OpenCV 显示与绘制

1. OpenCV 的基础绘制功能

OpenCV 中提供直线、椭圆、矩形、圆以及多边形的绘制功能。
在 OpenCV 的图形绘制中我们会经常使用以下两种结构：cv::Point 和 cv::Scalar
cv::Point 表示 2D 平面上的点，通过指定其在图像上的坐标位置 x 和 y 来实现。
语法结构如下：

```
Point pt;
pt.x = 10;
pt.y = 8;
```

或者

```
Point pt = Point(10, 8);
```

cv::Scalar 表示一个含有 4 个元素的向量，OpenCV 中用来传递像素值。语法结构如下：

```
Scalar( a, b, c )
```

其中 Blue = a, Green = b, Red = c。这里由于我们只有 BGR 三个颜色值，所以我们只需要定义三个变量，最后一个元素可以省略。
下面开始介绍直线、椭圆、矩形、圆以及多边形分别的语法结构：

(1) 直线：

```
cv::line(InputOutputArray img,
         Point           pt1,
         Point           pt2,
         const Scalar &  color,
         int             thickness = 1,
         int             lineType = LINE_8,
         int             shift = 0
         )
```

参数：
img: 图像
pt: 线段的起点
pt2: 线段的终点
color: 直线的颜色
thickness: 直线的粗细
lineType: 直线类型，具体可以参考下表
shift: 点坐标中的小数点位数

表 1 直线类型

FILLED	
LINE_4	4 联通线
LINE_8	8 联通线
LINE_AA	抗锯齿线

Tips:
这里的直线类型指的不是实线、虚线或者是点划线，而是指线的产生算法。如图所示，图 1 为 8 联通线、图 2 为 4 联通线。

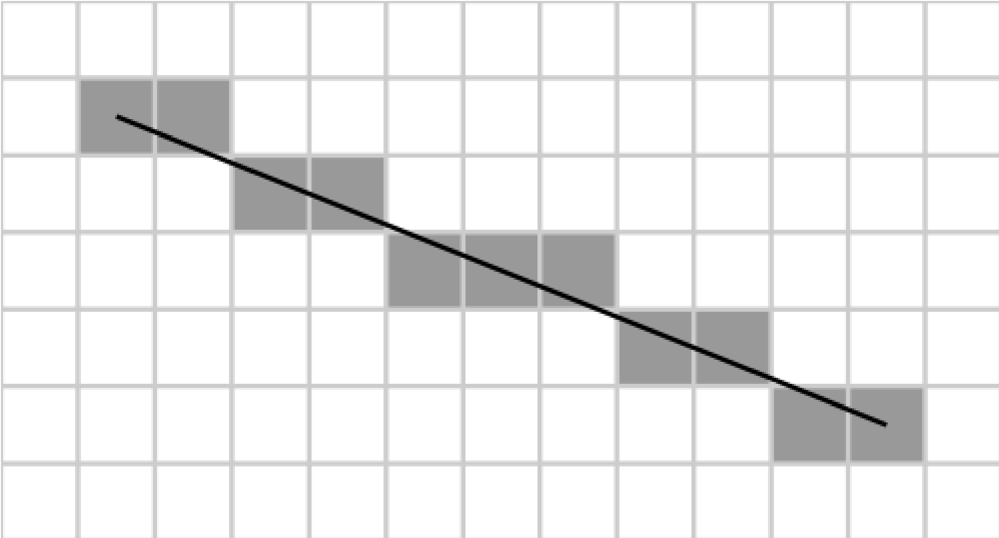

图 1 8 联通线

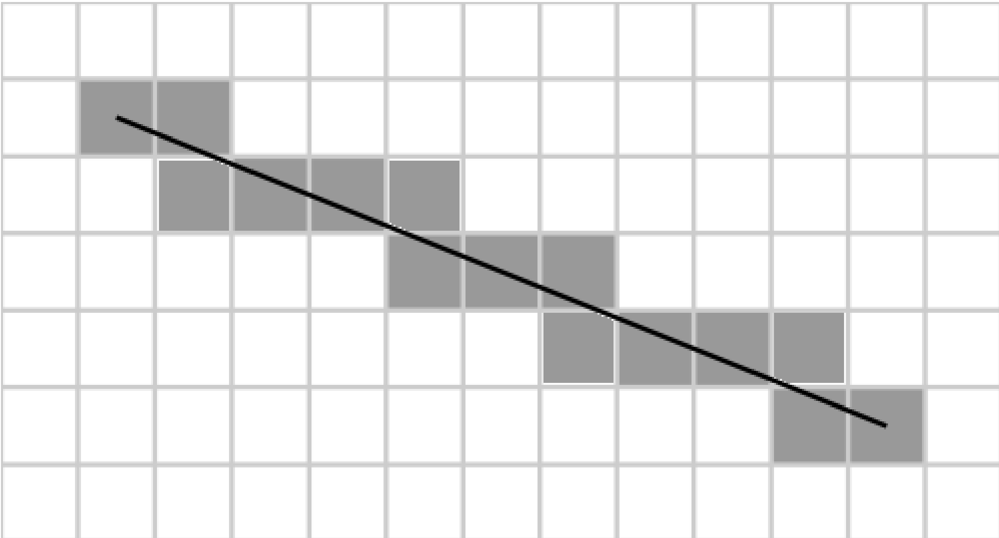

图 2 4 联通线

8 联通线是指下一个点连接上一个点的边或者角，4 联通线是指下一个点与上一个点边相连。4 联通线消除了 8 联通线的断裂瑕疵。

实例：

```
void MyLine( Mat img, Point start, Point end )
{
  int thickness = 2;
  int lineType =LINE_8;

  line(img,
        start,
        end,
        Scalar( 0, 0, 0 ),
        thickness,
        lineType);
}
```

(2) 椭圆

```
cv::ellipse(InputOutputArray img,
            Point           center,
            Size            axes,
            double          angle,
            double          startAngle,
            double          endAngle,
            const Scalar&   color,
            int             thickness = 1,
            int             lineType = LINE_8,
            int             shift = 0
            )
```

参数：
img: 图像
center: 椭圆中心
axes: 椭圆主轴尺寸的一半
angle: 椭圆旋转角度
startAngle: 椭圆弧的起始角度
engAngle: 椭圆弧的终止角度
color: 椭圆的颜色
thickness: 椭圆轮廓的粗细，如果不设置默认填充
lineType: 椭圆边界线类型
shift: 中心坐标和轴值的小数点位数

cv::ellipse 可以用来绘制椭圆线、实心椭圆、椭圆弧、椭圆扇面。如果想要绘制一个完整的椭圆，startAngle=0、endAngle=360。如果起始角度大于终止角度，他们会进行交换。下图 3 在绘制蓝色弧时各个参数的含义。

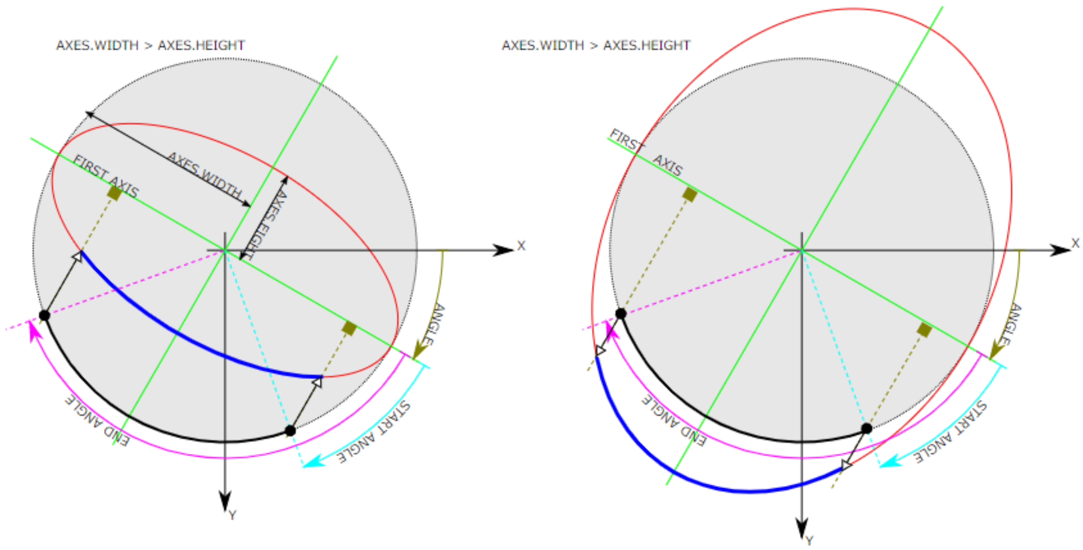

图 3 椭圆弧绘制中各参数意义

实例：

```
void MyEllipse( Mat img, double angle )
{
 int thickness = 2;
```

```
  int lineType = 8;
  ellipse( img,
           Point( w/2, w/2 ),
           Size( w/4, w/16),
           angle,
           0,
           360,
           Scalar( 255, 0, 0),
           thickness,
           lineType );
}
```

(3) 矩形：

```
cv::rectangle(InputOutputArray img,
              Point            pt1,
              Point            pt2,
              const Scalar&    color,
              int              thickness = 1,
              int              lineType = LINE_8,
              int              shift = 0
              )
```

参数：

img：图像

pt1：矩形的顶点

pt2:与 pt1 相对的矩形顶点

color：矩形颜色或者亮度

thickness：矩形轮廓的粗细

lineType：线条类型

shift：点坐标中的小数点位数

实例：

```
rectangle( rook_image,
           Point( 0, 7*w/8 ),
           Point( w, w),
           Scalar( 0, 255, 255 ),
           FILLED,
           LINE_8 );
```

(4) 多段线

```
cv::polylines(InputOutputArray   img,
              InputArrayOfArrays pts,
              bool               isClosed,
              const Scalar&      color,
              int                thickness = 1,
              int                lineType = LINE_8,
              int                shift = 0
              )
```

参数：
img：图像
pts：多边形曲线阵列
isClosed：所绘制多段线是否闭合
color：多段线颜色
thickness：多段线粗细
lineType：多段线种类
shift：点坐标中的小数点位数
多边形填充：

```
cv::fillPoly(InputOutputArray img,
             const Point** pts,
             const int*      npts
             int             ncontours
             const Scalar& color,
             int             lineType = LINE_8,
             int             shift = 0,
             Point           offset = Point()
             )
```

cv::fillPoly 可以填充由多边形包围的区域，可用于填充复杂区域。
参数：
img：图像
pts：多边形数组，每个多边形都是一组点
npts：顶点个数
ncontours：多边形轮廓个数
color：多边形颜色
lineType：多边形边界粗细
shift：点坐标的小数点位数
offset：可选择轮廓点偏移量
实例：

```
void MyPolygon( Mat img )
{
  int lineType = LINE_8;

  Point rook_points[1][20];
  rook_points[0][0] = Point( w/4, 7*w/8 );
  rook_points[0][1] = Point( 3*w/4, 7*w/8);
  rook_points[0][2] = Point( 3*w/4, 13*w/16 );
  rook_points[0][3] = Point( 11*w/16, 13*w/16 );
  rook_points[0][4] = Point( 19*w/32, 3*w/8 );
  rook_points[0][5] = Point( 3*w/4, 3*w/8 );
  rook_points[0][6] = Point( 3*w/4, w/8 );
  rook_points[0][7] = Point( 26*w/40, w/8 );
  rook_points[0][8] = Point( 26*w/40, w/4 );
  rook_points[0][9] = Point( 22*w/40, w/4 );
```

```
  rook_points[0][10] = Point( 22*w/40, w/8 );
  rook_points[0][11] = Point( 18*w/40, w/8 );
  rook_points[0][12] = Point( 18*w/40, w/4 );
  rook_points[0][13] = Point( 14*w/40, w/4 );
  rook_points[0][14] = Point( 14*w/40, w/8 );
  rook_points[0][15] = Point( w/4, w/8 );
  rook_points[0][16] = Point( w/4, 3*w/8 );
  rook_points[0][17] = Point( 13*w/32, 3*w/8 );
  rook_points[0][18] = Point( 5*w/16, 13*w/16 );
  rook_points[0][19] = Point( w/4, 13*w/16 );

  const Point* ppt[1] = { rook_points[0] };
  int npt[] = { 20 };

  fillPoly( img,
            ppt,
            npt,
            1,
            Scalar( 255, 255, 255 ),
            lineType );
}
```

(5) Demo

```
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>
#define w 400
using namespace cv;
void MyEllipse( Mat img, double angle );
void MyFilledCircle( Mat img, Point center );
void MyPolygon( Mat img );
void MyLine( Mat img, Point start, Point end );
int main( void ){
```

创建两个窗口和两个图像来进行图形绘制

```
    char atom_window[] = "Drawing 1: Atom";
    char rook_window[] = "Drawing 2: Rook";
    Mat atom_image = Mat::zeros( w, w, CV_8UC3 );
    Mat rook_image = Mat::zeros( w, w, CV_8UC3 );
```

原子图像绘制，创建了 MyEllipse 和 MyFilledCircle 两个函数

```
    MyEllipse( atom_image, 90 );
    MyEllipse( atom_image, 0 );
    MyEllipse( atom_image, 45 );
    MyEllipse( atom_image, -45 );
    MyFilledCircle( atom_image, Point( w/2, w/2) );
```

绘制国际象棋的车创建了 MyPolygon 和 MyLine 函数，且应用了矩形的绘制函数

```
    MyPolygon( rook_image );
    rectangle( rook_image,
               Point( 0, 7*w/8 ),
               Point( w, w),
               Scalar( 0, 255, 255 ),
               FILLED,
               LINE_8 );
    MyLine( rook_image, Point( 0, 15*w/16 ), Point( w, 15*w/16 ) );
    MyLine( rook_image, Point( w/4, 7*w/8 ), Point( w/4, w ) );
    MyLine( rook_image, Point( w/2, 7*w/8 ), Point( w/2, w ) );
    MyLine( rook_image, Point( 3*w/4, 7*w/8 ), Point( 3*w/4, w ) );
    imshow( atom_window, atom_image );
    moveWindow( atom_window, 0, 200 );
    imshow( rook_window, rook_image );
    moveWindow( rook_window, w, 200 );
    waitKey( 0 );
    return(0);
}
void MyEllipse( Mat img, double angle )
{
    int thickness = 2;
    int lineType = 8;
    ellipse( img,
             Point( w/2, w/2 ),
             Size( w/4, w/16 ),
             angle,
             0,
             360,
             Scalar( 255, 0, 0 ),
             thickness,
             lineType );
}
void MyFilledCircle( Mat img, Point center )
{
    circle( img,
            center,
            w/32,
            Scalar( 0, 0, 255 ),
            FILLED,
            LINE_8 );
}
void MyPolygon( Mat img )
```

```
{
    int lineType = LINE_8;
    Point rook_points[1][20];
    rook_points[0][0]  = Point(    w/4,   7*w/8 );
    rook_points[0][1]  = Point(  3*w/4,   7*w/8 );
    rook_points[0][2]  = Point(  3*w/4,  13*w/16 );
    rook_points[0][3]  = Point( 11*w/16, 13*w/16 );
    rook_points[0][4]  = Point( 19*w/32,  3*w/8 );
    rook_points[0][5]  = Point(  3*w/4,   3*w/8 );
    rook_points[0][6]  = Point(  3*w/4,     w/8 );
    rook_points[0][7]  = Point( 26*w/40,    w/8 );
    rook_points[0][8]  = Point( 26*w/40,    w/4 );
    rook_points[0][9]  = Point( 22*w/40,    w/4 );
    rook_points[0][10] = Point( 22*w/40,    w/8 );
    rook_points[0][11] = Point( 18*w/40,    w/8 );
    rook_points[0][12] = Point( 18*w/40,    w/4 );
    rook_points[0][13] = Point( 14*w/40,    w/4 );
    rook_points[0][14] = Point( 14*w/40,    w/8 );
    rook_points[0][15] = Point(    w/4,     w/8 );
    rook_points[0][16] = Point(    w/4,   3*w/8 );
    rook_points[0][17] = Point( 13*w/32,  3*w/8 );
    rook_points[0][18] = Point(  5*w/16, 13*w/16 );
    rook_points[0][19] = Point(    w/4,  13*w/16 );
    const Point* ppt[1] = { rook_points[0] };
    int npt[] = { 20 };
    fillPoly( img,
              ppt,
              npt,
              1,
              Scalar( 255, 255, 255 ),
              lineType );
}
void MyLine( Mat img, Point start, Point end )
{
    int thickness = 2;
    int lineType = LINE_8;
    line( img,
          start,
          end,
          Scalar( 0, 0, 0 ),
          thickness,
          lineType );
}
```

2. 轮廓

(1) 如何在图像中寻找物体轮廓

在二值图像中寻找轮廓

```
cv::findContours(InputArray             image,
                 OutputArrayOfArrays contours,
                 OutputArray             hierarchy,
                 int                     mode,
                 int                     method,
                 Point                   offset = Point()
                 )
```

参数：

image: 8 位单通道图像，非零像素视为 1，零值像素视为 0，因此所有的输入图像均被视为二值图像。因此在应用时我们一般会利用 Canny, compare, InRange, threshold, adaptiveThreshold 等函数处理得到二值图像。如果模式为 RETR_CCOMP 或者 RETR_FLOODFILL，输入也可以是 32 位的灰度图像。

contours: 检测到的轮廓，每个轮廓都储存为一组点向量，定义为 std::vector<std::vector<cv::Point>>。

hierarchy: 可选输出向量，定义为 std::vector<cv::Vec4i>，包含有图像拓扑信息。其内部元素的个数等同于所检测轮廓的个数。Vec4i 是 Vec<int, 4>的别名，定义为向量内每一个元素包含 4 个 int 型变量。hierarchy 向量内第 i 个轮廓的 4 个 int 型变量（hierarchy[i][0], hierarchy[i][1], hierarchy[i][2], hierarchy[i][3]）分别表示其同一层级下的后一个轮廓，前一个轮廓，子轮廓以及父轮廓的索引编号。如果轮廓 i 没有对应的上述轮廓，则 hierarchy[i][0], hierarchy[i][1], hierarchy[i][2], hierarchy[i][3]被置为-1。

mode: 轮廓的检索模式。具体见下表二。

method: 轮廓的近似方法。具体见表三。

offset: Point 偏移量

表二 轮廓检索模式

RETR_EXTERNAL	只检索最外围轮廓，对于所有轮廓设置 hierarchy[i][2] = hierarchy[i][3] = -1
RETR_LIST	检索所有轮廓但不建立任何轮廓间的层级关系
RETR_CCOMP	检索所有轮廓，但所有轮廓只建立两个等级关系，最外层为顶层，如果第二层轮廓内还包含有其他轮廓，则这些轮廓都会被归为顶层。
RETR_TREE	检索所有轮廓，构建一个嵌套式的完整的层级轮廓。
RETR_FLOODFILL	

表三 轮廓近似算法

CHAIN_APPROX_NONE	储存所有轮廓点
CHAIN_APPROX_SIMPLE	压缩水平垂直等信息，只保留拐点
CHAIN_APPROX_TC89_L1	使用 teh-Chinl chain[242]近似算法
CHAIN_APPROX_TC89_KCOS	使用 teh-Chinl chain 近似算法

[242]C-H Teh and Roland T. Chin. On the detection of dominant points on digital curves. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(8):859–872,

1989.

轮廓的绘制

```
cv::drawContours(InputOutputArray     image,
                 InputArrayOfArrays contours,
                 int                contourIdx,
                 const              Scalar& color,
                 int                thickness = 1,
                 int                lineType = LINE_8,
                 InputArray         hierarchy = noArray(),
                 int                maxLevel =INT_MAX,
                 Point              offset = Point()
                 )
```

参数：

image: 目的图像。

contours: 所有输入轮廓，每个轮廓储存为一组点向量。

contourIdx: 指示要绘制的轮廓线参数，如果为负，则绘制所有轮廓线。

color: 轮廓线颜色。

thickness: 绘制轮廓线的粗细，如果为负（thickness=FILLED）,则轮廓内部也会被绘制。当 thickness=FILLED 时，即使没有提供层级数据，也能成功绘制有孔的轮廓。

lineType: 直线绘制算法。

hierarchy: 可选层级信息，只有当想绘制多条轮廓时才需要用到。

mexLevel: 所绘制轮廓的最大层级。如果是 0，则只绘制指定的轮廓；如果是 1，则绘制指定轮廓以及嵌套轮廓；如果是 2，则绘制指定轮廓、嵌套轮廓以及嵌套轮廓的嵌套轮廓以此类推。该参数只在层级参数可用时可以使用。

offset: Point 的偏移

Demo

```
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;
Mat src_gray;
int thresh = 100;
RNG rng(12345);
void thresh_callback(int, void* );
int main( int argc, char** argv )
{
```

读取图像

```
    Mat src = imread( "D:/Picture/food.jpg", IMREAD_COLOR );
    if( src.empty() )
    {
        cout << "Could not open or find the image!\n" << endl;
```

```
        cout << "Usage: " << argv[0] << " <Input image>" << endl;
        return -1;}
```

转化为灰度图像，模糊处理以去除噪声

```
    cvtColor( src, src_gray, COLOR_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );
```

创建名为 Source 的窗口并在其中显示输入图像

```
    const char* source_window = "Source";
    namedWindow( source_window );
    imshow( source_window, src );
    const int max_thresh = 255;
    createTrackbar( "Canny thresh:", source_window, &thresh,
max_thresh, thresh_callback );
    thresh_callback( 0, 0 );
    waitKey();
    return 0;
}
void thresh_callback(int, void* )
{
    Mat canny_output;
```

Canny 边缘检测

```
    Canny( src_gray, canny_output, thresh, thresh*2 );
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
```

寻找轮廓的函数

```
    findContours( canny_output, contours, hierarchy, RETR_TREE,
CHAIN_APPROX_SIMPLE );
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( size_t i = 0; i< contours.size(); i++ )
    {
        Scalar color = Scalar( rng.uniform(0, 256),
rng.uniform(0,256), rng.uniform(0,256) );
```

轮廓绘制

```
        drawContours( drawing, contours, (int)i, color, 2, LINE_8,
hierarchy, 0 );
    }
    imshow( "Contours", drawing );
}
```

(2) 凸包函数

```
cv::convexHull(InputArray   points,
               OutputArray hull,
               bool          clockwise = false,
               bool          returnPoints = true
               )
```

参数：

points: 输入的二维点集，存储在 std::vector 或者 Mat 中

hull: 输出找到的凸包。

clockwise: 操作方向，当 clockwise=true 时，输出凸包为顺时针方向。否则输出凸包方向为逆时针方向。

returnPoints: 操作标识符，默认值为 true，此时返回各凸包的各点，否则返回凸包各点的索引。当输出数组为 std::vector 时，此标识被忽略。

Demo

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

```
    return 0;
}
void thresh_callback(int, void* )
{
    Mat canny_output;
    Canny( src_gray, canny_output, thresh, thresh*2 );
    vector<vector<Point> > contours;
    findContours( canny_output, contours, RETR_TREE,
CHAIN_APPROX_SIMPLE );
```

利用凸包函数找到图形中的凸包

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

绘制轮廓与凸包

```
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( size_t i = 0; i< contours.size(); i++ )
    {
        Scalar color = Scalar( rng.uniform(0, 256),
rng.uniform(0,256), rng.uniform(0,256) );
        drawContours( drawing, contours, (int)i, color );
        drawContours( drawing, hull, (int)i, color );
    }
    imshow( "Hull demo", drawing );
}
```

凸包的应用

(1) 凸包可以用来确定一组点集的边界。

(2) 凸包可以用来避免碰撞。

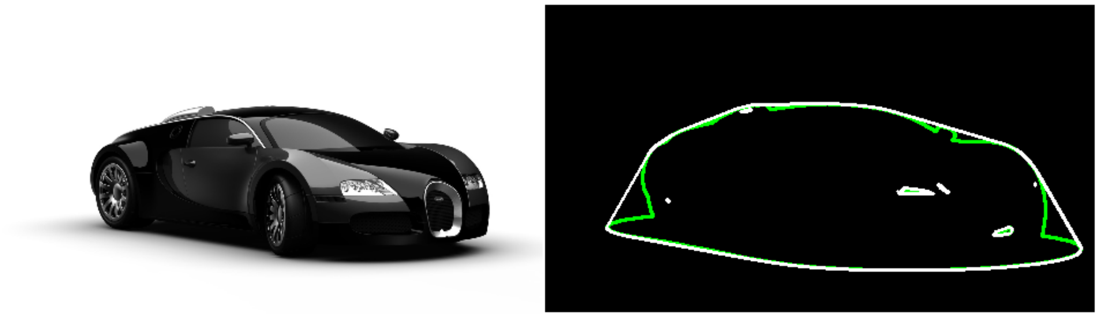

图 4 左：原始图像 右：凸包（白色）和轮廓（绿色）

(3) 为轮廓创造包围框和圆圈

cv::boundingRect (InputArray array)

计算点集或灰度图像非零像素的垂直边界矩形。

参数：

array: 输入灰度图像或二维点集，存储在 std::vector 或者 Mat 中。

cv::minEnclosingCircle(InputArray points,
Point2f& center,
float& radius
)

寻找一个二维点集封闭最小环形区域

参数：

points: 输入的二维点向量，存储在 std::vector 或者 Mat 中

center: 输出圆环的中心

radius: 输出圆环的半径

Demo

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

```
    waitKey();
    return 0;
}
void thresh_callback(int, void* )
{
    Mat canny_output;
    Canny( src_gray, canny_output, thresh, thresh*2 );
    vector<vector<Point> > contours;
    findContours( canny_output, contours, RETR_TREE,
CHAIN_APPROX_SIMPLE );
```

对于每个找到的轮廓，采取精度为±3 的多边形近似，且曲线必须闭合。然后对于每个多边形找到一个边界框，然后利用函数 minEnclosingCircle()找到多边形的最小封闭圆。

```
    vector<vector<Point> > contours_poly( contours.size() );
    vector<Rect> boundRect( contours.size() );
    vector<Point2f>centers( contours.size() );
    vector<float>radius( contours.size() );
    for( size_t i = 0; i < contours.size(); i++ )
    {
        approxPolyDP( contours[i], contours_poly[i], 3, true );
        boundRect[i] = boundingRect( contours_poly[i] );
        minEnclosingCircle( contours_poly[i], centers[i], radius[i] );
    }
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( size_t i = 0; i< contours.size(); i++ )
    {
        Scalar color = Scalar( rng.uniform(0, 256), rng.uniform(0,256),
rng.uniform(0,256) );
        drawContours( drawing, contours_poly, (int)i, color );
        rectangle(  drawing,  boundRect[i].tl(),  boundRect[i].br(),
color, 2 );
        circle( drawing, centers[i], (int)radius[i], color, 2 );
    }
    imshow( "Contours", drawing );
}
```

(4) 为轮廓创建旋转框和椭圆

cv::minAreaRect(InputArray points)

寻找输入二维点集的最小面积的旋转矩形

参数：

points: 输入二维点向量，存储在 std::vector<>或者 Mat 中

cv::fitEllipse(InputArray points)

在一组二维点集附近拟合一个椭圆

参数：

points: 输入二维点集，存储在 std::vector 或者 Mat 中。

Demo

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

```
    Canny( src_gray, canny_output, thresh, thresh*2 );
    vector<vector<Point> > contours;
    findContours( canny_output, contours, RETR_TREE,
CHAIN_APPROX_SIMPLE, Point(0, 0) );
```

利用 minAreaRct 函数在轮廓周围寻找最小旋转矩形，利用 fitEllipse 函数在矩形内部拟合椭圆。

```
    vector<RotatedRect> minRect( contours.size() );
    vector<RotatedRect> minEllipse( contours.size() );
    for( size_t i = 0; i < contours.size(); i++ )
    {
        minRect[i] = minAreaRect( contours[i] );
        if( contours[i].size() > 5 )
        {
            minEllipse[i] = fitEllipse( contours[i] );
        }
    }
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( size_t i = 0; i< contours.size(); i++ )
    {
        Scalar color = Scalar( rng.uniform(0, 256),
rng.uniform(0,256), rng.uniform(0,256) );
        // contour
        drawContours( drawing, contours, (int)i, color );
        // ellipse
        ellipse( drawing, minEllipse[i], color, 2 );
        // rotated rectangle
        Point2f rect_points[4];
        minRect[i].points( rect_points );
        for ( int j = 0; j < 4; j++ )
        {
            line( drawing, rect_points[j], rect_points[(j+1)%4],
color );
        }
    }
    imshow( "Contours", drawing );
}
```

(5) 图像矩

图像识别的核心问题是图像的特征提取，因此数据越简单越具有代表性，可以不受到光线、噪点、几何变形干扰，则越好。图像矩可以用来描述图像不同种类的几何特征，例如：大小、灰度、方向、形状等。针对一副图像，可以把像素坐标看成是一个二维随机变量(x, y)，那么一幅灰度图像可以用二维灰度图密度函数来表示。因此我们可以利用矩来描述灰度图像的特征。空间矩中零阶矩求面积，一阶矩确定重心，二阶矩确定主方向，二阶矩和三阶矩可以推导出七个不变矩，不变矩具有旋转，平移，缩放等不变性。

```
cv::moments( InputArray array,
             bool       binaryImage = false
           )
```

计算多边形或者栅格化形状所有的矩到三阶

参数：

array: 输入光栅图像或者二维点数组

binaryImage: 如果是真，则非零图像像素被视为 1，该参数只适用与图像

```
cv::Moments::Moments
(
// 空间矩(10 个)
double m00,double m10,double m01,double m20,double m11,double m02,double
    m30,double m21,double m12,double m03
// 中心矩（7 个）
double mu20, double mu11, double mu02, double mu30, double mu21 , double
mu12,double mu03
// 中心归一化矩（）
double nu20, double nu11, double nu02, double nu30, double nu21, double nu12,double
nu03;
)
```

a. 空间矩 Moments::mji 的计算公式：

$$m_{ji} = \sum_{x,y} \left(\mathrm{array}(x,y) \cdot x^j \cdot y^i \right)$$

对于二值图像，m00 就是轮廓的面积

b. 中心距 Moments::muji 计算公式：

$$mu_{ji} = \sum_{x,y} \left(\mathrm{array}(x,y) \cdot (x-\bar{x})^j \cdot (y-\bar{y})^i \right)$$

其中：$(\bar{x}, \bar{y})$为轮廓的质心：

$$\bar{x} = \frac{m_{10}}{m_{00}}, \ \bar{y} = \frac{m_{01}}{m_{00}}$$

c. 归一化的中心距 Moments::nuji 计算公式：

$$nu_{ji} = \frac{mu_{ji}}{m_{00}^{(i+j)/2+1}}.$$

```
cv::contourArea( InputArray contour,
                 bool        oriented = false
               )
```

计算轮廓面积

参数：

contour: 输入二维点向量，存储在 std::vector 或 Mat 中

oriented: 面向区域标识符，如果是 true，函数根据轮廓的方向返还一个带符号的面积值（顺时针或者逆时针）。通过这一特性可以根据面积的符号来确定轮廓的位置。如果是默认值 false，则面积以绝对值的方式返回。

实例：

```
vector<Point> contour;
contour.push_back(Point2f(0, 0));
contour.push_back(Point2f(10, 0));
contour.push_back(Point2f(10, 10));
contour.push_back(Point2f(5, 4));
double area0 = contourArea(contour);
vector<Point> approx;
approxPolyDP(contour, approx, 5, true);
double area1 = contourArea(approx);
cout << "area0 =" << area0 << endl <<
"area1 =" << area1 << endl <<
"approx poly vertices" << approx.size() << endl;
```

```
cv::arcLength( InputArray curve,
               bool        closed
             )
```

计算轮廓的周长或者曲线强度

参数：

curve: 输入二维点向量，存储在 std::vector 或者 Mat 中。

closed: 指示曲线是否闭合。

Demo

```
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
#include <iomanip>
using namespace cv;
using namespace std;
Mat src_gray;
int thresh = 100;
RNG rng(12345);
void thresh_callback(int, void* );
int main( int argc, char** argv )
{
    Mat src = imread("D:images.jpg", IMREAD_COLOR);
```

```
    if( src.empty() )
    {
        cout << "Could not open or find the image!\n" << endl;
        cout << "usage: " << argv[0] << " <Input image>" << endl;
        return -1;
    }
    cvtColor( src, src_gray, COLOR_BGR2GRAY );
    blur( src_gray, src_gray, Size(3,3) );
    const char* source_window = "Source";
    namedWindow( source_window );
    imshow( source_window, src );
    const int max_thresh = 255;
    createTrackbar( "Canny thresh:", source_window, &thresh,
max_thresh, thresh_callback );
    thresh_callback( 0, 0 );
    waitKey();
    return 0;
}
void thresh_callback(int, void* )
{
    Mat canny_output;
    Canny( src_gray, canny_output, thresh, thresh*2, 3 );
    vector<vector<Point> > contours;
    findContours( canny_output, contours, RETR_TREE,
CHAIN_APPROX_SIMPLE );
```

计算每个轮廓的所有矩

```
    vector<Moments> mu(contours.size() );
    for( size_t i = 0; i < contours.size(); i++ )
    {
        mu[i] = moments( contours[i] );
    }
```

计算轮廓的质心

```
    vector<Point2f> mc( contours.size() );
    for( size_t i = 0; i < contours.size(); i++ )
    {
        //add 1e-5 to avoid division by zero
        mc[i] = Point2f( static_cast<float>(mu[i].m10 / (mu[i].m00 +
1e-5)),
                         static_cast<float>(mu[i].m01 / (mu[i].m00 +
1e-5)) );
        cout << "mc[" << i << "]=" << mc[i] << endl;
    }
    Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
    for( size_t i = 0; i< contours.size(); i++ )
```

输出轮廓面积、长度，画出轮廓和质心

```
    {
        Scalar color = Scalar( rng.uniform(0, 256),
rng.uniform(0,256), rng.uniform(0,256) );
        drawContours( drawing, contours, (int)i, color, 2 );
        circle( drawing, mc[i], 4, color, -1 );
    }
    imshow( "Contours", drawing );
    cout << "\t Info: Area and Contour Length \n";
    for( size_t i = 0; i < contours.size(); i++ )
    {
        cout << " * Contour[" << i << "] - Area (M_00) = " <<
std::fixed << std::setprecision(2) << mu[i].m00
             << " - Area OpenCV: " << contourArea(contours[i]) << " -
Length: " << arcLength( contours[i], true ) << endl;
    }
}
```

(6) 多边形测试

```
cv::pointPolygonTest( InputArray contour,
                      Point2f     pt,
                      bool        measureDist
                    )
```

测试点是否在多边形中。该函数确定点是在轮廓内部，外部还是在轮廓上。返回正值表示在内部，负值表示在外部，0 值则在轮廓上。当 measureDist = false，返回值是+1，-1 和 0。否则返回值是点到轮廓最近边的距离。

参数：

contour: 输入轮廓

pt: 用来测试轮廓的点

measureDist: 如果 true，函数评估的是点到轮廓最近边的距离值，否则函数只是检测点与轮廓之间的关系。

Demo

```
#include "opencv2/highgui.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;
int main( void )
{
    const int r = 100;
    Mat src = Mat::zeros( Size( 4*r, 4*r ), CV_8U );
```

绘制图形

```
    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 );
    }
```

得到相应轮廓

```
    vector<vector<Point> > contours;
    findContours( src, contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
```

计算各点到轮廓的距离

```
    Mat raw_dist( src.size(), CV_32F );
    for( int i = 0; i < src.rows; i++ )
    {
        for( int j = 0; j < src.cols; j++ )
        {
```

```
            raw_dist.at<float>(i,j) =
(float)pointPolygonTest( contours[0], Point2f((float)j, (float)i),
true );
        }
    }
    double minVal, maxVal;
    Point maxDistPt; // inscribed circle center
    minMaxLoc(raw_dist, &minVal, &maxVal, NULL, &maxDistPt);
    minVal = abs(minVal);
    maxVal = abs(maxVal);
```

图形化显示距离

```
    Mat drawing = Mat::zeros( src.size(), CV_8UC3 );
    for( int i = 0; i < src.rows; i++ )
    {
        for( int j = 0; j < src.cols; j++ )
        {
            if( raw_dist.at<float>(i,j) < 0 )
            {
                drawing.at<Vec3b>(i,j)[0] = (uchar)(255 -
abs(raw_dist.at<float>(i,j)) * 255 / minVal);
            }
            else if( raw_dist.at<float>(i,j) > 0 )
            {
                drawing.at<Vec3b>(i,j)[2] = (uchar)(255 -
raw_dist.at<float>(i,j) * 255 / maxVal);
            }
            else
            {
                drawing.at<Vec3b>(i,j)[0] = 255;
                drawing.at<Vec3b>(i,j)[1] = 255;
                drawing.at<Vec3b>(i,j)[2] = 255;
            }
        }
    }
    circle(drawing, maxDistPt, (int)maxVal, Scalar(255,255,255));
    imshow( "Source", src );
    imshow( "Distance and inscribed circle", drawing );
    waitKey();
    return 0;
}
```