
# OpenCV 3.1.0 – 图像处理教程


# 讲师： 贾志刚

E-Mail: bfnh1998@hotmail.com

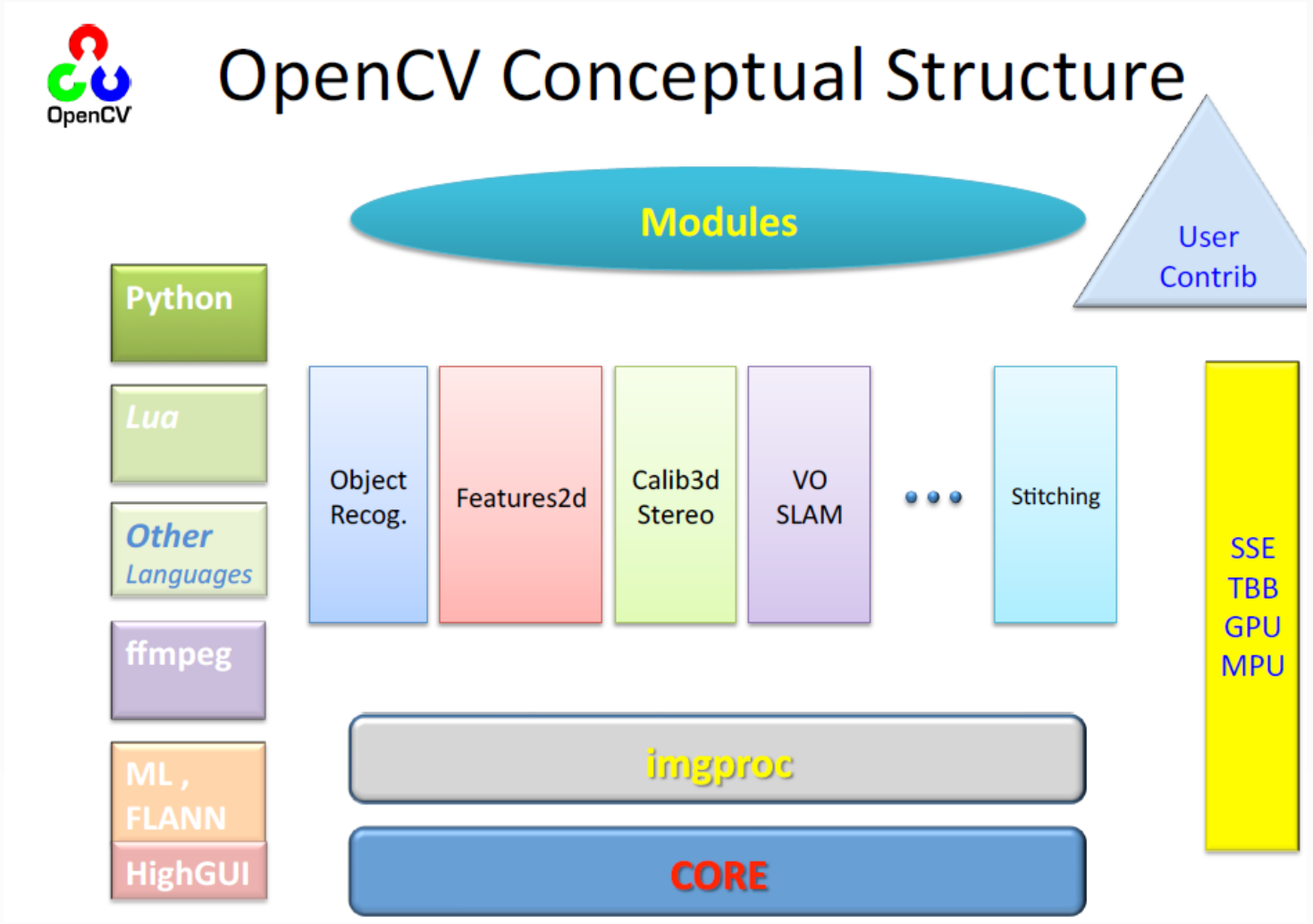
微博：流浪的鱼-GloomyFish

# OpenCV介绍与环境搭建

- OpenCV介绍
- 开发环境搭建
- 代码与演示

# OpenCV介绍

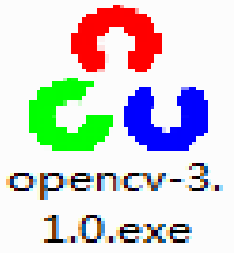
- OpenCV是计算机视觉开源库，主要算法涉及图像处理和机器学习相关方法。
- 是Intel公司贡献出来的，俄罗斯工程师贡献大部分C/C++带代码。
- 在多数图像处理相关的应用程序中被采用,BSD许可，可以免费应用在商业和研究领域
- 最新版本是OpenCV 3.1.0，当前SDK支持语言包括了Java、Python、IOS和Android版本。
- 官方主页： http://opencv.org/opencv-3-1.html
- 其它Matlab、Halcon


OpenCV
OpenCV Conceptual Structure
Modules
User Contrib
Python
Lua
Other Languages
ffmpeg
ML , FLANN
HighGUI
Object Recog.
Features2d
Calib3d Stereo
VO SLAM
Stitching
SSE
TBB
GPU
MPU
imgproc
CORE

# 核心模块

- HighGUI部分
- Image Process
- 2D Feature
- Camera Calibration and 3D reconstruction
- Video Analysis
- Object Detection
- Machine Learning
- GPU加速

# 开发环境搭建




- VS2015版本
- 下载OpenCV 3.1.0 版本
- 配置环境变量和在VS2015中引入头文件、库文件、连接库。

# 开发环境搭建

包括头文件：
D:\opencv3.1\opencv\build\include
D:\opencv3.1\opencv\build\include\opencv
D:\opencv3.1\opencv\build\include\opencv2

库文件
D:\opencv3.1\opencv\build\x64\vc14\lib

链接器
opencv_world310d.lib

# 测试代码

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

using namespace cv;
using namespace std;
int main(int argc, char** args) {
	Mat image = imread("girl.jpg", IMREAD_GRAYSCALE);
	if (image.empty()) {
		cout << "could not find the image resource..." << std::endl;
		return -1;
	}
	namedWindow("My Image", CV_WINDOW_AUTOSIZE);
	imshow("My Image", image);
	waitKey(0);

	return 0;
}
```

# 加载、修改、保存图像

- 加载图像（用cv::imread）
- 修改图像 (cv::cvtColor)
- 保存图像(cv::imwrite)
- 代码演示

# 加载图像（用cv::imread）

- **imread**功能是加载图像文件成为一个**Mat**对象，其中第一个参数表示图像文件名称
- 第二个参数，表示加载的图像是什么类型，支持常见的三个参数值
- IMREAD_UNCHANGED (<0) 表示加载原图，不做任何改变
- IMREAD_GRAYSCALE ( 0)表示把原图作为灰度图像加载进来
- IMREAD_COLOR (>0) 表示把原图作为RGB图像加载进来

**注意：**OpenCV支持JPG、PNG、TIFF等常见格式图像文件加载

# 显示图像 (cv::namedWindos 与cv::imshow)

- namedWindos功能是创建一个OpenCV窗口，它是由OpenCV自动创建与释放，你无需取销毁它。
- 常见用法namedWindow("Window Title", WINDOW_AUTOSIZE)
- WINDOW_AUTOSIZE会自动根据图像大小，显示窗口大小，不能人为改变窗口大小
- WINDOW_NORMAL,跟QT集成的时候会使用，允许修改窗口大小。
- imshow根据窗口名称显示图像到指定的窗口上去，第一个参数是窗口名称，第二参数是Mat对象

# 修改图像 (cv::cvtColor)

- cvtColor的功能是把图像从一个彩色空间转换到另外一个色彩空间，有三个参数，第一个参数表示源图像、第二参数表示色彩空间转换之后的图像、第三个参数表示源和目标色彩空间如：COLOR_BGR2HLS 、COLOR_BGR2GRAY 等
- cvtColor( image, gray_image, COLOR_BGR2GRAY );

# 保存图像(cv::imwrite)

- 保存图像文件到指定目录路径
- 只有8位、16位的PNG、JPG、Tiff文件格式而且是单通道或者三通道的BGR的图像才可以通过这种方式保存
- 保存PNG格式的时候可以保存透明通道的图片
- 可以指定压缩参数

# 矩阵的掩膜操作

- 获取图像像素指针
- 掩膜操作解释
- 代码演示

# 获取图像像素指针

- CV_Assert(myImage.depth() == CV_8U);
- Mat.ptr<uchar>(int i=0) 获取像素矩阵的指针，索引i表示第几行，从0开始计行数。
- 获得当前行指针const uchar* current= myImage.ptr<uchar>(row );
- 获取当前像素点P(row, col)的像素值 p(row, col) =current[col]

# 像素范围处理saturate_cast<uchar>

- saturate_cast<uchar>（-100），返回 0。
- saturate_cast<uchar>（288），返回255
- saturate_cast<uchar>（100），返回100
- 这个函数的功能是确保RGB值得范围在0~255之间

## 掩膜操作实现图像对比度调整

-红色是中心像素，从上到下，从左到右对每个像素做同样的处理操作，得到最终结果就是对比度提高之后的输出图像Mat对象

矩阵的掩膜操作十分简单，根据掩膜来重新计算每个像素的像素值，掩膜(mask 也被称为Kernel)

通过掩膜操作实现图像对比度提高。

$$I(i,j) = 5 * I(i,j) - [I(i-1,j) + I(i+1,j) + I(i,j-1) + I(i,j+1)]$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

# 代码实现

```
int main(int argc, char** argv)
{
    Mat myImage = imread("D:/test.jpg");
    CV_Assert(myImage.depth() == CV_8U);
    namedWindow("mask_demo", CV_WINDOW_AUTOSIZE);
    imshow("mask_demo", myImage);

    // clone current image
    Mat resultImage;
    myImage.copyTo(resultImage);
    int nchannels = myImage.channels();
    int height = myImage.rows;
    int cols = myImage.cols;
    int width = myImage.cols * nchannels;
    for (int row = 1; row < height - 1; row++) {
        const uchar* previous = myImage.ptr<uchar>(row - 1);
        const uchar* current = myImage.ptr<uchar>(row);
        const uchar* next = myImage.ptr<uchar>(row + 1);
        uchar* output = resultImage.ptr<uchar>(row);
        for (int col = nchannels; col < nchannels * (myImage.cols - 1); col++) {
            *output = saturate_cast<uchar>(5 * current[col] - previous[col] - next[col] - current[col - nchannels] - current[col + nchannels]);
            output++;
        }
    }

    namedWindow("mask_result", CV_WINDOW_AUTOSIZE);
    imshow("mask_result", resultImage);

    // 关闭
    waitKey(0);
    return 0;
}
```

# 函数调用filter2D功能

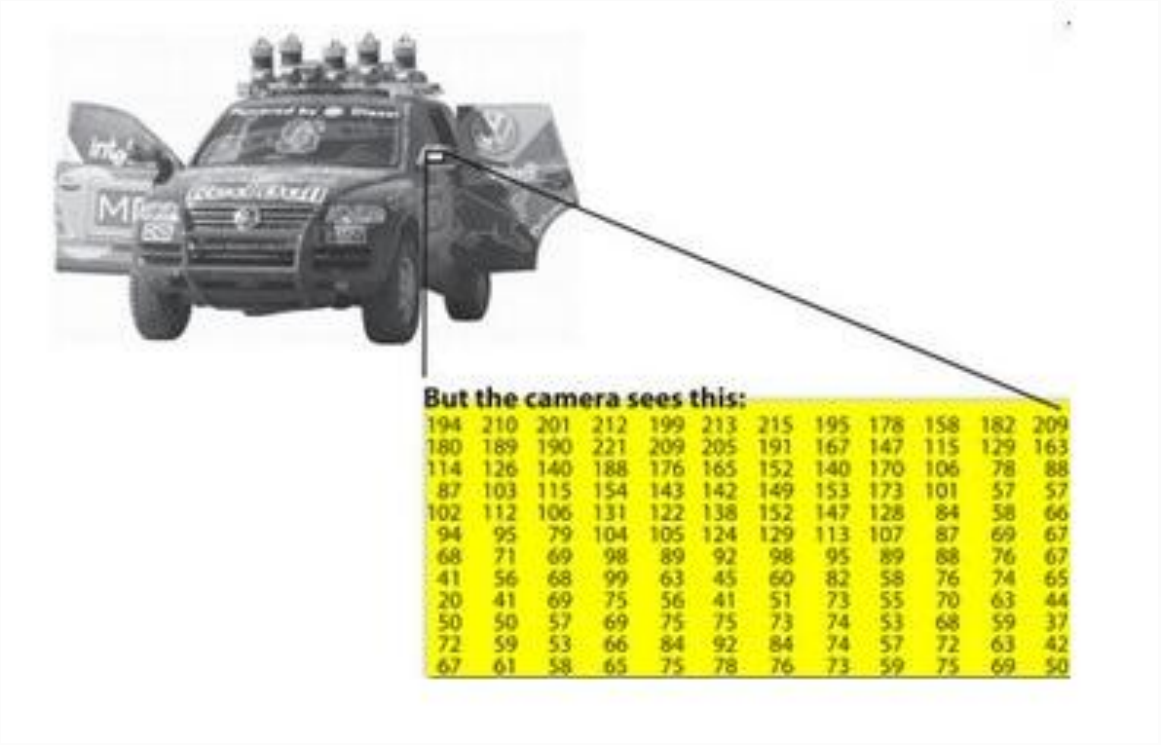
1. 定义掩膜：Mat kernel = (Mat_<char>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
2. filter2D( src, dst, src.depth(), kernel );其中src与dst是Mat类型变量、src.depth表示位图深度，有32、

```
Mat kernel = (Mat_<char>(3,3) <<  0, -1,  0,
                                 -1,  5, -1,
                                  0, -1,  0);
t = (double)getTickCount();
filter2D( src, dst1, src.depth(), kernel );
t = ((double)getTickCount() - t)/getTickFrequency();
cout << "Built-in filter2D time passed in seconds:      " << t << endl;

imshow( "Output", dst1 );
```

# Mat对象

- Mat对象与IplImage对象
- Mat对象使用
- Mat定义数组

# Mat对象


But the camera sees this:
194 210 201 212 199 213 215 195 178 158 182 209
180 189 190 221 209 205 191 167 147 115 129 163
114 126 140 188 176 165 152 140 170 106 78 88
87 103 115 154 143 142 149 153 173 101 57 57
102 112 106 131 122 138 152 147 128 84 58 66
94 95 79 104 105 124 129 113 107 87 69 67
68 71 69 98 89 92 98 95 89 88 76 67
41 56 68 99 63 45 60 82 58 76 74 65
20 41 69 75 56 41 51 73 55 70 63 44
50 50 57 69 75 75 73 74 53 68 59 37
72 59 53 66 84 92 84 74 57 72 63 42
67 61 58 65 75 78 76 73 59 75 69 50

# Mat对象与IplImage对象

- **Mat**对象OpenCV2.0之后引进的图像数据结构、自动分配内存、不存在内存泄漏的问题，是面向对象的数据结构。分了两个部分，头部与数据部分
- **IplImage**是从2001年OpenCV发布之后就一直存在，是C语言风格的数据结构，需要开发者自己分配与管理内存，对大的程序使用它容易导致内存泄漏问题

# Mat对象构造函数与常用方法

| |
|---|
| Mat () |
| Mat (int rows, int cols, int type) |
| Mat (Size size, int type) |
| Mat (int rows, int cols, int type, const Scalar &s) |
| Mat (Size size, int type, const Scalar &s) |
| Mat (int ndims, const int *sizes, int type) |
| Mat (int ndims, const int *sizes, int type, const Scalar &s) |

**常用方法：**

void copyTo(Mat mat)
void convertTo(Mat dst, int type)
Mat clone()
int channels()
int depth()
bool empty();
uchar* ptr(i=0)

# Mat对象使用

- 部分复制：一般情况下只会复制Mat对象的头和指针部分，不会复制数据部分

```
Mat A= imread(imgFilePath);
Mat B(A)  // 只复制
```

- 完全复制：如果想把Mat对象的头部和数据部分一起复制，可以通过如下两个API实现

```
Mat F = A.clone(); 或 Mat G; A.copyTo(G);
```

# Mat对象使用-四个要点

- 输出图像的内存是自动分配的
- 使用OpenCV的C++接口，不需要考虑内存分配问题
- 赋值操作和拷贝构造函数只会复制头部分
- 使用clone与copyTo两个函数实现数据完全复制

# Mat对象创建

- cv::Mat::Mat构造函数

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

Mat M(2,2,CV_8UC3, Scalar(0,0,25

其中前两个参数分别表示行(row)跟列(column)、第三个CV_8UC3中的8表示每个通道占8位、U表示无符号、C表示Char类型、3表示通道数目是3，第四个参数是向量表示初始化每个像素值是多少，向量长度对应通道数目一致

- 创建多维数组cv::Mat::create

```
int sz[3] = {2,2,2};
Mat  L(3,sz, CV_8UC1, Scalar::all(0));
```

- cv::Mat::create实现

```
Mat M;
M.create(4, 3, CV_8UC2);
M = Scalar(127,127);
cout << "M = " << endl << " " << M << endl <<
    endl;
uchar* firstRow = M.ptr<uchar>(0);
printf("%d", *firstRow);
```

```
M =
 [127, 127, 127, 127, 127, 127;
 127, 127, 127, 127, 127, 127;
 127, 127, 127, 127, 127, 127;
 127, 127, 127, 127, 127, 127]

127
```

# 定义小数组

```
Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0,
    -1, 0);
C = " << endl << " " << C << endl <<
```

```
C =
 [0, -1, 0;
  -1, 5, -1;
  0, -1, 0]
```

演示代码：

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

using namespace cv;
using namespace std;
int main(int argc, char** args) {
Mat image = imread("D:/test.jpg", IMREAD_GRAYSCALE);
if (image.empty()) {
cout << "could not find the image resource..." << std::endl;
return -1;
}
namedWindow("My Image", CV_WINDOW_AUTOSIZE);
imshow("My Image", image);

Mat M;
M.create(4, 3, CV_8UC2);
M = Scalar(127,127);
cout << "M = " << endl << " " << M << endl << endl;
uchar* firstRow = M.ptr<uchar>(0);
printf("%d\n", *firstRow);

Mat C = (Mat_<double>(3, 3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
cout << "C = " << endl << " " << C << endl << endl;

waitKey(0);
return 0;
}
```

# 图像操作

- 读写图像
- 读写像素
- 修改像素值

# 读写图像

- **imread** 可以指定加载为灰度或者RGB图像
- **Imwrite** 保存图像文件，类型由扩展名决定

# 读写像素

- 读一个GRAY像素点的像素值（CV_8UC1）

```
Scalar intensity = img.at<uchar>(y, x);
或者 Scalar intensity = img.at<uchar>(Point(x, y));
```

- 读一个RGB像素点的像素值

```
Vec3f intensity = img.at<Vec3f>(y, x);
float blue = intensity.val[0];
float green = intensity.val[1];
float red = intensity.val[2];
```

# 修改像素值

- 灰度图像

```
img.at<uchar>(y, x) = 128;
```

- RGB三通道图像

```
img.at<Vec3b>(y,x)[0]=128; // blue
img.at<Vec3b>(y,x)[1]=128; // green
img.at<Vec3b>(y,x)[2]=128; // red
```

- 空白图像赋值

```
img = Scalar(0);
```

- ROI选择

```
Rect r(10, 10, 100, 100);
Mat smallImg = img(r);
```

# Vec3b与Vec3F

- Vec3b对应三通道的顺序是blue、green、red的uchar类型数据。
- Vec3f对应三通道的float类型数据
- 把CV_8UC1转换到CV32F1实现如下：

```
src.convertTo(dst, CV_32F);
```

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

using namespace cv;
using namespace std;
int main(int argc, char** args) {
Mat image = imread("D:/test.jpg", IMREAD_COLOR);
if (image.empty()) {
cout << "could not find the image resource..." << std::endl;
return -1;
}

int height = image.rows;
int width = image.cols;
int channels = image.channels();
printf("height=%d width=%d channels=%d", height, width, channels);
for (int row = 0; row < height; row++) {
      for (int col = 0; col < width; col++) {
            if (channels == 3) {
                  image.at<Vec3b>(row, col)[0] = 0; // blue
                  image.at<Vec3b>(row, col)[1] = 0; // green
            }
       }
}

namedWindow("My Image", CV_WINDOW_AUTOSIZE);
imshow("My Image", image);
waitKey(0);
return 0;
}
```

# 图像混合

- 理论-线性混合操作
- 相关API (addWeighted)
- 代码演示

# 理论-线性混合操作

$$g(x) = (1-\alpha) f_0(x) + \alpha f_1(x)$$

其中 $\alpha$ 的取值范围为0~1之间

# 相关API (addWeighted)

```
void cv::addWeighted ( InputArray  src1,
                       double      alpha,
                       InputArray  src2,
                       double      beta,
                       double      gamma,
                       OutputArray dst,
                       int         dtype = -1
                     )
```

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) * \texttt{alpha} + \texttt{src2}(I) * \texttt{beta} + \texttt{gamma})$$

参数1：输入图像Mat – src1
参数2：输入图像src1的alpha值
参数3：输入图像Mat – src2
参数4：输入图像src2的alpha值
参数5：gamma值
参数6：输出混合图像

**注意点：**两张图像的大小和类型必须一致才可以

```
Mat src1, src2, dest;
src1 = imread("D:/vcprojects/images/LinuxLogo.jpg");
src2 = imread("D:/vcprojects/images/win7logo.jpg");
if (!src1.data) {
		printf("could not load LinuxLogo image...\n");
		return -1;
}
if (!src2.data) {
		printf("could not load win7logo image...\n");
		return -1;
}
if (src1.rows == src2.rows && src1.cols == src2.cols) {
		double alpha = 0.5;
		namedWindow("line-blend", CV_WINDOW_AUTOSIZE);
		addWeighted(src1, (1 - alpha), src2, alpha, 0.0, dest);

		imshow("line-blend", dest);
		waitKey(0);
		return 0;
}
else {
		printf("image size is not same...\n");
		return -1;
}
```

# 调整图像亮度与对比度

- 理论
- 代码演示

# 理论

- **图像变换**可以看作如下：

- 像素变换 – 点操作
- 邻域操作 – 区域

调整图像亮度和对比度属于像素变换-点操作

$$g(i,j)=\alpha f(i,j)+\beta \text{其中} \alpha>0,\beta \text{是增益变量}$$

# 重要的API

- Mat new_image = Mat::zeros( image.size(), image.type() ); 创建一张跟原图像大小和类型一致的空白图像、像素值初始化为0
- saturate_cast<uchar>(value)确保值大小范围为0~255之间
- Mat.at<Vec3b>(y,x)[index]=value 给每个像素点每个通道赋值

## 实现代码

```
Mat input, output;
input = imread("D:/vcprojects/images/test1.png");
namedWindow("input-image", CV_WINDOW_AUTOSIZE);
imshow("input-image", input);

if (!input.data) {
    printf("could not load image...\n");
    return -1;
}

int height = input.rows;
int width = input.cols;
double alpha = 1.2;
double beta = 50;
output = Mat::zeros(input.size(), input.type());
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        output.at<Vec3b>(y, x)[0] = saturate_cast<uchar>(alpha * input.at<Vec3b>(y, x)[0] + beta); // blue
        output.at<Vec3b>(y, x)[1] = saturate_cast<uchar>(alpha * input.at<Vec3b>(y, x)[1] + beta);  // green
        output.at<Vec3b>(y, x)[2] = saturate_cast<uchar>(alpha * input.at<Vec3b>(y, x)[2] + beta);  // red
    }
}

namedWindow("line-transform", CV_WINDOW_AUTOSIZE);
imshow("line-transform", output);
waitKey(0);

return 0;
```

# 绘制形状与文字

- 使用cv::Point与cv::Scalar
- 绘制线、矩形、园、椭圆等基本几何形状
- 随机生成与绘制文本
- 代码演示

# 使用cv::Point与cv::Scalar

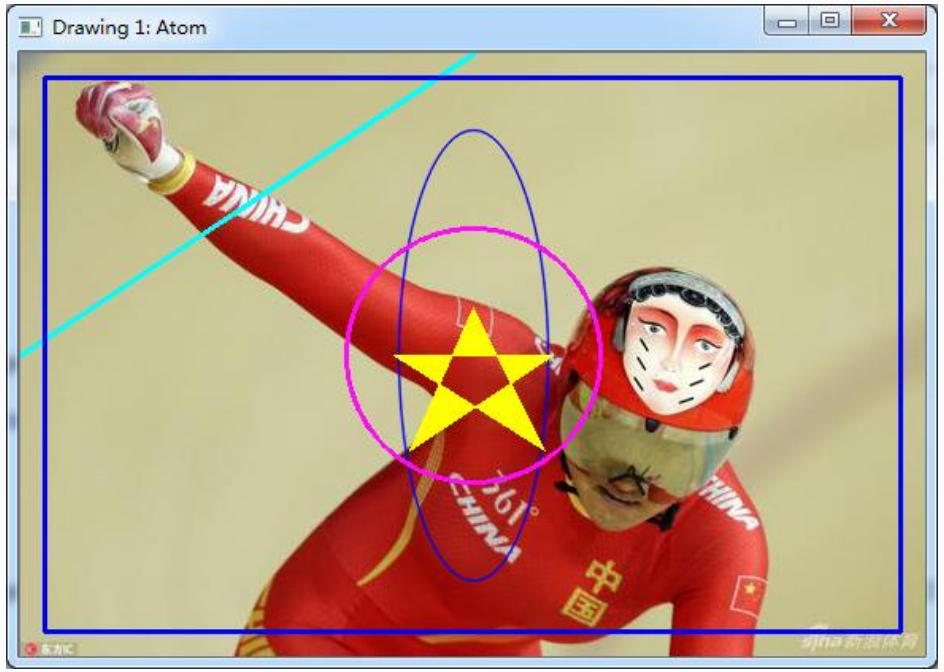
- Point表示2D平面上一个点x,y
  Point p;
  p.x = 10;
  p.y = 8;
  or
  p = Pont(10,8);
- Scalar表示四个元素的向量
  Scalar(a, b, c);// a = blue, b = green, c = red表示RGB三个通道

# 绘制线、矩形、园、椭圆等基本几何形状

- 画线 cv::line （LINE_4\LINE_8\LINE_AA）
- 画椭圆cv::ellipse
- 画矩形cv::rectangle
- 画圆cv::circle
- 画填充cv::fillPoly

# 填充矩形

- 可以通过多边形填充来填充矩形



# 随机数生成cv::RNG

- 生成高斯随机数gaussian (double sigma)
- 生成正态分布随机数uniform (int a, int b)

# 绘制添加文字

- putText函数 中设置*fontFace(cv::HersheyFonts)*,

 - fontFace, CV_FONT_HERSHEY_PLAIN

- fontScale , 1.0, 2.0~ 8.0

# 演示代码

```
int drawRandomLines(Mat image) {
    RNG rng(0xffffff);
    Point pt1, pt2;
    for (int i = 0; i < 100000; i++) {
        pt1.x = rng.uniform(0, image.cols);
        pt2.x = rng.uniform(0, image.cols);
        pt1.y = rng.uniform(0, image.rows);
        pt2.y = rng.uniform(0, image.rows);
        int r = rng.uniform(0, 255);
        int g = rng.uniform(0, 255);
        int b = rng.uniform(0, 255);
        line(image, pt1, pt2, Scalar(b, g, r), 1, LINE_8);
        putText(image, "Open CV Core Tutorial", Point(image.cols / 2-200, image.rows / 2),
            CV_FONT_HERSHEY_COMPLEX, 1.0, Scalar(0, 255, 0), 3, LINE_8);

        imshow(WINTITLE, image);
        if (waitKey(10) >= 0)
        {
            return -1;
        }
    }
    return 0;
}
```

# 模糊图像一

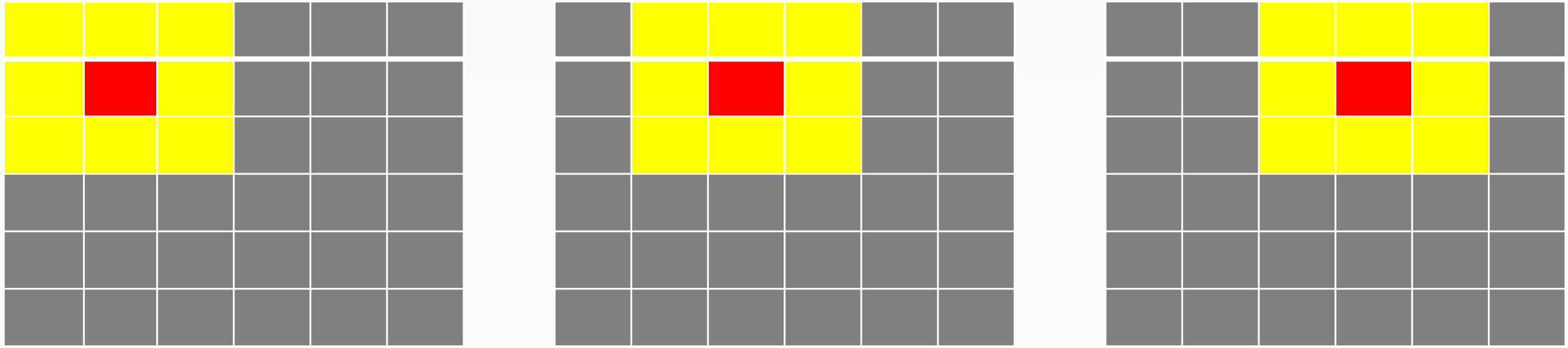
- 模糊原理
- 代码演示

# 模糊原理

- Smooth/Blur 是图像处理中最简单和常用的操作之一
- 使用该操作的原因之一就为了给图像预处理时候减低噪声
- 使用Smooth/Blur操作其背后是数学的卷积计算

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l)$$

- 通常这些卷积算子计算都是线性操作，所以又叫线性滤波



假设有6x6的图像像素点矩阵。

卷积过程：6x6上面是个3x3的窗口，从左向右，从上向下移动，黄色的每个像个像素点值之和取平均值赋给中心红色像素作为它卷积处理之后新的像素值。每次移动一个像素格。

# 模糊原理

- 归一化盒子滤波（均值滤波）

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

- 高斯滤波

$$G_0(x, y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2} + \frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

# 相关API

- 均值模糊
  - blur(Mat src, Mat dst, Size(xradius, yradius), Point(-1,-1));

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

- 高斯模糊
  - GaussianBlur(Mat src, Mat dst, Size(11, 11), sigmax, sigmay);

其中Size（x, y），x, y 必须是正数而且是奇数

# 演示代码

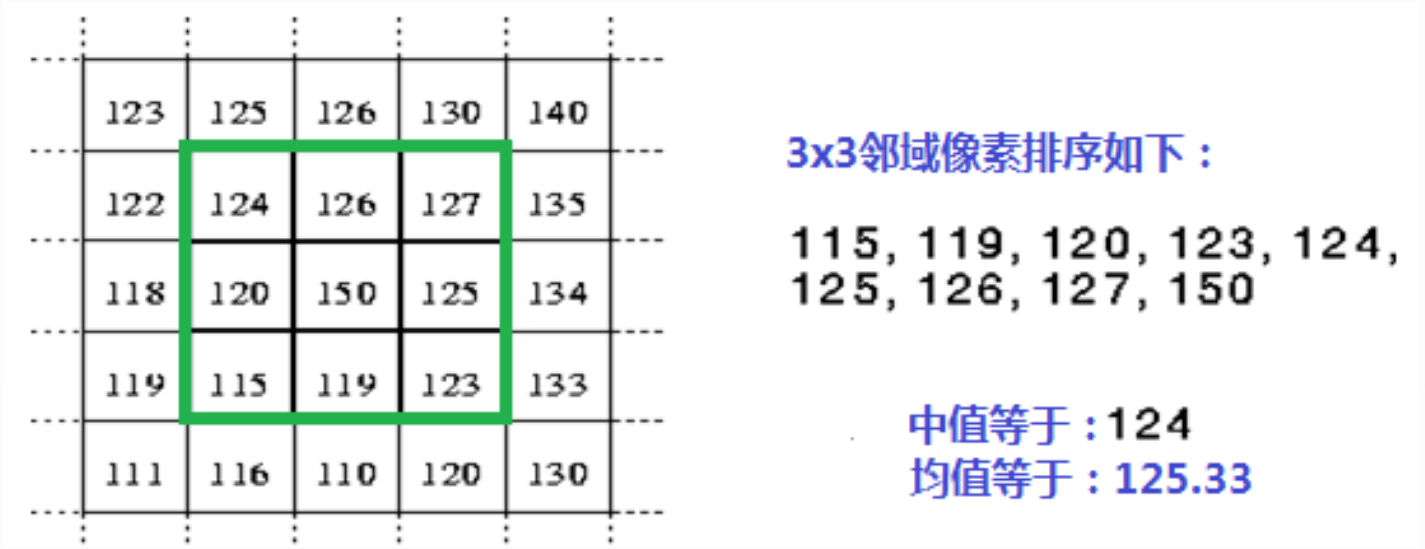
```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;

int main(int argc, char** argv) {
    Mat src, dest;
    src = imread("D:/vcprojects/images/test.png");
    if (!src.data) {
        printf("could not load LinuxLogo image...\n");
        return -1;
    }
    char source_title[] = "sourceImage";
    char dest_title[] = "resultImage";

    namedWindow(source_title, CV_WINDOW_AUTOSIZE);
    namedWindow(dest_title, CV_WINDOW_AUTOSIZE);
    // blur(src, dest, Size(15, 15), Point(-1, -1));

    GaussianBlur(src, dest, Size(11, 11), 5, 5);

    imshow(source_title, src);
    imshow(dest_title, dest);
    waitKey(0);
    return 0;
}
```

# 图像模糊二

- 中值滤波
- 双边滤波
- 代码演示

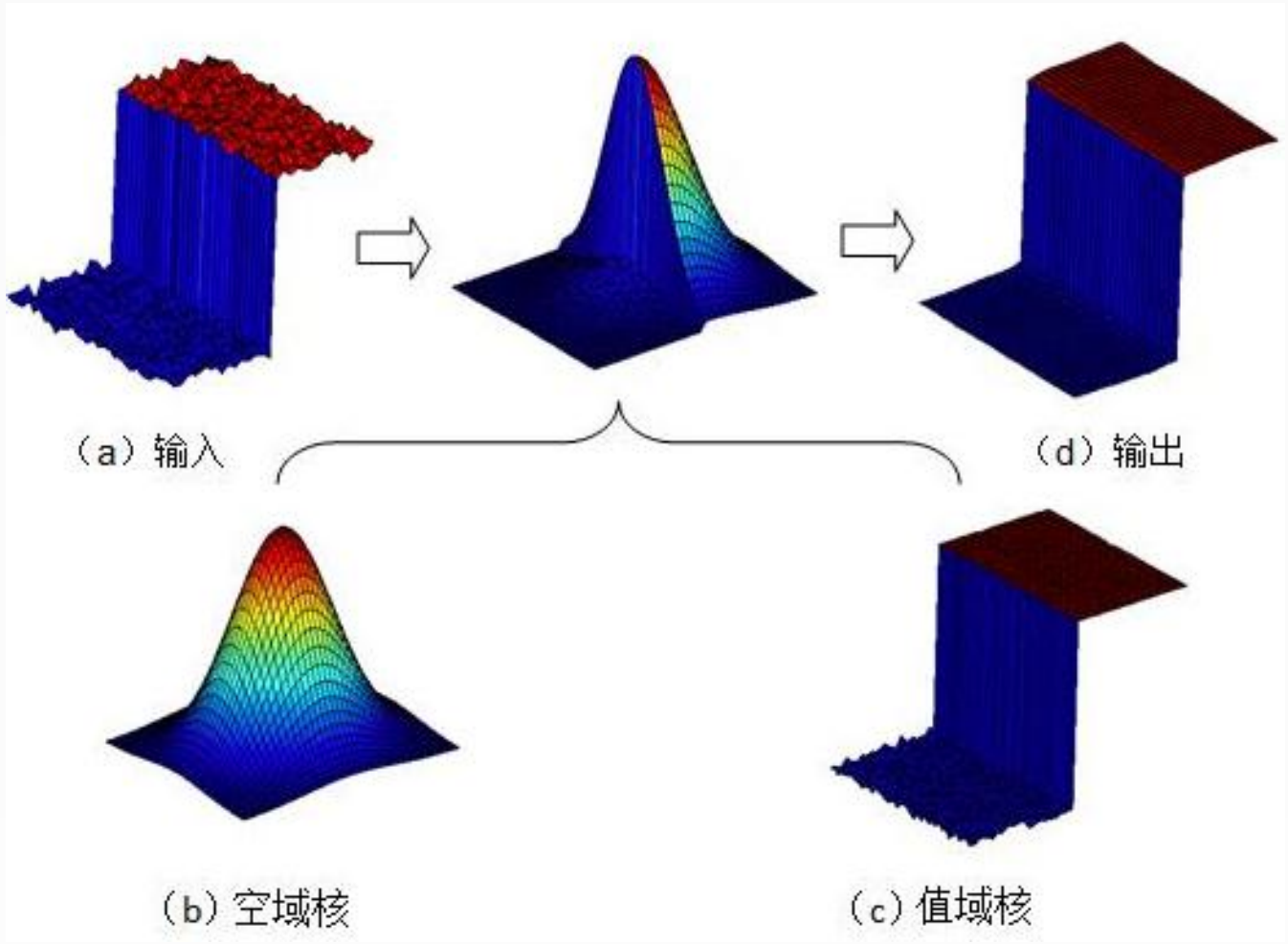
# 中值滤波

- 统计排序滤波器
- 中值对椒盐噪声有很好的抑制作用



# 双边滤波

- 均值模糊无法克服边缘像素信息丢失缺陷。原因是均值滤波是基于平均权重
- 高斯模糊部分克服了该缺陷，但是无法完全避免，因为没有考虑像素值的不同
- 高斯双边模糊 – 是边缘保留的滤波方法，避免了边缘信息丢失，保留了图像轮廓不变


（a）输入
（d）输出
（b）空域核
（c）值域核

# 相关API

- 中值模糊medianBlur（Mat src, Mat dest, ksize）
- 双边模糊bilateralFilter(src, dest, d=15, 150, 3);

- 15 –计算的半径，半径之内的像数都会被纳入计算，如果提供-1 则根据sigma space参数取值
- 150 – sigma color 决定多少差值之内的像素会被计算
- 3 – sigma space 如果d的值大于0则声明无效，否则根据它来计算d值

中值模糊的ksize大小必须是大于1而且必须是奇数。

# 演示代码

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

int main(int argc, char** argv) {
    Mat src, dest;
    src = imread("D:/vcprojects/images/cvtest.png");
    if (!src.data) {
        printf("could not load LinuxLogo image...\n");
        return -1;
    }

    char INPUT_WIN[] = "Source Image";
    char OUTPUT_WIN[] = "Filted out Image";
    namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_WIN, CV_WINDOW_AUTOSIZE);

    // filter image out
    // medianBlur(src, dest, 3);
    bilateralFilter(src, dest, 15, 150, 10);

    imshow(INPUT_WIN, src);
    imshow(OUTPUT_WIN, dest);
    waitKey(0);
    return 0;

}
```

# 膨胀与腐蚀

- 腐蚀
- 膨胀
- 代码演示

## 形态学操作(morphology operators)-膨胀

- 图像形态学操作 – 基于形状的一系列图像处理操作的合集，主要是基于集合论基础上的形态学数学
- 形态学有四个基本操作：腐蚀、膨胀、开、闭
- 膨胀与腐蚀是图像处理中最常用的形态学操作手段

# 形态学操作-膨胀

- 跟卷积操作类似，假设有图像A和结构元素B，结构元素B在A上面移动，其中B定义其中心为锚点，计算B覆盖下A的最大像素值用来替换锚点的像素，其中B作为结构体可以是任意形状

# 形态学操作-腐蚀

- 腐蚀跟膨胀操作的过程类似，唯一不同的是以最小值替换锚点重叠下图像的像素值

# 相关API

- getStructuringElement(int shape, Size ksize, Point anchor)
  - 形状 (MORPH_RECT \MORPH_CROSS \MORPH_ELLIPSE)
  - 大小
  - 锚点 默认是Point(-1, -1)意思就是中心像素
- dilate(src, dst, kernel)

$$\texttt{dst}(x,y) = \max_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

- erode(src, dst, kernel)

$$\texttt{dst}(x,y) = \min_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

# 动态调整结构元素大小

- TrackBar – createTrackbar(const String & trackbarname, const String winName,  int* value, int count, Trackbarcallback func, void* userdata=0)

其中最中要的是 callback 函数功能。如果设置为NULL就是说只有值update，但是不会调用callback的函数。

# 代码演示

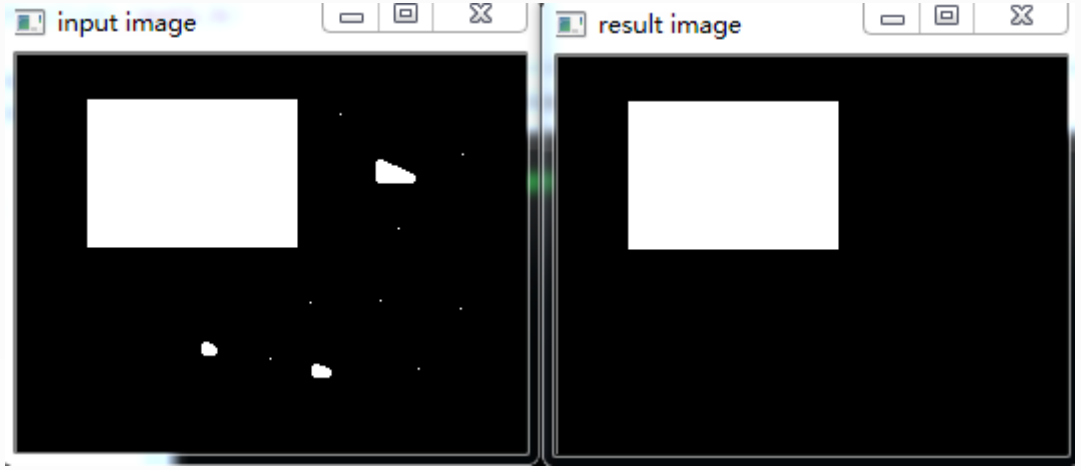
```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;

int main(int argc, char** argv) {
    Mat src, dest;
    src = imread("D:/vcprojects/images/cat.jpg");
    if (!src.data) {
        printf("could not load LinuxLogo image...\n");
        return -1;
    }

    char INPUT_WIN[] = "input image";
    char OUTPUT_WIN[] = "output image";
    namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_WIN, CV_WINDOW_AUTOSIZE);

    // dilate
    Mat kernel = getStructuringElement(MORPH_RECT, Size(5, 5), Point(-1, -1));
    // dilate(src, dest, kernel);

    // erosion
    erode(src, dest, kernel);

    imshow(INPUT_WIN, src);
    imshow(OUTPUT_WIN, dest);
    waitKey(0);
    return 0;
}
```

# 形态学操作

- 开操作- open
- 闭操作- close
- 形态学梯度- Morphological Gradient
- 顶帽 – top hat
- 黑帽 – black hat

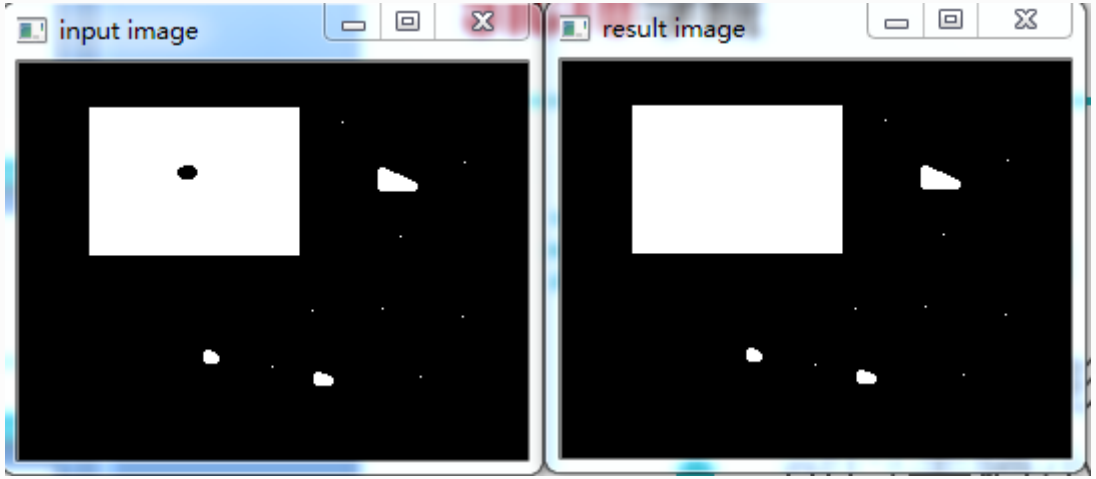
# 开操作- open

- 先腐蚀后膨胀 $dst = open(src, element) = dilate(erode(src, element))$
- 可以去掉小的对象，假设对象是前景色，背景是黑色



# 闭操作-close

- 先膨胀后腐蚀（bin2） $dst = close(src, element) = erode(dilate(src, element))$
- 可以填充小的洞（fill hole），假设对象是前景色，背景是黑色



# 形态学梯度- Morphological Gradient

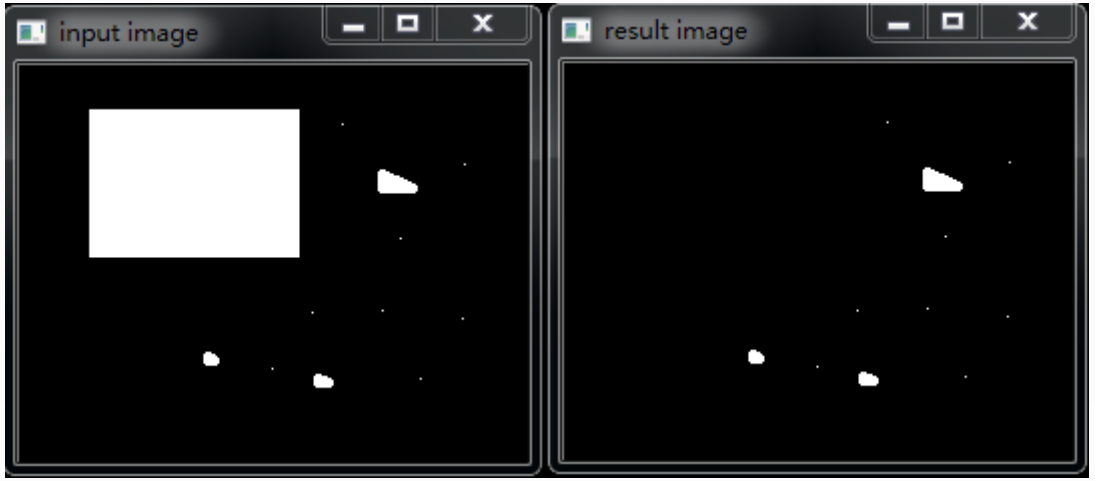
- 膨胀减去腐蚀

$$dst = morph_{grad}(src, element) = dilate(src, element) - erode(src, element)$$

- 又称为基本梯度（其它还包括-内部梯度、方向梯度）

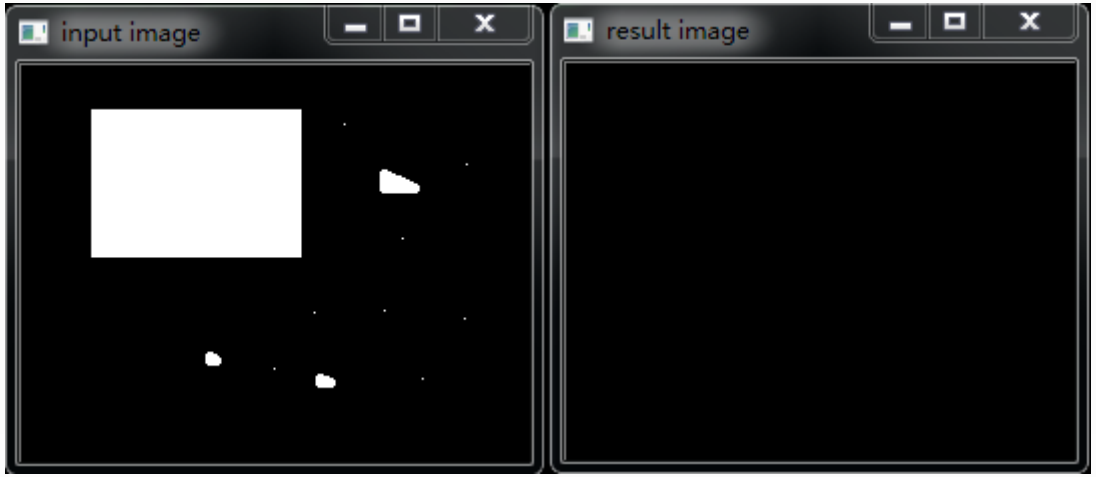
# 顶帽 – top hat

- 顶帽 是原图像与开操作之间的差值图像



# 黑帽

- 黑帽是闭操作图像与源图像的差值图像



# 相关API

- morphologyEx(src, dest, CV_MOP_BLACKHAT, kernel);
  - Mat src – 输入图像
  - Mat dest – 输出结果
  - int OPT – CV_MOP_OPEN/ CV_MOP_CLOSE/ CV_MOP_GRADIENT / CV_MOP_TOPHAT/ CV_MOP_BLACKHAT 形态学操作类型
  - Mat kernel 结构元素
  - int Iteration 迭代次数，默认是1

# 演示代码

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <math.h>

using namespace cv;
int main(int argc, char** argv) {
    Mat src, dest;
    src = imread("D:/vcprojects/images/bintest.png");
    if (!src.data) {
        printf("could not load image...\n");
    }

    char INPUT_WIN[] = "input image";
    char OUTPUT_WIN[] = "result image";
    namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_WIN, CV_WINDOW_AUTOSIZE);

    Mat kernel = getStructuringElement(MORPH_RECT, Size(11, 11), Point(-1, -1));

    morphologyEx(src, dest, CV_MOP_BLACKHAT, kernel);

    imshow(INPUT_WIN, src);
    imshow(OUTPUT_WIN, dest);
    waitKey(0);
    return 0;
}
```

# 形态学操作应用-提取水平与垂直线

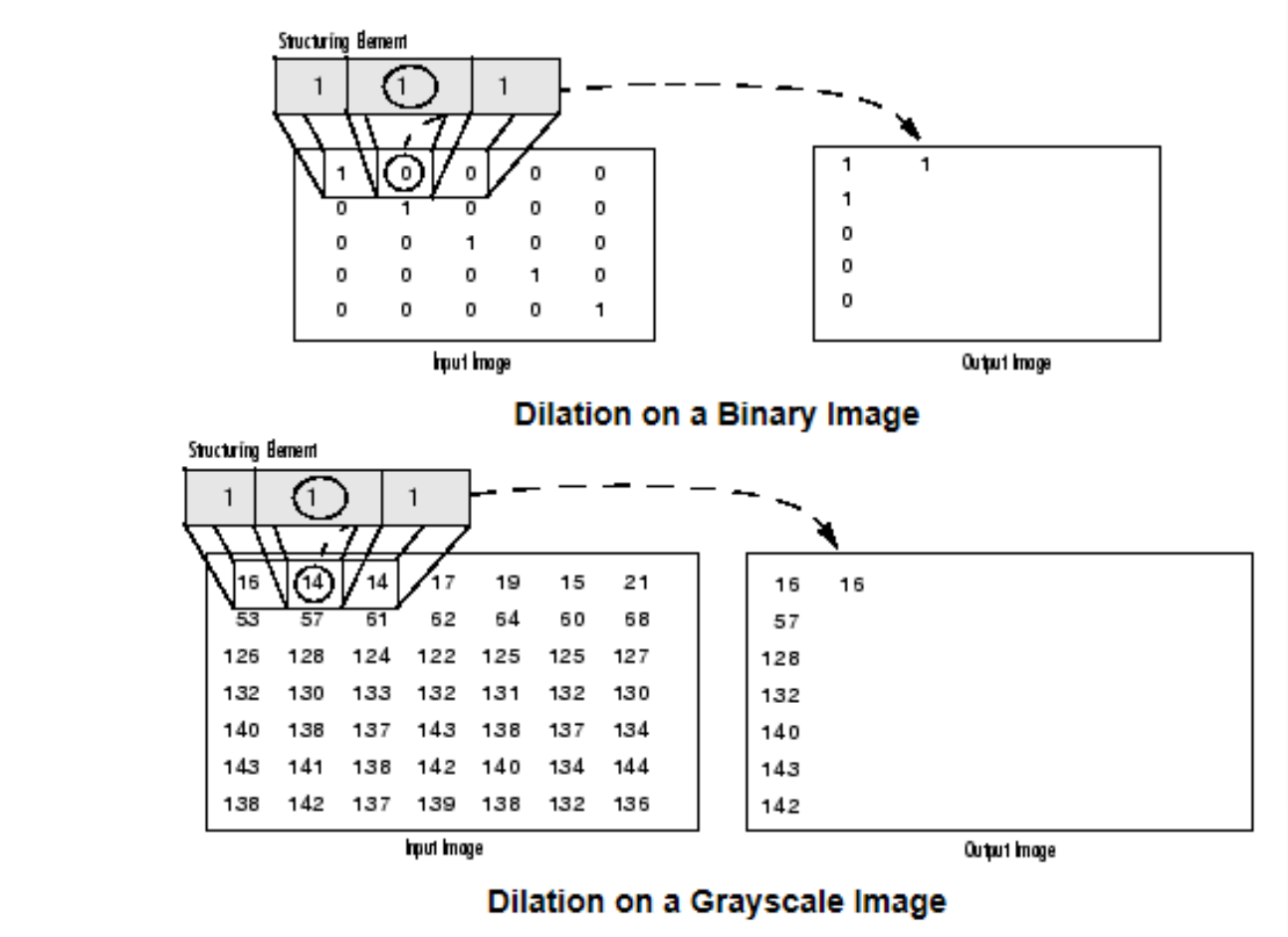
- 原理方法
- 实现步骤
- 代码演示

# 原理方法

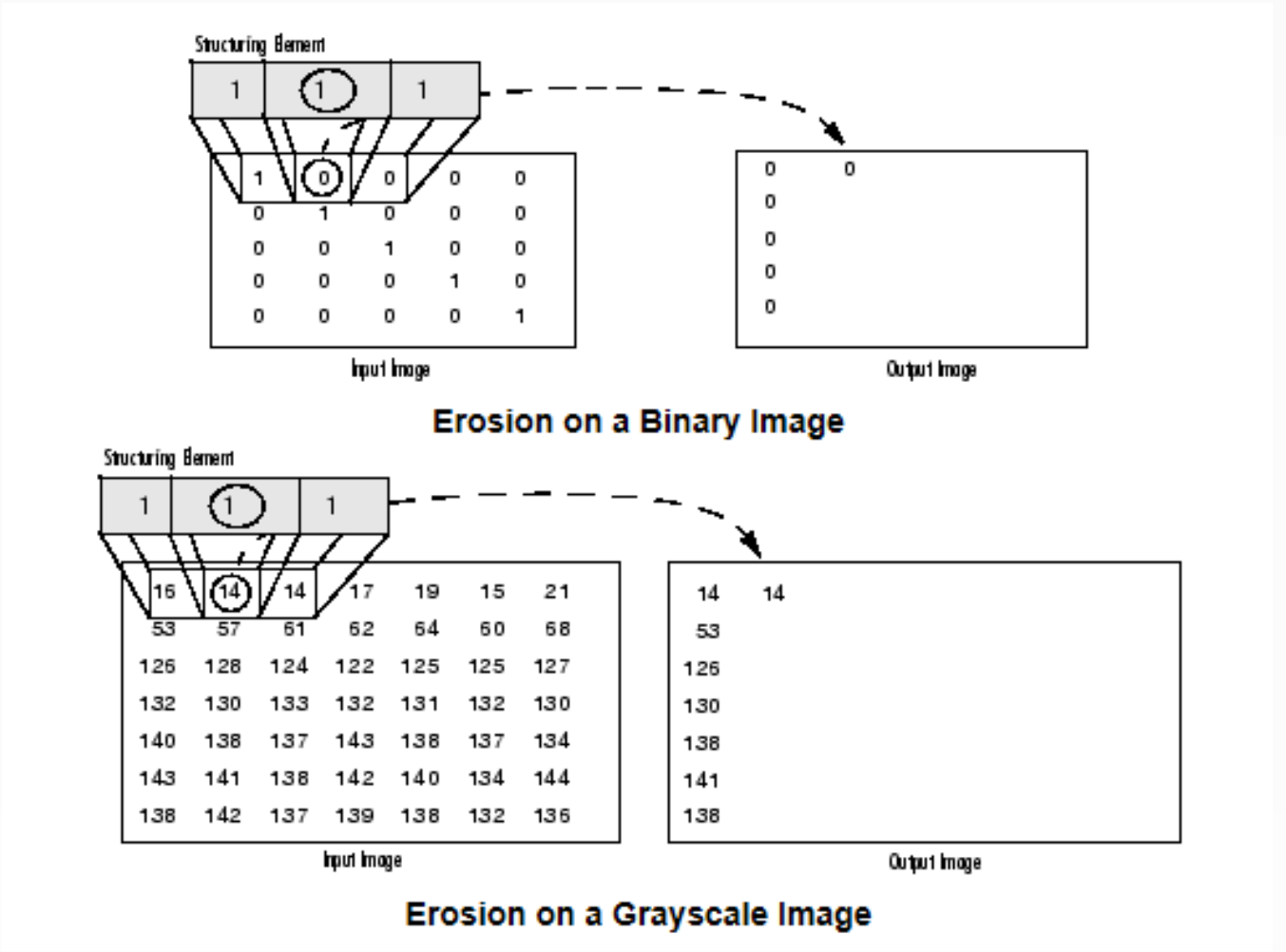
图像形态学操作时候，可以通过自定义的结构元素实现结构元素对输入图像一些对象敏感、另外一些对象不敏感，这样就会让敏感的对象改变而不敏感的对象保留输出。通过使用两个最基本的形态学操作 – **膨胀**与**腐蚀**，使用不同的结构元素实现对输入图像的操作、得到想要的结果。

- 膨胀，输出的像素值是结构元素覆盖下输入图像的最大像素值
- 腐蚀，输出的像素值是结构元素覆盖下输入图像的最小像素值

二值图像与灰度图像上的膨胀操作

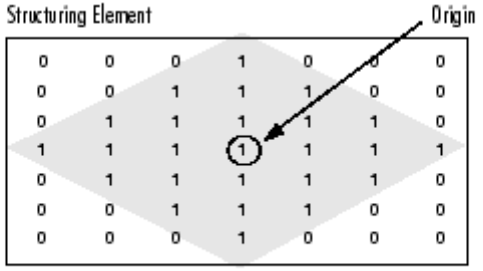


二值图像与灰度图像上的腐蚀操作



# 结构元素

- 上述膨胀与腐蚀过程可以使用任意的结构元素
- 常见的形状：矩形、园、直线、磁盘形状、砖石形状等各种自定义形状。

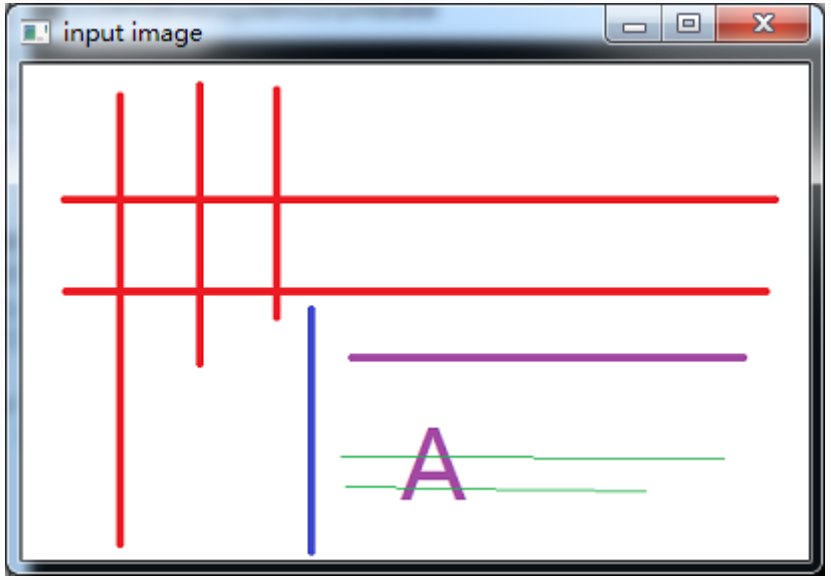



A Diamond-Shaped Structuring Element and its Origin

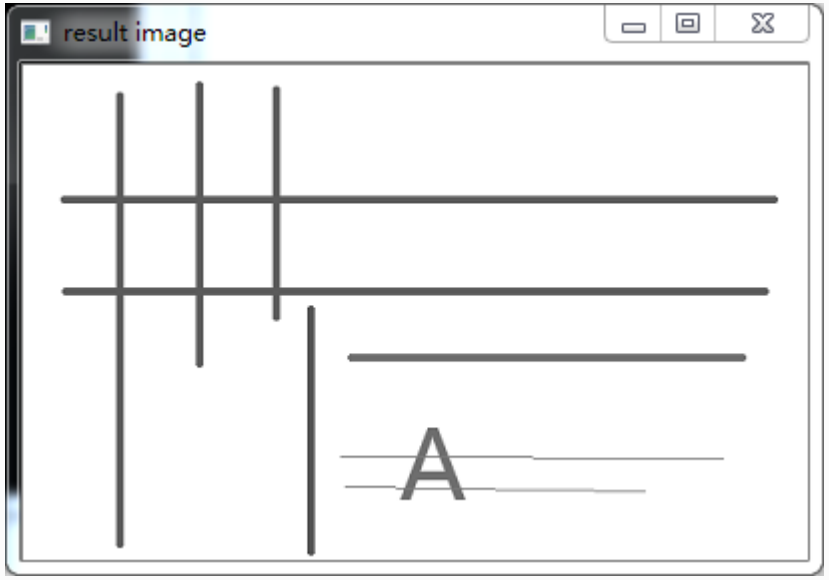
# 提取步骤

- 输入图像彩色图像 imread
- 转换为灰度图像 – cvtColor
- 转换为二值图像 – adaptiveThreshold
- 定义结构元素
- 开操作 （腐蚀+膨胀）提取 水平与垂直线

# 代码实现-第一步输入彩色图像 imread



# 转换为灰度图像 – cvtColor




```
Mat gray;
if (src.channels() == 3) {
    cvtColor(src, gray, CV_BGR2GRAY);
} else {
    gray = src;
}
imshow(OUTPUT_WIN, gray);
```

# 转换为二值图像 – adaptiveThreshold

- adaptiveThreshold(

```
Mat src, // 输入的灰度图像
Mat dest, // 二值图像
double maxValue, // 二值图像最大值
int adaptiveMethod // 自适应方法，只能其中之一 –
                                // ADAPTIVE_THRESH_MEAN_C ， ADAPTIVE_THRESH_GAUSSIAN_C
int thresholdType,// 阈值类型
int blockSize, // 块大小
double C // 常量C 可以是正数，0，负数
)
```

# 转换为二值图像 – adaptiveThreshold

- THRESH_BINARY

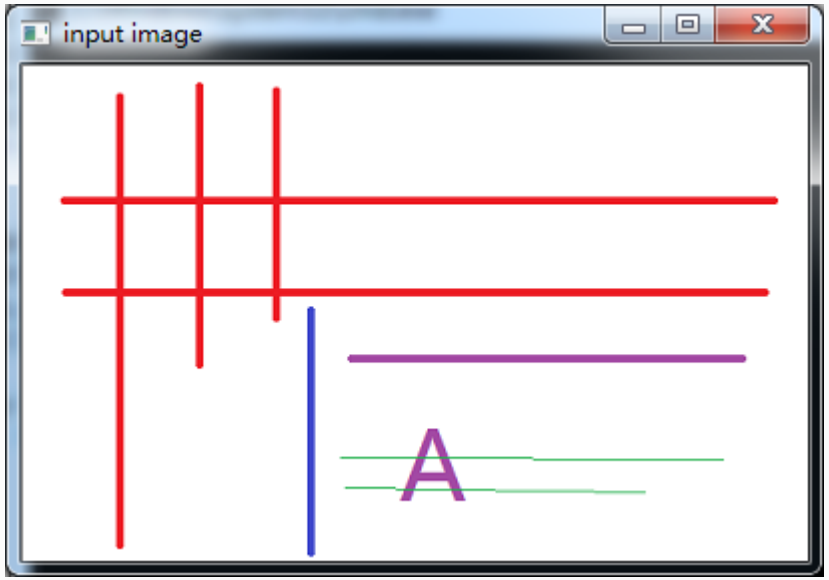
$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

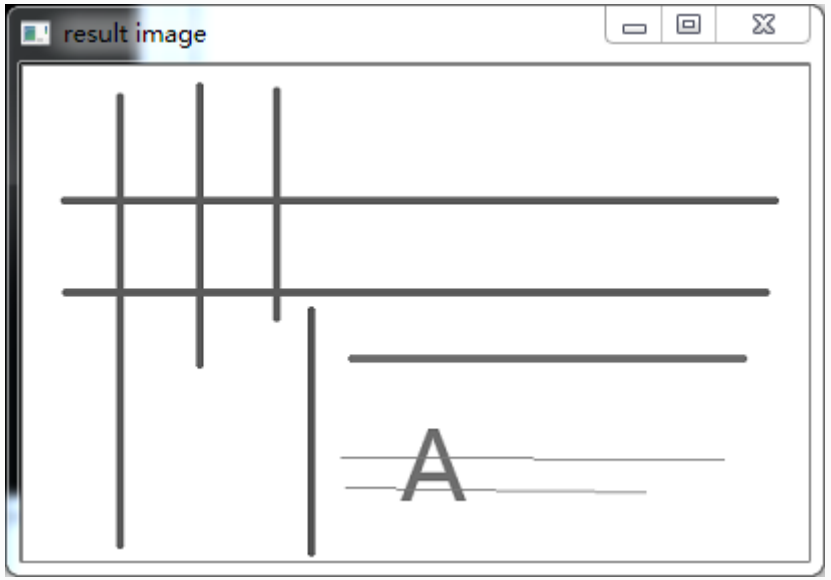
- THRESH_BINARY_INV

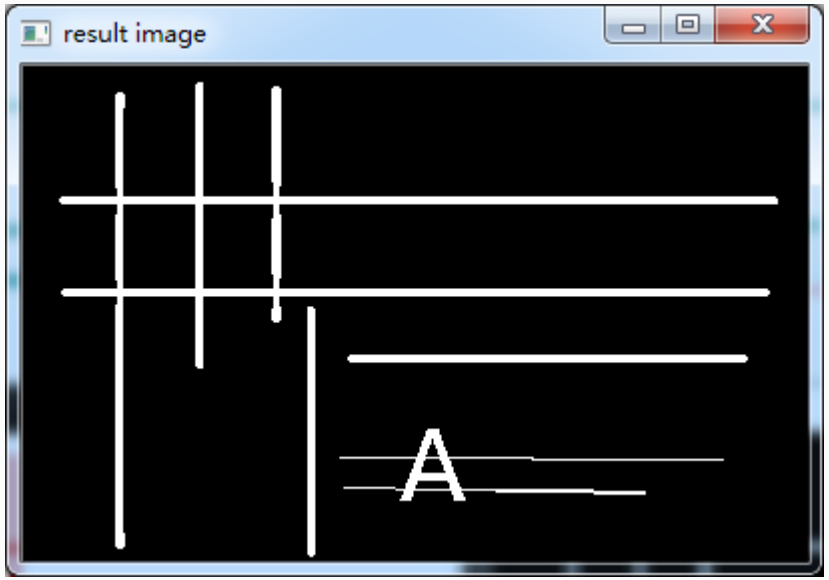
$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

阈值T = *sum*(blocksize X blockSize的像素平均值) - 常量C

# 转换为二值图像 – adaptiveThreshold



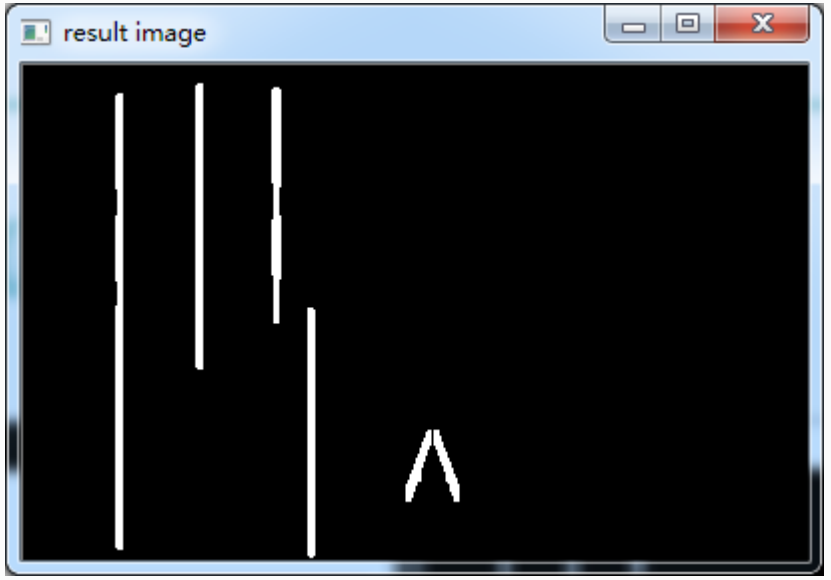



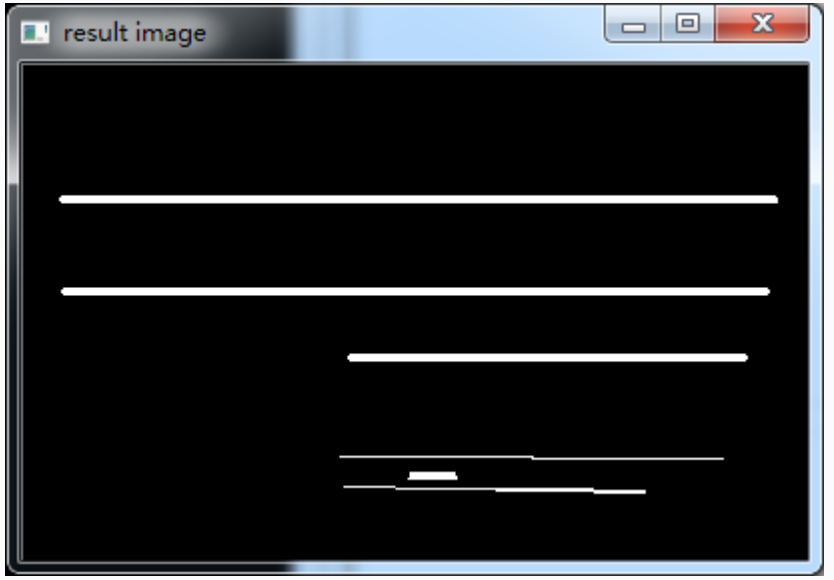



# 定义结构元素

- 一个像素宽的水平线 - 水平长度 width/30
- 一个像素宽的垂直线 – 垂直长度 height/30

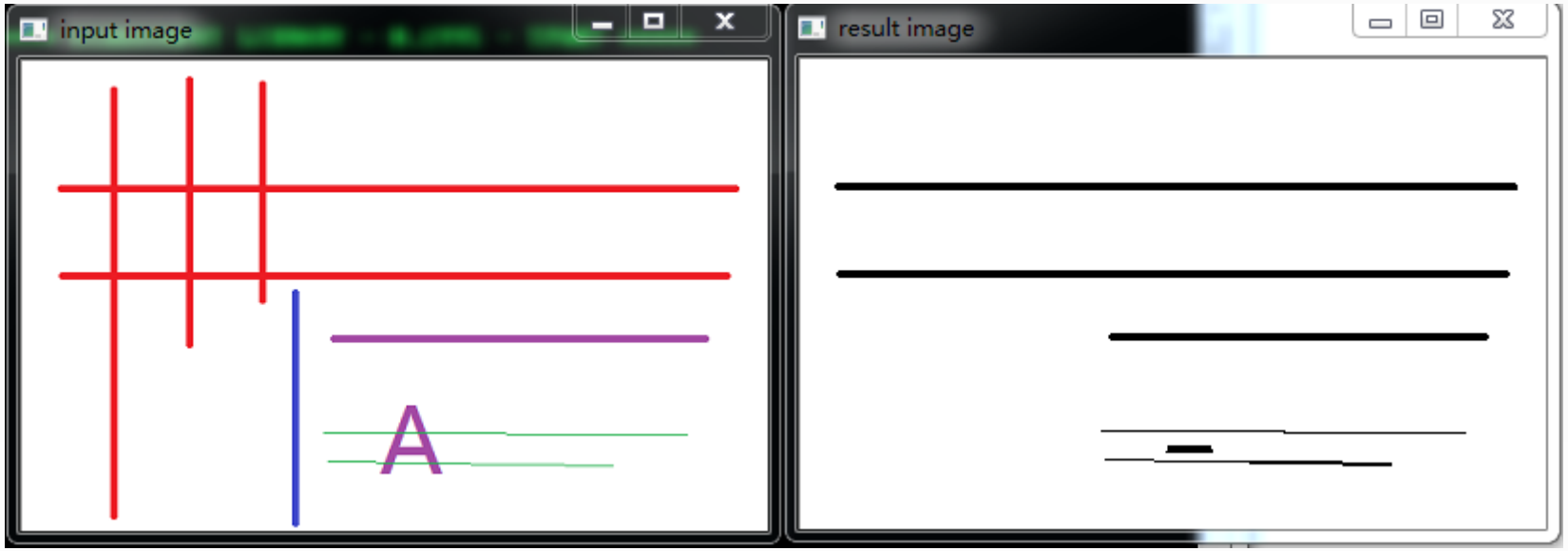
# 开操作(腐蚀+膨胀)-检测



# 后处理

- bitwise_not（Mat bin, Mat dst）像素取反操作，255 – SrcPixel
- 模糊（blur）



# 代码实现

```
Mat src, dest;
src = imread("D:/vcprojects/images/bin1.png");
if (!src.data) {
    printf("could not load image...\n");
    return -1;
}

char INPUT_WIN[] = "input image";
char OUTPUT_WIN[] = "result image";
namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
imshow(INPUT_WIN, src);

// conver to gray image
Mat gray;
if (src.channels() == 3) {
    cvtColor(src, gray, CV_BGR2GRAY);
} else {
    gray = src;
}

// convert to binary image
adaptiveThreshold(~gray, dest, 255, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY, 15, -2);

// create custom structure
int xsize = dest.cols / 30;
int ysize = dest.rows / 30;
Mat horline = getStructuringElement(MORPH_RECT, Size(xsize, 1), Point(-1, -1));
Mat vecline = getStructuringElement(MORPH_RECT, Size(1, ysize), Point(-1, -1));

// open operation - extract horizle lines
Mat hbin;
erode(dest, hbin, horline);
dilate(hbin, dest, horline);
imshow(OUTPUT_WIN, dest);
```

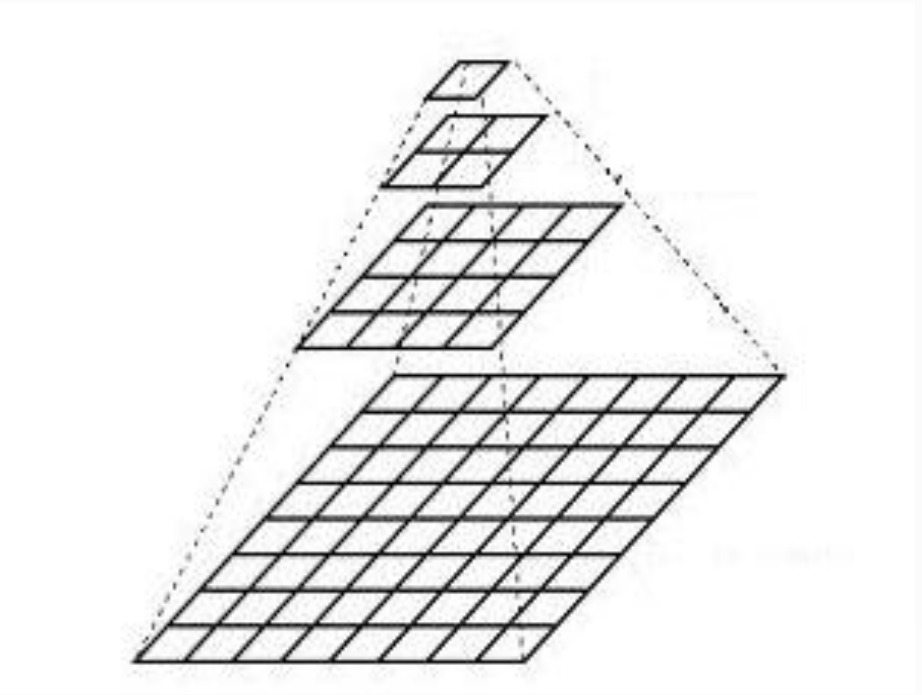
# 图像上采样和降采样

- 图像金字塔概念
- 采样API
- 代码演示

# 图像金字塔概念

1. 我们在图像处理中常常会调整图像大小，最常见的就是放大(zoom in)和缩小（zoom out），尽管几何变换也可以实现图像放大和缩小，但是这里我们介绍图像金字塔
2. 一个图像金字塔式一系列的图像组成，最底下一张是图像尺寸最大，最上方的图像尺寸最小，从空间上从上向下看就想一个古代的金字塔。

# 图像金字塔概念



# 图像金字塔概念

- 高斯金子塔 – 用来对图像进行降采样
- 拉普拉斯金字塔 – 用来重建一张图片根据它的上层降采样图片

# 图像金字塔概念 – 高斯金字塔

- 高斯金子塔是从底向上，逐层降采样得到。
- 降采样之后图像大小是原图像MxN的M/2 x N/2 ,就是对原图像删除偶数行与列，即得到降采样之后上一层的图片。
- 高斯金子塔的生成过程分为两步：
  - 对当前层进行高斯模糊
  - 删除当前层的偶数行与列

  即可得到上一层的图像，这样上一层跟下一层相比，都只有它的1/4大小。

$$\frac{1}{16}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

# 高斯不同(Difference of Gaussian-DOG)

- 定义：就是把同一张图像在不同的参数下做高斯模糊之后的结果相减，得到的输出图像。称为高斯不同(DOG)
- 高斯不同是图像的内在特征，在灰度图像增强、角点检测中经常用到。

# 采样相关API

- 上采样(cv::pyrUp) – zoom in 放大
- 降采样 (cv::pyrDown) – zoom out 缩小

pyrUp(Mat src, Mat dst, Size(src.cols*2, src.rows*2))
生成的图像是原图在宽与高各放大两倍
pyrDown(Mat src, Mat dst, Size(src.cols/2, src.rows/2))
生成的图像是原图在宽与高各缩小1/2

# 演示代码

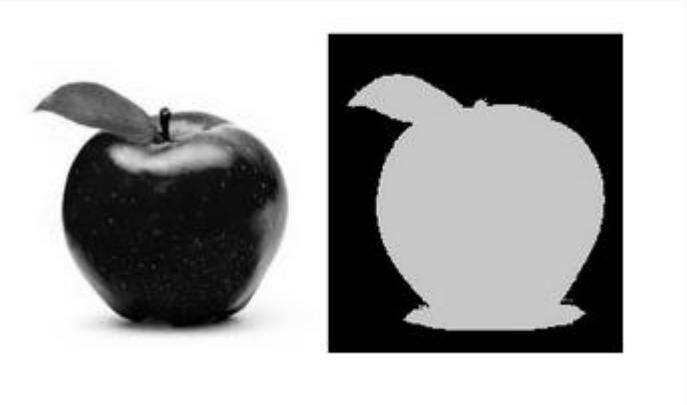
```
Mat src, dest;
src = imread("D:/vcprojects/images/cat.jpg");
if (!src.data) {
    printf("could not load image...");
    return -1;
}

char INPUT_WIN[] = "input image";
char OUTPUT_WIN[] = "show result";
namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
namedWindow(OUTPUT_WIN, CV_WINDOW_AUTOSIZE);
imshow(INPUT_WIN, src);

// zoom out
// pyrDown(src, dest, Size(src.cols / 2, src.rows / 2));
// imshow(OUTPUT_WIN, dest);

// zoom in
pyrUp(src, dest, Size(src.cols * 2, src.rows * 2));
imshow(OUTPUT_WIN, dest);

waitKey(0);
return 0;
```

# 基本阈值操作

- 图像阈值
- 阈值类型
- 代码演示

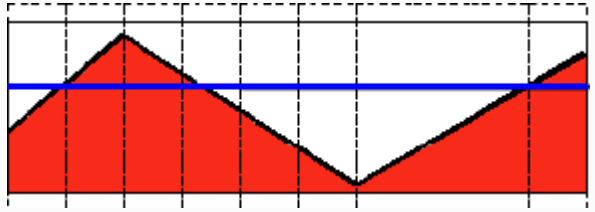
# 图像阈值（threshold）

- **阈值** 是什么？简单点说是把图像分割的标尺，这个标尺是根据什么产生的，阈值产生算法？阈值类型。（Binary segmentation）

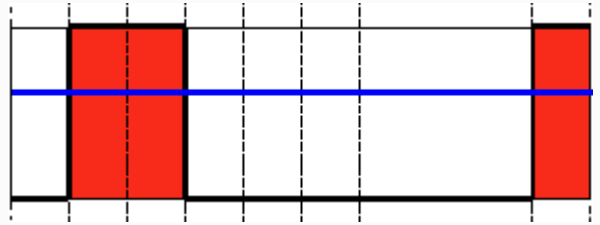


# 阈值类型—阈值二值化(threshold binary)

- 左下方的图表示图像像素点Src(x,y)值分布情况，蓝色水平线表示阈值

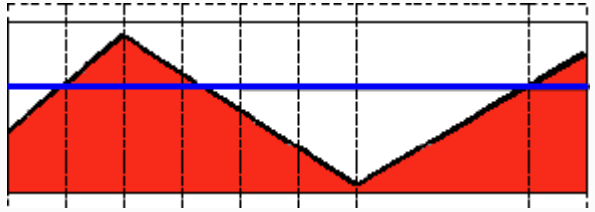


$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

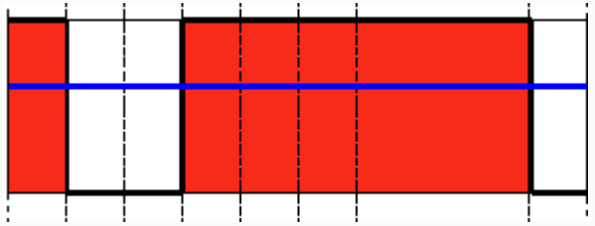


# 阈值类型—阈值反二值化(threshold binary Inverted)

- 左下方的图表示图像像素点Src(x,y)值分布情况，蓝色水平线表示阈值

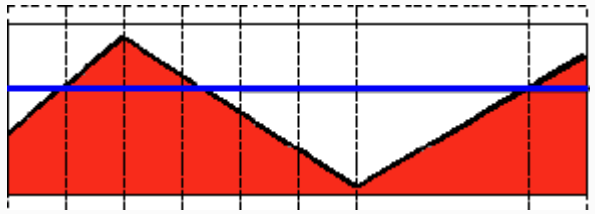


$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxVal} & \text{otherwise} \end{cases}$$

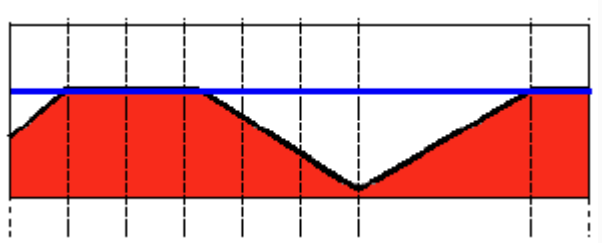


# 阈值类型—截断 (truncate)

- 左下方的图表示图像像素点Src(x,y)值分布情况，蓝色水平线表示阈值

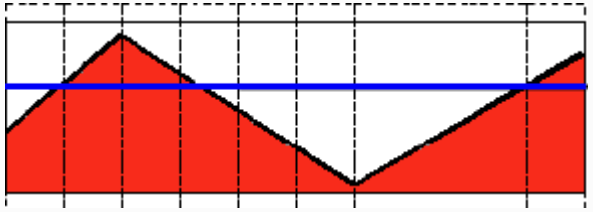


$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

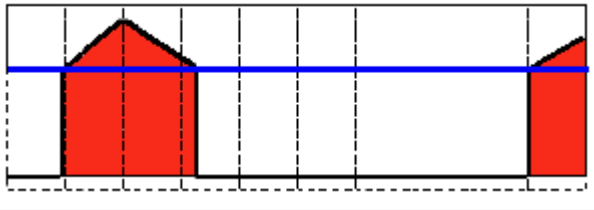


# 阈值类型—阈值取零 (threshold to zero)

- 左下方的图表示图像像素点Src(x,y)值分布情况，蓝色水平线表示阈值

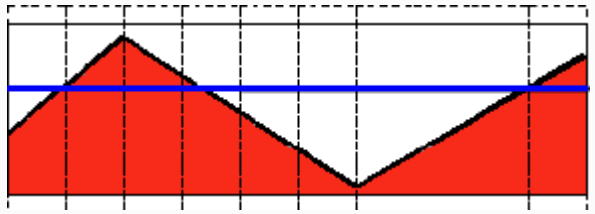


$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

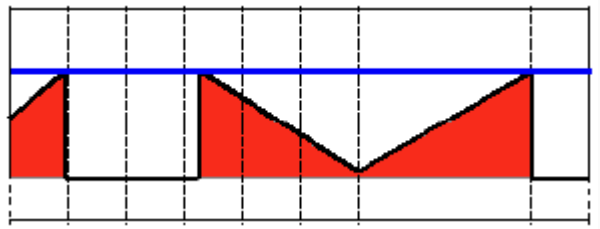


# 阈值类型—阈值反取零 (threshold to zero inverted)

- 左下方的图表示图像像素点Src(x,y)值分布情况，蓝色水平线表示阈值



$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$



| Enumerator | |
|---|---|
| THRESH_BINARY | $\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$ |
| THRESH_BINARY_INV | $\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxval} & \text{otherwise} \end{cases}$ |
| THRESH_TRUNC | $\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$ |
| THRESH_TOZERO | $\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$ |
| THRESH_TOZERO_INV | $\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$ |
| THRESH_MASK | |
| THRESH_OTSU | flag, use Otsu algorithm to choose the optimal threshold value |
| THRESH_TRIANGLE | flag, use Triangle algorithm to choose the optimal threshold value |

# 演示代码

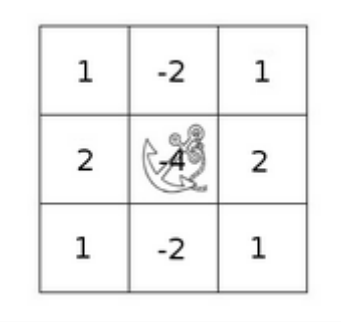
```
// conver to gray
if (src.channels() == 3) {
    cvtColor(src, temp, CV_BGR2GRAY);
} else {
    temp = src;
}

// apply threshold
createTrackbar(trackbar_value, OUTPUT_WIN, &threshold_value, maxvalue, threshold_demo);
threshold_demo(0, 0);

// loop for your end
for (;;) {
    int c;
    c = waitKey(20);
    if ((char)c == 27) { // ESC
        break;
    }
}
return 0;
```

# 自定义线性滤波

- 卷积概念
- 常见算子
- 自定义卷积模糊
- 代码演示

# 卷积概念

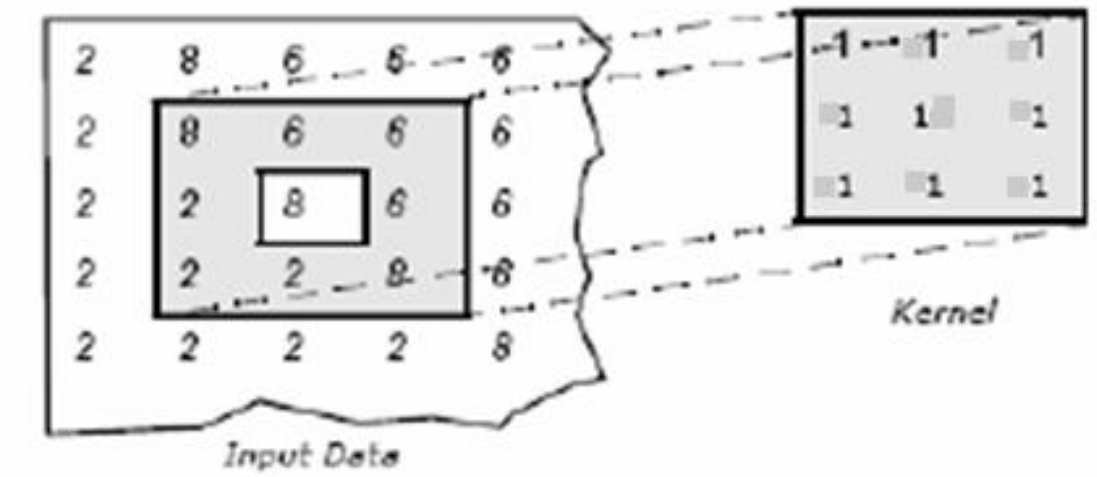
- **卷积**是图像处理中一个操作，是kernel在图像的每个像素上的操作。
- **Kernel**本质上一个固定大小的矩阵数组，其中心点称为锚点(anchor point)



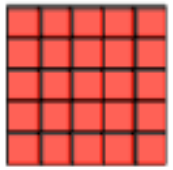
# 卷积如何工作

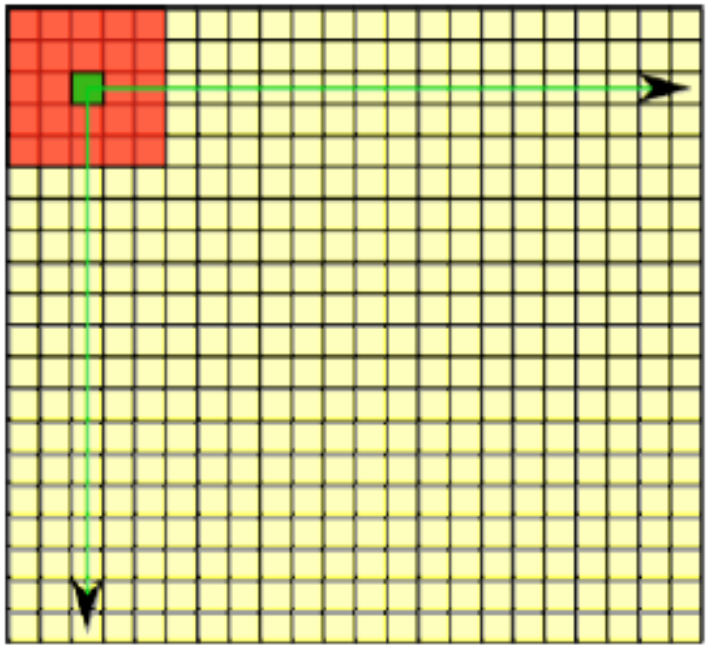
- 把kernel放到像素数组之上，求锚点周围覆盖的像素乘积之和（包括锚点），用来替换锚点覆盖下像素点值称为卷积处理。数学表达如下：

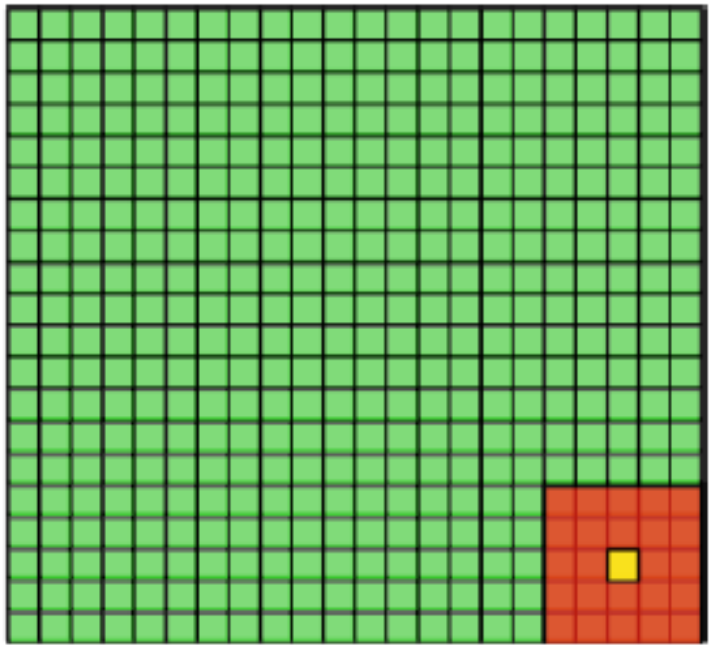
$$H(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j) K(i,j)$$




Sum = 8x1+6x1+6x1+2x1+8x1+6x1+2x1+2x1+8x1
New pixel = sum / (m*n)







# 常见算子

| | |
|---|---|
| +1 | 0 |
| 0 | -1 |

| | |
|---|---|
| 0 | +1 |
| -1 | 0 |

Robert算子

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

Sobel算子

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 4 | -1 |
| 0 | -1 | 0 |

拉普拉斯算子

# 自定义卷积模糊

- filter2D方法filter2D(
Mat src, //输入图像
Mat dst, // 模糊图像
int depth, // 图像深度32/8
Mat kernel, // 卷积核/模板
Point anchor, // 锚点位置
double delta // 计算出来的像素+delta
)
其中 kernel是可以自定义的卷积核

$$K = \frac{1}{3 \cdot 3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

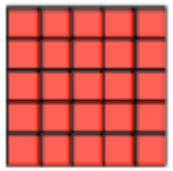
# 演示代码

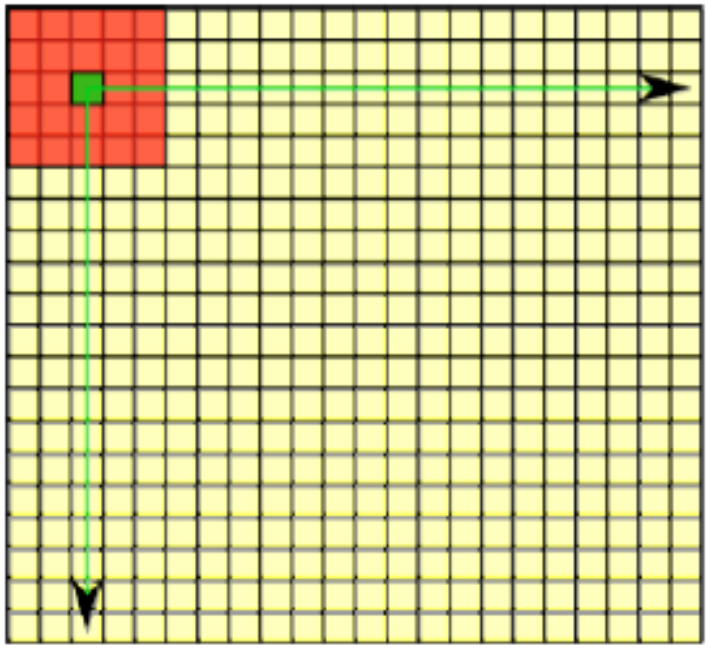
```
src = imread("D:/vcprojects/images/test.png");
if (!src.data) {
    printf("could not load image...\n");
    return -1;
}

char INPUT_WIN[] = "input image";
char OUTPUT_WIN[] = "result image";
namedWindow(INPUT_WIN, CV_WINDOW_AUTOSIZE);
namedWindow(OUTPUT_WIN, CV_WINDOW_AUTOSIZE);

imshow(INPUT_WIN, src);

int c = 0;
int index = 1;
while (true) {
    c = waitKey(500);
    if ((char)c == 27) {
        break;
    }
    ksize = 3 + 2 * (index % 5);
    kernel = Mat::ones(Size(ksize, ksize), CV_32F) / (float)(ksize*ksize);
    filter2D(src, dest, -1, kernel, Point(-1, -1), 0);
    imshow(OUTPUT_WIN, dest);
    index++;
}
return 0;
```

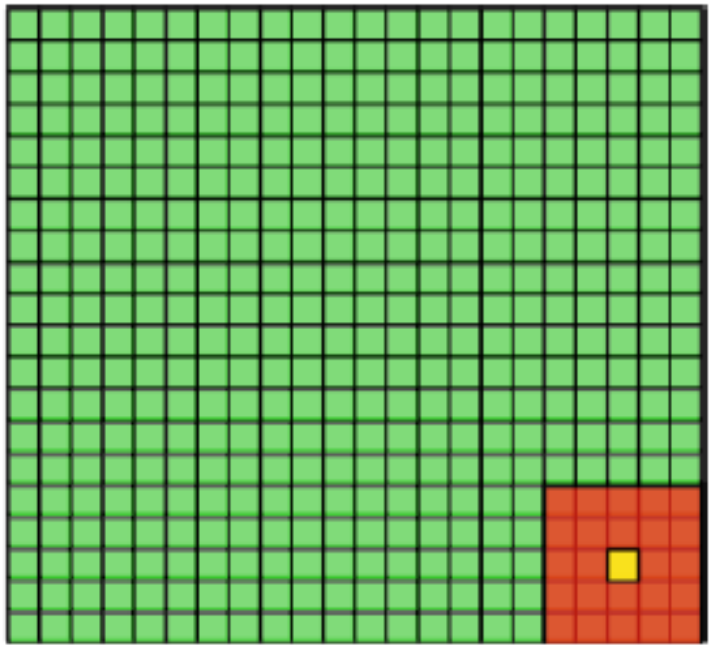
# 处理边缘

- 卷积边缘问题
- 处理边缘
- 代码演示

# 卷积边缘问题







# 卷积边界问题

- 图像卷积的时候边界像素，不能被卷积操作，原因在于边界像素没有完全跟kernel重叠，所以当3x3滤波时候有1个像素的边缘没有被处理，5x5滤波的时候有2个像素的边缘没有被处理。

# 处理边缘

在卷积开始之前增加边缘像素，填充的像素值为0或者RGB黑色，比如3x3在四周各填充1个像素的边缘，这样就确保图像的边缘被处理，在卷积处理之后再去掉这些边缘。openCV中默认的处理方法是： BORDER_DEFAULT，此外常用的还有如下几种：

- **BORDER_CONSTANT** – 填充边缘用指定像素值
- **BORDER_REPLICATE –** 填充边缘像素用已知的边缘像素值。
- **BORDER_WRAP** – 用另外一边的像素来补偿填充

BORDER_DEFAULT


BORDER_CONSTANT


## BORDER_REPLICATE – 通过插值计算


## BORDER_WRAP – 另外一边补偿


# API说明 – 给图像添加边缘API

- copyMakeBorder（
  - Mat src, // 输入图像
  - Mat dst, // 添加边缘图像
  - int top, // 边缘长度，一般上下左右都取相同值，
  - int bottom,
  - int left,
  - int right,
  - int borderType // 边缘类型
  - Scalar value
  )

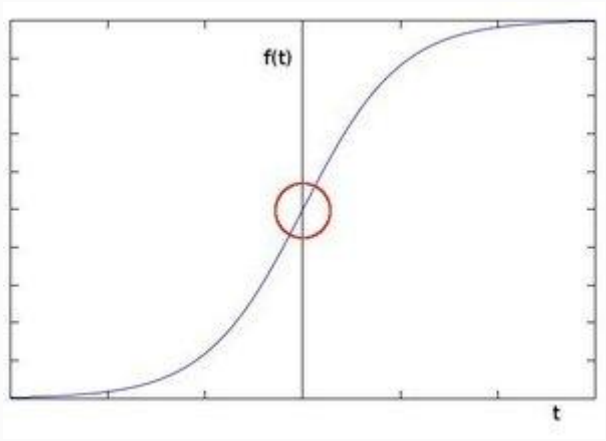
# 演示代码

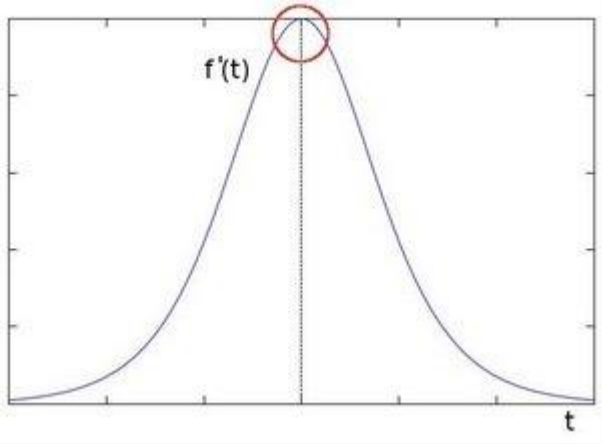
```
int flag = 0;
// int border_type = BORDER_DEFAULT;
int border_type = BORDER_WRAP;
RNG rng;
Scalar color;
int top = (int)(0.05*src.rows);
int bottom = (int)(0.05*src.rows);
int left = (int)(0.05*src.cols);
int right = (int)(0.05*src.cols);

while (true) {
    flag = waitKey(500);
    if ((char)flag == 27) {
        break;
    }
    if ((char)flag == 'c') {
        border_type = BORDER_CONSTANT;
    }
    else if ((char)flag == 'r') {
        border_type = BORDER_REPLICATE;
    }
    color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    copyMakeBorder(src, dest, top, bottom, left, right, border_type, color);
    imshow(OUTPUT_WIN, dest);
}
return 0;
```

# Sobel算子

- 卷积应用-图像边缘提取
- 相关API
- 代码演示

# 卷积应用-图像边缘提取



# 卷积应用-图像边缘提取

- 边缘是什么 – 是像素值发生跃迁的地方，是图像的显著特征之一，在图像特征提取、对象检测、模式识别等方面都有重要的作用。
- 如何捕捉/提取边缘 – 对图像求它的一阶导数

  delta = f(x) – f(x-1), delta越大，说明像素在X方向变化越大，边缘信号越强，
- 我已经忘记啦，不要担心，用Sobel算子就好！卷积操作！

# Sobel算子

- 是离散微分算子（discrete differentiation operator），用来计算图像灰度的近似梯度
- Soble算子功能集合高斯平滑和微分求导
- 又被称为一阶微分算子，求导算子，在水平和垂直两个方向上求导，得到图像X方法与Y方向梯度图像

# Sobel算子

水平梯度

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

垂直梯度

$$G = \sqrt{G_x^2 + G_y^2}$$

$$G = |G_x| + |G_y|$$

最终图像梯度

# Sobel算子

- 求取导数的近似值，kernel=3时不是很准确，OpenCV使用改进版本Scharr函数，算子如下：

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

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

# API说明cv::Sobel

```
cv::Sobel (
InputArray Src // 输入图像
OutputArray dst// 输出图像，大小与输入图像一致
int depth // 输出图像深度.
Int dx.  // X方向，几阶导数
int dy // Y方向，几阶导数.
int ksize, SOBEL算子kernel大小，必须是1、3、5、7.
double scale  = 1
double delta = 0
int borderType = BORDER_DEFAULT
)
```

| Input depth (src.depth()) | Output depth (ddepth) |
|---|---|
| CV_8U | -1/CV_16S/CV_32F/CV_64F |
| CV_16U/CV_16S | -1/CV_32F/CV_64F |
| CV_32F | -1/CV_32F/CV_64F |
| CV_64F | -1/CV_64F |

# API说明cv::Scharr

```
cv::Scharr (
InputArray Src // 输入图像
OutputArray dst// 输出图像，大小与输入图像一致
int depth // 输出图像深度.
Int dx.  // X方向，几阶导数
int dy // Y方向，几阶导数.
double scale  = 1
double delta = 0
int borderType = BORDER_DEFAULT
)
```

# 其它API

- GaussianBlur( src, dst, Size(3,3), 0, 0, BORDER_DEFAULT );
- cvtColor( src,  gray, COLOR_RGB2GRAY );
- addWeighted( A, 0.5,B, 0.5, 0, AB);
- convertScaleAbs(A, B)// 计算图像A的像素绝对值，输出到图像B

$$\texttt{dst}(I) = \texttt{saturate\_cast<uchar>}(|\texttt{src}(I) * \texttt{alpha} + \texttt{beta}|)$$

# 演示代码

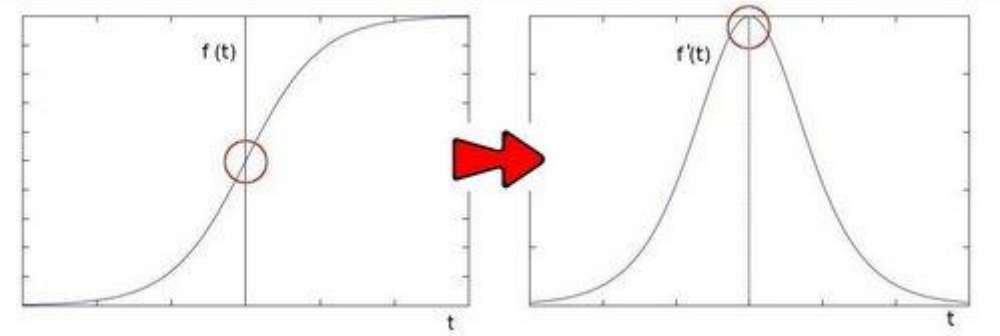
```
int main(int argc, char** argv) {
    Mat src, dest;
    src = imread("D:/vcprojects/images/test.png");
    if (!src.data) {
        printf("could not load image...\n");
        return -1;
    }

    char INPUT_TITLE[] = "input image";
    char OUTPUT_TITLE[] = "sobel-demo";
    namedWindow(INPUT_TITLE, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_TITLE, CV_WINDOW_AUTOSIZE);

    GaussianBlur(src, src, Size(3, 3), 0, 0, BORDER_DEFAULT);
    cvtColor(src, src, CV_BGR2GRAY);
    imshow(INPUT_TITLE, src);

    Mat xgrad, ygrad;
    Sobel(src, xgrad, CV_16S, 1, 0, 3);
    Sobel(src, ygrad, CV_16S, 0, 1, 3);

    convertScaleAbs(xgrad, xgrad);
    convertScaleAbs(ygrad, ygrad);

    addWeighted(xgrad, 0.5, ygrad, 0.5, 0, dest);
    imshow(OUTPUT_TITLE, dest);

    waitKey(0);
    return 0;
}
```

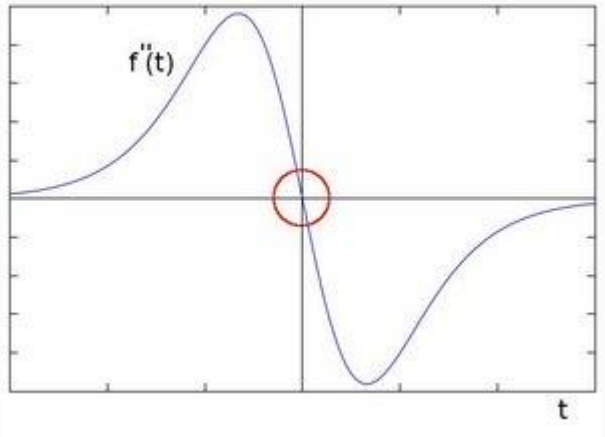

input image
sobel-demo

# Laplance算子

- 理论
- API使用
- 代码演示

# 理论




**解释：**在二阶导数的时候，最大变化处的值为零即边缘是零值。通过二阶导数计算，依据此理论我们可以计算图像二阶导数，提取边缘。

# Laplance算子

- 二阶导数我不会，别担心 ->拉普拉斯算子(Laplance operator)

$$Laplace(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

- Opencv已经提供了相关API - cv::Laplance

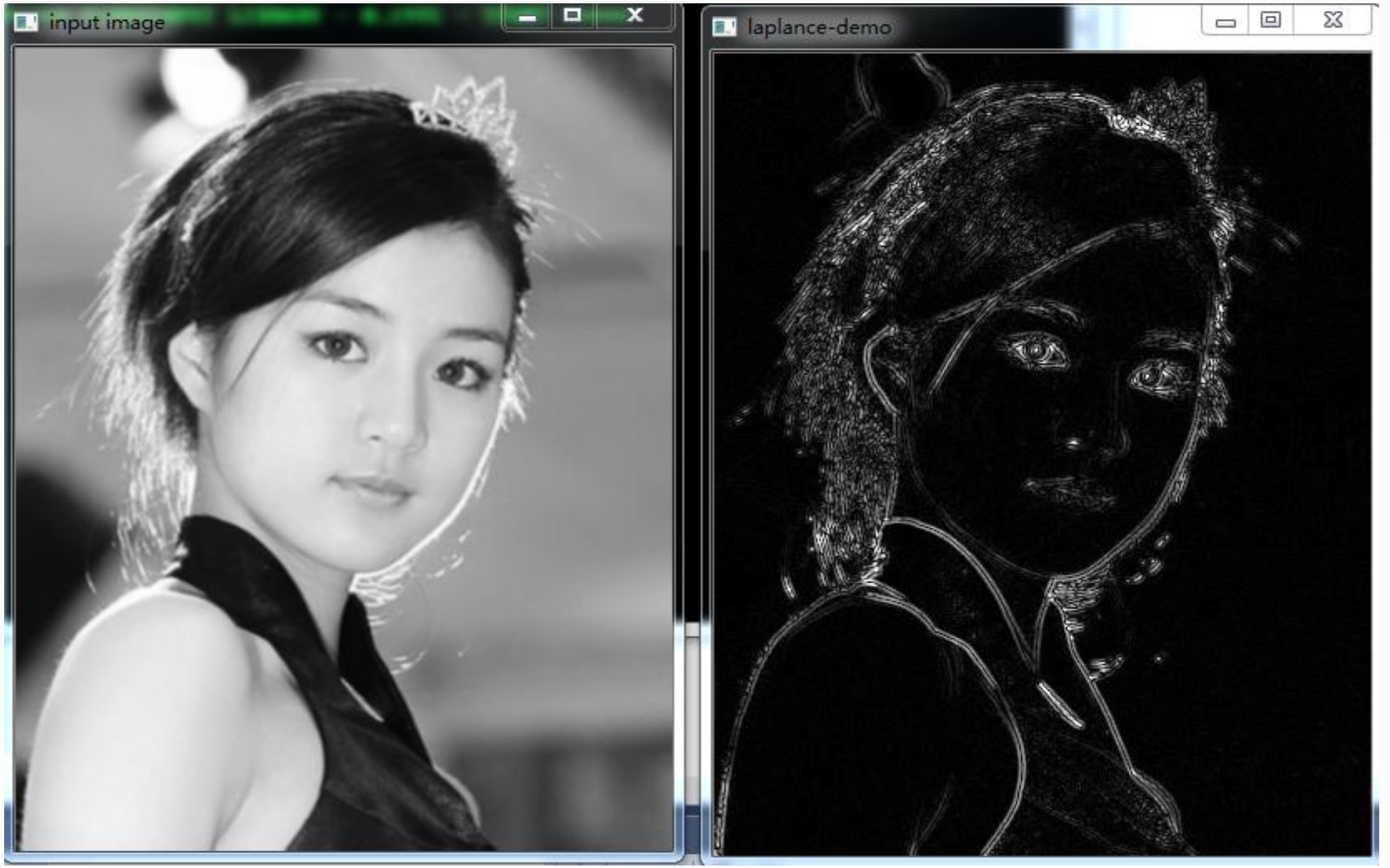
# 处理流程

- 高斯模糊 – 去噪声GaussianBlur()
- 转换为灰度图像cvtColor()
- 拉普拉斯 – 二阶导数计算Laplacian()
- 取绝对值convertScaleAbs()
- 显示结果

# API使用cv::Laplacian

```
Laplacian(
InputArray src,
OutputArray dst,
int depth, //深度CV_16S
int kisze, // 3
double scale = 1,
double delta =0.0,
int borderType = 4
)
```

# 显示图像



# 演示代码

```
Mat src, dest;
src = imread("D:/vcprojects/images/test.png");
if (!src.data) {
    printf("could not load image...\n");
    return -1;
}

char INPUT_TITLE[] = "input image";
char OUTPUT_TITLE[] = "laplance-demo";
namedWindow(INPUT_TITLE, CV_WINDOW_AUTOSIZE);
namedWindow(OUTPUT_TITLE, CV_WINDOW_AUTOSIZE);

// pre-process, blur
GaussianBlur(src, src, Size(3, 3), 0);

// convert to gray
cvtColor(src, src, CV_BGR2GRAY);

// display input
imshow(INPUT_TITLE, src);

// laplance
Laplacian(src, dest, CV_16S, 3);
convertScaleAbs(dest, dest);

imshow(OUTPUT_TITLE, dest);

waitKey(0);
return 0;
```

# Canny边缘检测

- Canny算法介绍
- API cv::Canny()
- 代码演示

# Canny算法介绍

- **Canny**是边缘检测算法，在1986年提出的。
- 是一个很好的边缘检测器
- 很常用也很实用的图像处理方法

# Canny算法介绍 – 五步 in cv::Canny

1. 高斯模糊 - GaussianBlur
2. 灰度转换 - cvtColor
3. 计算梯度 – Sobel/Scharr
4. 非最大信号抑制
5. 高低阈值输出二值图像

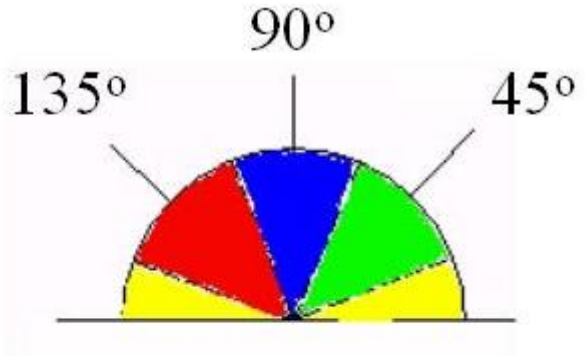
# Canny算法介绍 - 非最大信号抑制

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

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

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan(\frac{G_y}{G_x})$$




其中黄色区域取值范围为0~22.5 与157.5～180

绿色区域取值范围为22.5 ~ 67.5

蓝色区域取值范围为67.5~112.5

红色区域取值范围为112.5~157.5

# Canny算法介绍-高低阈值输出二值图像

- T1， T2为阈值，凡是高于T2的都保留，凡是小于T1都丢弃，从高于T2的像素出发，凡是大于T1而且相互连接的，都保留。最终得到一个输出二值图像。
- 推荐的高低阈值比值为 T2: T1 = 3:1/2:1其中T2为高阈值，T1为低阈值

# API – cv::Canny

```
Canny（
InputArray src, // 8-bit的输入图像
OutputArray edges,// 输出边缘图像， 一般都是二值图像，背景是黑色
double threshold1,// 低阈值，常取高阈值的1/2或者1/3
double threshold2,// 高阈值
int aptertureSize,// Soble算子的size，通常3x3，取值3
bool L2gradient // 选择 true表示是L2来归一化，否则用L1归一化
）
```

$$L_2 \text{ norm} = \sqrt{(dI/dx)^2 + (dI/dy)^2}$$

$$L_1 \text{ norm} = |dI/dx| + |dI/dy|$$

默认情况一般选择是L1，参数设置为false

# 演示代码

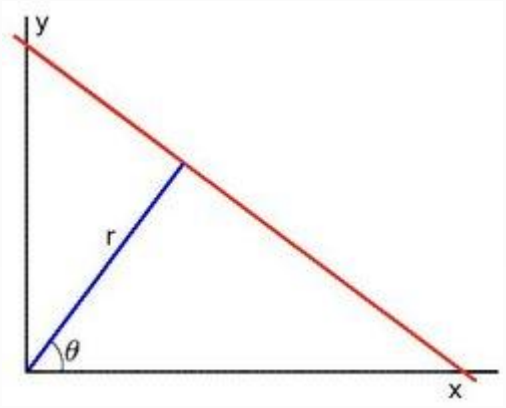
```
    if (!src.data) {
        printf("could not load image...\n");
        return -1;
    }

    char INPUT_TITLE[] = "input image";
    namedWindow(INPUT_TITLE, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_TITLE, CV_WINDOW_AUTOSIZE);
    imshow(INPUT_TITLE, src);

    dest.create(src.size(), src.type());
    cvtColor(src, gray_src, CV_BGR2GRAY);
    createTrackbar("Threshold :", OUTPUT_TITLE, &high_threshold, max_value, CannyDemo);
    CannyDemo(0, 0);

    waitKey(0);
}

void CannyDemo(int, void*) {
    blur(gray_src, edges_dest, Size(3, 3), Point(-1, -1), BORDER_DEFAULT);
    double lowt = high_threshold / rate;
    Canny(edges_dest, edges_dest, lowt, high_threshold, 3, true);
    dest = Scalar::all(0);

    // 使用遮罩层，只有非零的元素才被copy到模板中。
    src.copyTo(dest, edges_dest);
    imshow(OUTPUT_TITLE, dest);
}
```

# 霍夫变换-直线

- 霍夫直线变换介绍
- 相关API学习
- 代码演示

# 霍夫直线变换介绍

- Hough Line Transform用来做直线检测
- 前提条件 – 边缘检测已经完成
- 平面空间到极坐标空间转换




$x=\rho\cos\theta, y=\rho\sin\theta$

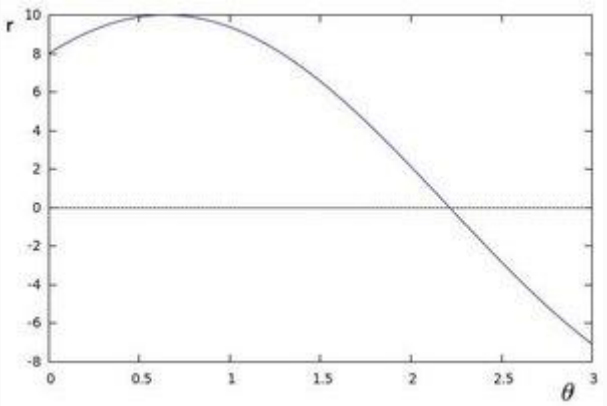
$\rho^2=x^2+y^2,\ \tan\theta=y/x\ (x\neq 0)$

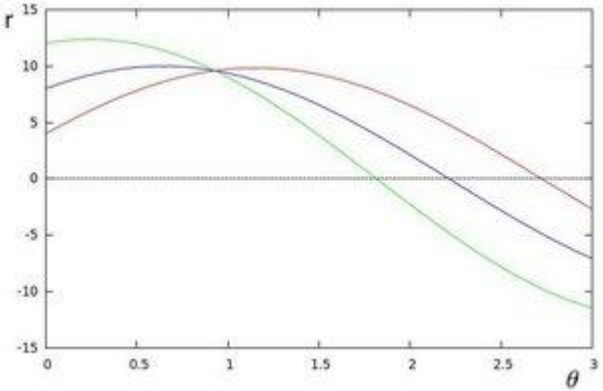
# 霍夫直线变换介绍

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$$

$$r = x\cos\theta + y\sin\theta$$

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta$$

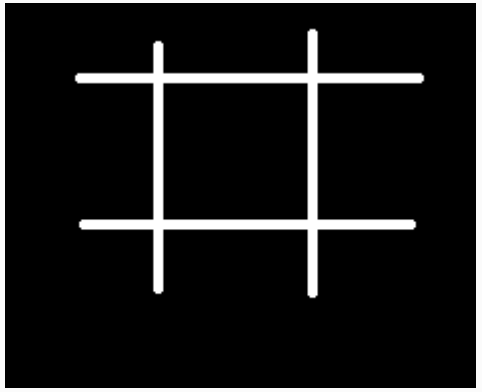


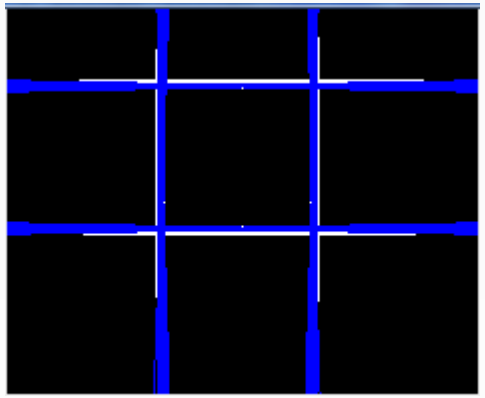



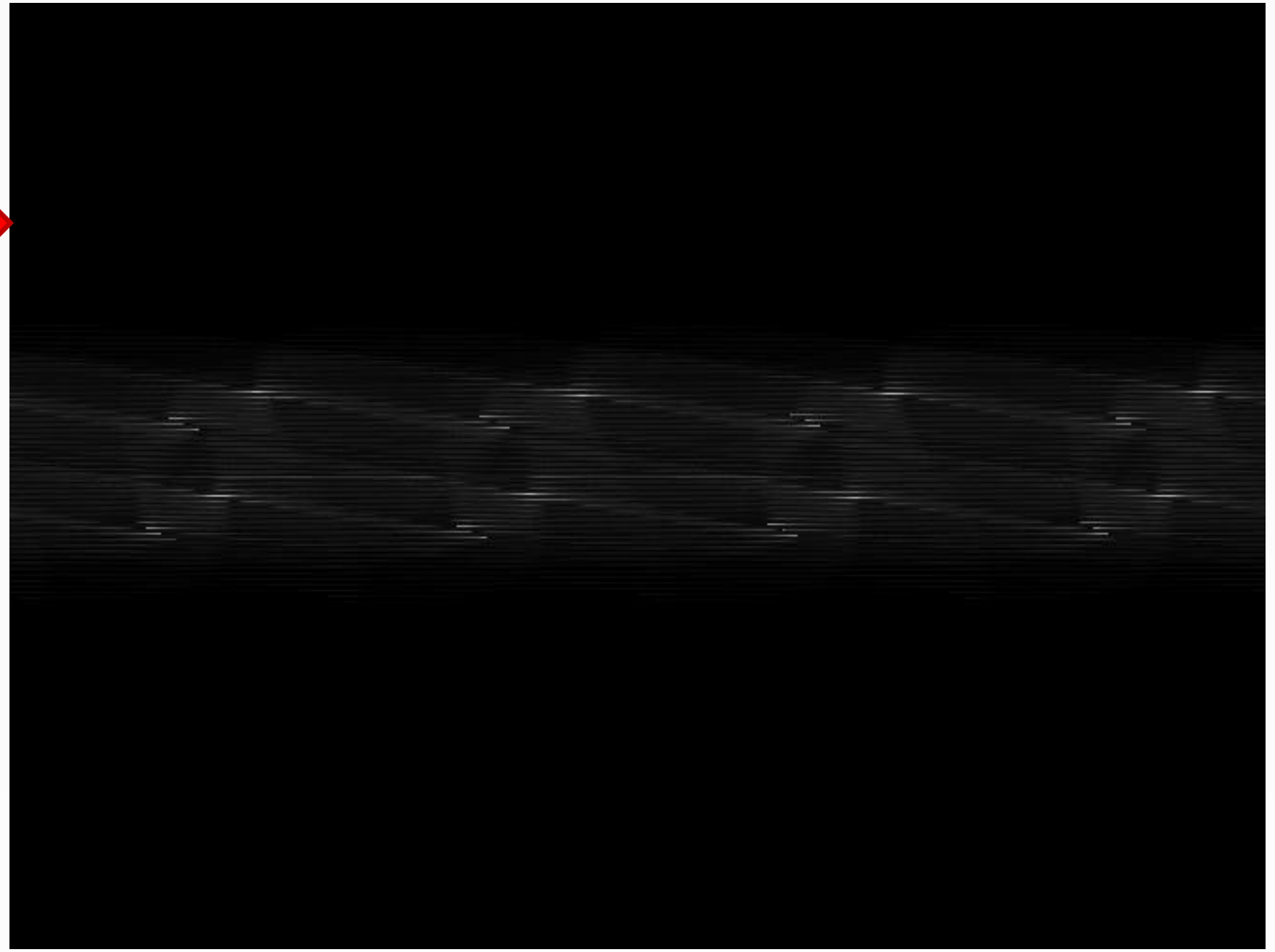
# 霍夫直线变换介绍

- 对于任意一条直线上的所有点来说
- 变换到极坐标中，从[0~360]空间，可以得到r的大小
- 属于同一条直线上点在极坐标空(r, theta)必然在一个点上有最强的信号出现，根据此反算到平面坐标中就可以得到直线上各点的像素坐标。从而得到直线

## 从平面坐标变换到霍夫空间（极坐标）







# 相关API学习

- 标准的霍夫变换 cv::HoughLines从平面坐标转换到霍夫空间，最终输出是$(\theta, r_\theta)$ 表示极坐标空间
- 霍夫变换直线概率 cv::HoughLinesP最终输出是直线的两个点$(x_0, y_0, x_1, y_1)$

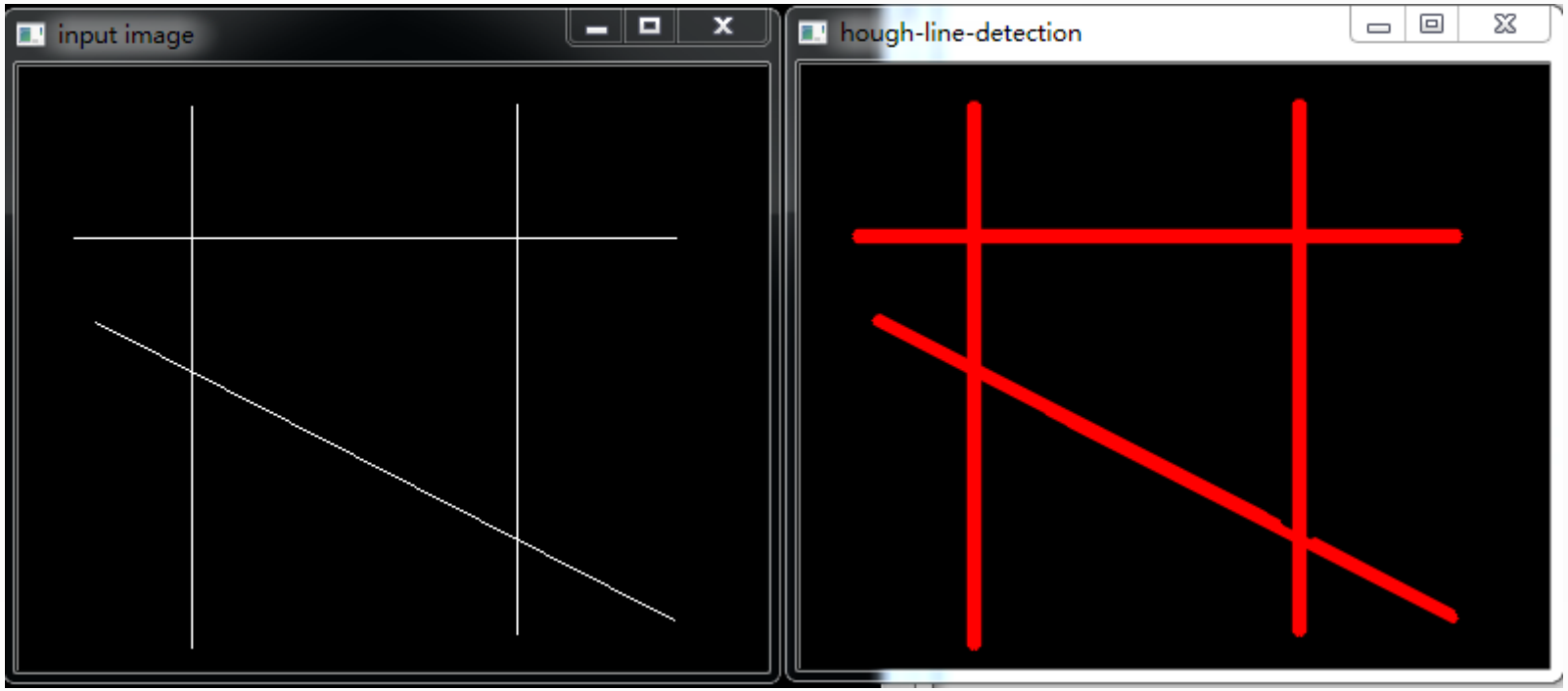
# 相关API学习

```
cv::HoughLines(
InputArray src, // 输入图像，必须8-bit的灰度图像
OutputArray lines, // 输出的极坐标来表示直线
double rho, // 生成极坐标时候的像素扫描步长
double theta, //生成极坐标时候的角度步长，一般取值CV_PI/180
int threshold, // 阈值，只有获得足够交点的极坐标点才被看成是直线
double srn=0;// 是否应用多尺度的霍夫变换，如果不是设置0表示经典霍夫变换
double stn=0;//是否应用多尺度的霍夫变换，如果不是设置0表示经典霍夫变换
double min_theta=0; // 表示角度扫描范围 0 ~180之间， 默认即可
double max_theta=CV_PI
) // 一般情况是有经验的开发者使用，需要自己反变换到平面空间
```

# 相关API学习

```
cv::HoughLinesP(
InputArray src, // 输入图像，必须8-bit的灰度图像
OutputArray lines, // 输出的极坐标来表示直线
double rho, // 生成极坐标时候的像素扫描步长
double theta, //生成极坐标时候的角度步长，一般取值CV_PI/180
int threshold, // 阈值，只有获得足够交点的极坐标点才被看成是直线
double minLineLength=0;// 最小直线长度
double maxLineGap=0;// 最大间隔
)
```

# HoughLinesP检测效果



# 演示代码

## 先进行边缘检测

```
imshow(INPUT_TITLE, src);
Canny(src, src_gray, 150, 200, 3);
cvtColor(src_gray, dest, CV_GRAY2BGR);
```

## 霍夫直线检测

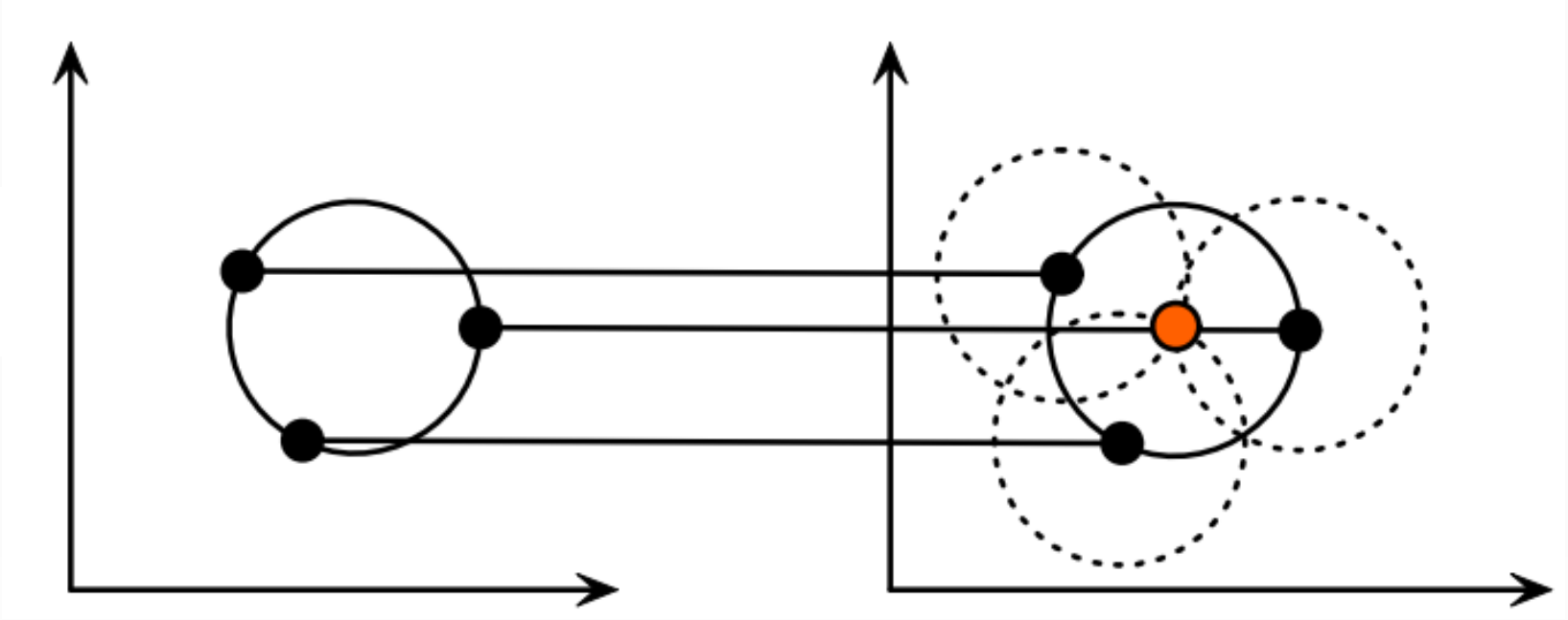
```
//
vector<Vec4f> plines;
HoughLinesP(src_gray, plines, 1, CV_PI / 180, 10, 0, 10);
for (size_t i = 0; i < plines.size(); i++) {
    Vec4f hl = plines[i];
    line(dest, Point(hl[0], hl[1]), Point(hl[2], hl[3]), Scalar(0, 0, 255), 3, CV_AA);
}
imshow(OUTPUT_TITLE, dest);
```

# 霍夫圆变换

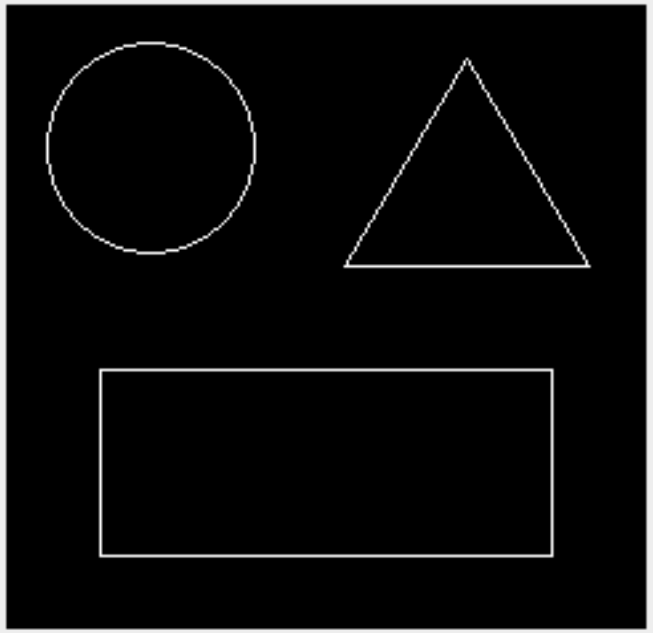
- 霍夫圆检测原理
- 相关API
- 代码演示

# 霍夫圆检测原理

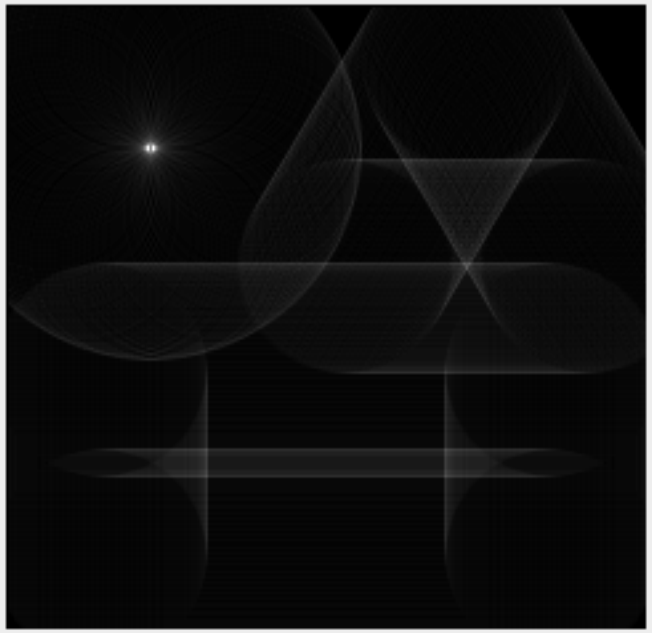
$$x = a + R\cos(\theta)$$
$$y = b + R\sin(\theta)$$



Hough[theta][a][b]++



原图



极坐标空间效果,最亮一点表示圆心

# 霍夫圆变换原理

- 从平面坐标到极坐标转换三个参数 $C(x_0, y_0, r)$其中$x_0, y_0$是圆心
- 假设平面坐标的任意一个圆上的点，转换到极坐标中：$C(x_0, y_0, r)$ 处有最大值，霍夫变换正是利用这个原理实现圆的检测。

# 相关API cv::HoughCircles

- 因为霍夫圆检测对噪声比较敏感，所以首先要对图像做中值滤波。
- 基于效率考虑，Opencv中实现的霍夫变换圆检测是基于图像梯度的实现，分为两步：

  1. 检测边缘，发现可能的圆心
  2. 基于第一步的基础上从候选圆心开始计算最佳半径大小

# HoughCircles参数说明

**HoughCircles**(
InputArray image, // 输入图像 ,必须是8位的单通道灰度图像
OutputArray circles, // 输出结果，发现的圆信息
Int method, // 方法 - HOUGH_GRADIENT
Double dp, // dp = 1;
Double mindist, // 10 最短距离-可以分辨是两个圆的，否则认为是同心圆- *src_gray.rows/8*
Double param1, // canny edge detection low threshold
Double param2, // 中心点累加器阈值 – 候选圆心
Int minradius, // 最小半径
Int maxradius//最大半径
)

# 演示代码

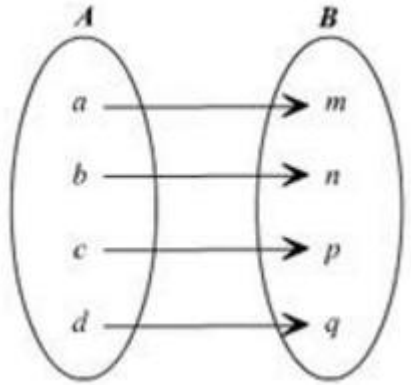
```
char INPUT_TITLE[] = "input image";
char OUTPUT_TITLE[] = "hough circle demo";
namedWindow(INPUT_TITLE, CV_WINDOW_AUTOSIZE);
namedWindow(OUTPUT_TITLE, CV_WINDOW_AUTOSIZE);

// show input image
// medianBlur(src, src, 5);
cvtColor(src, src, CV_BGR2GRAY);
GaussianBlur(src, dest, Size(5, 5), 0, 0);
imshow(INPUT_TITLE, src);

// 基于灰度空间
vector<Vec3f> circles;
HoughCircles(dest, circles, HOUGH_GRADIENT, 1, 10, 100, 30, 5, 50);
// 重新转回到RGB色彩空间
cvtColor(dest, dest, CV_GRAY2BGR);
for (size_t i = 0; i < circles.size(); i++) {
    Vec3f c3 = circles[i];
    circle(dest, Point(c3[0], c3[1]), c3[2], Scalar(0, 0, 255), 3, LINE_AA);
    circle(dest, Point(c3[0], c3[1]), 2, Scalar(0, 0, 255), 3, LINE_AA);
}
imshow(OUTPUT_TITLE, dest);

waitKey(0);
return 0;
```

# 像素重映射(cv::remap)

- 什么是像素重映射
- API介绍
- 代码演示

# 什么是像素重映射

- **简单点说就是把输入图像中各个像素按照一定的规则映射到另外一张图像的对应位置上去，形成一张新的图像。**




$$g(x,y) = f(h(x,y))$$

g(x,y)是重映射之后的图像，h(x,y)是功能函数，f是源图像

# 什么是像素重映射

假设有映射函数 $h(x,y) = (I.cols - x, y)$

# API介绍cv::remap

```
Remap(
InputArray src,// 输入图像
OutputArray dst,// 输出图像
InputArray  map1,// x 映射表 CV_32FC1/CV_32FC2
InputArray map2,// y 映射表
int interpolation,// 选择的插值方法，常见线性插值，可选择立方等
int borderMode,// BORDER_CONSTANT
const Scalar borderValue// color
)
```

# API介绍cv::remap

```
if (x > src.cols*0.25 && x < src.cols*0.75 && y > src.rows*0.25 && y < src.rows*0.75) {
    map_x.at<float>(y, x) = 2 * (x - src.cols*0.25f) + 0.5f;
    map_y.at<float>(y, x) = 2 * (y - src.rows*0.25f) + 0.5f;
} else {
    map_x.at<float>(y, x) = 0;
    map_y.at<float>(y, x) = 0;
}
```

缩小一半

```
map_x.at<float>(y, x) = (float)x;
map_y.at<float>(y, x) = (float)(src.rows - y);
```

Y方向对调

```
map_x.at<float>(y, x) = (float)(src.cols - x);
map_y.at<float>(y, x) = (float)y;
```

X方向对调

```
map_x.at<float>(y, x) = (float)(src.cols - x);
map_y.at<float>(y, x) = (float)(src.rows - y);
```

XY方向同时对调

# 演示代码

```
int main(int argc, char** argv) {
    src = imread("D:/vcprojects/images/test.png");
    if (!src.data) {
        printf("could not load image...\n");
        return -1;
    }
    char input_win[] = "input image";
    namedWindow(input_win, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_TITLE, CV_WINDOW_AUTOSIZE);
    imshow(input_win, src);
    map_x.create(src.size(), CV_32FC1);
    map_y.create(src.size(), CV_32FC1);
    while (true) {
        index = waitKey(500);
        if ((char)index == 27) {
            break;
        }
        update_remap();
        remap(src, dst, map_x, map_y, INTER_LINEAR, BORDER_CONSTANT, Scalar(0, 255, 255));
        imshow(OUTPUT_TITLE, dst);
    }

    return 0;
}
```

# 直方图均衡化

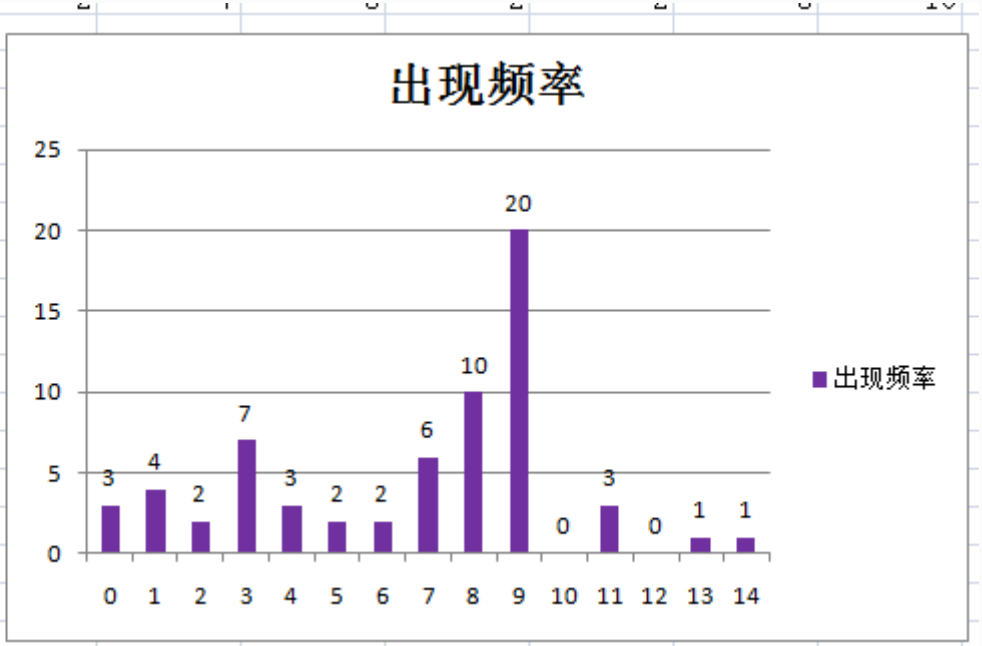
- 什么是直方图
- 直方图均衡化
- API说明
- 代码演示

# 什么是直方图(Histogram)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 7 | 9 | 4 |
| 2 | 3 | 4 | 1 | 0 | 0 | 0 | 9 |
| 3 | 3 | 3 | 9 | 1 | 11 | 3 | 3 |
| 8 | 8 | 8 | 9 | 11 | 13 | 8 | 8 |
| 8 | 8 | 8 | 9 | 1 | 6 | 8 | 8 |
| 7 | 7 | 7 | 9 | 4 | 5 | 7 | 7 |
| 9 | 9 | 9 | 9 | 14 | 9 | 9 | 9 |
| 9 | 9 | 9 | 9 | 9 | 11 | 9 | 9 |

假设有图像数据8x8，像素值范围0~14共15个灰度等级，统计得到各个等级出现次数及直方图如右侧所示

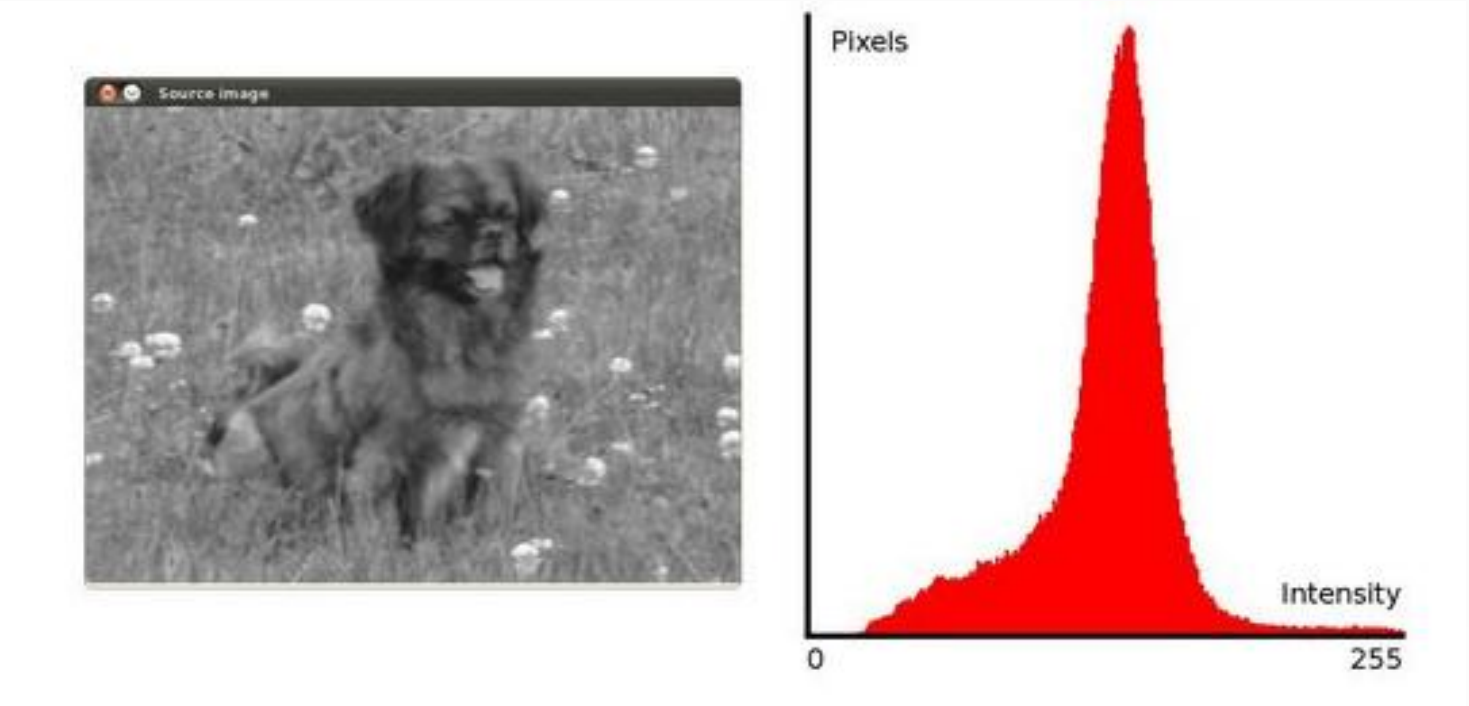
| 像素等级 | 出现频率 |
|---|---|
| 0 | 3 |
| 1 | 4 |
| 2 | 2 |
| 3 | 7 |
| 4 | 3 |
| 5 | 2 |
| 6 | 2 |
| 7 | 6 |
| 8 | 10 |
| 9 | 20 |
| 10 | 0 |
| 11 | 3 |
| 12 | 0 |
| 13 | 1 |
| 14 | 1 |



# 什么是直方图

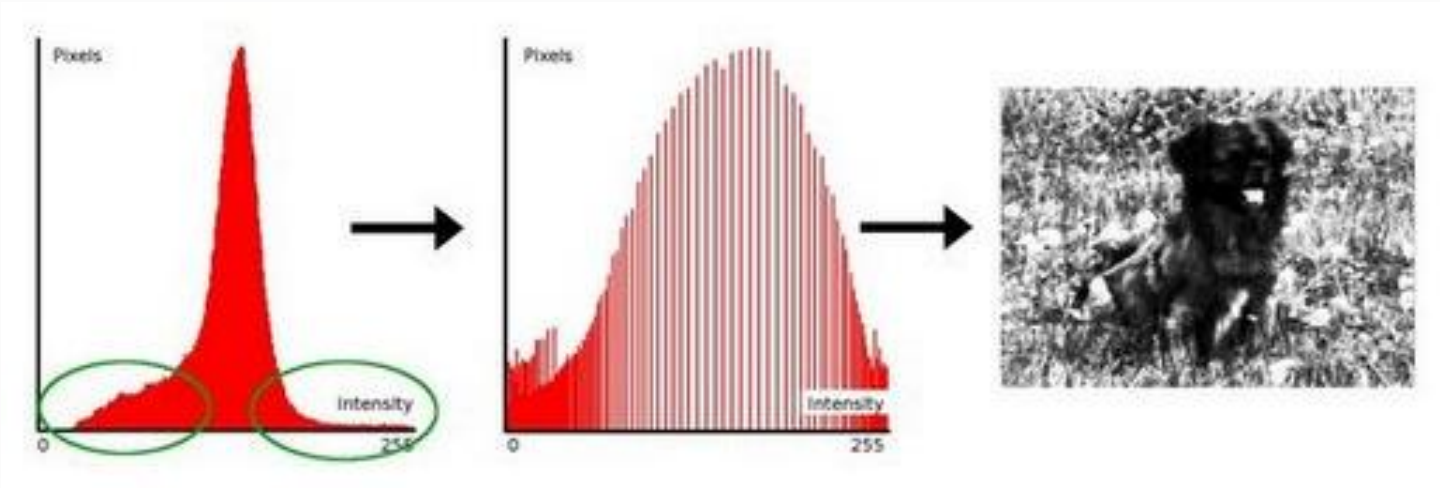
- 图像直方图，是指对整个图像像在灰度范围内的像素值(0~255)统计出现频率次数，据此生成的直方图，称为图像直方图-直方图。直方图反映了图像灰度的分布情况。是图像的统计学特征。

# 图像直方图



# 直方图均衡化

是一种提高图像对比度的方法，拉伸图像灰度值范围。



# 直方图均衡化

- 如何实现，通过上一课中的remap我们知道可以将图像灰度分布从一个分布映射到另外一个分布，然后在得到映射后的像素值即可。

$$H^{'}(i) = \sum_{0 \le j < i} H(j)$$

$$equalized(x,y) = H^{'}(src(x,y))$$

# API说明cv::equalizeHist

```
equalizeHist(
InputArray src,//输入图像，必须是8-bit的单通道图像
OutputArray dst// 输出结果
)
```

# 演示代码

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <math.h>

using namespace cv;
int main(int argc, char** argv) {
    Mat src, dst;
    src = imread("D:/vcprojects/images/cat.jpg");
    if (!src.data) {
        printf("could not load image...\n");
        return -1;
    }

    cvtColor(src, src, CV_BGR2GRAY);
    equalizeHist(src, dst);
    char INPUT_T[] = "input image";
    char OUTPUT_T[] = "result image";
    namedWindow(INPUT_T, CV_WINDOW_AUTOSIZE);
    namedWindow(OUTPUT_T, CV_WINDOW_AUTOSIZE);

    imshow(INPUT_T, src);
    imshow(OUTPUT_T, dst);

    waitKey(0);
    return 0;
}
```

# 直方图计算

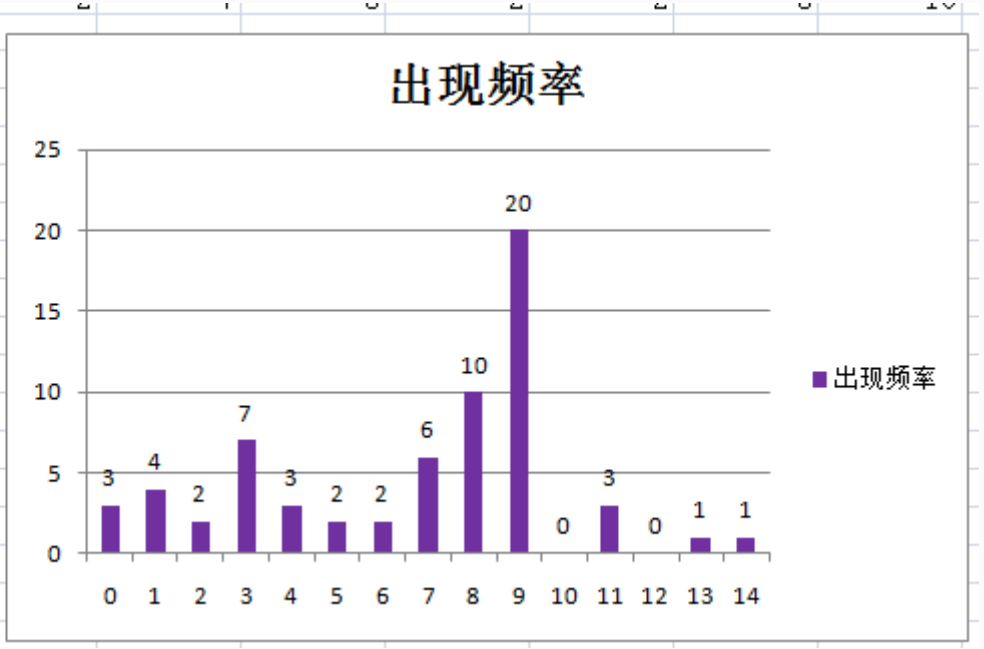
- 直方图概念
- API学习
- 代码演示

# 直方图概念

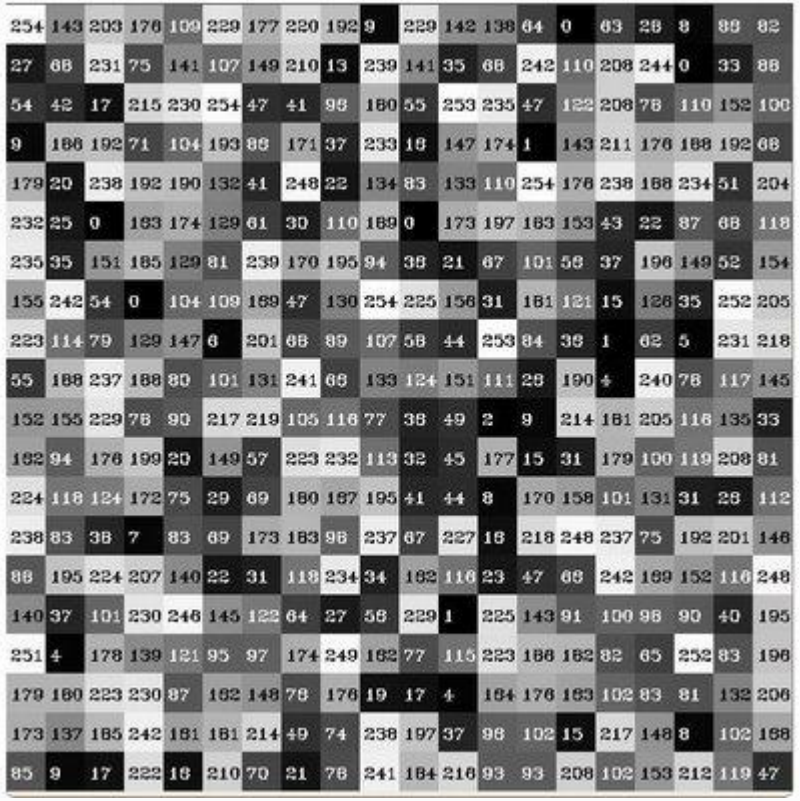
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 5 | 6 | 7 | 9 | 4 |
| 2 | 3 | 4 | 1 | 0 | 0 | 0 | 9 |
| 3 | 3 | 3 | 9 | 1 | 11 | 3 | 3 |
| 8 | 8 | 8 | 9 | 11 | 13 | 8 | 8 |
| 8 | 8 | 8 | 9 | 1 | 6 | 8 | 8 |
| 7 | 7 | 7 | 9 | 4 | 5 | 7 | 7 |
| 9 | 9 | 9 | 9 | 14 | 9 | 9 | 9 |
| 9 | 9 | 9 | 9 | 9 | 11 | 9 | 9 |

假设有图像数据8x8，像素值范围0~14共15个灰度等级，统计得到各个等级出现次数及直方图如右侧所示，每个紫色的长条叫BIN

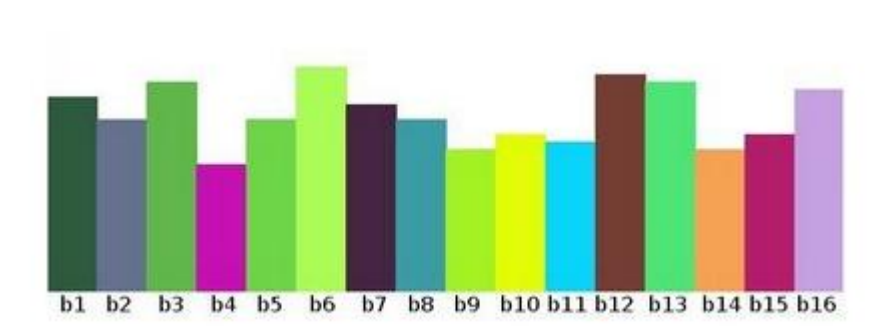
| 像素等级 | 出现频率 |
|---|---|
| 0 | 3 |
| 1 | 4 |
| 2 | 2 |
| 3 | 7 |
| 4 | 3 |
| 5 | 2 |
| 6 | 2 |
| 7 | 6 |
| 8 | 10 |
| 9 | 20 |
| 10 | 0 |
| 11 | 3 |
| 12 | 0 |
| 13 | 1 |
| 14 | 1 |



# 直方图概念



$$[0,255] = [0,15] \cup [16,31] \cup \ldots \cup [240,255]$$
$$range = bin_1 \cup bin_2 \cup \ldots \cup bin_{n=15}$$



# 直方图概念

- 上述直方图概念是基于图像像素值，其实对图像梯度、每个像素的角度、等一切图像的属性值，我们都可以建立直方图。这个才是直方图的概念真正意义，不过是基于图像像素灰度直方图是最常见的。
- 直方图最常见的几个属性：

- dims 表示维度，对灰度图像来说只有一个通道值dims=1
- bins 表示在维度中子区域大小划分，bins=256，划分为256个级别
- range 表示值得范围，灰度值范围为[0~255]之间

# API学习

**split**(// 把多通道图像分为多个单通道图像
const Mat &src, //输入图像
Mat* mvbegin）// 输出的通道图像数组

**calcHist**(
const Mat* images,//输入图像指针
int images,// 图像数目
const int* channels,// 通道数
InputArray mask,// 输入mask，可选，不用
OutputArray hist,//输出的直方图数据
int dims,// 维数
const int* histsize,// 直方图级数
const float* ranges,// 值域范围
bool uniform,// true by default
bool accumulate// false by defaut
)

# 演示代码

```
int histsize = 256;
float range[] = { 0, 256 };
const float* histRanges = { range };
bool uniform = true;
bool accumulate = false;
Mat b_hist, g_hist, r_hist;
calcHist(&bgr_planes[0], 1, 0, Mat(), b_hist, 1, &histsize, &histRanges, uniform, accumulate);
calcHist(&bgr_planes[1], 1, 0, Mat(), g_hist, 1, &histsize, &histRanges, uniform, accumulate);
calcHist(&bgr_planes[2], 1, 0, Mat(), r_hist, 1, &histsize, &histRanges, uniform, accumulate);

// draw histogram image
int hist_w = 512;
int hist_h = 400;
int bin_w =cvRound((double)512 / histsize);
Mat histImage(hist_h, hist_w, CV_8UC3, Scalar(0, 0, 0));

normalize(b_hist, b_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat());
normalize(g_hist, g_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat());
normalize(r_hist, r_hist, 0, histImage.rows, NORM_MINMAX, -1, Mat());

for (size_t i = 1; i < histsize; i++) {
    line(histImage, Point(bin_w*(i - 1), hist_h - cvRound(b_hist.at<float>(i - 1))),
        Point(bin_w*(i - 1), hist_h - cvRound(b_hist.at<float>(i))), Scalar(255, 0, 0), 2, LINE_AA);

    line(histImage, Point(bin_w*(i - 1), hist_h - cvRound(g_hist.at<float>(i - 1))),
        Point(bin_w*(i - 1), hist_h - cvRound(g_hist.at<float>(i))), Scalar(0, 255, 0), 2, LINE_AA);

    line(histImage, Point(bin_w*(i - 1), hist_h - cvRound(r_hist.at<float>(i - 1))),
        Point(bin_w*(i - 1), hist_h - cvRound(r_hist.at<float>(i))), Scalar(0, 0, 255), 2, LINE_AA);

}

imshow(OUTPUT_T, histImage);
waitKey(0);
return 0;
```

# 直方图比较

- 直方图比较方法
- 相关API
- 代码演示

# 直方图比较方法-概述

对输入的两张图像计算得到直方图H1与H2，归一化到相同的尺度空间
然后可以通过计算H1与H2的之间的距离得到两个直方图的相似程度进
而比较图像本身的相似程度。Opencv提供的比较方法有四种：
-Correlation 相关性比较
-Chi-Square 卡方比较
-Intersection 十字交叉性
-Bhattacharyya distance 巴氏距离

# 直方图比较方法-相关性计算(CV_COMP_CORREL)

$$d(H_1,H_2)=\frac{\sum_I(H_1(I)-\bar{H}_1)(H_2(I)-\bar{H}_2)}{\sqrt{\sum_I(H_1(I)-\bar{H}_1)^2\sum_I(H_2(I)-\bar{H}_2)^2}}$$

其中

$$\bar{H}_k=\frac{1}{N}\sum_J H_k(J)$$

其中N是直方图的BIN个数，$\bar{H}$是均值

# 直方图比较方法-卡方计算(CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I)}$$

H1,H2分别表示两个图像的直方图数据

# 直方图比较方法-十字计算(CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

H1,H2分别表示两个图像的直方图数据

# 直方图比较方法-巴氏距离计算(CV_COMP_BHATTACHARYYA )

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H}_1 \bar{H}_2 N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

$\bar{H}$是均值

H1,H2分别表示两个图像的直方图数据

# 相关API

- 首先把图像从RGB色彩空间转换到HSV色彩空间 cvtColor
- 计算图像的直方图，然后归一化到[0~1]之间calcHist 和normalize;
- 使用上述四种比较方法之一进行比较compareHist

# 相关API cv::compareHist

```
compareHist(
InputArray h1, // 直方图数据，下同
InputArray H2,
int method// 比较方法，上述四种方法之一
)
```

# 演示代码

加载图像数据

```
Mat base, test1, test2;
base = imread("D:/vcprojects/images/lena.jpg");
if (!base.data) {
    printf("could not load image...\n");
    return -1;
}
test1 = imread("D:/vcprojects/images/lena.png");
test2 = imread("D:/vcprojects/images/lenanoise.png");
```

从RGB空间转换到HSV空间

```
cvtColor(base, base, CV_BGR2HSV);
cvtColor(test1, test1, CV_BGR2HSV);
cvtColor(test2, test2, CV_BGR2HSV);
```

计算直方图并归一化

```
int h_bins = 50; int s_bins = 60;
int histSize[] = { h_bins, s_bins };
// hue varies from 0 to 179, saturation from 0 to 255
float h_ranges[] = { 0, 180 };
float s_ranges[] = { 0, 256 };
const float* ranges[] = { h_ranges, s_ranges };
// Use the o-th and 1-st channels
int channels[] = { 0, 1 };
MatND hist_base;
MatND hist_test1;
MatND hist_test2;

calcHist(&base, 1,  channels, Mat(), hist_base, 2, histSize, ranges, true, false);
normalize(hist_base, hist_base, 0, 1, NORM_MINMAX, -1, Mat());

calcHist(&test1, 1, channels, Mat(), hist_test1, 2, histSize, ranges, true, false);
normalize(hist_test1, hist_test1, 0, 1, NORM_MINMAX, -1, Mat());

calcHist(&test2, 1, channels, Mat(), hist_test2, 2, histSize, ranges, true, false);
normalize(hist_test2, hist_test2, 0, 1, NORM_MINMAX, -1, Mat());
```

比较直方图，并返回值

```
double basebase = compareHist(hist_base, hist_base, CV_COMP_CORREL);
double basetest1 = compareHist(hist_base, hist_test1, CV_COMP_CORREL);
double basetest2 = compareHist(hist_base, hist_test2, CV_COMP_CORREL);
double tes1test2 = compareHist(hist_test1, hist_test2, CV_COMP_CORREL);
printf("test1 compare with test2 correlation value :%f", tes1test2);
```

# 直方图反向投影(Back Projection)

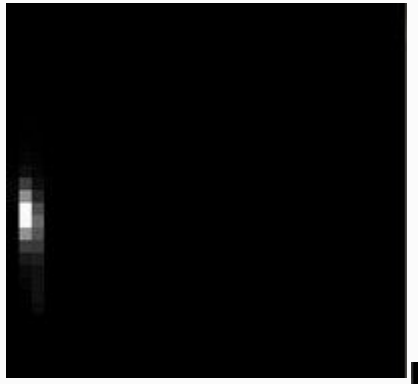
- 反向投影
- 实现步骤与相关API
- 代码演示

# 反向投影

- 反向投影是反映直方图模型在目标图像中的分布情况
- 简单点说就是用直方图模型去目标图像中寻找是否有相似的对象。通常用HSV色彩空间的HS两个通道直方图模型

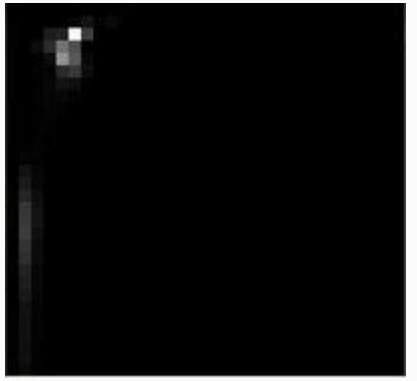
# 反向投影

直方图模型建立



反向投影结果

待检测图像



# 反向投影 – 步骤

1.建立直方图模型
2.计算待测图像直方图并映射到模型中
3.从模型反向计算生成图像

# 实现步骤与相关API

- 加载图片imread
- 将图像从RGB色彩空间转换到HSV色彩空间cvtColor
- 计算直方图和归一化calcHist与normalize
- Mat与MatND其中Mat表示二维数组，MatND表示三维或者多维数据，此处均可以用Mat表示。
- 计算反向投影图像 - calcBackProject

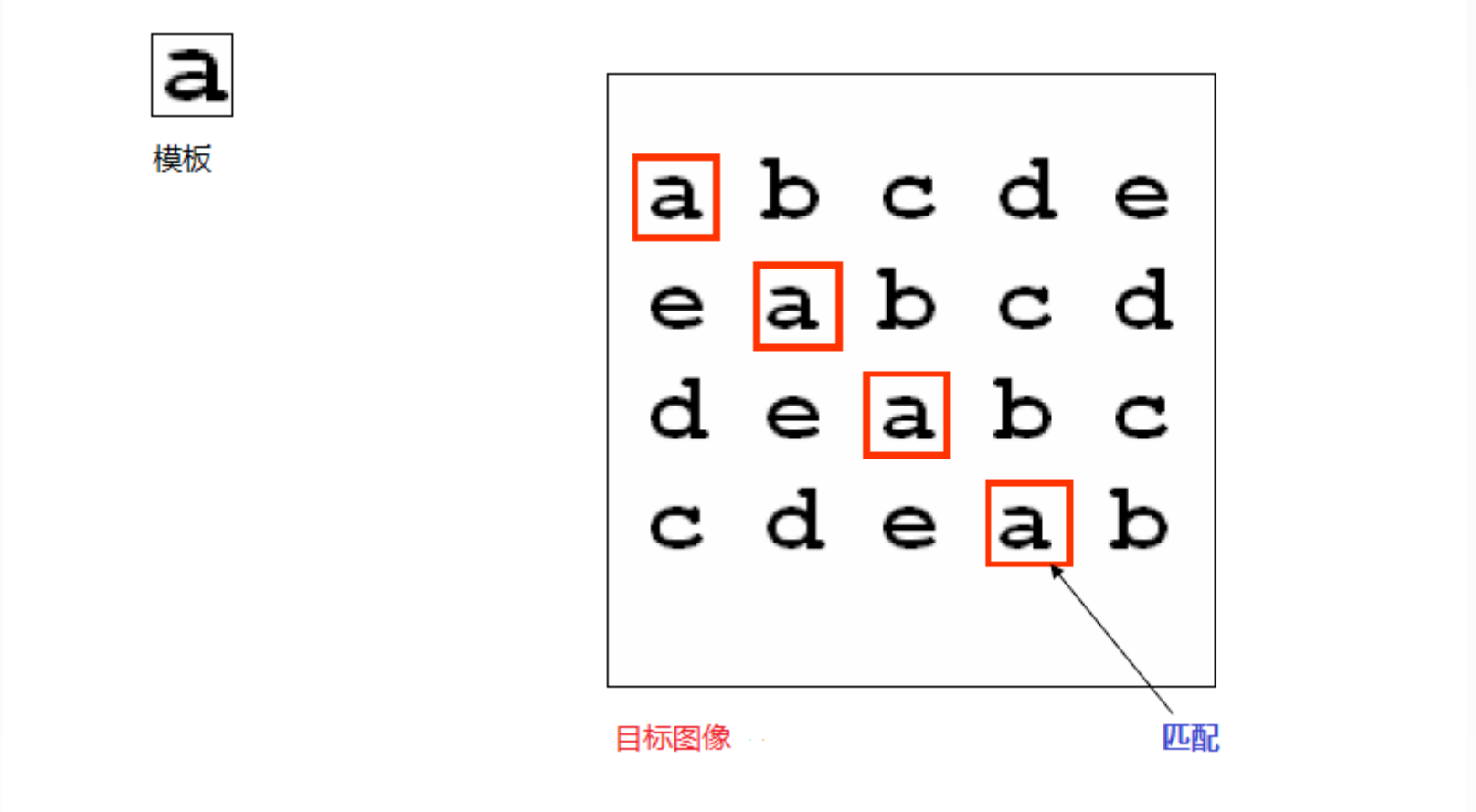
# 演示代码

```
void Hist_And_Backprojection(int, void*) {
    Mat hist;
    int histSize = MAX(bins, 2); // 最少两个梯度
    float hue_range[] = { 0, 180 };
    const float* ranges = { hue_range };
    calcHist(&hue, 1, 0, Mat(), hist, 1, &histSize, &ranges, true, false);
    normalize(hist, hist, 0, 255, NORM_MINMAX, -1, Mat());

    Mat backproj;
    calcBackProject(&hue, 1, 0, hist, backproj, &ranges, 1, true);

    imshow("BackProj", backproj);

    int w = 400; int h = 400;
    int bin_w = cvRound((double)w / histSize);
    Mat histImg = Mat::zeros(w, h, CV_8UC3);
    for (size_t i = 0; i < bins; i++) {
        rectangle(histImg, Point(i*bin_w, h),
            Point((i + 1)*bin_w, h - cvRound(hist.at<float>(i)*h / 255.0)),
            Scalar(0, 0, 255), 2, LINE_AA);
    }
    imshow("Histogram", histImg);

}
```

# 模板匹配(Template Match)

- 模板匹配介绍
- 相关API介绍
- 代码演示

# 模板匹配介绍



# 模板匹配介绍

- 模板匹配就是在整个图像区域发现与给定子图像匹配的小块区域。
- 所以模板匹配首先需要一个模板图像T（给定的子图像）
- 另外需要一个待检测的图像-源图像S
- 工作方法，在带检测图像上，从左到右，从上向下计算模板图像与重叠子图像的匹配度，匹配程度越大，两者相同的可能性越大。

# 模板匹配介绍 – 匹配算法介绍

OpenCV中提供了六种常见的匹配算法如下：

1. 计算平方不同 $$R(x,y)=\sum_{x',y'}(T(x',y')-I(x+x',y+y'))^2$$

2. 计算相关性 $$R(x,y)=\sum_{x',y'}(T(x',y')\cdot I(x+x',y+y'))$$

3. 计算相关系数

$$R(x,y)=\sum_{x',y'}(T'(x',y')\cdot I'(x+x',y+y'))$$

$$T'(x',y')=T(x',y')-1/(w\cdot h)\cdot\sum_{x'',y''}T(x'',y'')$$
$$I'(x+x',y+y')=I(x+x',y+y')-1/(w\cdot h)\cdot\sum_{x'',y''}I(x+x'',y+y'')$$

# 模板匹配介绍 – 匹配算法介绍

计算归一化平方不同

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

计算归一化相关性

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

计算归一化相关系数

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

| | |
|---|---|
| TM_SQDIFF | $$R(x,y)=\sum_{x',y'}(T(x',y')-I(x+x',y+y'))^2$$ |
| TM_SQDIFF_NORMED | $$R(x,y)=\frac{\sum_{x',y'}(T(x',y')-I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'}T(x',y')^2\cdot\sum_{x',y'}I(x+x',y+y')^2}}$$ |
| TM_CCORR | $$R(x,y)=\sum_{x',y'}(T(x',y')\cdot I(x+x',y+y'))$$ |
| TM_CCORR_NORMED | $$R(x,y)=\frac{\sum_{x',y'}(T(x',y')\cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'}T(x',y')^2\cdot\sum_{x',y'}I(x+x',y+y')^2}}$$ |
| TM_CCOEFF | $$R(x,y)=\sum_{x',y'}(T'(x',y')\cdot I'(x+x',y+y'))$$ where $$T'(x',y')=T(x',y')-1/(w\cdot h)\cdot\sum_{x'',y''}T(x'',y'')$$ $$I'(x+x',y+y')=I(x+x',y+y')-1/(w\cdot h)\cdot\sum_{x'',y''}I(x+x'',y+y'')$$ |
| TM_CCOEFF_NORMED | $$R(x,y)=\frac{\sum_{x',y'}(T'(x',y')\cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'}T'(x',y')^2\cdot\sum_{x',y'}I'(x+x',y+y')^2}}$$ |

# 相关API介绍cv::matchTemplate

matchTemplate(

InputArray image,// 源图像，必须是8-bit或者32-bit浮点数图像

InputArray templ,// 模板图像，类型与输入图像一致

OutputArray result,**// 输出结果，必须是单通道32位浮点数，假设源图像WxH,模板图像wxh,**
**则结果必须为W-w+1, H-h+1的大小。**
int method,//使用的匹配方法

InputArray mask=noArray()//(optional)
)

# 相关API介绍cv::matchTemplate

```
enum    cv::TemplateMatchModes {
          cv::TM_SQDIFF = 0,
          cv::TM_SQDIFF_NORMED = 1,
          cv::TM_CCORR = 2,
          cv::TM_CCORR_NORMED = 3,
          cv::TM_CCOEFF = 4,
          cv::TM_CCOEFF_NORMED = 5
        }
```

# 演示代码

```
void Match_Demo(int, void*) {
    Mat img_display;
    src.copyTo(img_display);
    int result_rows = src.rows - temp.rows + 1;
    int result_cols = src.cols - temp.cols + 1;
    dst.create(Size(result_cols, result_rows), CV_32FC1);

    matchTemplate(src, temp, dst, match_method);
    normalize(dst, dst, 0, 1, NORM_MINMAX, -1, Mat());

    double minValue, maxValue;
    Point minLoc;
    Point maxLoc;
    Point matchLoc;

    minMaxLoc(dst, &minValue, &maxValue, &minLoc, &maxLoc, Mat());
    if (match_method == TM_SQDIFF || match_method == TM_SQDIFF_NORMED) {
        matchLoc = minLoc;
    }
    else {
        matchLoc = maxLoc;
    }

    rectangle(img_display, matchLoc, Point(matchLoc.x + temp.cols, matchLoc.y + temp.rows), Scalar::all(0), 2, LINE_AA);
    rectangle(dst, matchLoc, Point(matchLoc.x + temp.cols, matchLoc.y + temp.rows), Scalar::all(0), 2, LINE_AA);

    imshow(OUTPUT_T, dst);
    imshow(match_t, img_display);
    return;
}
```

# 轮廓发现(find contour in your image)

- 轮廓发现(find contour)
- 代码演示


#1
#2

# 轮廓发现(find contour)

- **轮廓发现**是基于图像边缘提取的基础寻找对象轮廓的方法。所以边缘提取的阈值选定会影响最终轮廓发现结果
- **API介绍**
  - findContours发现轮廓
  - drawContours绘制轮廓

## 轮廓发现(find contour)

在二值图像上发现轮廓使用API cv::findContours(
InputOutputArray binImg, // 输入图像，非0的像素被看成1,0的像素值保持不变，8-bit
OutputArrayOfArrays contours,// 全部发现的轮廓对象
OutputArray, hierachy// 图该的拓扑结构，可选，该轮廓发现算法正是基于图像拓扑结构实现。
int mode, // 轮廓返回的模式
int method,// 发现方法
Point offset=Point()// 轮廓像素的位移，默认（0, 0）没有位移
)

## 轮廓绘制(draw contour)

在二值图像上发现轮廓使用API cv::findContours之后对发现的轮廓数据进行绘制显示

```
drawContours(
InputOutputArray  binImg, // 输出图像
 OutputArrayOfArrays  contours,//  全部发现的轮廓对象
Int contourIdx// 轮廓索引号
const Scalar & color,// 绘制时候颜色
int  thickness,// 绘制线宽
int  lineType ,// 线的类型LINE_8
InputArray hierarchy,// 拓扑结构图
int maxlevel,// 最大层数，  0只绘制当前的，1表示绘制绘制当前及其内嵌的轮廓
Point offset=Point()// 轮廓位移，可选
```

# 演示代码

- 输入图像转为灰度图像cvtColor
- 使用Canny进行边缘提取，得到二值图像
- 使用findContours寻找轮廓
- 使用drawContours绘制轮廓

# 演示代码

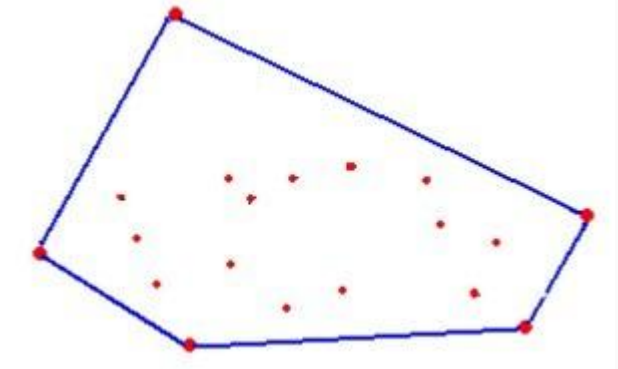
```
void Demo_Contours(int, void*) {
    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;
    Canny(src, dst, threshold_value, threshold_value * 2, 3, false);
    findContours(dst, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));

    Mat drawImg = Mat::zeros(dst.size(), CV_8UC3);
    for (size_t i = 0; i < contours.size(); i++) {
        Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
        drawContours(drawImg, contours, i, color, 2, LINE_8, hierarchy, 0, Point(0, 0));
    }
    imshow(output_win, drawImg);
}
```

# 凸包-Convex Hull

- 概念介绍
- API说明
- 代码演示

# 概念介绍

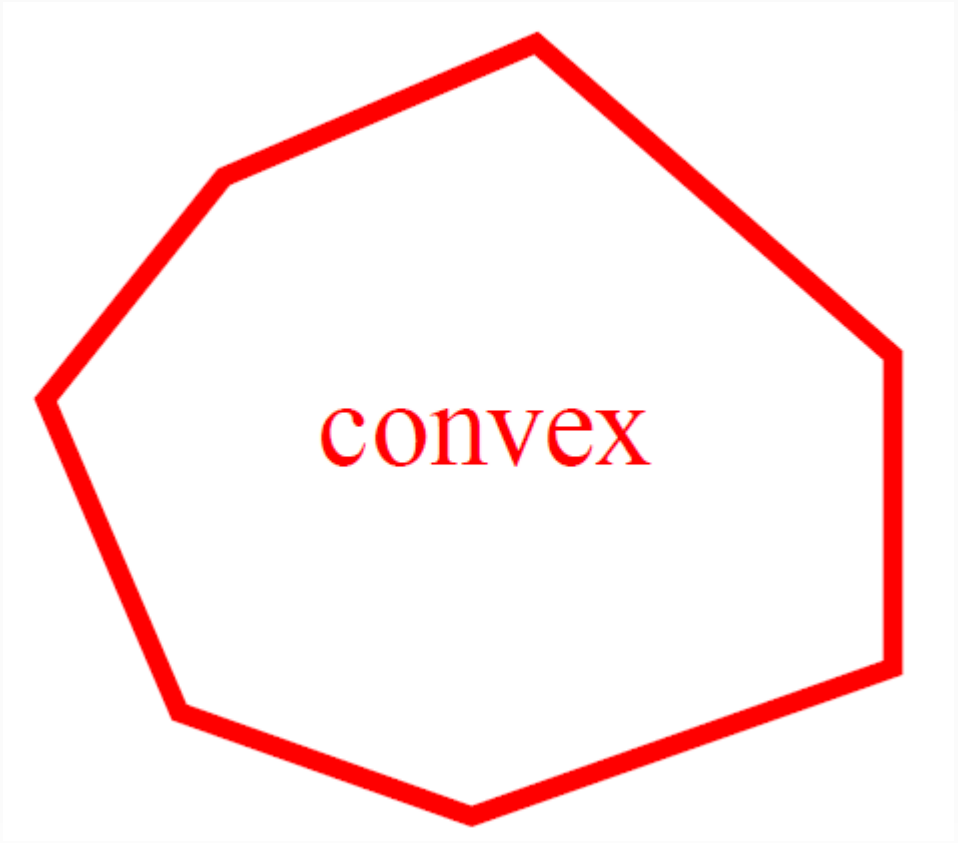
- **什么是凸包(Convex Hull)，在一个多变形边缘或者内部任意两个点的连线都包含在多边形边界或者内部。**

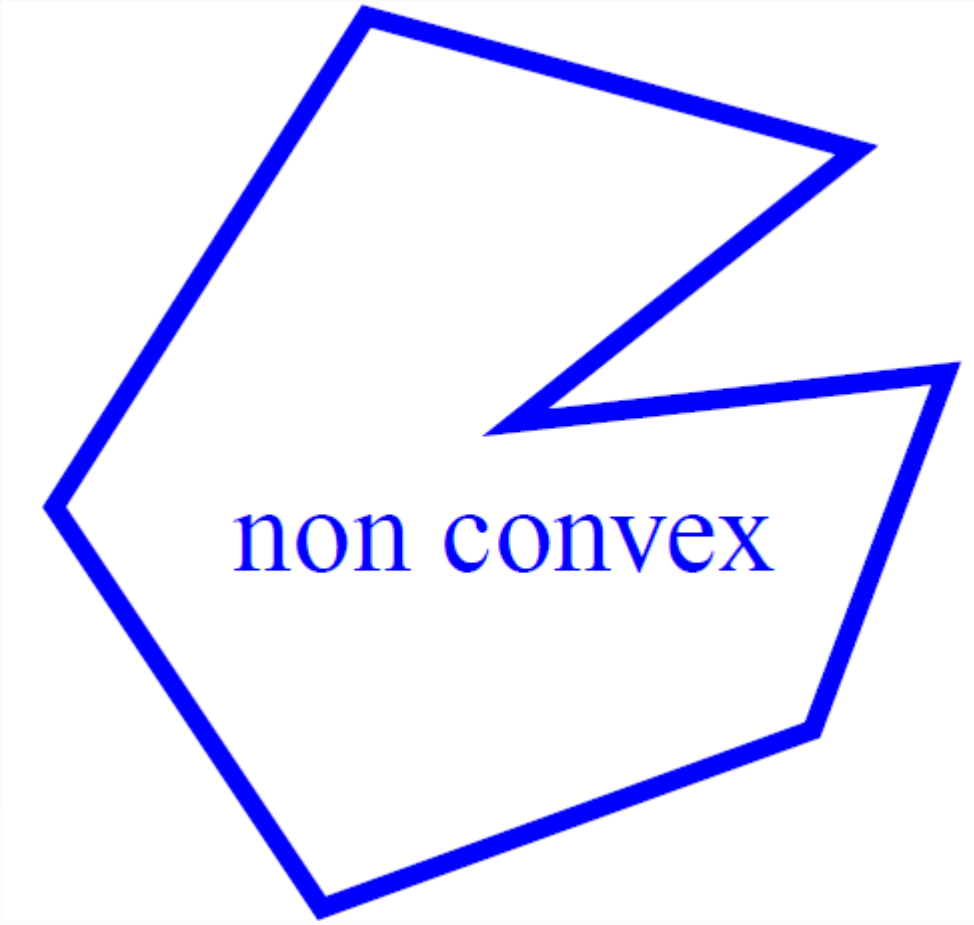


**正式定义：**
包含点集合S中所有点的最小凸多边形称为凸包

- **检测算法**

**- Graham扫描法**


convex

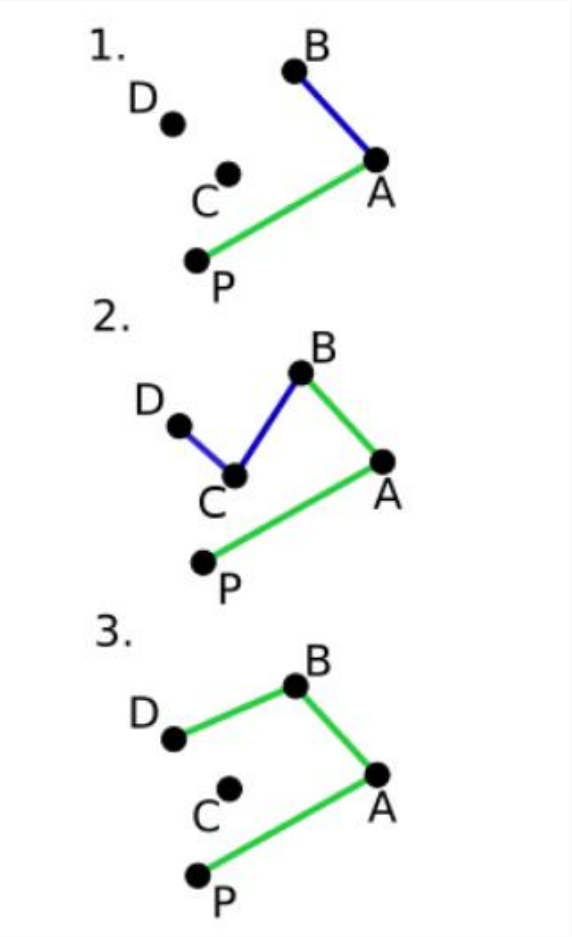

non convex

# 概念介绍-Graham扫描算法

- 首先选择Y方向最低的点作为起始点p0
- 从p0开始极坐标扫描，依次添加p1....pn（排序顺序是根据极坐标的角度大小，逆时针方向）
- 对每个点pi来说，如果添加pi点到凸包中导致一个左转向（逆时针方法）则添加该点到凸包，反之如果导致一个右转向（顺时针方向）删除该点从凸包中

# 概念介绍-Graham扫描算法

- No worry,我们只是需要了解，OpenCV已经实现了凸包发现算法和API提供我们使用。



# API说明cv::convexHull

- convexHull(

InputArray points,// 输入候选点，来自findContours
OutputArray hull,// 凸包
bool clockwise,// default true, 顺时针方向
bool returnPoints）// true 表示返回点个数，如果第二个参数是vector<Point>则自动忽略

# 代码演示

- 首先把图像从RGB转为灰度
- 然后再转为二值图像
- 在通过发现轮廓得到候选点
- 凸包API调用
- 绘制显示。

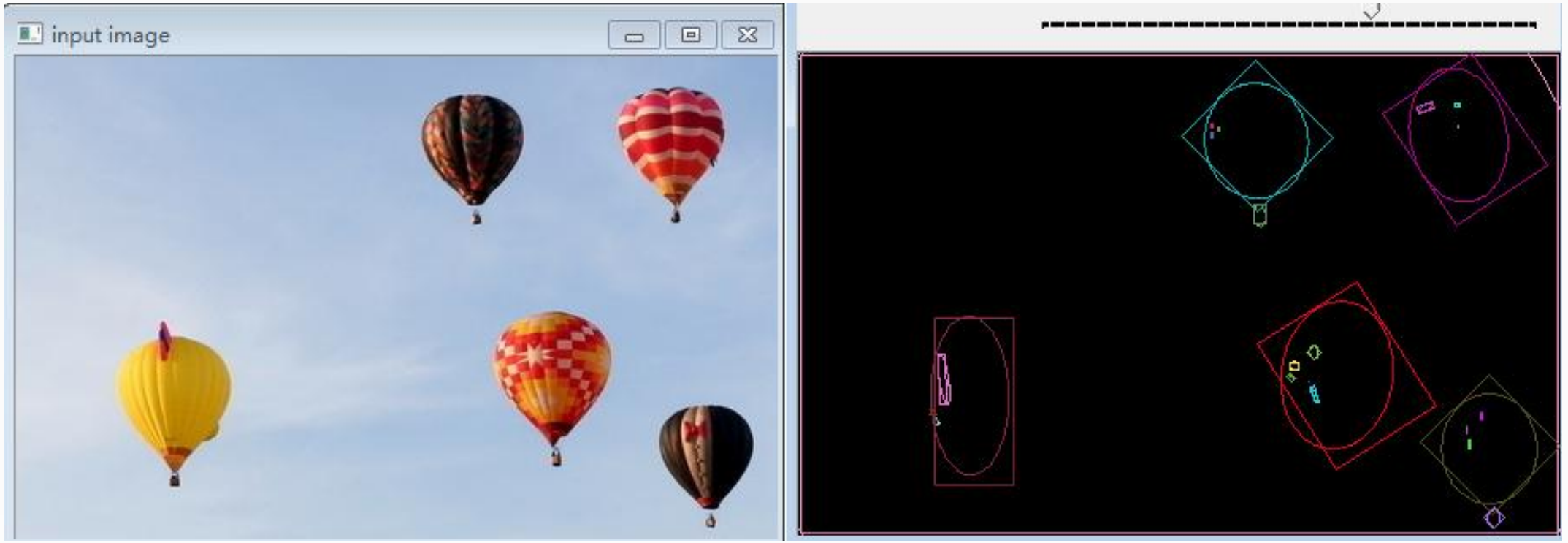
# 演示代码

```
cvtColor(src, src_gray, CV_BGR2GRAY);
blur(src_gray, src_gray, Size(3, 3), Point(-1, -1));
```

```
void Threshold_Callback(int, void*) {
    Mat threshold_output;
    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;

    threshold(src_gray, threshold_output, threshold_value, 255, THRESH_BINARY);
    findContours(threshold_output, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));

    vector<vector<Point>> hull(contours.size());
    for (size_t i = 0; i < contours.size(); i++) {
        convexHull(Mat(contours[i]), hull[i], false);
    }

    Mat dst = Mat::zeros(threshold_output.size(), CV_8UC3);
    for (size_t i = 0; i < contours.size(); i++) {
        Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
        drawContours(dst, hull, i, color, 1, LINE_8, hierarchy, 0, Point(0, 0));
        drawContours(dst, contours, i, color, 1, LINE_8, hierarchy, 0, Point(0, 0));
    }
    imshow(output_win, dst);
}
```

# 轮廓周围绘制矩形框和圆形框

- API介绍
- 代码演示


input image

# 轮廓周围绘制矩形 -API

- approxPolyDP(InputArray curve, OutputArray approxCurve, double epsilon, bool closed)

基于RDP算法实现,目的是减少多边形轮廓点数

```
void cv::approxPolyDP ( InputArray  curve,
                        OutputArray approxCurve,
                        double      epsilon,
                        bool        closed
                      )
```

# 轮廓周围绘制矩形-API

- cv::boundingRect(InputArray points)得到轮廓周围最小矩形左上交点坐标和右下角点坐标，绘制一个矩形
- cv::minAreaRect(InputArray  points)得到一个旋转的矩形，返回旋转矩形

# 轮廓周围绘制圆和椭圆-API

- cv::minEnclosingCircle(InputArray points, //得到最小区域圆形
  Point2f& center, // 圆心位置
  float& radius)// 圆的半径
- cv::fitEllipse(InputArray  points)得到最小椭圆

# 演示代码-步骤

- 首先将图像变为二值图像
- 发现轮廓，找到图像轮廓
- 通过相关API在轮廓点上找到最小包含矩形和圆，旋转矩形与椭圆。
- 绘制它们。

# 演示代码

```
Mat threshold_output;
vector<vector<Point>> contours;
vector<Vec4i> hierarchy;

threshold(gray_src, threshold_output, threshold_v, 255, THRESH_BINARY);
findContours(threshold_output, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));
```

```
// 初始化数组大小
vector<vector<Point>> contours_poly(contours.size());
vector<Rect> boundRects(contours.size());
vector<Point2f> centers(contours.size());
vector<float> radius(contours.size());

vector<RotatedRect> minRects(contours.size());
vector<RotatedRect> minEllipses(contours.size());

// 得到最小矩形与圆形的坐标信息
for (size_t i = 0; i < contours.size(); i++) {
    // 从这些点得到闭合曲线， 3表示跟原来点的最大距离，true表示是闭合的
    // approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);
    // boundRects[i] = boundingRect(Mat(contours_poly[i]));
    // minEnclosingCircle(Mat(contours_poly[i]), centers[i], radius[i]);
    minRects[i] = minAreaRect(Mat(contours[i]));
    if (contours[i].size() > 5) {
        minEllipses[i] = fitEllipse(Mat(contours[i]));
    }
}
```

```
// 绘制
drawImg = Mat::zeros(threshold_output.size(), CV_8UC3);
for (size_t i = 0; i < contours.size(); i++) {
    Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    drawContours(drawImg, contours_poly, i, color, 1, LINE_8, vector<Vec4i>(), 0, Point(0, 0));

    // draw rectangle and circle
    // Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    // rectangle(drawImg, boundRects[i].tl(), boundRects[i].br(), color, 1, LINE_AA, 0);
    // circle(drawImg, centers[i], radius[i], color, 1, LINE_AA, 0);

    // draw rotate rectangle and ellipse
    ellipse(drawImg, minEllipses[i], color, 1, 8);
    Point2f rect_points[4];
    minRects[i].points(rect_points);
    for (int j = 0; j < 4; j++) {
        line(drawImg, rect_points[j], rect_points[(j + 1)%4], color, 1, 8, 0);
    }
}
```

# 图像矩(Image Moments)

- 矩的概念介绍
- API介绍与使用
- 代码演示

# 矩的概念介绍

- 几何矩

- 几何矩 $M_{ji}=\sum_{x,y}(P(x,y)\cdot x^{j}\cdot y^{i})$其中(i + j)和等于几就叫做几阶距
- 中心距 $mu_{ji}=\sum_{x,y}(P(x,y)\cdot(x-\bar{x})^{j}\cdot(y-\bar{y})^{i})$其中$\bar{x},\bar{y}$表示它的中心质点。
- 中心归一化距 $nu_{ji}=\frac{mu_{ji}}{m_{00}^{(i+j)/2+1}}$.

# 矩的概念介绍

- 图像中心Center(x0, y0)

$$x_0 = \frac{m_{10}}{m_{00}} \quad y_0 = \frac{m_{01}}{m_{00}}$$

# API介绍与使用 – cv::moments 计算生成数据

| spatial moments |
| --- |
| double m00 |
| double m10 |
| double m01 |
| double m20 |
| double m11 |
| double m02 |
| double m30 |
| double m21 |
| double m12 |
| double m03 |

| central moments |
| --- |
| double mu20 |
| double mu11 |
| double mu02 |
| double mu30 |
| double mu21 |
| double mu12 |
| double mu03 |

| central normalized moments |
| --- |
| double nu20 |
| double nu11 |
| double nu02 |
| double nu30 |
| double nu21 |
| double nu12 |
| double nu03 |

# API介绍与使用-计算矩cv::moments

**moments**(
InputArray  array,//输入数据
bool  binaryImage=false // 是否为二值图像
)

**contourArea**(
InputArray  contour,//输入轮廓数据
bool  oriented// 默认false、返回绝对值)

**arcLength**(
InputArray  curve,//输入曲线数据
bool  closed// 是否是封闭曲线)

# 演示代码-步骤

- 提取图像边缘
- 发现轮廓
- 计算每个轮廓对象的矩
- 计算每个对象的中心、弧长、面积

# 演示代码

```
cvtColor(src, gray_src, CV_BGR2GRAY);
GaussianBlur(gray_src, gray_src, Size(3, 3), 0, 0);
```

```
Canny(gray_src, canny_output, threshold_value, threshold_value * 2, 3, false);
findContours(canny_output, contours, hieracrhy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));
```

```
// calculate center of each moments
vector<Point2f> objcenters(contours.size());
for (size_t i = 0; i < contours.size(); i++) {
    objcenters[i] = Point(static_cast<float>(mu[i].m10 / mu[i].m00), static_cast<float>(mu[i].m01 / mu[i].m00));
}

Mat drawImg = Mat::zeros(canny_output.size(), CV_8UC3);
for (size_t i = 0; i < contours.size(); i++) {
    Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    drawContours(drawImg, contours, i, color, 1, 8, hieracrhy);
    circle(drawImg, objcenters[i], 3, color, 2, 8, 0);
}
```

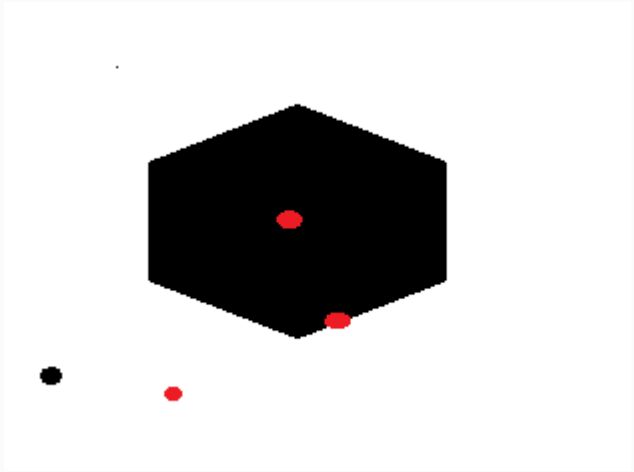
```
// calculate area and arc length
for (size_t i = 0; i < contours.size(); i++) {
    printf("[[Moments %d]] area : %.2f  arclength : %.2f \n", i, contourArea(contours[i], false), arcLength(contours[i], true));
    Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    drawContours(drawImg, contours, i, color, 1, 8, hieracrhy);
    circle(drawImg, objcenters[i], 3, color, 2, 8, 0);
}
```

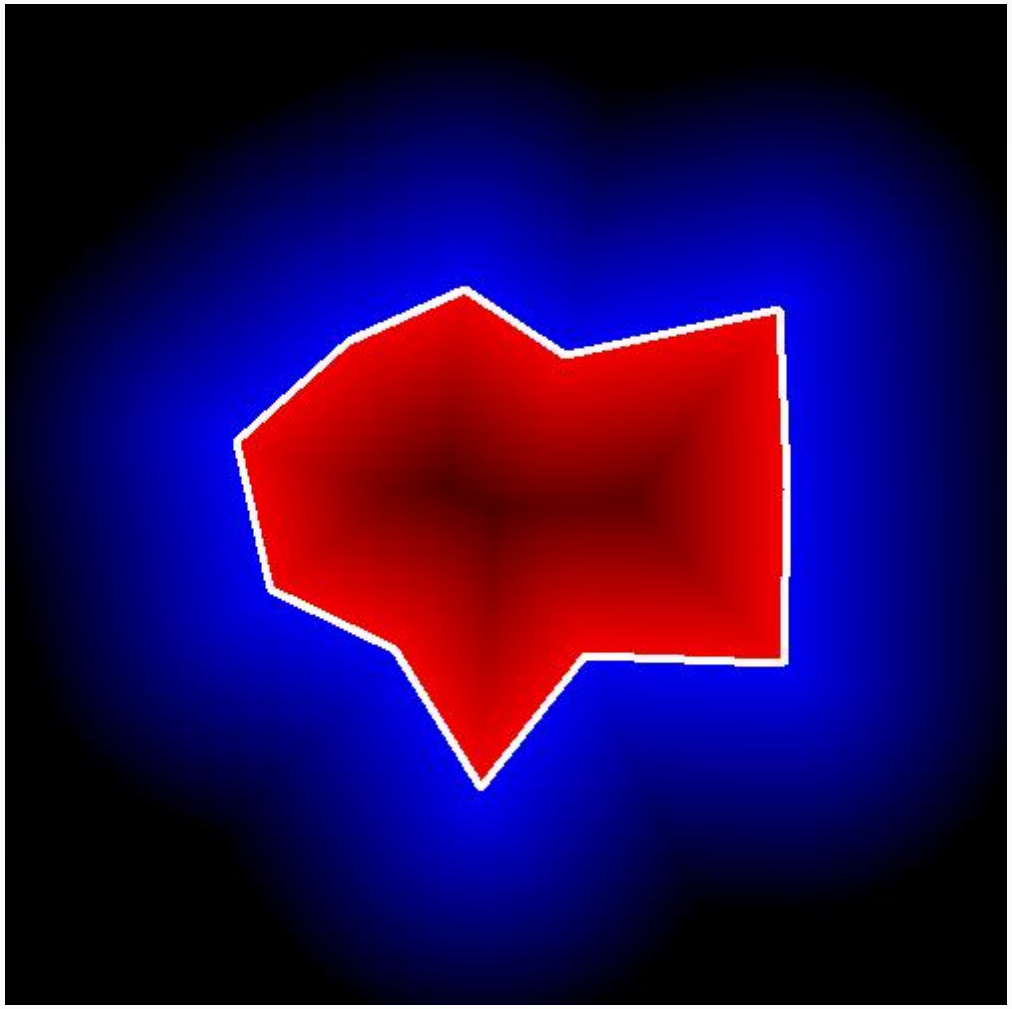
# 点多边形测试

- 概念介绍
- API介绍
- 代码演示

## 概念介绍 - 点多边形测试

- 测试一个点是否在给定的多边形内部，边缘或者外部





# API介绍 cv::pointPolygonTest

**pointPolygonTest**(
InputArray  contour,// 输入的轮廓
Point2f  pt, // 测试点
bool  measureDist // 是否返回距离值，如果是false，1表示在内面，0表示在边界上，-1表示在外部，true返回实际距离
)

返回数据是double类型

# 演示代码-步骤

- 构建一张400x400大小的图片， Mat::Zero(400, 400, CV_8UC1)
- 画上一个六边形的闭合区域line
- 发现轮廓
- 对图像中所有像素点做点 多边形测试，得到距离，归一化后显示。

# 演示代码-细节

```
double minValue, maxValue;
minMaxLoc(raw_dist, &minValue, &maxValue, 0, 0, Mat());
```

- 内部

```
drawImg.at<Vec3b>(row, col)[0] = (uchar)((abs(distance)/maxValue)*255);
```

- 外部

```
drawImg.at<Vec3b>(row, col)[1] = (uchar)(255*(abs(distance)/minValue));
```

- 边缘线

```
drawImg.at<Vec3b>(row, col)[0] = 255;
drawImg.at<Vec3b>(row, col)[1] = saturate_cast<uchar>(minValue);
drawImg.at<Vec3b>(row, col)[2] = saturate_cast<uchar>(minValue);
```

# 演示代码

```
const int r = 100;
Mat src = Mat::zeros(r * 4, r * 4, CV_8UC1);

vector<Point2f> vert(6);
vert[0] = Point(3 * r / 2, static_cast<int>(1.34*r));
vert[1] = Point(1 * r, 2 * r);
vert[2] = Point(3 * r / 2, static_cast<int>(2.866*r));
vert[3] = Point(5 * r / 2, static_cast<int>(2.866*r));
vert[4] = Point(3 * r, 2 * r);
vert[5] = Point(5 * r / 2, static_cast<int>(1.34*r));

for (int i = 0; i < 6; i++) {
    line(src, vert[i], vert[(i + 1) % 6], Scalar(255), 3, 8, 0);
}
```

```
vector<vector<Point>> contours;
vector<Vec4i> hieracrhy;

// full data copy by clone
Mat src_copy = src.clone();
// find contours
findContours(src_copy, contours, hieracrhy, RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0));

Mat raw_dist = Mat(src_copy.size(), CV_32FC1);
for (int row = 0; row < src_copy.rows; row++) {
    for (int col = 0; col < src_copy.cols; col++) {
        raw_dist.at<float>(row, col) = (float)pointPolygonTest(contours[0], Point2f((float)col, (float)row), true);
    }
}
```

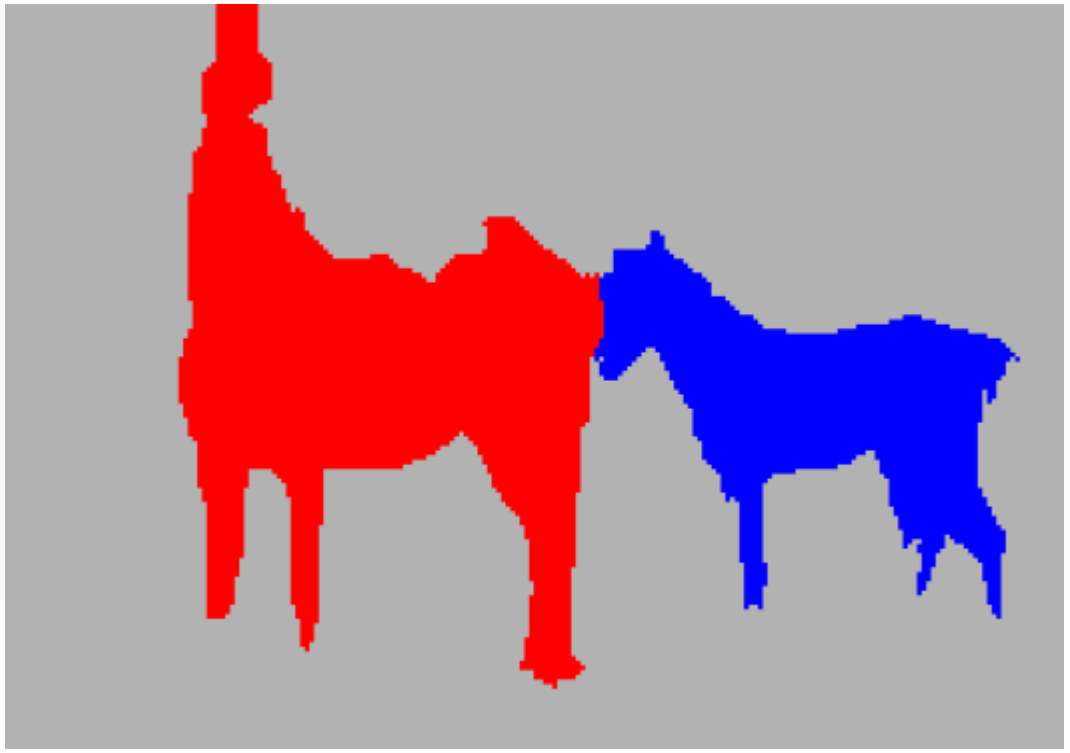
# 演示代码

```
double minValue, maxValue;
minMaxLoc(raw_dist, &minValue, &maxValue, 0, 0, Mat());
minValue = abs(minValue);
maxValue = abs(maxValue);

Mat drawImg = Mat::zeros(raw_dist.size(), CV_8UC3);
for (int row = 0; row < src_copy.rows; row++) {
    for (int col = 0; col < src_copy.cols; col++) {
        float distance = raw_dist.at<float>(row, col);
        if (distance > 0) {
            drawImg.at<Vec3b>(row, col)[0] = (uchar)((abs(distance)/maxValue)*255);
        }
        else if (distance < 0) {
            drawImg.at<Vec3b>(row, col)[1] = (uchar)(255*(abs(distance)/minValue));
        } else {
            drawImg.at<Vec3b>(row, col)[0] = 255;
            drawImg.at<Vec3b>(row, col)[1] = saturate_cast<uchar>(minValue);
            drawImg.at<Vec3b>(row, col)[2] = saturate_cast<uchar>(minValue);
        }
    }
}
```

# 基于距离变换与分水岭的图像分割

- 什么是图像分割
- 距离变换与分水岭介绍
- 相关API
- 代码演示

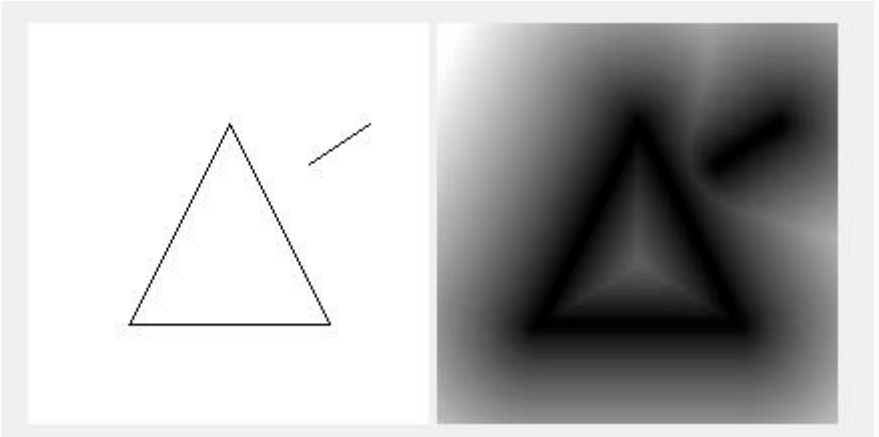
# 什么是图像分割(Image Segmentation)



# 什么是图像分割

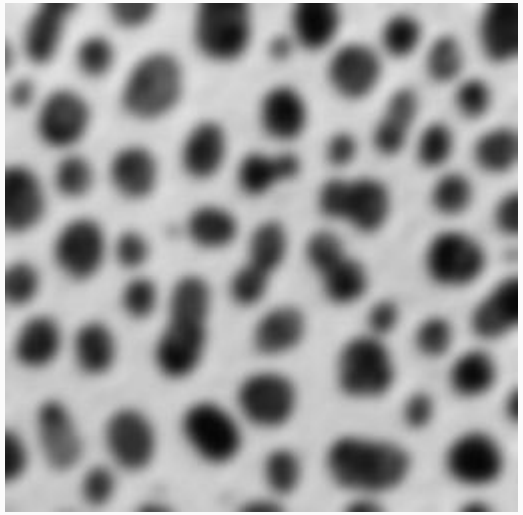
- 图像分割(Image Segmentation)是图像处理最重要的处理手段之一
- 图像分割的目标是将图像中像素根据一定的规则分为若干(N)个cluster集合，每个集合包含一类像素。
- 根据算法分为监督学习方法和无监督学习方法，图像分割的算法多数都是无监督学习方法 - KMeans

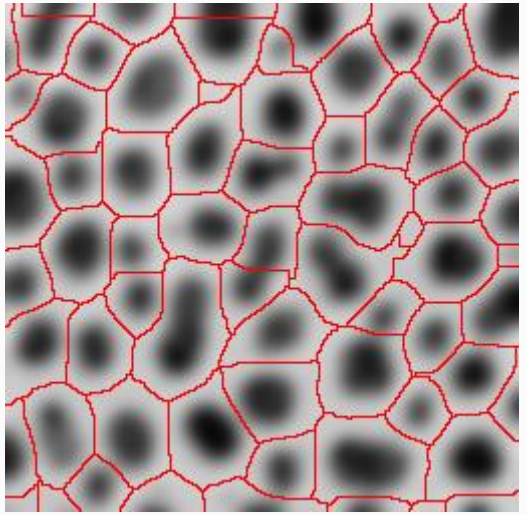
# 距离变换与分水岭介绍



还记得上节课的内容，测试点多边形得到结果跟距离变换相似

# 距离变换与分水岭介绍





# 距离变换与分水岭介绍

- 距离变换常见算法有两种
  - 不断膨胀/ 腐蚀得到
  - 基于倒角距离

- 分水岭变换常见的算法
  - 基于浸泡理论实现

# 相关API

- cv::distanceTransform(InputArray src, OutputArray dst, OutputArray labels, int distanceType, int maskSize, int labelType=DIST_LABEL_CCOMP)

distanceType = DIST_L1/DIST_L2,

maskSize = 3x3,最新的支持5x5，推荐3x3、

labels离散维诺图输出

dst输出8位或者32位的浮点数，单一通道，大小与输入图像一致

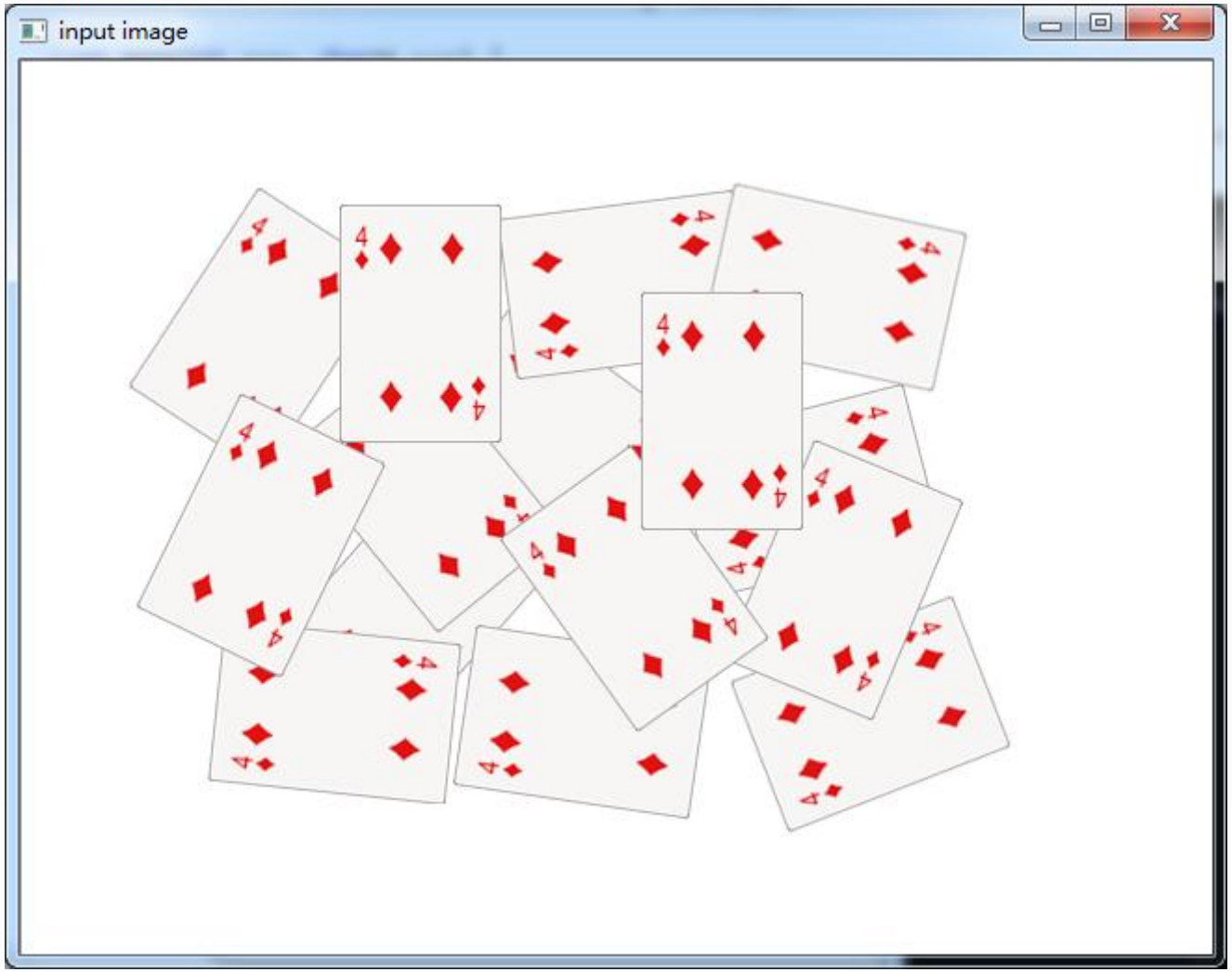
- cv::watershed(InputArray image, InputOutputArray markers)

# 处理流程

1.将白色背景变成黑色-目的是为后面的变换做准备
2. 使用filter2D与拉普拉斯算子实现图像对比度提高，sharp
3. 转为二值图像通过threshold
4. 距离变换
5. 对距离变换结果进行归一化到[0~1]之间
6. 使用阈值，再次二值化，得到标记
7. 腐蚀得到每个Peak - erode
8.发现轮廓 – findContours
9. 绘制轮廓- drawContours
10.分水岭变换 watershed
11. 对每个分割区域着色输出结果

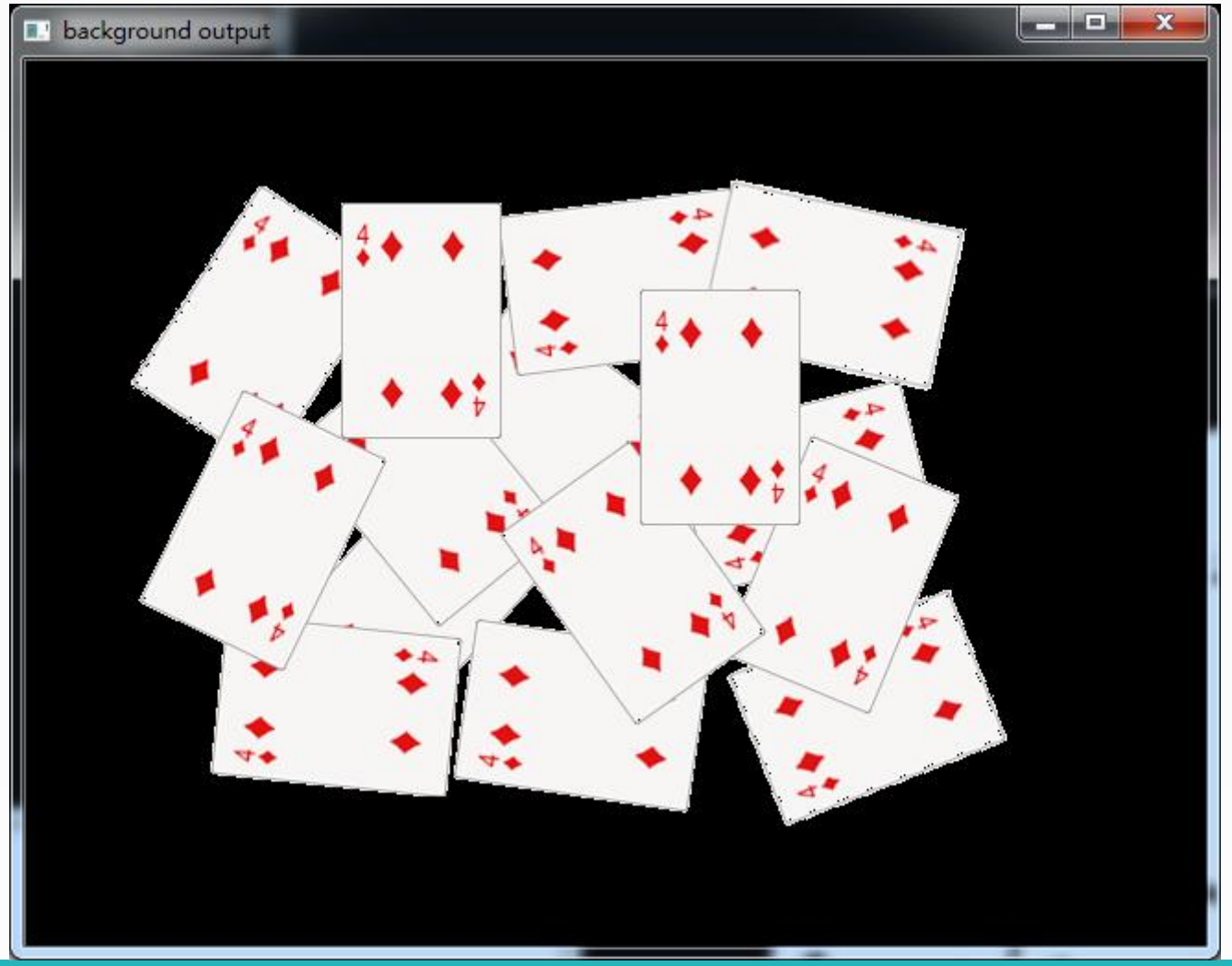
# 演示代码-加载图像

```
char input_win[] = "input image";
char watershed_win[] = "watershed segmentation demo";
Mat src = imread("D:/vcprojects/images/cards.png");
// Mat src = imread("D:/kuaidi.jpg");
if (src.empty()) {
    printf("could not load image...\n");
    return -1;
}
namedWindow(input_win, CV_WINDOW_AUTOSIZE);
//namedWindow(watershed_win, CV_WINDOW_AUTOSIZE);
imshow(input_win, src);
```



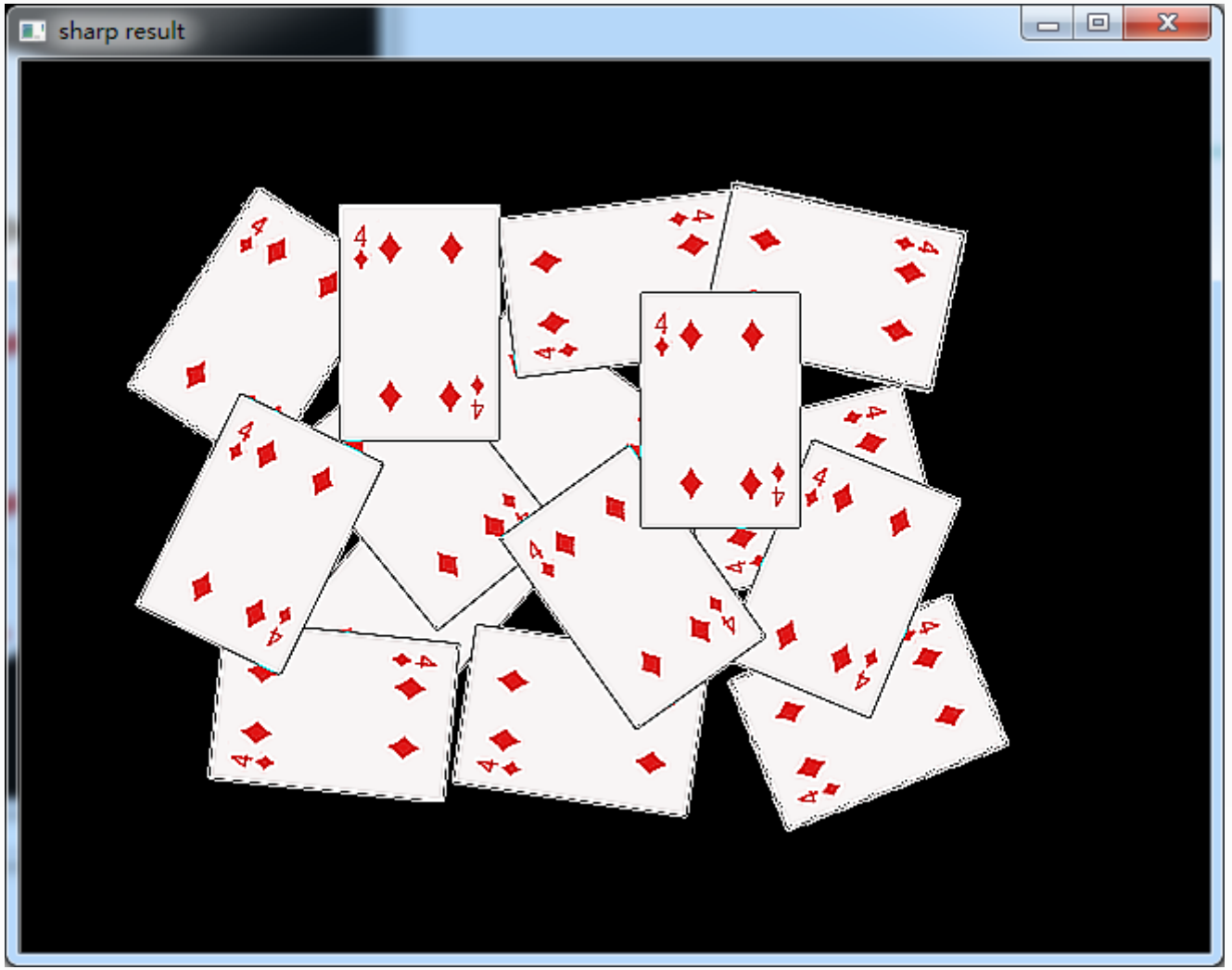
# 演示代码-去背景

```
// change the background
for (int row = 0; row < src.rows; row++) {
    for (int col = 0; col < src.cols; col++) {
        if (src.at<Vec3b>(row, col) == Vec3b(255, 255, 255)) {
            src.at<Vec3b>(row, col)[0] = 0;
            src.at<Vec3b>(row, col)[1] = 0;
            src.at<Vec3b>(row, col)[2] = 0;
        }
    }
}
namedWindow("background output", CV_WINDOW_AUTOSIZE);
imshow("background output", src);
```



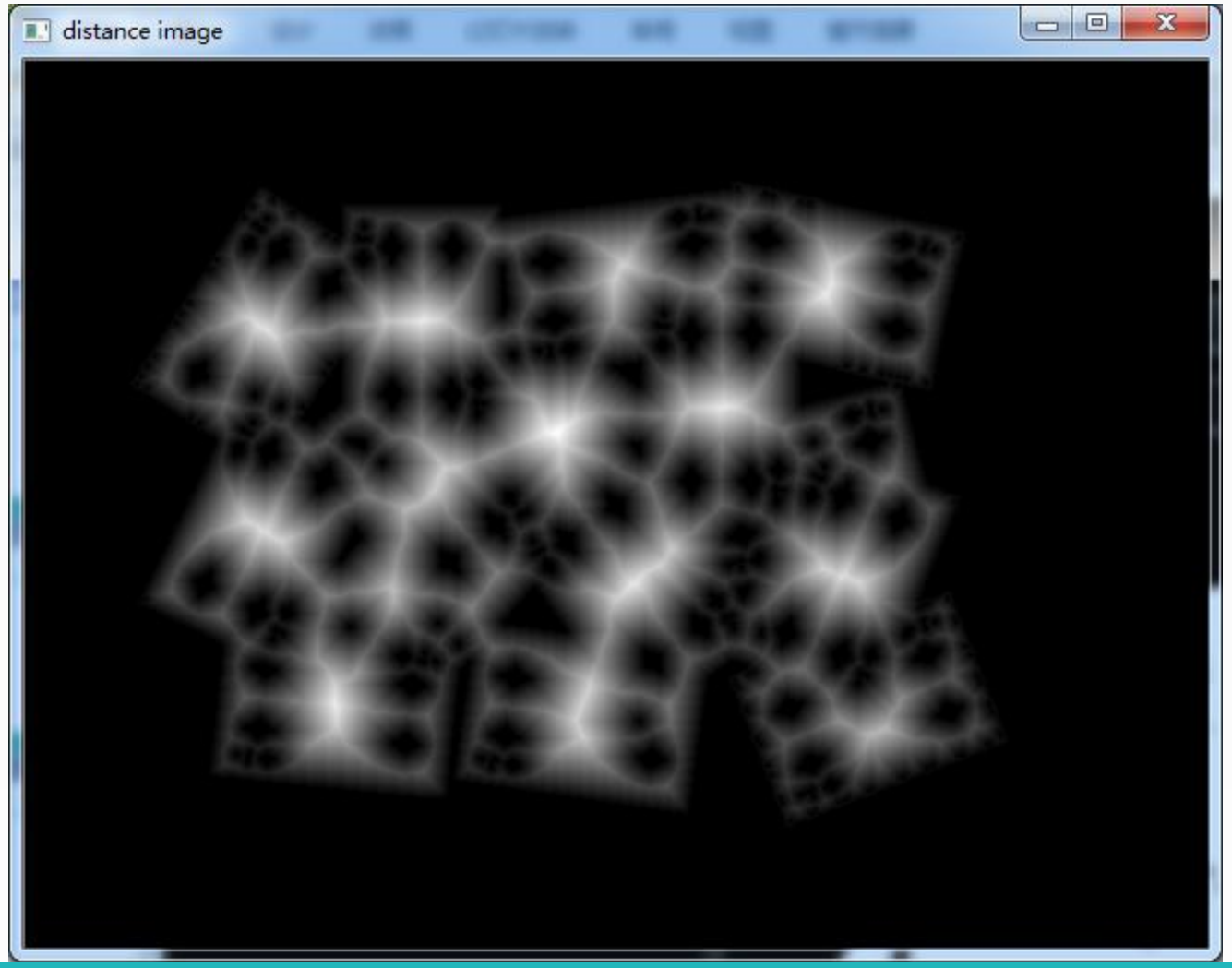
# 演示代码-Sharp

```
// create kernel
Mat kernel = (Mat_<float>(3, 3) << 1, 1, 1,
    1, -8, 1,
    1, 1, 1);

// make it more sharp
Mat imgLaplance;
Mat sharp = src;
filter2D(sharp, imgLaplance, CV_32F, kernel, Point(-1, -1), 0);
src.convertTo(sharp, CV_32F);
Mat imgResult = sharp - imgLaplance;
// 显示
imgResult.convertTo(imgResult, CV_8UC3);
imgLaplance.convertTo(imgLaplance, CV_8UC3);
namedWindow("sharp result", CV_WINDOW_AUTOSIZE);
imshow("sharp result", imgResult);
```



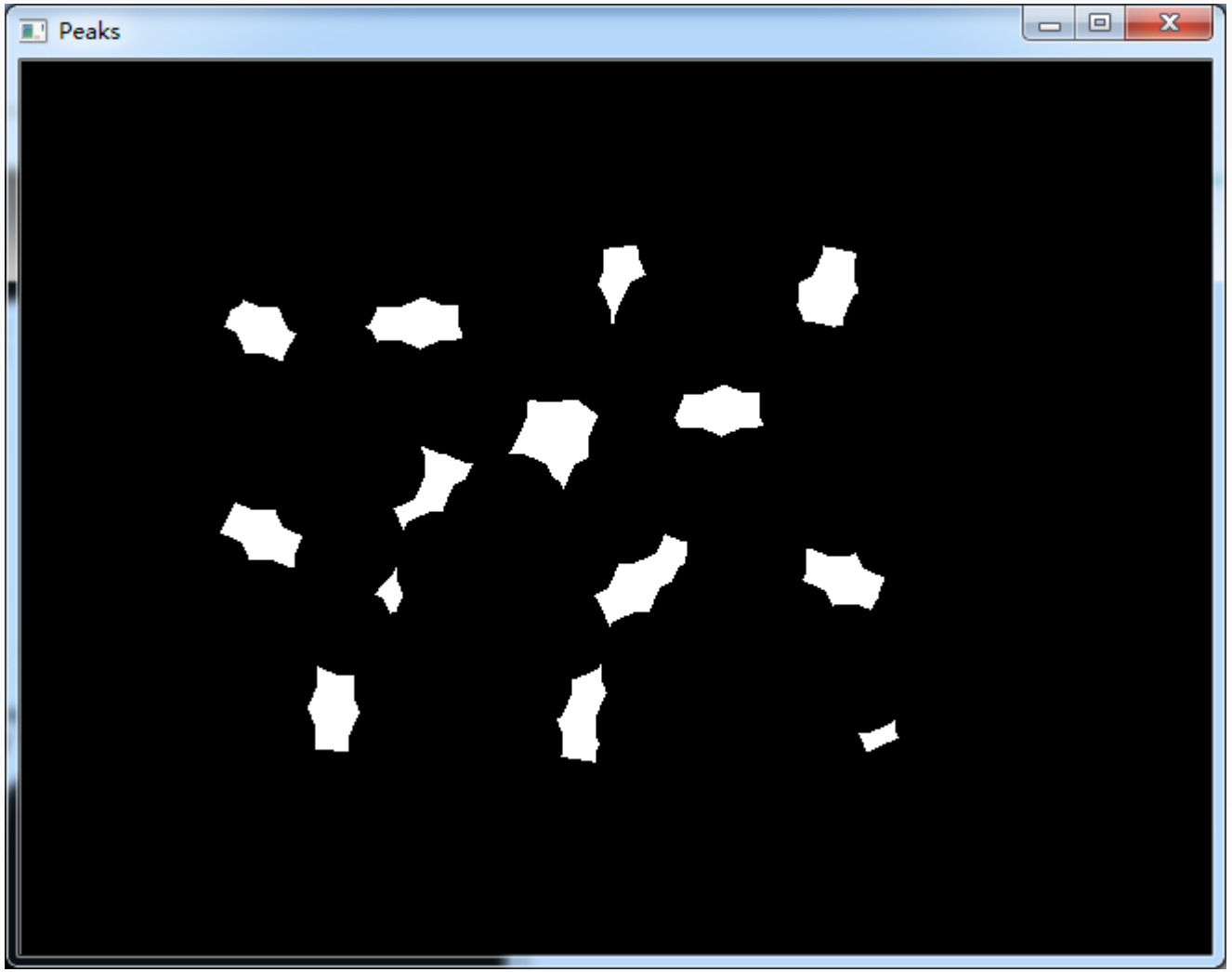
# 演示代码-二值距离变换

```
// 二值图像
Mat binImg;
cvtColor(src, binImg, CV_BGR2GRAY);
threshold(binImg, binImg, 40, 255, CV_THRESH_BINARY | CV_THRESH_OTSU);
imshow("Binary Image", binImg);

// distance transformation
Mat dist;
distanceTransform(binImg, dist, CV_DIST_L2, 3);
normalize(dist, dist, 0, 1., NORM_MINMAX);
imshow("distance image", dist);

// try to get marker
threshold(dist, dist, .4, 1., CV_THRESH_BINARY);
```



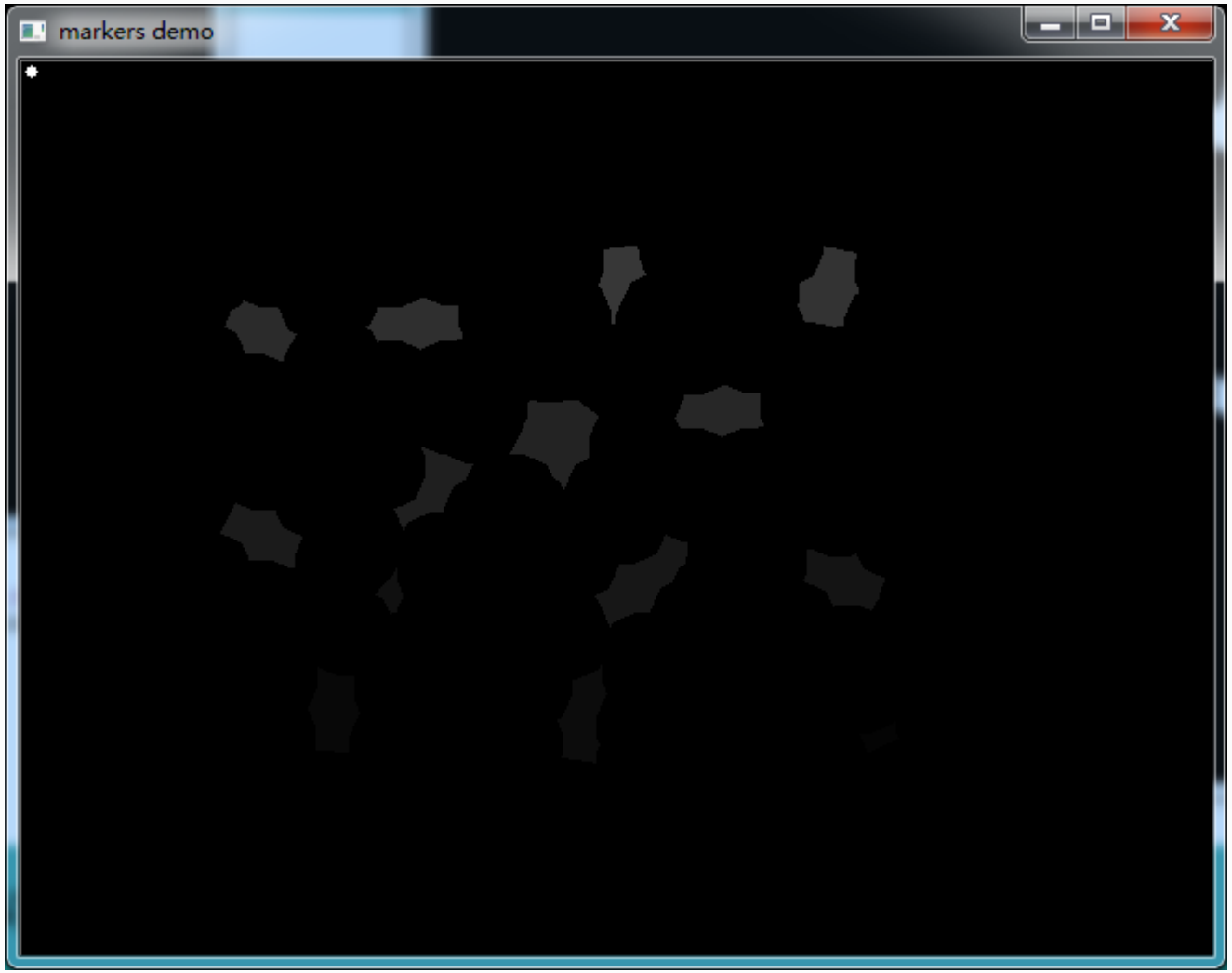
# 演示代码-二值腐蚀

```
// try to get marker
threshold(dist, dist, .4, 1., CV_THRESH_BINARY);

// erode the distance image
Mat kernel1 = Mat::ones(13, 13, CV_8UC1);
erode(dist, dist, kernel1);
imshow("Peaks", dist);
```



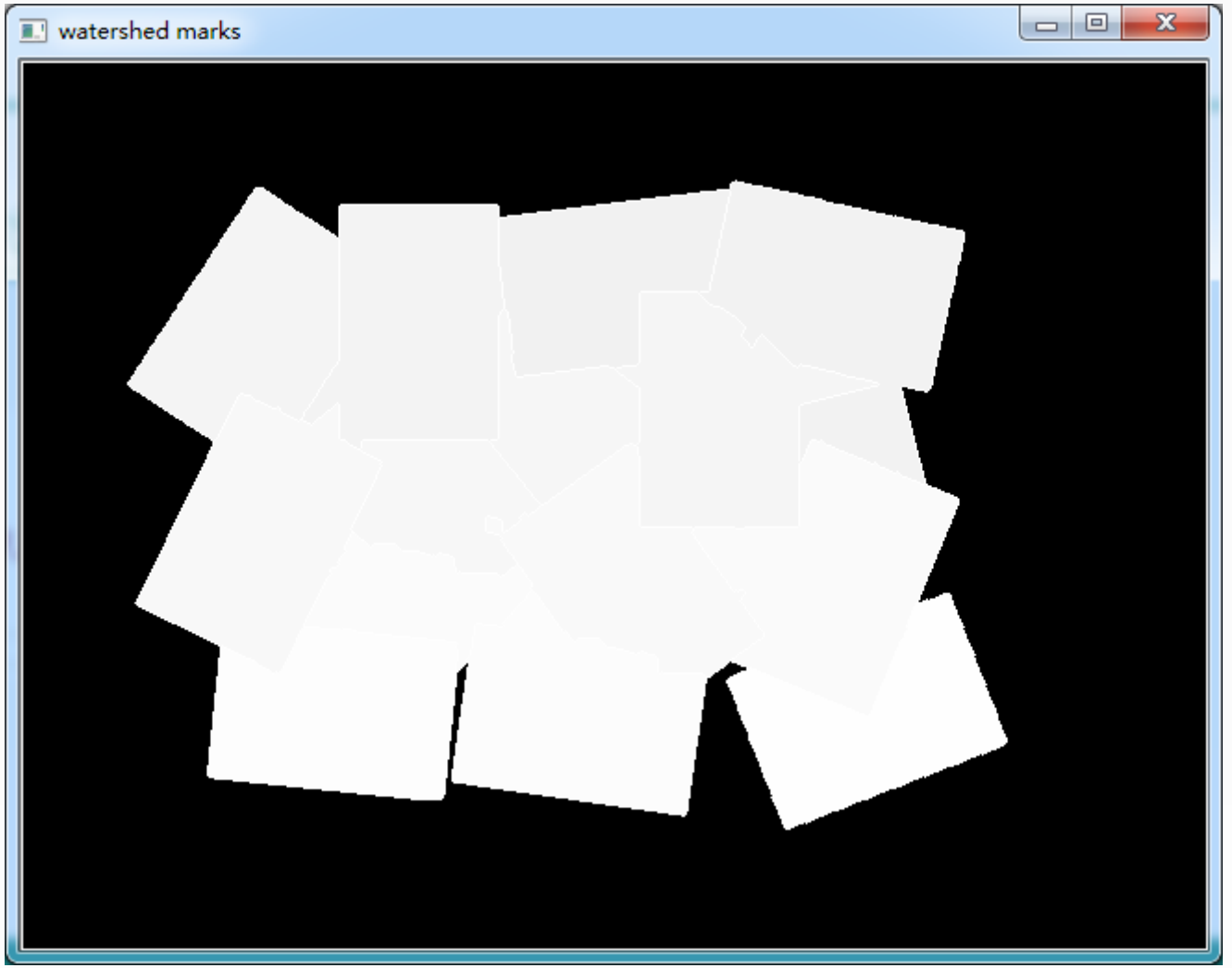
# 演示代码-标记

```
Mat dist_8u;
dist.convertTo(dist_8u, CV_8U);
// find contours
vector<vector<Point>> contours;
findContours(dist_8u, contours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, Point(0, 0));

// create markers image
Mat markers = Mat::zeros(dist.size(), CV_32SC1);
// draw markers
for (size_t i = 0; i < contours.size(); i++) {
    drawContours(markers, contours, static_cast<int>(i), Scalar::all(static_cast<int>(i) + 1),
}
// draw background circle
circle(markers, Point(5, 5), 3, CV_RGB(255, 255, 255), -1);
imshow("markers demo", markers*1000);
```



# 演示代码-分水岭变换

```
// perform watershed transform
watershed(src, markers);

Mat mark = Mat::zeros(markers.size(), CV_8UC1);
markers.convertTo(mark, CV_8UC1);
bitwise_not(mark, mark);
imshow("watershed marks", mark);
```



# 演示代码-着色效果

```
// fill with color and display final result
Mat dst = Mat::zeros(markers.size(), CV_8UC3);
for (int row = 0; row < markers.rows; row++) {
    for (int col = 0; col < markers.cols; col++) {
        int index = markers.at<int>(row, col);
        if (index > 0 && index <= static_cast<int>(contours.size())) {
            dst.at<Vec3b>(row, col) = colors[index-1];
        }
        else {
            dst.at<Vec3b>(row, col) = Vec3b(0, 0, 0);
        }
    }
}
imshow("Final Result", dst);
```