图");

imshow("【4】原画+logo图",image);

运行结果如下图所示：







通过改变选定ROI的两种方式中的参数可以调节logo图在原图中的位置；通过改变addweight（）函数中的权重参数可以调节logo图在原图中的颜色深浅。

矩阵（图像）的掩码操作

参考网址：

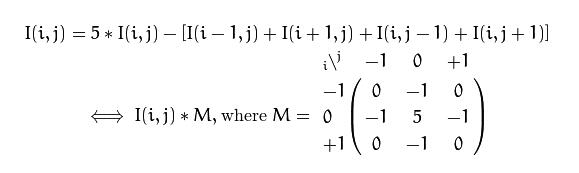
<http://blog.csdn.net/qq_22033759/article/details/48168559>

<http://docs.opencv.org/2.4/doc/tutorials/core/mat-mask-operations/mat-mask-operations.html?highlight=mask%20operations%20matrices>

<http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/core/mat-mask-operations/mat-mask-operations.html#maskoperationsfilter>

1 图像进行掩码操作的目的和原理

对一张图像进行掩码操作的主要目的是为了增强图像的对比度。由于图像在程序中是以矩阵的形式进行存储的，掩码操作其实就是对图像矩阵中的每一个像素值做一个处理得到新的像素值。处理的方法是利用原像素点和其周围上下左右四个点的像素值做一个加权平均，公式如下图：



如上图所示，比如对坐标是（0,0）的像素点进行掩码操作，也就是数值为5的像素点，其操作后得到的新像素值=5\*5-（-1-1-1-1）=29。

2 opencv中实现掩码操作的两种方法

第一种是基本的利用指针遍历矩阵的每个像素点的方法（矩阵的边界无法使用加权平均），其程序代码如下（封装成一个函数）：

void Sharpen(const Mat& myImage, Mat& Result)

{

// 刚进入函数的时候，我们要确保输入图像是无符号字符类型的。为了做到这点，

//我们使用了 CV\_Assert 函数。若该函数括号内的表达式为false，则会抛出一个错误。

CV\_Assert(myImage.depth() == CV\_8U);

//创建了一个与输入有着相同大小和类型的输出图像

Result.create(myImage.size(), myImage.type());

const int nChannels = myImage.channels();

for (int j = 1; j < myImage.rows - 1; ++j)

{

const uchar\* previous = myImage.ptr<uchar>(j - 1);

const uchar\* current = myImage.ptr<uchar>(j);

const uchar\* next = myImage.ptr<uchar>(j + 1);

uchar\* output = Result.ptr<uchar>(j);

for (int i = nChannels; i < nChannels\*(myImage.cols - 1); ++i)

{

\*output++ = saturate\_cast<uchar>(5 \* current[i]

- current[i - nChannels] - current[i + nChannels] - previous[i] - next[i]);

}

}

//在图像的边界上，上面给出的公式会访问不存在的像素位置（比如(0,-1)）。

//因此我们的公式对边界点来说是未定义的。

//一种简单的解决方法，是不对这些边界点使用掩码，而直接把它们设为0：

Result.row(0).setTo(Scalar(0));

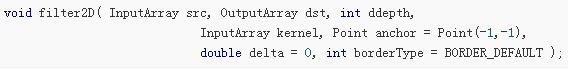
Result.row(Result.rows - 1).setTo(Scalar(0));

Result.col(0).setTo(Scalar(0));

Result.col(Result.cols - 1).setTo(Scalar(0));

}

第一种是自己写代码实现，第二种则是利用OpenCv提供的filter2D函数来实现，需要自定义一个内核。该函数定义如下：



* 前两个参数为输入和输出图
* 第三个参数为图像的类型
* 第四个为要输入的内核
* 第五个为模板的中心点
* 第五个为补充图像的值
* 第六个是边界类型（后三个一般用不到，取省缺值）

其相关代码如下：

//第二种方法；

//需要先定义一个内核；

Mat kern = (Mat\_<char>(3, 3) << 0, -1, 0,

-1, 5, -1,

0, -1, 0);

//调用filter2D函数；

filter2D(image, result, CV\_8U, kern);

对比之下，使用第二种方法的代码要比第一种要稍简单。下面是测试效果图和测试程序：



测试小程序代码：

#include<iostream>

#include<opencv2\core\core.hpp>//Mat类包含在这个头文件中；

#include <opencv2\highgui\highgui.hpp>//显示图像用的，因为用到了显示图片，所以需要包含进去；

#include<opencv2/imgproc/imgproc.hpp>//函数cvtColor()包含在这个头文件中；

using namespace cv;

using namespace std;

void Sharpen(const Mat&,Mat&);

int main()

{

Mat image=imread("D:\\12.jpg");//Mat是一个n维矩阵类，声明在<opencv2/core/core.hpp>中;

Mat result;

//第一种方法；

//Sharpen(image,result);

//第二种方法；

//需要先定义一个内核；

Mat kern = (Mat\_<char>(3, 3) << 0, -1, 0,-1, 5, -1,0, -1, 0);

//调用filter2D函数；

filter2D(image, result, CV\_8U, kern);

namedWindow("原图",CV\_WINDOW\_AUTOSIZE );

namedWindow("掩码效果图",CV\_WINDOW\_AUTOSIZE );

imshow("原图",image);

imshow("掩码效果图",result);

waitKey(0);//等待直到用户按下一个按键之后退出。

return 0;

}

void Sharpen(const Mat& myImage, Mat& Result)

{

// 刚进入函数的时候，我们要确保输入图像是无符号字符类型的。为了做到这点，

//我们使用了 CV\_Assert 函数。若该函数括号内的表达式为false，则会抛出一个错误。

CV\_Assert(myImage.depth() == CV\_8U);

//创建了一个与输入有着相同大小和类型的输出图像

Result.create(myImage.size(), myImage.type());

const int nChannels = myImage.channels();

for (int j = 1; j < myImage.rows - 1; ++j)

{

const uchar\* previous = myImage.ptr<uchar>(j - 1);

const uchar\* current = myImage.ptr<uchar>(j);

const uchar\* next = myImage.ptr<uchar>(j + 1);

uchar\* output = Result.ptr<uchar>(j);

for (int i = nChannels; i < nChannels\*(myImage.cols - 1); ++i)

{

\*output++ = saturate\_cast<uchar>(5 \* current[i]

- current[i - nChannels] - current[i + nChannels] - previous[i] - next[i]);

}

}

//在图像的边界上，上面给出的公式会访问不存在的像素位置（比如(0,-1)）。

//因此我们的公式对边界点来说是未定义的。

//一种简单的解决方法，是不对这些边界点使用掩码，而直接把它们设为0：

Result.row(0).setTo(Scalar(0));

Result.row(Result.rows - 1).setTo(Scalar(0));

Result.col(0).setTo(Scalar(0));

Result.col(Result.cols - 1).setTo(Scalar(0));

}

改变图像的对比度和亮度

（changing the contrast and brightness of an image）

1 学习到的内容

* 访问像素值
* 用0初始化矩阵
* saturate\_cast 是做什么用的，以及它为什么有用
* 一些有关像素变换的精彩内容

2 对比度和亮度变换的原理

一般来说，图像处理算子是带有一幅或多幅输入图像、产生一幅输出图像的函数。图像变换可分为以下两种：

* 点算子（像素变换）
* 邻域（基于区域的）算子

两种常用的点过程（即点算子），是用常数对点进行乘法和加法运算：



两个参数alpha （> 0） 和 beta 一般称作 增益 和 偏置 参数。我们往往用这两个参数来分别控制 对比度 和 亮度 。其中f(i,j)是源图像像素点，g(i,j)是输出图像对应的像素点。

3 实现代码

有两种方法可以实现上面对比度和亮度的控制。第一种采用for循环遍历每个像素点；第二种是采用Mat::convertTo()函数，达到的效果一样。测试代码如下：

/\*...........程序内容：改变图像的对比度和亮度(两种方法)..........\*/

#include<iostream>

#include<opencv2\core\core.hpp>

#include <opencv2\highgui\highgui.hpp>

using namespace cv;

using namespace std;

double alpha; /\*\*< 控制对比度 \*/

int beta; /\*\*< 控制亮度 \*/

int main()

{

Mat image=imread("D:\\12.jpg");

//构造一个初始化为0，并且和image同尺寸同类型的矩阵；

Mat new\_image = Mat::zeros( image.size(), image.type() );

/// 初始化：手动设置alpha和beta值；

cout << " Basic Linear Transforms " << endl;

cout << "-------------------------" << endl;

cout << "\* Enter the alpha value [1.0-3.0]: ";

cin >> alpha;

cout << "\* Enter the beta value [0-100]: ";

cin >> beta;

//第一种方法：遍历每一个像素点；

//执行运算 new\_image(i,j) = alpha\*image(i,j) + beta

//我们要访问图像的每一个像素。因为是对RGB图像进行运算，

//每个像素有三个值（R、G、B），所以我们要分别访问它们

for( int y = 0; y < image.rows; y++ )

{

for( int x = 0; x < image.cols; x++ )

{

for( int c = 0; c < 3; c++ )

{

new\_image.at<Vec3b>(y,x)[c] = saturate\_cast<uchar>( alpha\*( image.at<Vec3b>(y,x)[c] ) + beta );

}

}

}

/\*.................................................................................\*/

/\*

1、为了访问图像的每一个像素，我们使用这一语法： image.at<Vec3b>(y,x)[c]

其中， y 是像素所在的行， x 是像素所在的列， c 是R、G、B（0、1、2）之一。

2、因为 \alpha \cdot p(i,j) + \beta 的运算结果可能超出像素取值范围，

还可能是非整数（如果alpha 是浮点数的话），所以我们要用 saturate\_cast

对结果进行转换，以确保它为有效值。

\*/

//第二种方法：采用Mat::conventTo()函数；

//image.convertTo(new\_image, -1, alpha, beta);

/\*.................................................................................\*/

/// 创建窗口

namedWindow("Original Image", 1);

namedWindow("New Image", 1);

/// 显示图像

imshow("Original Image", image);

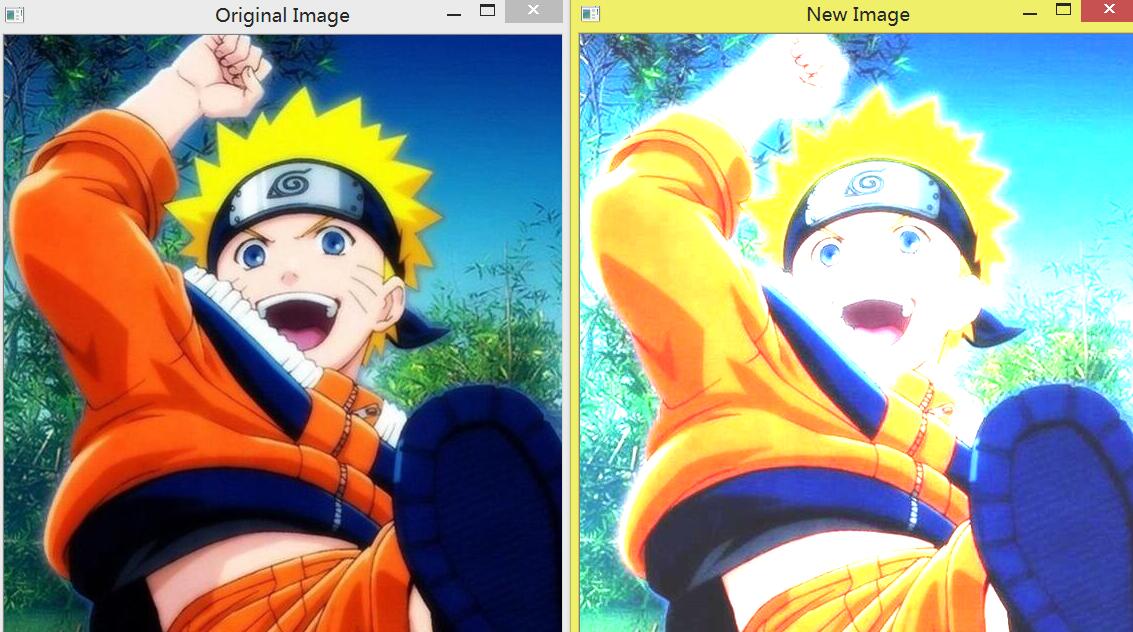
imshow("New Image", new\_image);

waitKey(0);//等待直到用户按下一个按键之后退出。

return 0;

}

比如当alpha=2,beta=50时，对比度和亮度变换的效果如下：



下面是函数Mat::convertTo()的原型：



C/C++语言知识整理

1 标识符const在程序代码中的作用

const是用来声明一个常量的，当你不想让一个值被改变时就用const,

const int max && int const max 是没有区别的，都可以。

不涉及到指针const很好理解。

涉及到指针的情况：

int b=100;

const int \* a=&b [1];

int const \*a=&b; [2]

int\* const a=&b; [3]

const int\* const a =&b; [4]

如果const位于星号的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；如果const位于星号的右侧，const就是修饰指针本身，即指针本身是常量。因此，[1]和[2]的情况相同，都是指针所指向的内容为常量（const放在变量声明符的位置无关），这种情况下不允许对内容进行更改操作，如不能\*a = 3 ；[3]为指针本身是常量，而指针所指向的内容不是常量，这种情况下不能对指针本身进行更改操作，如a++是错误的；[4]为指针本身和指向的内容均为常量。

2 字符’#’在程序代码中的作用

程序中以‘#’开头的语句成为预处理命令，一般有3个用途：

2.1 宏定义

在C语言源程序中常常用一个标识符来表示一个较为复杂的字符串，称为“宏”。宏定义有两种，包括不带参数的宏定义和带参数的宏定义，其中被定义为“宏”的标识符称为“宏名”，在程序进行编译预处理时，对程序中所有出现的宏名，都用宏定义中的字符串去替换。

第一种不带参数的宏定义格式如下：

#define 标识符 字符串

比如经常用到的：

#define PI 3.1415926

注意点：

* “标识符”一般用大写字母或者具有直观意义的标识符命名（用小写字母也不会错）；
* 宏定义不是说明或语句，行后不加分号；
* 宏定义必须写在函数之外，其作用域从宏定义命令起到源程序结束，如果要终止其作用域可使用#undef命令，比如要在某处终止宏定义PI，需在该处加上：#undef PI
* 不带参数的宏定义的目的其实就是用一个简单的标识符代替程序中经常要出现，比较繁杂的字符串或者需要随时更改的数值参数等，减少书写麻烦。

第二种是带参数的宏定义，格式如下：

#define 宏名（参数表） 字符串

例如：

#define M（y） y\*y+5\*y /\*宏定义\*/

……

float k;

k=M(6); /\*宏调用\*/

则k=6\*6+5\*6

说白了，带参数的宏定义的效果和一个简单的函数类似，带参数的宏定义可以嵌套使用。

2.2 文件包含

“文件包含”是C预处理程序的另一个重要功能，所谓的“文件包含”是将另一个源文件的内容包含进来。它的一般形式为：

#include“文件名”或者<文件名>

例如：

#include”iostream.h”

#include<iostream>

文件包含的功能是将指定的文件插入该命令行位置取代该命令行，从而将指定的文件和当前的源程序文件连成一个源文件。

对于程序设计人员来说，文件包含是很有用的 ，一个大的程序可以分成多个模块，有多个程序员分别编程；有些公用的符号常量或宏定义可以单独组成一个文件，在其它文件的开头包含该命令即可使用；这样可以提高效率。

2.3 条件编译

在程序编译时，如果希望按不同的条件去编译不同的程序部分时，就需要条件编译，条件编译有以下几种形式：

形式1：

#ifdef 标识符

程序段1

#else

程序段2

#endif

它的作用是如果标识符已经被#define命令定义过则对程序段1进行编译；否则对程序段2进行编译。如果没有程序段2（为空），则本格式中的#else可以没有。

形式2：

#ifndef 标识符

程序段1

#else

程序段2

#endif

与第一种格式刚好相反，如果标识符未被#define定义过则对程序段1进行编译，否则对程序段2进行编译。

形式3：

#if 常量表达式

程序段1

#else

程序段2

#endif

它的功能是，如果常量表达式的值为真（非0），则对程序段1进行编译，否则对程序段2进行编译。因此可以使程序在不同的条件下，完成不同的功能。

3 关于名字空间（namespace）

在使用opencv库的图像处理程序中，我们会看到程序开始是会有：using namespace cv；还有using namespace std;这样的语句，这里的“cv”、“std”是“名字空间”，名字空间的使用是C++对原先C语言的扩展，C语言中是没有“名字空间”的概念的。下面简单介绍一下关于名字空间的内容：

3.1 为什么需要使用名字空间

其实，引入名字空间的概念是为了解决程序中名字冲突的问题，即在程序运行过程中，遇到相同名字的变量，系统能不能正确的区分它们。

举个例子，因为一个大型的程序往往是由很多个程序员各自完成相应的部分，最后组合成一个完整的程序的，这样，难免会出现不同的程序员使用相同的标识符的情形，这样就会出现名字冲突。于是名字空间出现了。

3.2 什么是名字空间

所谓名字空间就是由一个程序设计者命名的内存区域。程序设计者可以根据需要指定一些有名字的空间域，把一些全局标识符分别放在各个名字空间中，从而和其它全局标识符隔开。

名字空间的作用类似于操作系统中的目录和文件之间的关系，不同目录中的文件可以重名，只要调用文件时指明文件所在的目录，这样就避免出现名字冲突的问题了。

3.3 如何使用名字空间

使用名字空间时可以根据需要设置名字空间，不同的名字空间代表不同的域；注意：不同的名字空间不能同名。

使用名字空间的语法如下：

Namespace 名字空间名

{

定义成员

}

其中的成员类型可以是：常量、变量、函数、结构体、类、模板等。还可以是名字空间，也就是名字空间的嵌套。这样，要使用某个名字空间中的某个成员时，可以通过“名字空间名：：成员名”的方法进行调用。

针对以下三种情况，C++对名字空间的使用提供了3种简化方式：

（1）、使用名字空间别名

遇到名字空间的名字比较长或者嵌套的情况,这种情况下，每次调用名字空间中的成员都需要码一长串的名字空间名，很不方便，C++提供了“使用名字空间别名”的方法简化名字空间的书写，比如：

namespace Information

{………}

可以使用一个很短的别名代替：

namespace Info=Information;

此时Info和Information完全等价，可以替换。

（2）、使用“using名字空间的成员名；”

可以在程序中使用“using名字空间的成员名”来简化名字空间的成员访问，using后面必须是由名字空间限定的成员。比如：

using ns::tax;

这样后面访问tax（）时就相当于访问ns::tax()，这样也可以简化名字空间的使用。

注意点：

不能在同一个作用域中用using声明不同名字空间的相同名字的成员名，这样会编译出错。

（3）、使用“using namespace 名字空间名”

在遇到一段程序中要经常访问某个名字空间中的多个成员时，可以使用“using namespace 名字空间名”。

注意点：

如果使用using namespace引入多个名字空间，要保证这些名字空间中不能有同名的成员，否则同样会引起同名冲突。

4 用typedef声明新类型名

用typedef声明新类型名来代替已有的类型，其实就相当于给原来的类型取了一个别名，并且可以用这个新取的别名代替原来类型名来定义变量，举个例子：

typedef int count;

这就是给int类型取了一个新的名字count，定义之后，count类型就代表int类型，于是我们就可以在程序中用count定义变量：

count a=10;

完全等价于：

int a=10;

typedef也可以声明结构体类型：

typedef struct

{

int month;

int day;

int year;

}date;

这样，date就是一个新声明的结构体类型，可以用于定义变量，比如：

date birthday；

date \*p;

总结：

1、typedef可以用来声明各种新类型名，但是不能直接用来定义变量；

2、用typedef只是已经存在的类型增加一个类型名，并没有创造新的类型；

3、当在不同源程序中用到同一类型数据时，常用typedef声明一些数据类型，把它们单独放在一个头文件中，然后在需要用到它们的文件中用#include命令把它们包含进来，以提高编程效率，用typedef有利于程序的通用和移植。

1. S是
2. 是
3. S
4. 是

为程序窗口添加滑动条

（函数createTrackbar()的使用）

1 滑动条函数createTrackbar()简介

Opencv提供了一个很有用的函数createTrackbar()，即为程序窗口添加一个滑动条，如下图所示：



滑动条上的按钮停靠位置其实就相当于一个变量x，而滑动条的最大长度就是该变量的取值范围，通过鼠标移动滑动条上的按钮，变量x可以任意改变，而一旦x改变，createTrackbar()就自动调用用户设定的回调函数，一般x定义成一个全局变量，这样回调函数中也可以使用x作为一个参数，从而达到一些特定的效果。

2 滑动条函数createTrackbar()参数说明

createTrackbar()函数定义在opencv的highgui（GUI Graphical User Interface 图形用户界面）模块中，所以使用createTrackbar()函数时需要包含头文件：#include <opencv2/highgui.hpp>。

createTrackbar()函数的声明如下：



createTrackbar()函数的参数一般有5个：

第1个参数是滑动条的名字；

第2个参数是滑动条所在窗口的名字，也就是需要在哪个窗口创建滑动条；

第3个参数是滑动条的位置变量&slider，它是一个整型指针变量；

第4个参数是设置滑动条的最大范围max，它是一个整型变量；

第5个参数是回调函数的名字，只要滑动条位置一发生改变，系统自动调用该回调函数。

3 createTrackbar()函数应用举例

下面是一个实现两幅图像的线性混合的例子：

#include <opencv2/core/core.hpp>

#include <opencv2/highgui/highgui.hpp>

#include<iostream>

using namespace cv;

/// 全局变量的声明与初始化

const int alpha\_slider\_max = 100;

int alpha\_slider;

double alpha;

double beta;

/// 声明存储图像的变量

Mat src1;

Mat src2;

Mat dst;

/\*\*

\* @function on\_trackbar

\* @定义响应滑动条的回调函数

\*/

void on\_trackbar( int, void\* )

{

//这里alpha\_slider是int类型，需要先将其强制转换成double类型，提高精度；

alpha = (double) alpha\_slider/alpha\_slider\_max ;

beta = ( 1.0 - alpha );

//加权函数addWeighted()的用法参考相关笔记；

addWeighted( src1, alpha, src2, beta, 0.0, dst);

imshow( "Linear Blend", dst );

}

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

{

/// 加载图像 (两图像的大小与类型要相同)

src1 = imread("D:\\12.jpg");

//两幅图线性混合需要其尺寸和类型完全相同，这里src2是一张尺寸和类型与src1相同的黑色（0）图像；

src2=Mat::zeros(src1.size(),src1.type());

//通过Mat.data判断是否成功读入图像，若成功则返回1，若读取失败返回0；

//若读取失败，程序直接return -1，程序结束；

if( !src1.data ) { printf("Error loading src1 \n"); return -1; }

if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

/// 初始化为零

alpha\_slider = 0;

/// 创建窗体

namedWindow("Linear Blend");

/// 在创建的窗体中创建一个滑动条控件

char \*TrackbarName="alpha";

createTrackbar( TrackbarName, "Linear Blend", &alpha\_slider, alpha\_slider\_max, on\_trackbar );

/// 结果在回调函数中显示

on\_trackbar( alpha\_slider, 0 );

/// 按任意键退出

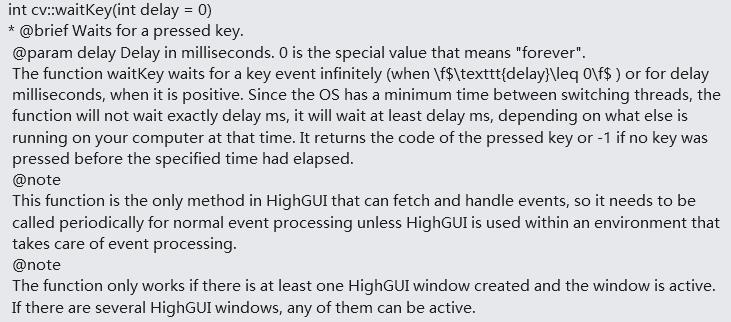
waitKey(0);

return 0;

}

Opencv小常识

1 waitKey();的使用



* 函数描述：等待一个按键；
* 参数delay：int型，单位是ms，省缺值为0，意为一直等待按键按下，没有按键按下程序就一直等在此处；
* 参数delay是一个正整数时，比如waitKey(10),表示程序在此暂停10ms，起到延时效果；
* waitKey()不管参数是多少，如果没有触发键盘，waitKey()返回值总是“-1”，如果一旦触发键盘上的任何按键，waitKey()返回值是该按键对应的ASCII码；
* 利用waitKey()函数的按键功能可以控制HighGUI窗口的很多事件，举例如下：

while( true )

{

int c;

c = waitKey(10);

if( (char)c == 27 )

{ break; }

if( (char)c == 'u' )

{ pyrUp( tmp, dst, Size( tmp.cols\*2, tmp.rows\*2 ) );

printf( "\*\* Zoom In: Image x 2 \n" );

}

else if( (char)c == 'd' )

{ pyrDown( tmp, dst, Size( tmp.cols/2, tmp.rows/2 ) );

printf( "\*\* Zoom Out: Image / 2 \n" );

}

描述：

* 如果没有按键按下，程序在此暂停10ms（函数返回值-1）；
* 如果按下“Esc”键（Esc的ASCII码=27），程序跳出该循环；
* 如果按下小写字母’u’,程序执行pyrUp（）函数等内容；
* 如果按下小写字母’d’,程序执行pyrDown（）函数等内容；

Opencv中基本图形的绘制

下面主要介绍怎样在图像中绘制基本图形如直线、圆、椭圆、矩形、任意填充多边形。

1 Point和Scalar两个结构体的介绍

Point数据结构表示了图像中坐标x和y表示的2D点，可定义为：

Point pt;

pt.x = 10;

pt.y = 8;

或者

Point pt = Point(10, 8);

Scalar表示具有4个元素的数组。该类型在OpenCV中被大量用于传递像素值，比如可以用来表示RGB彩色图像中的某点像素值（有R、G、B3通道，只需3个参数，第4个参数省略），比如：

Scalar( a, b, c )

那么定义的RGB颜色值为： Red = c, Green = b and Blue = a。

2 直线段的绘制

Opencv提供line()函数，可以在图像中指定两点间绘制直线，line()函数定义如下：

CV\_EXPORTS\_W void line(InputOutputArray img, Point pt1, Point pt2, const Scalar& color,int thickness = 1, int lineType = LINE\_8, int shift = 0);

* img：直线被绘制在img图像上面；
* pt1和pt2：指定直线的两个端点坐标；
* color:指定绘制直线段用什么颜色的线，比如Scalar( 0, 0, 255) ，即红色，而Scalar( 0, 255，0) 为绿色，Scalar( 255, 0, 0) 为蓝色；
* thickness：指定直线段的线粗，省缺值为1，该参数设置必须大于等于0，当线粗为负数时编译会出错；
* lineType:指定绘制矩形所用线的类型，省缺值为8；

3 椭圆/椭圆弧的绘制

利用opencv提供的ellipse()函数，可以方便的在图像中绘制出指定大小和位置的椭圆/任意角度的椭圆弧，ellipse()函数定义如下：

CV\_EXPORTS\_W void ellipse(InputOutputArray img, Point center, Size axes,

double angle, double startAngle, double endAngle,

const Scalar& color, int thickness = 1,

int lineType = LINE\_8, int shift = 0);

参数解释：

* img：椭圆被绘制在img图像上面；
* center：指定椭圆的中心坐标，比如Point（400,400）；
* axes：指定椭圆的外切矩形的长宽，比如Size（600,200）；
* angle:指定椭圆整体的旋转角度，比如angle=0，表示椭圆是水平放置的（即大径线水平），当angle值为正整数a时，表示椭圆整体顺时针旋转a度；
* startAngle和endAngle:表示椭圆弧的起始角和终止角，当其分别为0和360时，绘制出完整椭圆；
* color:指定绘制椭圆用什么颜色，比如Scalar( 0, 0, 255) ，即红色，而Scalar( 0, 255，0) 为绿色，Scalar( 255, 0, 0) 为蓝色；
* thickness：指定椭圆的线粗，省缺值为1，当线粗大于等于0时绘制空心椭圆，当线粗为负整数时，椭圆内部将被填充；
* lineType:指定椭圆线的类型，省缺值为8；

4 圆的绘制

利用opencv提供的circle()函数，可以在图像中画指定位置和大小的圆（空心圆和实心圆），circle()函数定义如下：

CV\_EXPORTS\_W void circle(InputOutputArray img, Point center, int radius,

const Scalar& color, int thickness = 1,

int lineType = LINE\_8, int shift = 0);

参数解释：

* img：圆被绘制在img图像上面；
* center：指定圆的中心坐标，比如Point（400,400）；
* radius：指定圆的半径大小；
* color:指定绘制圆用什么颜色，比如Scalar( 0, 0, 255) ，即红色，而Scalar( 0, 255，0) 为绿色，Scalar( 255, 0, 0) 为蓝色；
* thickness：指定圆的线粗，省缺值为1，当线粗大于等于0时绘制空心圆，当线粗为负整数时，圆内部将被填充；
* lineType:指定绘制圆所用线的类型，省缺值为8；

5 矩形的绘制

Opencv中提供了rectangle()函数来在图像中的指定位置绘制指定大小的矩形，其定义如下：

CV\_EXPORTS\_W void rectangle(InputOutputArray img, Point pt1, Point pt2,

const Scalar& color, int thickness = 1,

int lineType = LINE\_8, int shift = 0);

参数解释：

* img：矩形被绘制在img图像上面；
* pt1和pt2：指定矩形的两个对角顶点的坐标；
* color:指定绘矩形用什么颜色，比如Scalar( 0, 0, 255) ，即红色，而Scalar( 0, 255，0) 为绿色，Scalar( 255, 0, 0) 为蓝色；
* thickness：指定矩形的线粗，省缺值为1，当线粗大于等于0时绘制空心矩形，当线粗为负整数时，矩形内部将被填充；
* lineType:指定绘制矩形所用线的类型，省缺值为8；

6 绘制填充多边形

利用opencv提供的fillPoly()函数可以绘制任意形状的填充多边形，其函数定义如下：

CV\_EXPORTS void fillPoly(Mat& img, const Point\*\* pts,

const int\* npts, int ncontours,

const Scalar& color, int lineType = LINE\_8, int shift = 0,

Point offset = Point() );

函数解释：

* ing:填充多边形将被画到图像 img 上;
* \*pts:指向填充多边形的顶点集合的指针,一个Point类型的数组，数组元素是每个多边形顶点；
* npt：一个int型常指针，指定要绘制的填充多边形顶点数目；
* ncontours：要绘制的填充多边形轮廓数量，绘制一个多边形就填1；
* color:填充多边形的颜色定义，比如 Scalar( 255, 255, 255), 为白色；
* lineType:指定绘制填充多边形所用线的类型，省缺值为8；

7 综合实例（来自opencv\sources\samples\cpp\tutorial\_code\core\Matrix/draw1.cpp）

#include <opencv2/core.hpp>

#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

#define w 400

using namespace cv;

/// Function headers

void MyEllipse( Mat img, double angle );

void MyFilledCircle( Mat img, Point center );

void MyPolygon( Mat img );

void MyLine( Mat img, Point start, Point end );

/\*\*

\* @function main

\* @brief Main function

\*/

int main( void ){

/// Windows names

char atom\_window[] = "Drawing 1: Atom";

char rook\_window[] = "Drawing 2: Rook";

/// Create black empty images

Mat atom\_image = Mat::zeros( w, w, CV\_8UC3 );

Mat rook\_image = Mat::zeros( w, w, CV\_8UC3 );

/// 1. Draw a simple atom:

/// -----------------------

/// 1.a. Creating ellipses

MyEllipse( atom\_image, 90 );

MyEllipse( atom\_image, 0 );

MyEllipse( atom\_image, 45 );

MyEllipse( atom\_image, -45 );

/// 1.b. Creating circles

MyFilledCircle( atom\_image, Point( w/2, w/2) );

/// 2. Draw a rook

/// ------------------

/// 2.a. Create a convex polygon

MyPolygon( rook\_image );

/// 2.b. Creating rectangles

rectangle( rook\_image,

Point( 0, 7\*w/8 ),

Point( w, w),

Scalar( 0, 255, 255 ),

-1,

8 );

/// 2.c. Create a few lines

MyLine( rook\_image, Point( 0, 15\*w/16 ), Point( w, 15\*w/16 ) );

MyLine( rook\_image, Point( w/4, 7\*w/8 ), Point( w/4, w ) );

MyLine( rook\_image, Point( w/2, 7\*w/8 ), Point( w/2, w ) );

MyLine( rook\_image, Point( 3\*w/4, 7\*w/8 ), Point( 3\*w/4, w ) );

/// 3. Display your stuff!

imshow( atom\_window, atom\_image );

moveWindow( atom\_window, 0, 200 );

imshow( rook\_window, rook\_image );

moveWindow( rook\_window, w, 200 );

waitKey( 0 );

return(0);

}

/// Function Declaration

/\*\*

\* @function MyEllipse

\* @brief Draw a fixed-size ellipse with different angles

\*/

void MyEllipse( Mat img, double angle )

{

int thickness = 2;

int lineType = 8;

ellipse( img,

Point( w/2, w/2 ),

Size( w/4, w/16 ),

angle,

0,

360,

Scalar( 255, 0, 0 ),

thickness,

lineType );

}

/\*\*

\* @function MyFilledCircle

\* @brief Draw a fixed-size filled circle

\*/

void MyFilledCircle( Mat img, Point center )

{

int thickness = -1;

int lineType = 8;

circle( img,

center,

w/32,

Scalar( 0, 0, 255 ),

thickness,

lineType );

}

/\*\*

\* @function MyPolygon

\* @function Draw a simple concave polygon (rook)

\*/

void MyPolygon( Mat img )

{

int lineType = 8;

/\*\* Create some points \*/

Point rook\_points[1][20];

rook\_points[0][0] = Point( w/4, 7\*w/8 );

rook\_points[0][1] = Point( 3\*w/4, 7\*w/8 );

rook\_points[0][2] = Point( 3\*w/4, 13\*w/16 );

rook\_points[0][3] = Point( 11\*w/16, 13\*w/16 );

rook\_points[0][4] = Point( 19\*w/32, 3\*w/8 );

rook\_points[0][5] = Point( 3\*w/4, 3\*w/8 );

rook\_points[0][6] = Point( 3\*w/4, w/8 );

rook\_points[0][7] = Point( 26\*w/40, w/8 );

rook\_points[0][8] = Point( 26\*w/40, w/4 );

rook\_points[0][9] = Point( 22\*w/40, w/4 );

rook\_points[0][10] = Point( 22\*w/40, w/8 );

rook\_points[0][11] = Point( 18\*w/40, w/8 );

rook\_points[0][12] = Point( 18\*w/40, w/4 );

rook\_points[0][13] = Point( 14\*w/40, w/4 );

rook\_points[0][14] = Point( 14\*w/40, w/8 );

rook\_points[0][15] = Point( w/4, w/8 );

rook\_points[0][16] = Point( w/4, 3\*w/8 );

rook\_points[0][17] = Point( 13\*w/32, 3\*w/8 );

rook\_points[0][18] = Point( 5\*w/16, 13\*w/16 );

rook\_points[0][19] = Point( w/4, 13\*w/16 );

const Point\* ppt[1] = { rook\_points[0] };

int npt[] = { 20 };

fillPoly( img,

ppt,

npt,

1,

Scalar( 255, 255, 255 ),

lineType );

}

/\*\*

\* @function MyLine

\* @brief Draw a simple line

\*/

void MyLine( Mat img, Point start, Point end )

{

int thickness = 2;

int lineType = 8;

line( img,

start,

end,

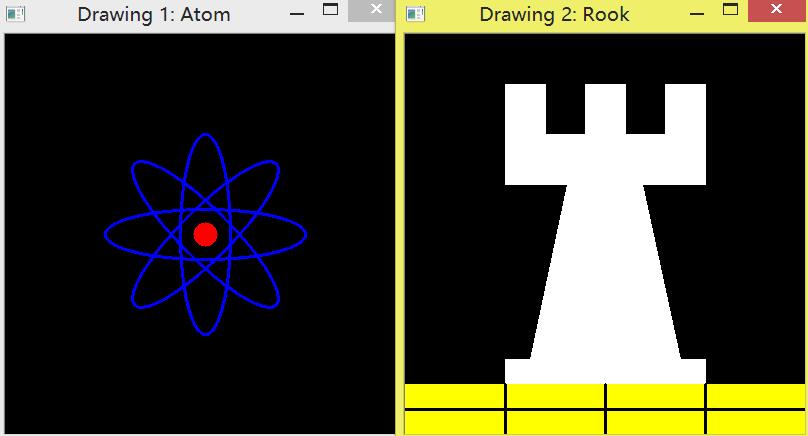
Scalar( 0, 0, 0 ),

thickness,

lineType );

}

程序效果：



霍夫线变换、霍夫圆变换

霍夫变换(Hough Transform)是图像处理中的一种特征提取技术，该过程在一个参数空间中通过计算累计结果的局部最大值得到一个符合该特定形状的集合作为霍夫变换结果。

霍夫线变换是一种用来寻找直线的方法. 在使用霍夫线变换之前, 首先要对图像进行边缘检测的处理，也即霍夫线变换的直接输入只能是边缘二值图像。

而霍夫圆变换是一种在图像中寻找圆的方法。

1 霍夫线变换

* 1. 霍夫线变换原理概述

图像中的某一点可以在平面直角坐标系中用（x,y）表示，也可以在极坐标中用极径极角（rho,theta）表示，通过某个点（x0,y0）的所有直线在以theta为横坐标、rho为纵坐标的坐标系中组成一条正弦曲线，而通过另一点（x1,y1）的所有直线在以theta为横坐标、rho为纵坐标的坐标系中组成另一条正弦曲线，这两条正弦曲线会存在交点（rho，theta），该点就唯一确定了平面直角坐标系中的经过点（x0,y0）和点（x1,y1）的直线信息，霍夫线变换利用概率统计的方法，将图像上经过所有点的所有直线映射到theta-rho坐标系中，当theta-rho坐标系中达到一定数量的正弦线交与某一点时，我们就认为该点代表平面直角坐标系中的一条直线，用这样的方法找出图像中的所有直线。Opencv提供了两个函数：标准霍夫线变换HoughLines()和统计概率霍夫线变换HoughLinesP()

1.2 *标准霍夫线变换HoughLines()函数定义：*

CV\_EXPORTS\_W void HoughLines( InputArray image, OutputArray lines,

double rho, double theta, int threshold,

double srn = 0, double stn = 0,

double min\_theta = 0, double max\_theta = CV\_PI );

参数解释：

* 参数image：待输入的8位、单通道二值图；
* 参数lines：输出检测到的所有直线，数据类型是二维vector向量容器，其中每个向量代表一条直线信息，包括rho和theta，可以定义为：vector<Vec2f> lines，则检测到的直线条数为：lines.size()；
* 参数rho：极径以像素为单位值的分辨率（resolution），可以填1；
* 参数theta：极角参数以弧度为单位的分辨率，可填1度，即PI/180；
* 参数threshold:阈值，极坐标中交于某点的正弦线数量达到该数值时，将该点确定为一条有效直线；
* 其他参数暂时取省缺值；

1.3 统计概率霍夫线变换HoughLinesP()函数定义：

CV\_EXPORTS\_W void HoughLinesP( InputArray image, OutputArray lines,

double rho, double theta, int threshold,

double minLineLength = 0, double maxLineGap = 0 );

参数解释：

* 参数image：待输入的8位、单通道二值图；
* 参数lines：输出检测到的所有直线，数据类型是4维vector向量容器，其中每个向量代表一条直线段两端点的坐标，可以定义为：vector<Vec4i> lines，则检测到的直线条数为：lines.size()；
* 参数rho：极径以像素为单位值的分辨率（resolution），可以填1；
* 参数theta：极角参数以弧度为单位的分辨率，可填1度，即PI/180；
* 参数threshold:阈值，极坐标中交于某点的正弦线数量达到该数值时，将该点确定为一条有效直线；
* 其他参数暂时取省缺值；

1.3 Sample（取自opencv tutorials）

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>

using namespace cv;

using namespace std;

void help()

{

cout << "\nThis program demonstrates line finding with the Hough transform.\n"

"Usage:\n"

"./houghlines <image\_name>, Default is pic1.jpg\n" << endl;

}

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

{

Mat src = imread("D:\\14.jpg");

if(src.empty())

{

help();

cout << "can not open "<< endl;

return -1;

}

Mat dst, cdst;

Canny(src, dst, 50, 200, 3);

cvtColor(dst, cdst, CV\_GRAY2BGR);

#if 0

vector<Vec2f> lines;

HoughLines(dst, lines, 1, CV\_PI/180, 100, 0, 0 );

//怎样由rho和theta获得绘制直线段所需的端点坐标值，没懂？

for( size\_t i = 0; i < lines.size(); i++ )

{

float rho = lines[i][0], theta = lines[i][1];

Point pt1, pt2;

double a = cos(theta), b = sin(theta);

double x0 = a\*rho, y0 = b\*rho;

pt1.x = cvRound(x0 + 1000\*(-b));

pt1.y = cvRound(y0 + 1000\*(a));

pt2.x = cvRound(x0 - 1000\*(-b));

pt2.y = cvRound(y0 - 1000\*(a));

line( cdst, pt1, pt2, Scalar(0,0,255), 3, CV\_AA);

}

#else

vector<Vec4i> lines;

HoughLinesP(dst, lines, 1, CV\_PI/180, 100);

for( size\_t i = 0; i < lines.size(); i++ )

{

Vec4i l = lines[i];

line( cdst, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0255,0,255), 1, CV\_AA);

}

#endif

imshow("source", src);

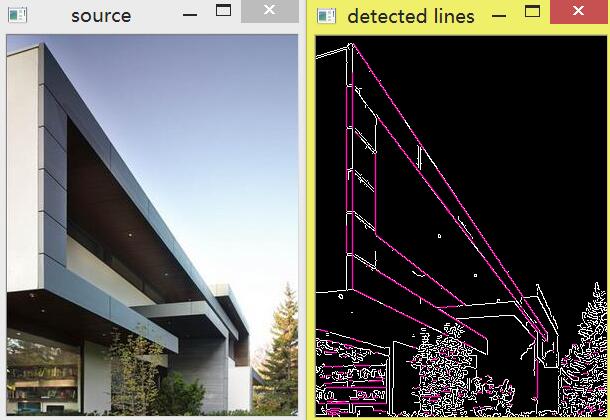
imshow("detected lines", cdst);

waitKey(0);

return 0;

}

检测效果：



2 霍夫圆变换

2.1 霍夫圆变换原理概述

霍夫圆变换的目的是要在图像中检测圆。基本原理和前面的霍夫线变换类似, 只是点对应的二维极径极角空间被三维的圆心点x, y还有半径r空间取代。通过opencv函数HoughCircles()来获取图像中检测到的圆集，再利用绘圆函数circle()将检测到的圆描绘出来。

2.2 霍夫圆变换函数HoughCircles()定义

CV\_EXPORTS\_W void HoughCircles( InputArray image, OutputArray circles,

int method, double dp, double minDist,

double param1 = 100, double param2 = 100,

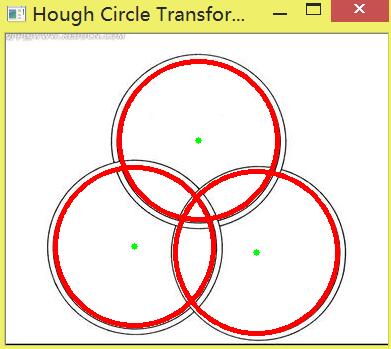
int minRadius = 0, int maxRadius = 0 );

参数解释：

* 参数image：待输入的8位、单通道二值图；
* 参数circles：输出检测到的所有圆，数据类型是3维vector向量容器，其中每个向量包含的圆心坐标x(center)、y(center)以及半径r，可以定义为：vector<Vec3f> lines，则检测到的圆数量为：lines.size()；
* 参数method：指定检测方法. 现在OpenCV中只有霍夫梯度法,只能填：CV\_HOUGH\_GRADIENT;
* 参数dp:累加器图像的反比分辨率，填1；
* 参数minDist: 检测到圆心之间的最小距离,这个值不能太小也不能太大；
* 其他参数暂时取省缺值；

2.3 sample（取自opencv tutorials）

效果图：



代码：

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>

#include <stdio.h>

using namespace cv;

using namespace std;

/\*\* @function main \*/

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

{

Mat src, src\_gray;

/// Read the image

src = imread("D:\\15.jpg", 1 );

if( !src.data )

{ return -1; }

/// Convert it to gray

cvtColor( src, src\_gray, CV\_BGR2GRAY );

/// Reduce the noise so we avoid false circle detection

GaussianBlur( src\_gray, src\_gray, Size(9, 9), 2, 2 );

vector<Vec3f> circles;//该模板类定义在名字空间std中；

/// Apply the Hough Transform to find the circles

HoughCircles( src\_gray, circles, CV\_HOUGH\_GRADIENT, 1, src\_gray.rows/16, 100, 100, 0, 0 );

/// Draw the circles detected

for( size\_t i = 0; i < circles.size(); i++ )

{

Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));

int radius = cvRound(circles[i][2]);

// circle center

circle( src, center, 3, Scalar(0,255,0), -1, 8, 0 );

// circle outline

circle( src, center, radius, Scalar(0,0,255), 3, 8, 0 );

}

/// Show your results

namedWindow( "Hough Circle Transform Demo", CV\_WINDOW\_AUTOSIZE );

imshow( "Hough Circle Transform Demo", src );

waitKey(0);

return 0;

}

Opencv中调用鼠标事件

在opencv中提供了专门针对调用鼠标事件的回调函数。对于鼠标的任意操作，通过回调函数可以实现任何响应。opencv为鼠标操作准备了两个函数：cvSetMouseCallback()和CvMouseCallback()。通过在主函数中设置cvSetMouseCallback()，并根据自己的目的编写响应的回调函数实现对鼠标操作的响应。

1 设置回调函数cvSetMouseCallback()

函数定义如下：

CVAPI(void) cvSetMouseCallback( const char\* window\_name, CvMouseCallback on\_mouse,

void\* param CV\_DEFAULT(NULL));

参数解释：

* 参数windowname:设置对鼠标事件做出响应的窗口名；
* 参数on\_mouse:回调函数的名字，可以任意取名；
* 最后一个参数可取默认值NULL；

2 专门的鼠标操作回调函数CvMouseCallback()

函数定义如下：

typedef void (CV\_CDECL \*CvMouseCallback )(int event, int x, int y, int flags, void\* param);

函数解释：

* 回调函数的返回值类型为void类型；
* 回调函数名可以任意取，只要和上面的cvSetMouseCallback()中回调函数名一致即可；
* 形参event:鼠标事件类型，常见鼠标操作的event事件如下：

CV\_EVENT\_MOUSEMOVE =0, //鼠标移动

CV\_EVENT\_LBUTTONDOWN =1, //按下左键

CV\_EVENT\_RBUTTONDOWN =2, //按下右键

CV\_EVENT\_MBUTTONDOWN =3, //按下中键

CV\_EVENT\_LBUTTONUP =4, //放开左键

CV\_EVENT\_RBUTTONUP =5, //放开右键

CV\_EVENT\_MBUTTONUP =6, //放开中键

CV\_EVENT\_LBUTTONDBLCLK =7, //左键双击

CV\_EVENT\_RBUTTONDBLCLK =8, //右键双击

CV\_EVENT\_MBUTTONDBLCLK =9, //中键双击

CV\_EVENT\_MOUSEWHEEL =10, //滚轮滚动

CV\_EVENT\_MOUSEHWHEEL =11 //横向滚轮滚动

* 形参x和y：表示鼠标在对应窗口中的当前坐标，窗口坐标系原点在窗口左上角，向右为x轴正向，向下为y轴正向；
* 形参flags：表示拖拽、组合键等标志，与event搭配使用，内容如下：

CV\_EVENT\_FLAG\_LBUTTON =1, //左键拖拽

CV\_EVENT\_FLAG\_RBUTTON =2, //右键拖拽

CV\_EVENT\_FLAG\_MBUTTON =4, //中键拖拽

CV\_EVENT\_FLAG\_CTRLKEY =8, //按住CTRL拖拽

CV\_EVENT\_FLAG\_SHIFTKEY =16, //按住Shift拖拽

CV\_EVENT\_FLAG\_ALTKEY =32 //按住ALT拖拽

* 形参param：表示传入或传出的数据（暂时还不会用）；

3 实例程序

程序内容为创建一个绘图板，利用鼠标左键拖住移动，可以进行绘图操作；双击鼠标左键，绘图板清空。源代码如下：

//................程序说明...................

//程序功能：简单实现opencv调用鼠标事件

//环境说明：VS2012+opencv3.0.0

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include<opencv2/core/core.hpp>

#include <iostream>

using namespace cv;

using namespace std;

/// 全局变量

char\* Mouse = "Source image";

Mat src;

bool downl=false;

Point beg=Point(0,0);

Point fin=Point(0,0);

//回调函数声明

void on\_mouse1(int event,int x,int y,int flags,void\* a);

/\*\* @function main \*/

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

{

//加载源图像

src = Mat::zeros(600,600,CV\_8UC3);

namedWindow(Mouse,CV\_WINDOW\_AUTOSIZE);

//设置调用鼠标事件函数

cvSetMouseCallback(Mouse,on\_mouse1,NULL);

imshow(Mouse,src);

//等待用户按任意按键退出程序

waitKey(0);

return 0;

}

//回调函数实现

void on\_mouse1(int event,int x,int y,int flags,void\* a)

{

if (event == CV\_EVENT\_LBUTTONDOWN) //左键按下

{

beg = Point(x, y);

downl = true;

}

else if (event == CV\_EVENT\_LBUTTONUP)//左键放开

{

downl = false;

}

//左键按下且鼠标移动时进行绘图操作

if (downl ==true&&event == CV\_EVENT\_MOUSEMOVE)

{

fin = cvPoint(x, y);

line(src, beg, fin, Scalar(255, 0, 0), 5);

waitKey(5); //可以解决画图时卡顿的问题

imshow(Mouse, src);

beg = fin;

}

//双击鼠标左键时清空绘图面板

else if(downl==false&&event==CV\_EVENT\_LBUTTONDBLCLK)

{

src = Mat::zeros(600,600,CV\_8UC3);

waitKey(5); //可以解决画图时卡顿的问题

imshow(Mouse, src);

}

}

寻找图像轮廓

学习内容

学习利用函数findContour()寻找单通道二值图像的轮廓；

学习利用函数drawContours()绘制绘制图像轮廓；

学习利用函数contourArea()求图像轮廓内部连通区域的面积；

学习利用函数arcLength()求轮廓的长度（周长）；

findContour()函数介绍

findContour()是opencv库提供用于在一张二值图中寻找轮廓的函数，使用的算法是：Suzuki85算法（铃木算法）（没研究过），该函数有2中重载模式，其函数声明如下：

void findContours( InputOutputArray image, OutputArrayOfArrays contours,

OutputArray hierarchy, int mode,

int method, Point offset = Point());

void findContours( InputOutputArray image, OutputArrayOfArrays contours,

int mode, int method, Point offset = Point());

可以看出，第二种重载模式除了少了一个参数hierarchy之外，其他没有区别，可以把第1个重载函数看做完整版，第2个重载函数看做第1个函数少了hierarchy时的简化版。下面解释各参数的含义：

参数image：源图像image必须是一个8位、单通道的二值图象，可以使用函数compare(), inRange()、threshold()、adaptiveThreshold()、Canny()等函数从灰度图或者彩色图中得到。

参数contours：从源图像中检测到的轮廓就存储到contours容器中，contours的定义类型必须是：vector<vector<Point> > contours; 注意：外边的vector向量的每个元素都是一个轮廓，比如contours.[i]表示第i+1条轮廓（第1条轮廓是conyour.[0]）；而里面的vector向量的元素是某条轮廓的所有组成点，所以这里contours的类型类似于二维数组，数组元素是点Point。

参数huerarchy:这个参数包含了检测到的所有轮廓之间的层级关系（继承关系）。Hierarchy的定义类型必须为：vector<Vec4i> hierarchy;即hierarchy也是一个向量，该向量的元素个数等于检测到的轮廓条数。而每个元素如hierarchy[i]又是一个包含4个元素的整型向量：hierarchy[i][0]、hierarchy[i][1]、hierarchy[i][2]、hierarchy[i][3]。这里的4个元素都是int类型，hierarchy[i][0]表示与轮廓contours[i]同层级的下一个轮廓编号（next contour）（注意hierarchy[i][0]不一定等于i+1），如果contours[i]已经是同层级的最后一个轮廓了，那么此时hierarchy[i][0]=-1；hierarchy[i][1] 表示与轮廓contours[i]同层级的上一个轮廓编号（previous contour），如果contours[i]是同层级的第一个轮廓，那么此时hierarchy[i][0]=-1；hierarchy[i][2] 表示轮廓contours[i]的子轮廓编号（first child contour），如果contours[i]不存在子轮廓（即contours[i]轮廓内不再有其他轮廓）那么此时hierarchy[i][2]=-1；hierarchy[i][3] 表示轮廓contours[i]的父轮廓编号（parent contour），如果contours[i]不存在父轮廓（即contours[i]轮廓没有被其他轮廓包含）那么此时hierarchy[i][3]=-1。

参数mode：mode表示检测轮廓的方法。有4种取值：

CV\_RETR\_EXTERNAL：只检测最外层级的轮廓，这里相当于给所有轮廓设置了hierarchy[i][2]= hierarchy[i][3]=-1；即子轮廓和父轮廓都不存在；

CV\_RETR\_LIST：检测所有轮廓，但轮廓之间没有继承关系；

这里可以知道，当mode为前面这两种方式时，没有用到任何层级关系，hierarchy参数就没用到，这里其实就是findContours()函数的第二种声明的用法；

CV\_RETR\_TREE：检测所有轮廓，并且建立所有的继承(包含)关系；

CV\_RETR\_CCOMP：检测所有轮廓，但是仅仅建立两层包含关系。外轮廓放到顶层，外轮廓包含的第一层内轮廓放到底层，如果内轮廓还包含轮廓，那就把这些内轮廓放到顶层去。

参数method：表示组成一条轮廓的点向量的存储方法，可取值如下：

CV\_CHAIN\_APPROX\_NONE：存储轮廓所有的组成点；

CV\_CHAIN\_APPROX\_SIMPLE：只存储水平，垂直，对角直线的起始点；

CV\_CHAIN\_APPROX\_TC89\_L1，CV\_CHAIN\_APPROX\_TC89\_KCOS：使用the flavors of Teh-Chin chain近似算法的一种。

参数offset: 偏移量，用于移动所有轮廓点。当轮廓是从图像的ROI提取的，并且需要在整个图像中分析时，这个参数将很有用。其他一般情况下，这个参数取省缺值。

drawContours()函数介绍

drawContours()函数一般是和findcontours()函数一起使用的，用于绘制出findcontours()函数找出的图像轮廓。其函数声明如下：

void drawContours( InputOutputArray image, InputArrayOfArrays contours,

int contourIdx, const Scalar& color,

int thickness = 1, int lineType = LINE\_8,

InputArray hierarchy = noArray(),

int maxLevel = INT\_MAX, Point offset = Point() );

参数解释：

参数image:目标图像，将轮廓绘制在此目标图像中，一般情况下才目标图像应该和原来找轮廓的图像尺寸大小一致；

参数contours：包含所有输入轮廓的容器，每个轮廓以点向量的形式存储，这里的contours和findContours()函数中的contours是同一个变量，先利用findContours()函数将所有轮廓信息存储在contours中，再利用drawContours()函数把contours中存储的轮廓绘制在图像中；

参数contourIdx:表明要绘制轮廓的编号，因为contours中存储的所有轮廓都有一个从0开始的编号，表示第几个轮廓，一般是和for循环一起使用绘制出所有或是特定的轮廓；注意，当该参数=-1时，表示绘制出所有轮廓，此时就不需要for循环了；

参数color：表示绘制轮廓时用什么颜色；

参数thickness：表示用多粗的线绘制轮廓，有省缺值1；注意，当thickness是负数时（比如取CV\_FILLED），轮廓内部将会被填充；

参数lineType：表示绘制轮廓线的连通属性，有省缺值LINE\_8,一般直接取8；

参数hierarchy：关于轮廓层级关系的可选择的信息，和findContours()函数中的hierarchy是同一个参数，只有当findContours()函数中用到这个参数时（也就是用第一种声明），它才是有效的；当只想绘制出一部分轮廓时才会用到参数hierarchy，具体请参考下一个参数maxLevel；当要绘制所有轮廓时这个参数用不到；

参数maxLevel：表示轮廓的最大层级，如果取0，只有指定的轮廓被绘制出；如果取1，则绘制出指定轮廓及其相同层级的其他轮廓；如果取2，则在1的基础上再绘制出下一层级的轮廓等等（不太明白）；

参数offset：偏移量，用于移动所有轮廓点。当轮廓是从图像的ROI提取的，并且需要在整个图像中分析时，这个参数将很有用。其他一般情况下，这个参数取省缺值。

函数contourArea()函数介绍

函数contourArea()

Mat类和IplImage和CvMat之间的相互转换

对于存储图像，早期的opencv1.0时代使用的是支持C接口的IplImage和CvMat结构；而从opencv2开始使用支持C++接口的Mat类。这里简单总结一下Mat类和IplImage和CvMat之间的相互转换。

1 Mat->IplImage

这一转换非常方便，可以采用简单的赋值运算，例如：

Mat img=Mat：：zeros（3,3，CV\_8UC3）;

IplImage \_img=img;

如果需要调用IplImage类型的指针的话，可以使用去地址操作符：&\_img。

2 Mat->CvMat

和Mat转化成IplImage类似，可以采用赋值操作：

Mat img=Mat：：zeros（3,3，CV\_8UC3）;

CvMat cv\_img=img;

需要注意的是，这两种赋值转化方式的结果是，IplImage和CvMat与Mat类共用同一矩阵数据，没有另外开辟内存空间。

3 IplImage-> Mat

注意的是，opencv2.x版本和opencv3.0以后版本采用的方式不同。

针对于opencv2.x版本：

Mat::Mat(const IplImage\* img, bool copyData=false);

默认情况下，新的Mat类型与原来的IplImage类型共享图像数据，转换只是创建一个Mat矩阵头。当将参数copyData设为true后，就会复制整个图像数据。

而在opencv3.0以后，上面的方法行不通了，但是可以用以下办法：

IplImage \* ipl = ...;

Mat m =cvarrToMat(ipl);

4 CvMat->Mat

与IplImage的转换类似，可以选择是否复制数据：

Mat::Mat(const CvMat\* m, bool copyData=false);

立体匹配的几种方法和一些有帮助的约束条件

1 交叉相关

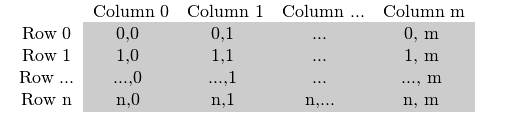
寻找两幅图像中的对应关系，最早用的是交叉相关技术。图像的交叉相关的定义，暂且先理解为和图像的平滑滤波过程类似，需要先定义一个模板H[x,y],然后将模板在源图像中进行移动，将原像素点F[x,y]及其领域像素与模板中对应权重的乘积之和，得到新图像对应像素点G[x,y]。这和数字图像的卷积操作很类似。交叉相关和卷积的区别是：交叉相关是将模板平移到图像点[x,y]不经过翻转，而卷积需要将模板进行翻转。

其具体过程是，对于图像I1中的像素点P1，搜索I2上的选定区域，对P1和P2的领域进行交叉相关运算，把交叉相关响应最大的像素，作为P1的最佳匹配点P2，然后就可以利用P1P2像素对寻找对应3D点的深度等信息。

OpenCV中操作图像像素的常用方法

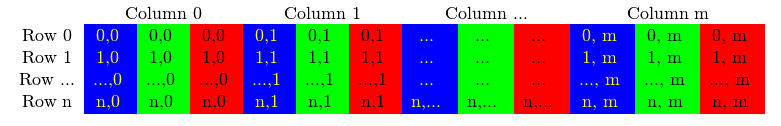
使用OpenCV库处理图像时，经常需要取用某个具体像素坐标P(x,y)处的像素值大小，有时也需要对该像素点进行赋值操作。也就是对具体坐标像素点进行取值和和赋值操作。下面总结一下常用到的3中操作图像像素的方法，在此之前，简单了解一下OpenCV中的图像存储方式：

灰度图像（颜色通道数=1）在OpenCV中是以二维矩阵的方式存储的，如下图所示：



请注意，像素坐标是（0,0）开始的，坐标点P（x,y）代表的是第x+1行、第y+1列的像素点。

对于常用的RGB彩色图像（颜色通道数=3），其图像自OpenCV中任然是以二维矩阵存储的，只不过，矩阵中的列会包含多个子列，其子列个数与通道数相等。下图是RGB颜色模型的矩阵：



所以说，对于一张分辨率为n x m的RGB彩色图，其在OpenCV中对应的二维矩阵的大小是：n x 3\*m。

下面的sample程序实现了3种操作图像像素（赋值和取值）的方法：

/\*

程序功能：

实现使用opencv操作图像像素的3种方式：指针、迭代器、动态地址；并计算了每种操作的时间效率，在执行相同的操作任务下（复制源图像到输出图像），

使用指针操作的效率明显优于另外两种方法。

时间：2017/9/21

参考资料：

1 http://blog.csdn.net/hjl240/article/details/52462340

2 http://www.opencv.org.cn/opencvdoc/2.3.2/html/doc/tutorials/core/how\_to\_scan\_images/how\_to\_scan\_images.html#howtoscanimagesopencv

\*/

#include<opencv2/opencv.hpp>

#include<iostream>

using namespace cv;

using namespace std;

Mat source = imread("sample.jpg");

//使用指针操作像素

void Pointer\_Method(Mat &source, Mat &output)

{

double start = static\_cast<double>(getTickCount());

int nr = source.rows; //行数

int nc = source.cols \* source.channels(); //每行元素的总元素数量

for (int i = 0; i<nr; i++)

{

uchar\* data1 = source.ptr<uchar>(i);//获取第i行的首地址

uchar\* data2 = output.ptr<uchar>(i);

for (int j = 0; j<nc; j++)

{

data2[j] = data1[j];

}

}

double end = static\_cast<double>(getTickCount());

double time = (end - start) / getTickFrequency();//计算时间

cout << "指针操作运行时间为：" << time << "秒" << endl;

imshow("指针操作", output);

}

//使用迭代器操作像素

void Iterator\_Method(Mat &source, Mat &output)

{

double start = static\_cast<double>(getTickCount());

Mat\_<Vec3b>::iterator it1 = source.begin<Vec3b>();//初始位置的迭代器

Mat\_<Vec3b>::iterator itend1 = source.end<Vec3b>();//终止位置的迭代器

Mat\_<Vec3b>::iterator it2 = output.begin<Vec3b>();//初始位置的迭代器

Mat\_<Vec3b>::iterator itend2 = output.end<Vec3b>();//终止位置的迭代器

for (; it1 != itend1; it1++)

{

//处理BGR三个通道

(\*it2)[0] = (\*it1)[0];//B

(\*it2)[1] = (\*it1)[1];//G

(\*it2)[2] = (\*it1)[2];//R

it2++;

}

double end = static\_cast<double>(getTickCount());

double time = (end - start) / getTickFrequency();//计算时间

cout << "迭代器操作运行时间为：" << time << "秒" << endl;

imshow("迭代器操作", output);

}

//使用动态地址操作像素

void On\_The\_Fly\_Method(Mat &source, Mat &output)

{

double start = static\_cast<double>(getTickCount());

int rowNumber = source.rows;

int colNumber = source.cols;

for (int i = 0; i < rowNumber; i++)

for (int j = 0; j < colNumber; j++)

{

//处理BGR三个通道

output.at<Vec3b>(i, j)[0] = source.at<Vec3b>(i, j)[0];//B

output.at<Vec3b>(i, j)[1] = source.at<Vec3b>(i, j)[1];//G

output.at<Vec3b>(i, j)[2] = source.at<Vec3b>(i, j)[2];//R

//若处理的是灰度图像

//output.at<uchar>(i, j) = source.at<uchar>(i, j);

}

double end = static\_cast<double>(getTickCount());

double time = (end - start) / getTickFrequency();//计算时间

cout << "动态地址操作运行时间为：" << time << "秒" << endl;

imshow("动态地址操作", output);

}

//主函数

int main()

{

//首先得初始化输出矩阵

Mat output1=Mat(source.rows, source.cols, source.type());

Mat output2 = Mat(source.rows, source.cols, source.type());

Mat output3 = Mat(source.rows, source.cols, source.type());

//调用三种方法函数

Pointer\_Method(source, output1);

Iterator\_Method(source, output2);

On\_The\_Fly\_Method(source, output3);

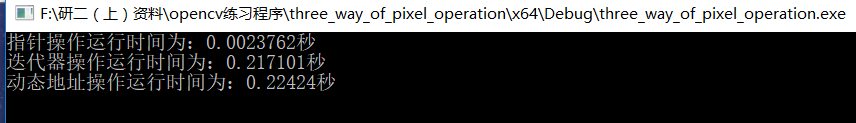
imshow("源图像", source);

waitKey();

return 0;

}

下图显示了相同任务（复制一幅图像）下3种方法的效率对比：



可以看出，使用指针操作的效率远远优于迭代和动态地址操作。