**Core组件进阶**

1. 访问图像中的像素
   1. 图像在内存中的存储方式

图像在内存中一般在内存足够大的系统中可以连续存储，连续存储有助于提升图像扫描速度，可以用isContinuous()来判断矩阵是否是连续存储的。

另外OpenCV的矩阵中包含很多子列，子列的个数与通道数相等，通道顺序是反过来的BGR而不是RGB.

* 1. 颜色空间缩减

三通道的图像，像素颜色最多有1600,0000种，如此之多的颜色格式，在很多系统中是不必要的，所以需要用到颜色空间缩减 – color space reduction。

颜色空间缩减通过当前颜色空间值除以某个输入值实现，即把对颜色空间值抽取一些值。而不是每个都要。

通过缩减运算公式:

在处理图像是以上的公式的每次的运算本身就要耗费很多开销，如果把像素值预先做成表，在后续图像的处理中通过查表方式会快一些。

* 1. LUT函数：Look up table
  2. 计时函数

|  |  |  |
| --- | --- | --- |
| 函数名 | 功能 | 实例 |
| GetTickCount() | 返回某个时间到当前的时钟周期数 | double time0 = static\_cast<double>(getTickCount());  time0 = ((double)getTickCount() - time0)/getTickFrequency(); |
| GetTickFrequency() | 返回CPU一秒钟所走的周期数 |

* 1. 访问图像中像素的三类方法
     1. 用指针操作像素

用指针的方法特点是速度比较快：

void colorReduce(Mat& inputImage, Mat& outputImage, int div)

{

//参数准备

outputImage = inputImage.clone(); //拷贝实参到临时变量

int rowNumber = outputImage.rows; //行数

int colNumber = outputImage.cols\*outputImage.channels(); //列数 x 通道数=每一行元素的个数

//双重循环，遍历所有的像素值

for(int i = 0;i < rowNumber;i++) //行循环

{

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

for(int j = 0;j < colNumber;j++) //列循环

{

// ---------【开始处理每个像素】-------------

data[j] = data[j]/div\*div + div/2;

// ----------【处理结束】---------------------

} //行处理结束

}

}

* + 1. 用迭代器操作像素

用迭代器操作像素的特点是安全。

void colorReduce(Mat& inputImage, Mat& outputImage, int div)

{

//参数准备

outputImage = inputImage.clone(); //拷贝实参到临时变量

//获取迭代器

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

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

//存取彩色图像像素

for(;it != itend;++it)

{

// ------------------------【开始处理每个像素】--------------------

(\*it)[0] = (\*it)[0]/div\*div + div/2;

(\*it)[1] = (\*it)[1]/div\*div + div/2;

(\*it)[2] = (\*it)[2]/div\*div + div/2;

// ------------------------【处理结束】----------------------------

}

}

* + 1. 动态地址计算

动态地址运算配合at方法：

Vec3b向量：彩色图像的Mat，返回一个有三个8位数组成的向量。

image.at<Vec3b>(i,j)[channel] = value;

void colorReduce(Mat& inputImage, Mat& outputImage, int div)

{

//参数准备

outputImage = inputImage.clone(); //拷贝实参到临时变量

int rowNumber = outputImage.rows; //行数

int colNumber = outputImage.cols; //列数

//存取彩色图像像素

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

{

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

{

// ------------------------【开始处理每个像素】--------------------

outputImage.at<Vec3b>(i,j)[0] = outputImage.at<Vec3b>(i,j)[0]/div\*div + div/2; //蓝色通道

outputImage.at<Vec3b>(i,j)[1] = outputImage.at<Vec3b>(i,j)[1]/div\*div + div/2; //绿色通道

outputImage.at<Vec3b>(i,j)[2] = outputImage.at<Vec3b>(i,j)[2]/div\*div + div/2; //红是通道

// -------------------------【处理结束】----------------------------

} // 行处理结束

}

}

1. ROI区域图像叠加和图像混合
   1. 感兴趣的区域: ROI

ROI, region of interest.

* ROI的好处

圈定区域并在这个区域做一些操作可以减少处理时间，增加精度。

* 定义ROI区域的方法

1. 矩形区域Rect

Mat imageROI;

ImageROI = image(Rect(500,250,logo.cols,logo.rows));

其中Rect的：

第一个参数和第二个参数指定矩形的左上角坐标

第二个参数和第三个参数指定矩形的长度

1. 行或列的范围Range

ImageROI=image(Range(250,250+logoImage.rows), Range(200,200+logoImage.cols));

起始索引到终止索引的一段连续序列。

* ROI 实例

利用感兴趣区域实现图像叠加：

bool ROI\_AddImage()

{

// 【1】读入图像

Mat srcImage1= imread("dota\_pa.jpg");

Mat logoImage= imread("dota\_logo.jpg");

if( !srcImage1.data ) { printf("读取srcImage1错误~！ \n"); return false; }

if( !logoImage.data ) { printf("读取logoImage错误~！ \n"); return false; }

// 【2】定义一个Mat类型并给其设定ROI区域

Mat imageROI= srcImage1(Rect(200,250,logoImage.cols,logoImage.rows));

// 【3】加载掩模（必须是灰度图）

Mat mask= imread("dota\_logo.jpg",0);

//【4】将掩膜拷贝到ROI

logoImage.copyTo(imageROI,mask);

// 【5】显示结果

namedWindow("<1>利用ROI实现图像叠加示例窗口");

imshow("<1>利用ROI实现图像叠加示例窗口",srcImage1);

return true;

}

* 1. 线性混合操作

线性混合操作是一种典型的二元（两个输入）的像素操作，公式：

(x)

(x) 分别表示两幅图像或是两段视频产生时间上的叠化效果

Cross-dissolve。

* 1. 计算数组加权和：addWeighted()函数

void addWeighted(InputArray src1, double alpha, InputArray src2,

double beta, double gamma, OutputArray dst, int dtype=-1)

|  |  |
| --- | --- |
| 参数列表 | 描述 |
| src1 | 表示要加权的第一个数组，常常填一个Mat |
| alpha | 表示第一个数组的权重 |
| src2 | 表示第二个数组，它需要和第一个数组有相同的尺寸和通道数 |
| beta | 第二个数组的权重值 |
| gamma | 一个加到权重综合上的标量值 |
| 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 |
| dtype | 输出阵列的可选深度，当两个输入数组具有相同的深度时，这个参数设置为-1，即等同于src1.depth() |

下面公式表示函数计算的两个数组的加权和，结果输出给第四个参数。

公式：dst = src1[I] \* alpha + src2[I]\*beta + gamma;

I 为多维数组的索引值。

多通道数组每个通道需要独立的进行处理。

深度CV\_32S是，函数不适用，回导致溢出或是结果不对。

1. addWeighted函数实现图像的线性混合：

bool LinearBlending()

{

//【0】定义一些局部变量

double alphaValue = 0.1;

double betaValue;

Mat srcImage2, srcImage3, dstImage;

// 【1】读取图像 ( 两幅图片需为同样的类型和尺寸 )

srcImage2 = imread("mogu.jpg");

srcImage3 = imread("rain.jpg");

if( !srcImage2.data ) { printf("读取srcImage2错误！ \n"); return false; }

if( !srcImage3.data ) { printf("读取srcImage3错误！ \n"); return false; }

// 【2】进行图像混合加权操作

betaValue = ( 1.0 - alphaValue );

addWeighted( srcImage2, alphaValue, srcImage3, betaValue, 0.0, dstImage);

// 【3】显示原图窗口

imshow( "<2>线性混合示例窗口【原图】", srcImage2 );

imshow( "<3>线性混合示例窗口【效果图】", dstImage );

return true;

}

* 1. 初级图像混合

利用addWeighted（）函数结合定义感兴趣区域ROI，实现自定义区域的线性混合

bool ROI\_LinearBlending()

{

//【1】读取图像

Mat srcImage4= imread("dota\_pa.jpg",1);

Mat logoImage= imread("dota\_logo.jpg");

if( !srcImage4.data ) { printf("读取srcImage4错误~！ \n"); return false; }

if( !logoImage.data ) { printf("读取logoImage错误~！ \n"); return false; }

//【2】定义一个Mat类型并给其设定ROI区域

Mat imageROI;

//方法一

imageROI= srcImage4(Rect(200,250,logoImage.cols,logoImage.rows));

//方法二

//imageROI= srcImage4(Range(250,250+logoImage.rows),Range(200,200+logoImage.cols));

//【3】将logo加到原图上

addWeighted(imageROI,0.5,logoImage,0.3,0.,imageROI);

//【4】显示结果

imshow("<4>区域线性图像混合示例窗口",srcImage4);

return true;

}