上海交通大学电工电子实验教学中心

工程实践与科技创新

2-B 指导书

图像处理部分

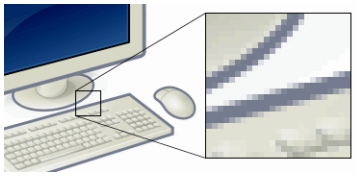
[在此处键入文档摘要。摘要通常为文档内容的简短概括。在此处键入文档摘要。摘要通常为文档内容的简短概括。]

目录

图像基础

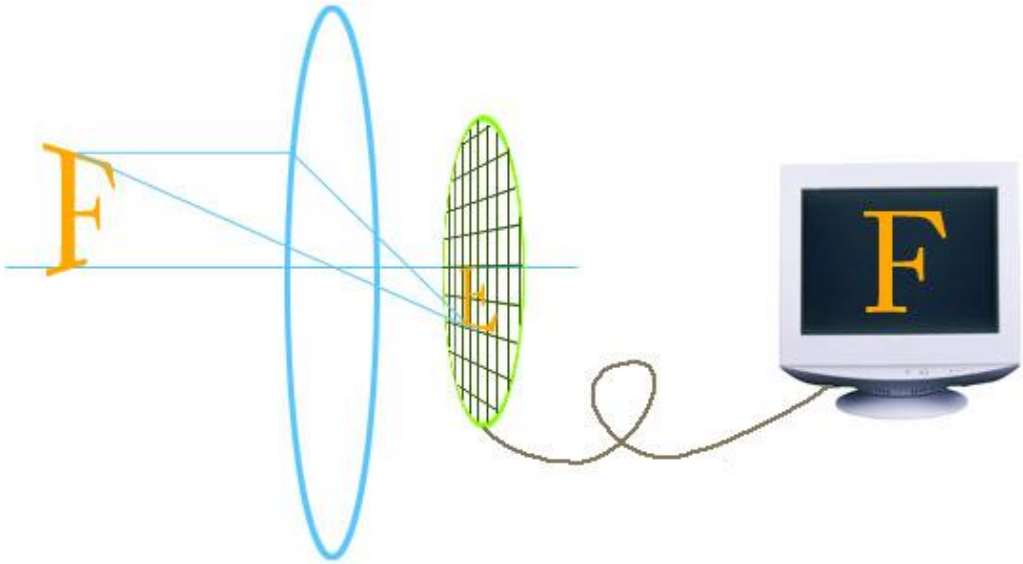
像素

如下图所示，将数字图像放大后可以看出，数字图像是由很多个点组成，每个点称为像素（pixel）。像素是数字图像的基本单位，具有某种颜色。



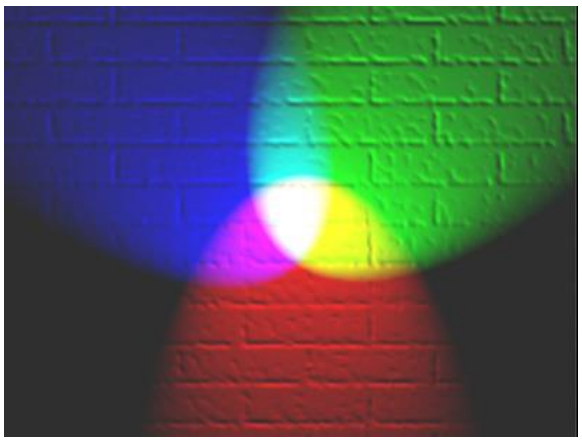
摄像头原理

下图为摄像头、数码相机等图像输入设备的原理图。绿色的圈代表感光元件，一般为CCD 或 CMOS，几十万几百万个紧密排列在一起，构成感光平面，每个元件感应极小区域的颜色信息，对应一个像素。



RGB 颜色空间

光的三原色为红（R）、绿（G）、蓝（B），通过这三种颜色不同强度的混合，可以得到任意颜色。



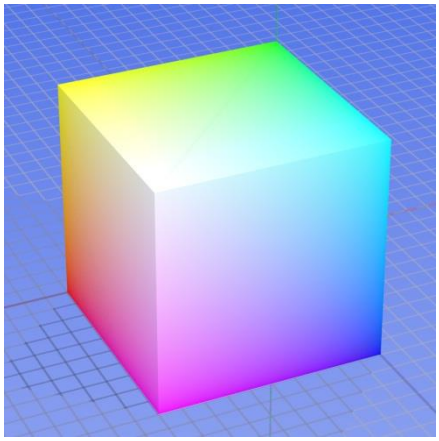
想象一个三维空间，建立一个直角坐标系，每个轴代表一种原色，取值范围是[0,1]。任意一种颜色通过分解出三原色的强度，并把强度作为坐标(R,G,B)，就可以对应到这个空间中的某个点。这样的空间称作颜色空间，而采用红绿蓝三原色作为三个坐标分量的颜色空间称作 RGB 颜色空间。

例如红色本身就是原色，因此分解出红绿蓝三原色的强度分别为 1、0、0，它的坐标就是(1,0,0)。

黄色分解出红绿蓝三原色的强度分别为 1、1、0，它的坐标就是(1,1,0)。

白色分解出红绿蓝三原色的强度分别为 1、1、1，它的坐标就是(1,1,1)。

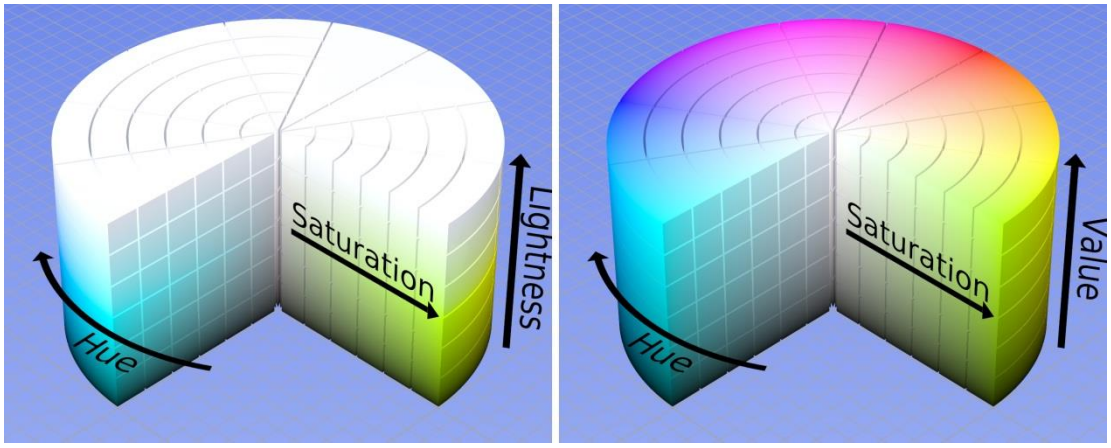
黑色就是没有光线，因此三原色强度均是 0，它就是原点。



RGB 颜色空间是数字图像的基本存储与表示方式，无需做额外的变换，处理起来简单快速。

HSV 颜色空间

另一种颜色空间是利用颜色的另外三种属性建立圆柱坐标系。三个坐标分量分别为色度（Hue）、饱和度（Saturation）、亮度（Luminance/Brightness/Intensity/Value），不同的坐标系有不同的取值范围，但基本概念类似。色度一般称作色轮，上面的颜色按照白光光谱顺序排列，通常 0 度是红色，120 度是绿色，240 度是蓝色。饱和度描述了颜色的鲜艳程度，一般取值[0,1]，1 是最鲜艳的颜色，即色轮的颜色，0 是没有颜色，即灰色。亮度的定义有很多种，最暗的颜色一般都是黑色，HSL 空间最亮的颜色是白色，HSV 空间最亮的是色轮颜色。



这种坐标系分离了色度、饱和度和亮度，是一种比较符合心理感受的颜色空间。因此可以单独修改某一分量，而不影响其他效果。如下图就是只修改了色度的两幅图像。

这种颜色空间对某一分量（如颜色）的判断也不容易受到其他影响。但是这种颜色空间需要从 RGB 空间转换，因此需要消耗时间，另外在很亮、很暗或者饱和度较低的情况下，色度的判断可能不是很准确。

数字图像的分类与表示

数字图像根据像素的颜色可以分成彩色图像、灰度图像和二值图像。

彩色图像最常见的是 24 位彩色图像，每个像素由 8 位二进制表示红色分量，8 位二进制表示绿色分量，8 位二进制表示蓝色分量，合在一起是 24 位。由于每个分量是 8 位二进制，即取值为 0~255 的整数，因此总共能表示 2553 种颜色。

灰度图像最常见的是 8 位灰度图像，即每个像素只由 8 位二进制表示。由于没有颜色，相当于彩色图舍弃了色度和饱和度，因此只需要记录三分之一的信息。这种图像也可以用 24 位彩色图像表示，方法是把像素的红绿蓝分量设置为相同的数值，就是对应的灰色。

二值图像顾名思义，每个像素只有两种取值，0 或者 1，只需要 1 位二进制就可以表示。通常用白色表示 1，黑色表示 0。因此用灰度图像的 255 表示 1，0 表示 0；彩色图像的 (255, 255, 255) 表示 1，(0, 0, 0) 表示 0。

图像的深度是指图像中每个通道的二进制位数。24 位彩色图像和 8 位灰度图像的深度都是 8，而二值图像的深度为 1。

通道

从一个 24 位彩色图像中，将每个像素的红色分量提取出来，将构成一个 8 位灰度图，则这个灰度图称作原来彩色图像的红色通道（Channel）。同样也有绿色通道和蓝色通道。显然绿色的物体在红色和蓝色通道里会相对较黑，而在绿色通道里相对较白。类似的，在 HSV 空间则有色度通道、饱和度通道以及亮度通道。

直方图

直方图（Histogram）是一种统计学图表，由一系列高度不等的纵向条纹或线段表示数据分布的情况。一般横轴表示数据类别，纵轴表示分布情况。图像直方图是用来表示图像中灰度分布的图表。横轴表示某种灰度，纵轴表示图像中有多少像素具有这种灰度。横轴划分的越多，直方图就越精细。下面是同一幅灰度图像的直方图，只有精细程度不同。

OpenCV 基础

简介

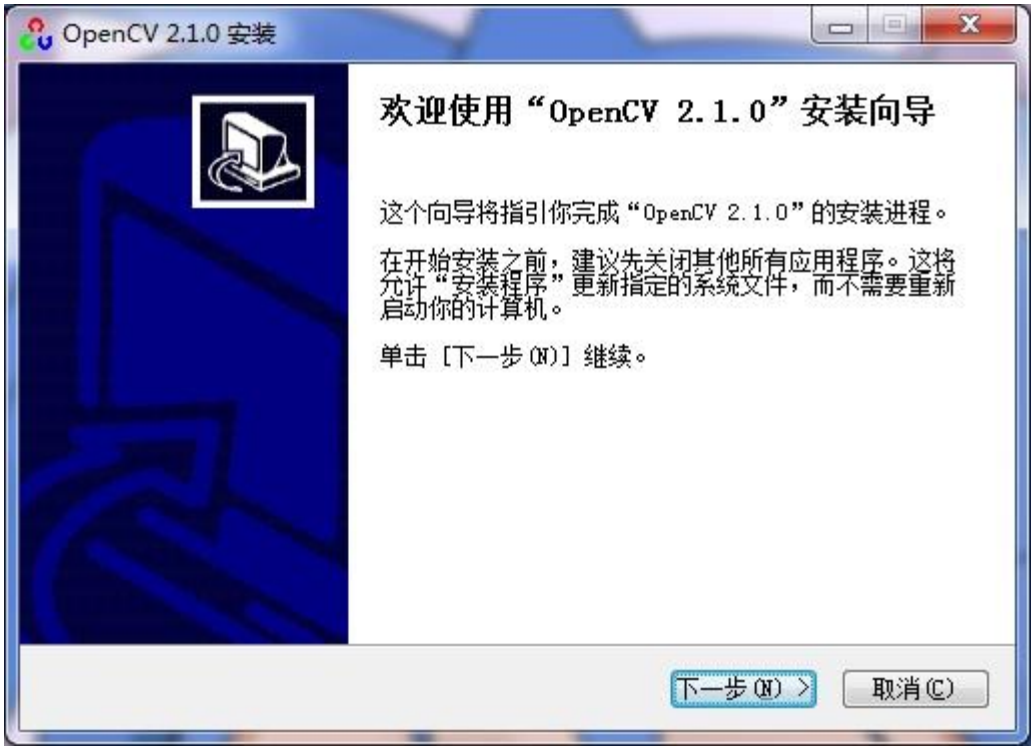
OpenCV 的全称是 Open Source Computer Vision Library，是一个跨平台的计算机视觉库，其中文官方网站为 http://www.opencv.org.cn。OpenCV 可用于开发实时的图像处理、计算机视觉以及模式识别程序。最新版本为 2.1。

OpenCV 可用于解决如下领域的问题：人机交互、物体识别、图像分割、人脸识别、动作识别、运动跟踪、机器人等等。

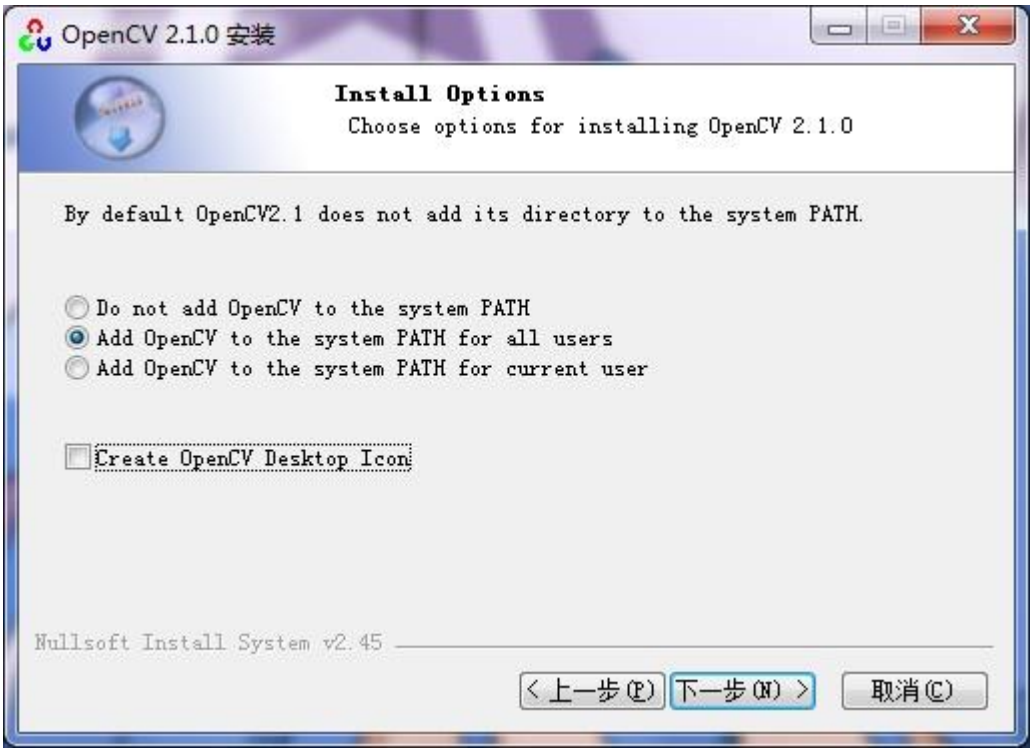
课程推荐的开发环境为 OpenCV2.1+VS2008，安装 VS2008 之后，可以从 OpenCV 的中文官方网站下载 OpenCV for Windows(VC2008 专用版)，这样可以不用自己动手编译 OpenCV。

安装

首先启动安装程序



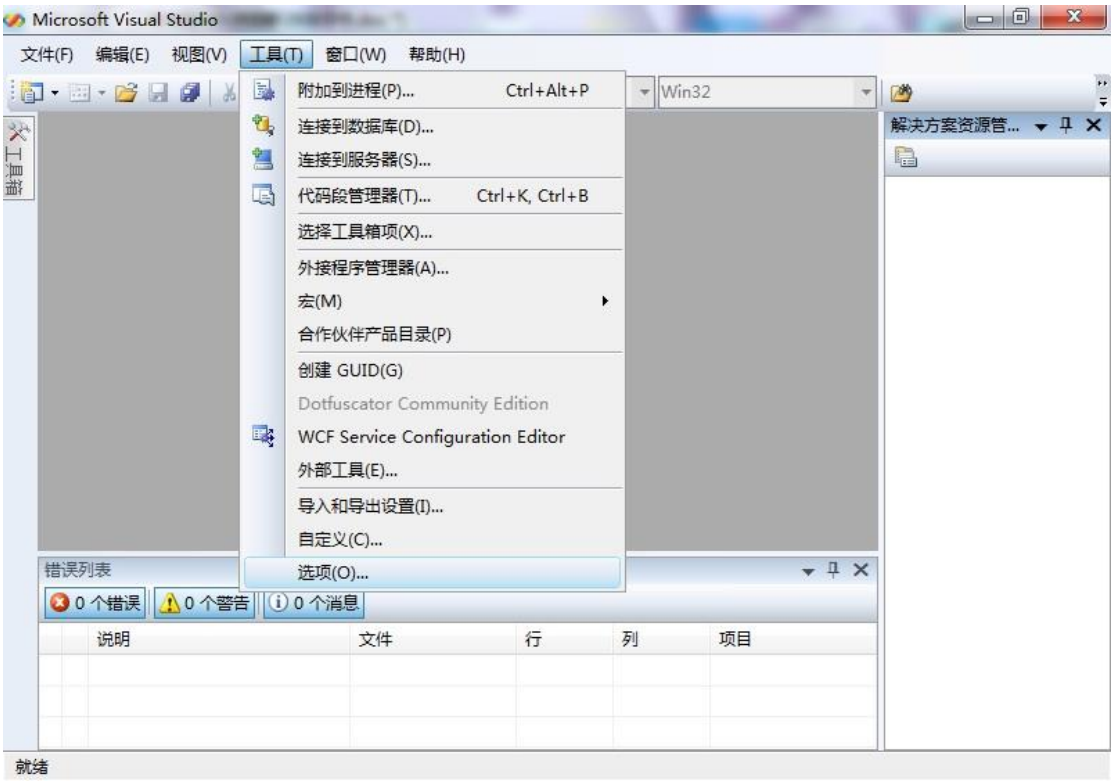
一直点下一步，直到这里



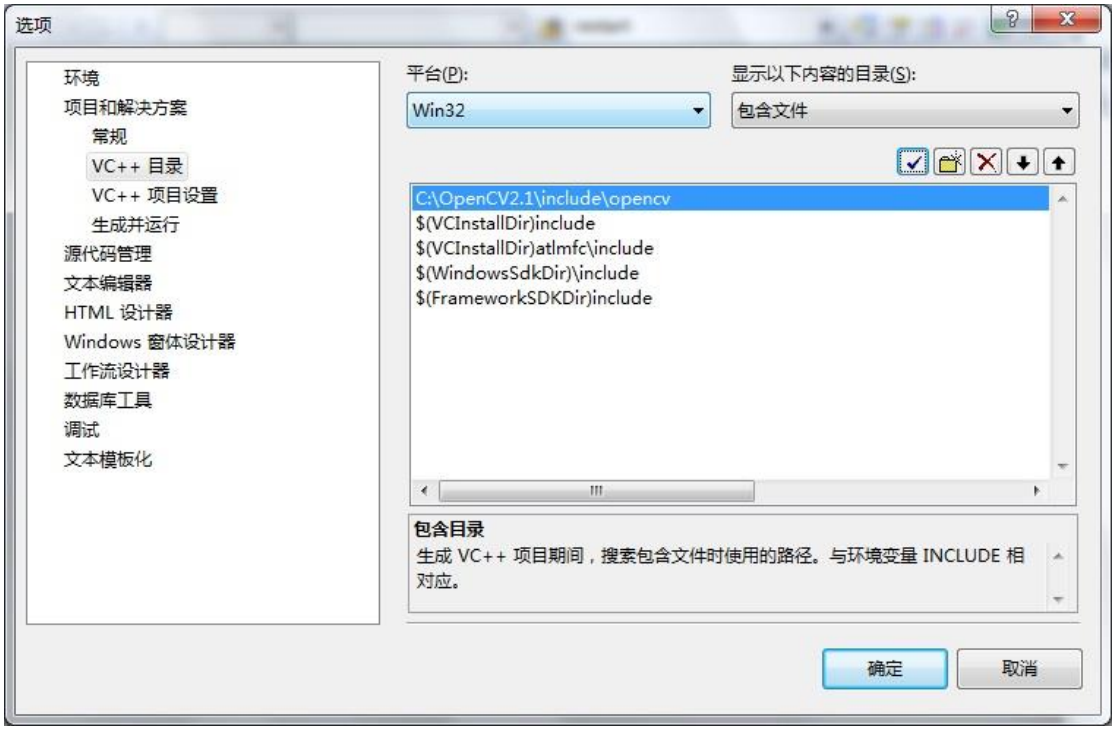
选择第二项或第三项，将 OpenCV 添加到系统路径中。然后一直点下一步完成安装。

OpenCV 环境设置

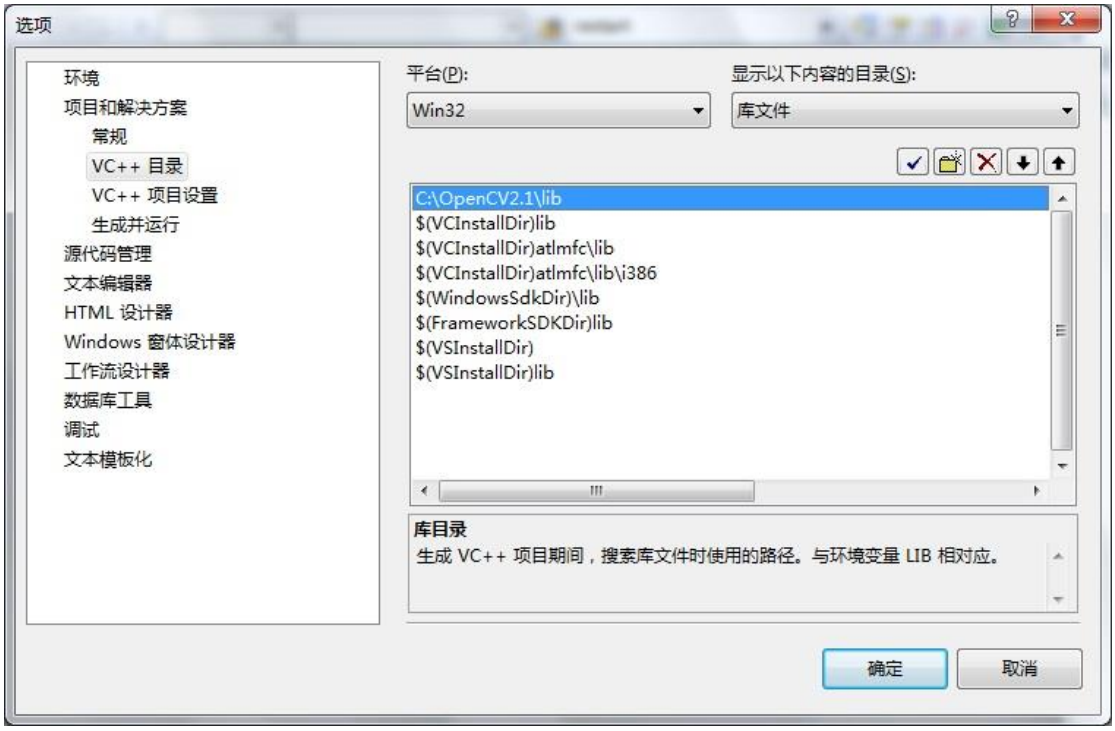
启动 VS2008，进入工具——选项



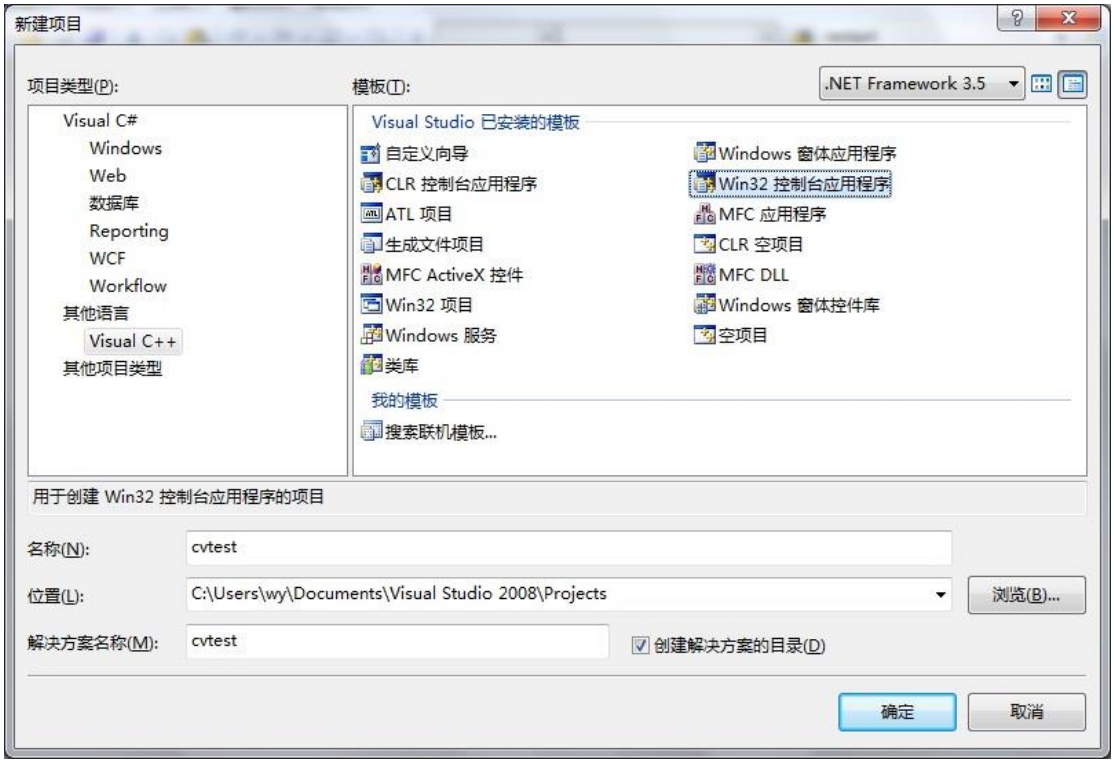
将 OpenCV 的头文件路径添加进来



再将 OpenCV 的库文件路径添加进来

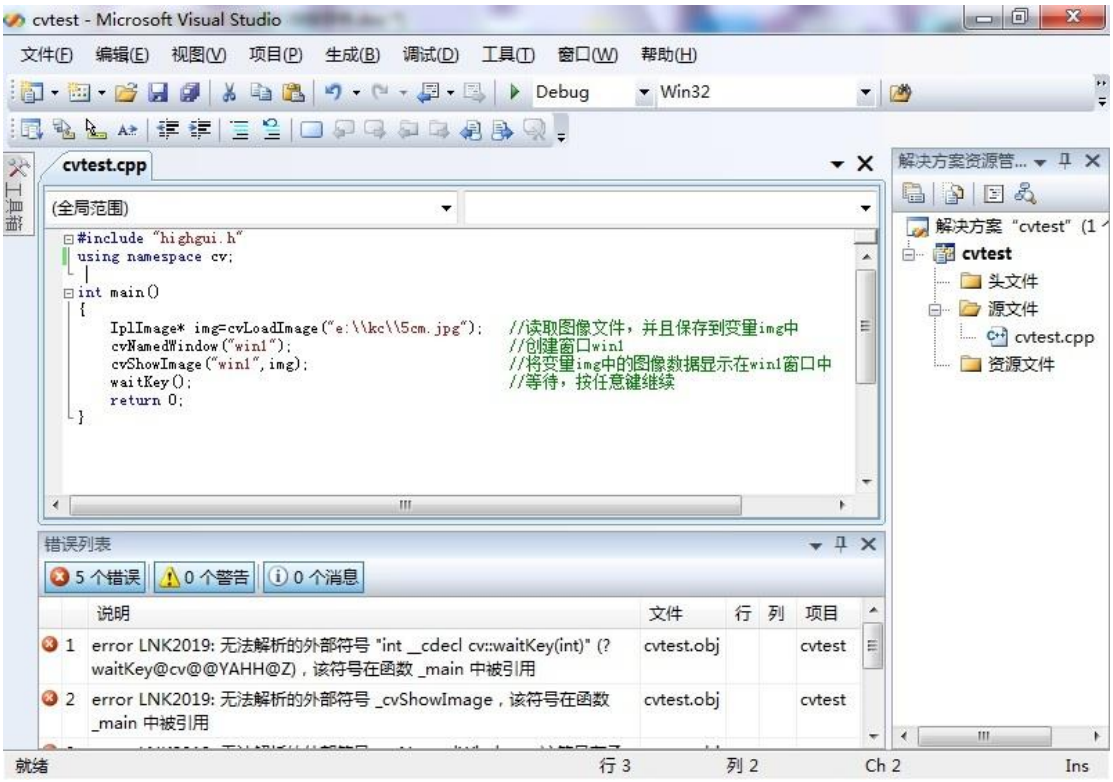


确定之后新建一个 VC 控制台项目

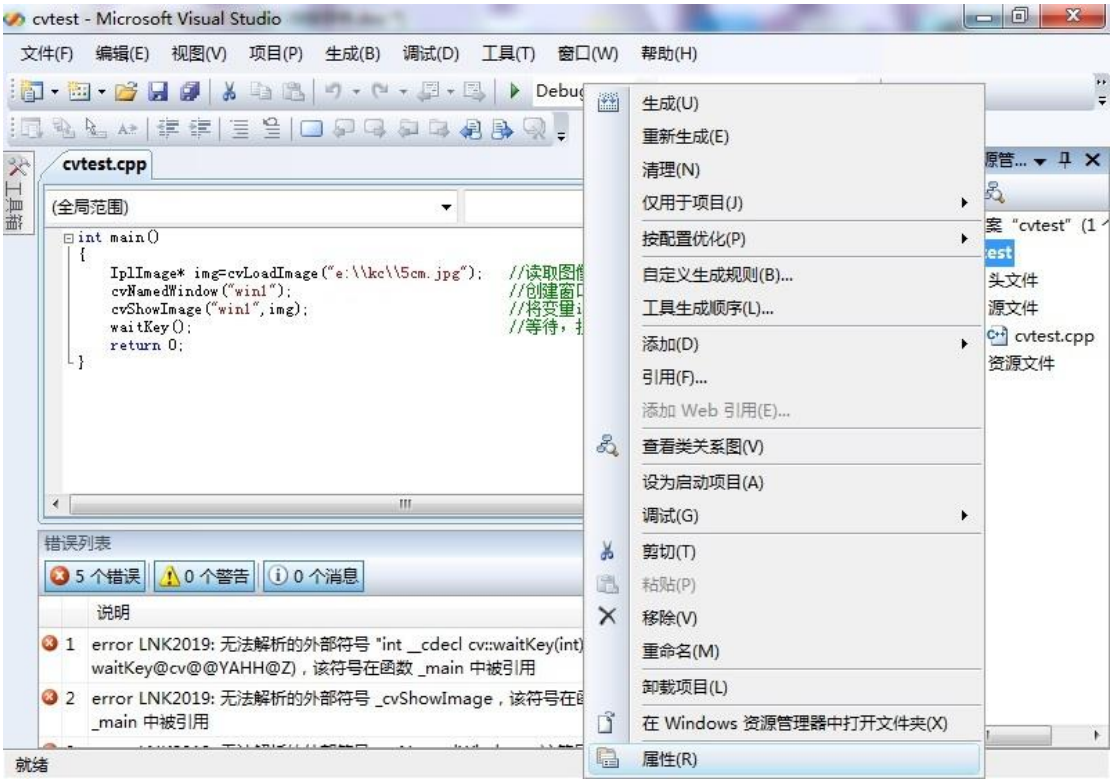


为了演示简便，这里创建为空项目

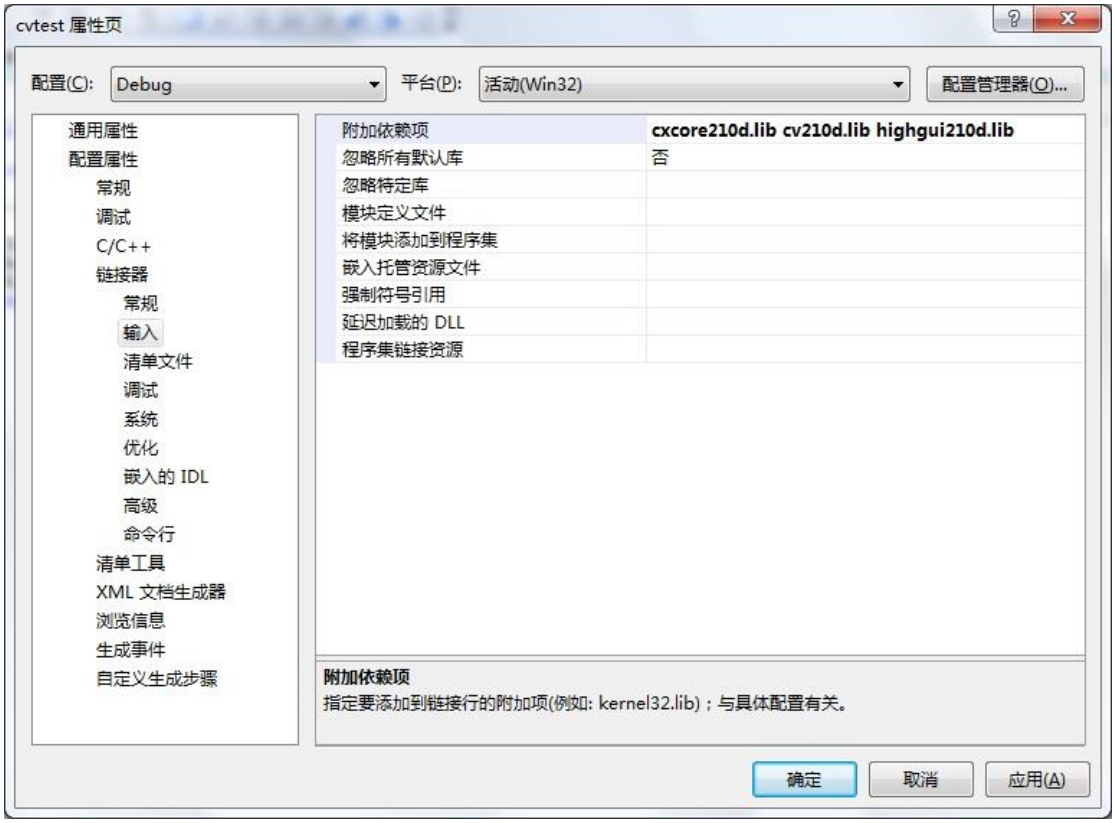
输入一些基本的测试代码，运行后提示有错误，是由于缺少库文件



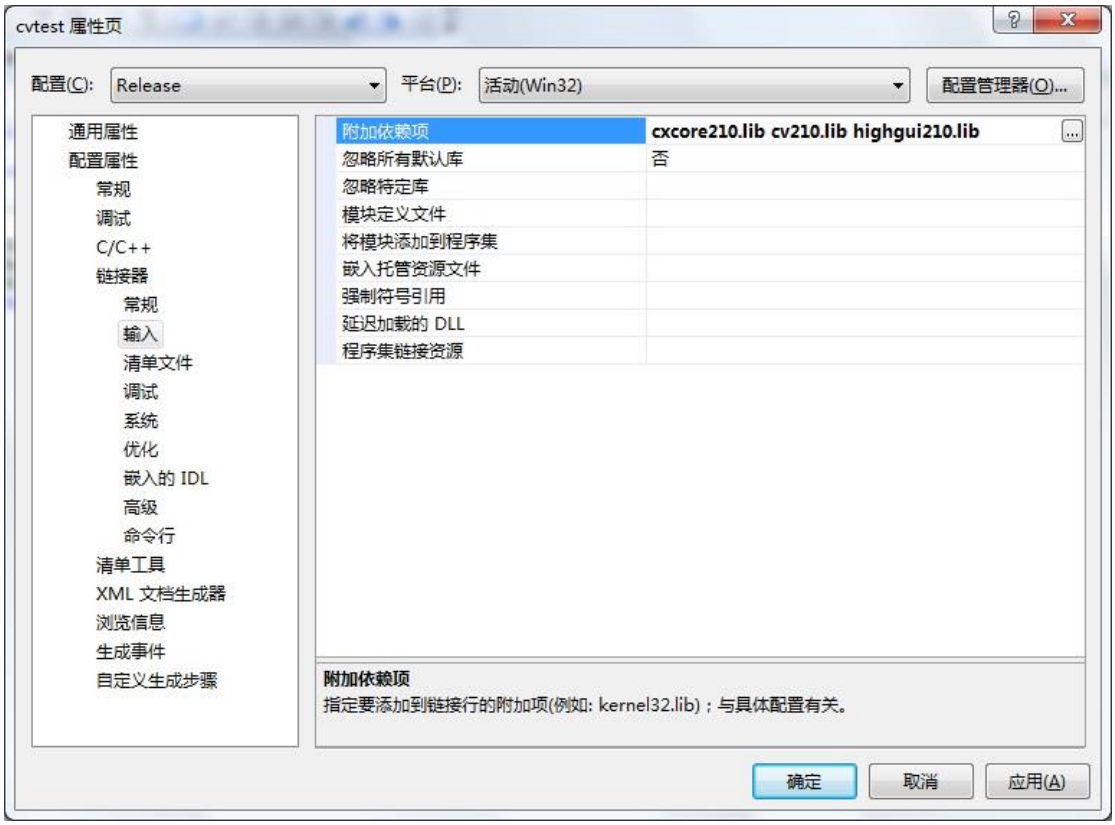
右键点击项目名称，选择属性



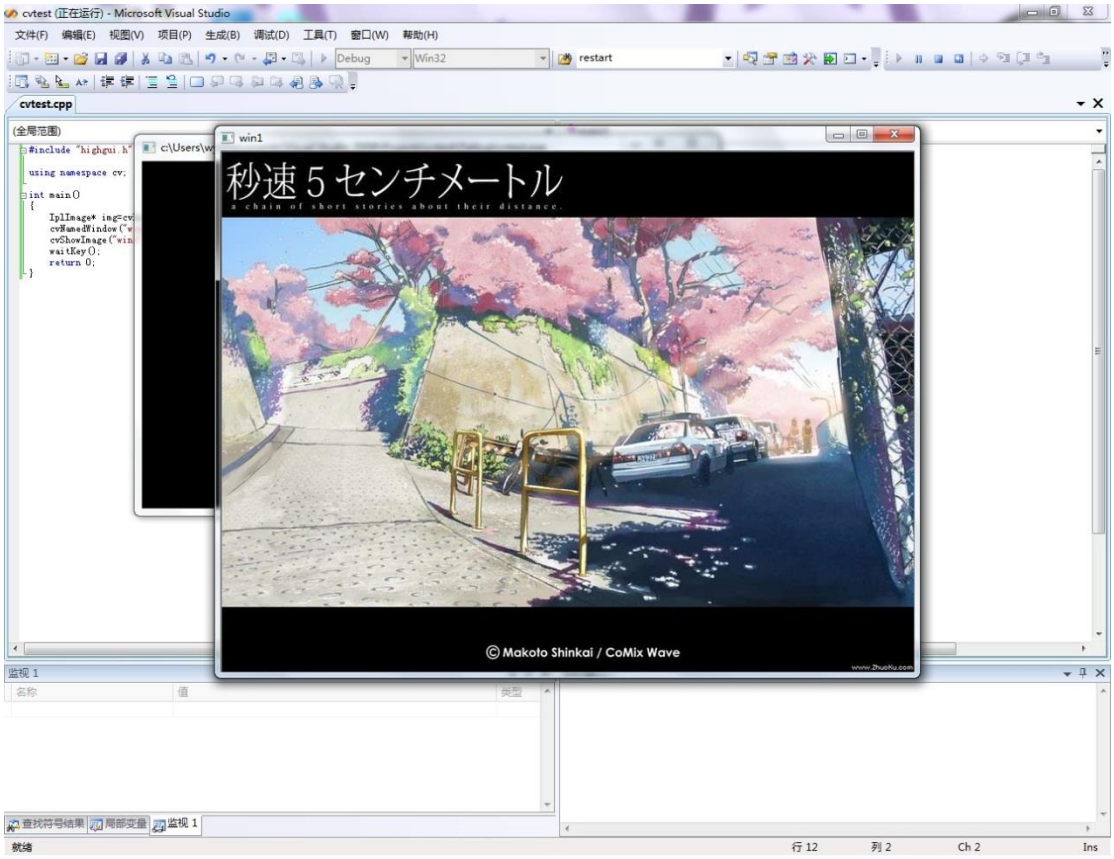
将 OpenCV 的库文件添加进项目。先添加 debug 版本的库文件



再添加 release 版本的库文件

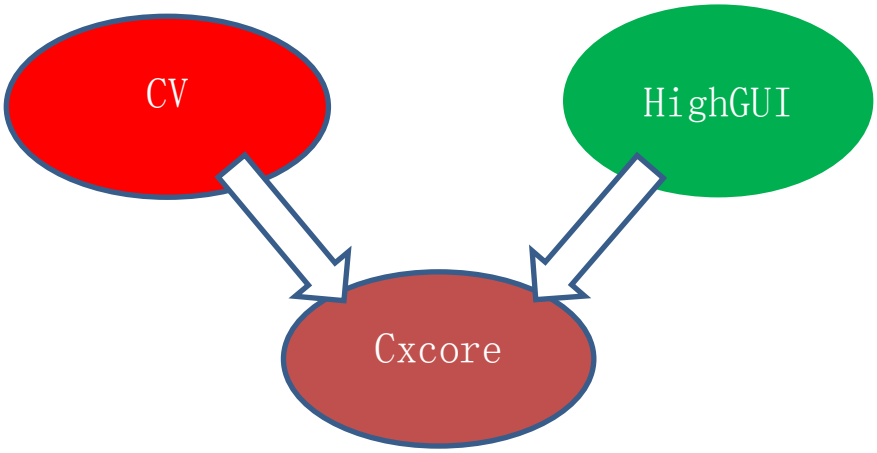


确定之后就能正常运行程序



OpenCV 的架构

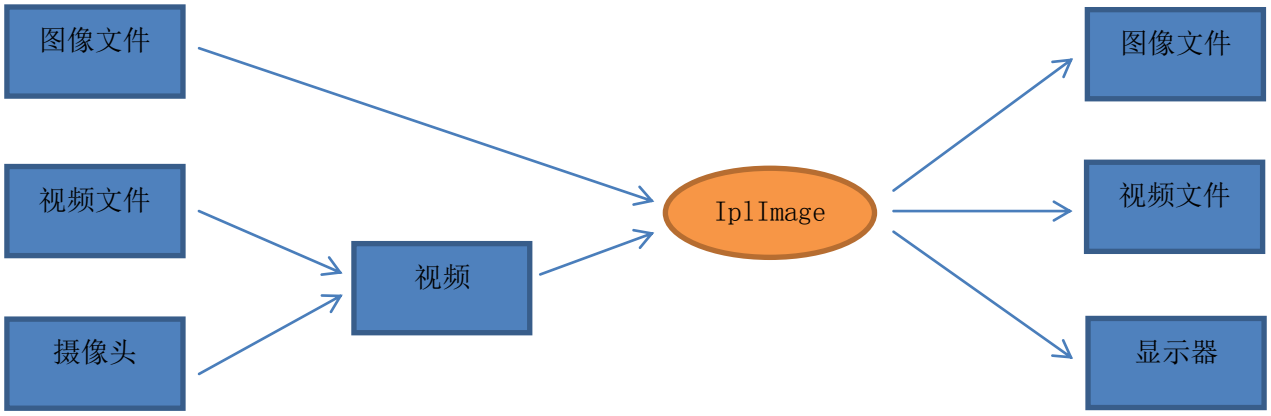
设置好 OpenCV 以后，我们了解一下 OpenCV 的基本架构。最常用的有三个模块，分别是 CV、HighGUI 以及 Cxcore。他们的关系如下：



Cxcore 包括了基本的数据结构及运算，是整个 OpenCV 的基础。CV 模块主要包括各种常用的图像处理算法，能极大地简化程序的编写，使用时需要#include "cv.h"。HighGUI 模块可以用于实现简易的图形界面，以及非常便利的图像和视频文件读写，并且支持摄像头作为视频输入，使用时需要#include "highgui.h"。由于后面两个模块都是以 Cxcore 作为基础的，因此只要使用了这两个模块的任意一个，就无需再包含 cxcore 的头文件。

HighGUI 模块 IO

首先介绍 HighGUI 模块。HighGUI 模块除了实现简易的图形界面，也实现了方便的输入输出。



上图展示了输入输出的几种途径，下面分别说明。

输入

输入分为两大类，图像输入与视频输入。

图像输入是从图像文件中读取图像，使用 cvLoadImage 函数，以图像文件名作为参数，如果读取成功，将得到储存着图像的 IplImage 类型的指针。

视频就是连续播放的图像，其中一幅图像称作一帧。OpenCV 对于视频的处理是通过对视频中的每幅图像分别处理实现的，因此视频输入最终也是得到 IplImage 类型的指针。

视频输入的途径有两种，一种是从视频文件得到视频，另一种是从摄像头得到视频。对于视频文件，cvCreateFileCapture 函数以文件名作为参数，返回视频，即 CvCapture 类型的指针；对于摄像头，cvCreateCameraCapture 函数以摄像头编号作为参数（只有一个摄像头可以输入-1）返回视频。成功得到视频之后，使用 cvQueryFrame 从视频中得到一帧图像，并返回该图像的 IplImage 指针。处理后再用 cvQueryFrame 得到下一帧图像。

输出

输出分为三类，图像文件输出是最简单的。只需要使用 cvSaveImage 函数，并且提供文件名与 IplImage 指针即可。

要输出视频文件，首先要创建写入器，用 cvCreateVideoWriter 函数，并且提供文件名、编码器（输入-1 可以在运行时从列表中选择）、帧率（每秒播放多少幅图像）以及视频分辨率（CvSize 类型，可以简单的用 cvSize(width, height)函数构造）。反复调用 cvWriteFrame 函数将每一帧图像输入写入器。最后不要忘记用 cvReleaseVideoWriter 函数关闭写入器，结束视频文件的写入。*在初期需要大量试验时，把小车运行的视频输出至视频文件保存起来，以后每次试验时可以直接读取视频文件进行初步测试，然后再用小车实验。*

因为通常编写的是控制台程序，要在显示器上输出图像或视频，首先要创建一个窗口。cvNamedWindow 函数可以创建一个窗口，其参数为窗口的名字。然后可以用 cvShowImage 函数将一个 IplImage 指针指向的图像显示在指定名字的窗口中，这样就显示了静态的图像。如果要显示视频，则通过不断的调用 cvShowImage 函数，将视频的每一帧图像显示出来即可。

IplImage 介绍

根据前面的介绍，各种输入输出方式都离不开图像，而且 OpenCV 主要用于图像处理，因此保存图像的数据结构是非常重要的。在 Cxcore 中提供了 IplImage 结构用于保存图像。其重要的成员变量有：

width——图像的宽度

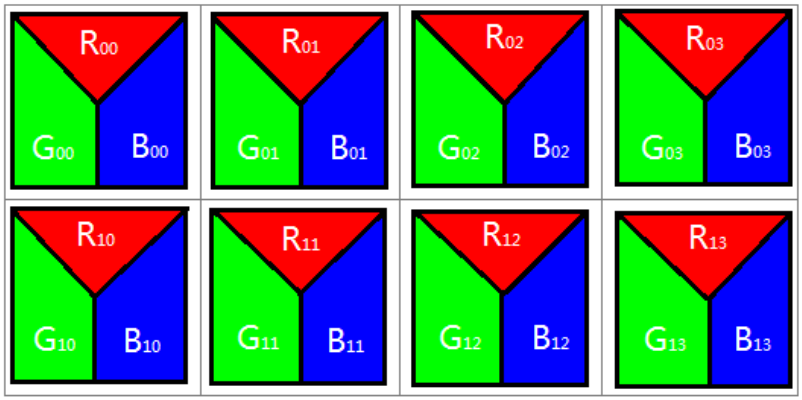
height——图像的高度

imageData——图像数据

宽度与高度是图像的基本属性，以像素为单位，限制了像素坐标的最大值，因此多用于循环遍历像素时的终止条件。

图像数据是一个一维数组，数组的每个元素保存了一个像素的某一通道的值。对于像素来说是按行优先的顺序储存，对于通道来说是交叉储存的。下图说明了这一点。

一幅4*2的彩色图像：



上面的图像在一维数组中的表示

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
B_{00}	G_{00}	R_{00}	B_{01}	G_{01}	R_{01}	B_{02}	G_{02}	R_{02}	B_{03}	G_{03}	R_{03}	B_{10}	G_{10}	R_{10}	B_{11}	G_{11}	R_{11}	B_{12}	G_{12}	R_{12}	B_{13}	G_{13}	R_{13}

因此已知某像素的二维坐标，访问该像素时，首先要把二维坐标转换为一维的数组下标。图像的左上角为原点，x 轴向右，y 轴向下，则从原点到(x,y)点有 y*width+x 个像素。

对于灰度图像，由于只有一个通道，因此每个像素在数组中只占一个元素，用上面的公式得到的像素个数就可以直接作为下标访问该像素的灰度值。而对于彩色图像，由于每个像素占用三个数组元素，因此像素个数*3 才是该像素的第一个分量数组下标，即蓝色，随后两个元素是像素的绿色和红色分量。

上述计算过于繁琐，使得代码可读性较差，因此将这些操作包装起来，得到简洁直观的代码。具体程序如下：

```
template<class T> class Image {
    private:
    IplImage* imgp;
    public:
    Image(IplImage* img=0) {imgp=img;}
    ~Image(){imgp=0;}
    inline T* operator[](const int rowIndx) {
        return ((T *)(imgp->imageData + rowIndx*imgp->widthStep));
    }
};
typedef struct{
    unsigned char b,g,r;
} RgbPixel;
typedef Image<RgbPixel>       RgbImage;
typedef Image<unsigned char>  BwImage;
```

将上面的代码放置在程序的最前面，使用起来非常简单。对于彩色图像，首先构造这个类,RgbImage imgclass(img),然后通过 imgclass[i][j].r 就可以访问位于图像 img 中(j,i)坐标的像素的红色分量，其他分量以此类推。对于灰度图像，也要先构造类 BwImage imgclass(img)，通过 imgclass[i][j]即可访问访问位于图像 img 中(j,i)坐标的像素灰度值。坐标与下标顺序不同，是由于 c 语言的程序中，第一个下标分量是指定某一行，第二个分量指定某一列，而行数正是图像的 y 轴坐标，列数正是图像的 x 轴坐标。

HighGUI 模块 UI

接下来继续介绍 HighGUI 模块,除了程序控制的输入输出以外,还有用户控制接口(User Interface)。包括键盘输入、鼠标输入以及滑块输入。

键盘

C 语言同样可以键盘输入，比如 cin、scanf 等等，但这与 HighGUI 模块的键盘输入不同。运行 OpenCV 程序时，如果用 cvNamedWindow 创建了窗口并且显示图像，则可以看到两个窗口。一个窗口是 C 语言程序运行时打开的控制台窗口，另一个是 OpenCV 程序中创建的显示图像的窗口。C 语言的输入只能在控制台窗口里进行，而 HighGUI 的输入则只能在图像窗口中进行。

HighGUI 的键盘输入函数为 cvWaitKey(time)，当程序运行到这条语句，将被暂停 time 毫秒。在这段时间内如果有某个键被按下，该函数会返回按下的键值，并且程序立刻继续运行；如果超过 time 毫秒仍没有键按下，函数将返回-1，并且继续运行。如果 time 等于 0 或者省略这个参数，程序将一直等待，直到有键按下。这个函数除了用于选择功能以外，还经常用于跳出循环。通常程序末尾也会写上这个函数,防止程序直接结束而来不及查看结果。

cvWaitKey 函数对于任意一个图像窗口都有效果，然而当没有图像窗口时，这个函数将被忽略，即不等待而直接返回-1。

鼠标

鼠标输入稍微复杂一点。首先要了解回调函数（Callback）。回调函数与用户编写的普通函数没有多少差别，但是除了用户可以调用以外，主要是由系统自动调用的。类似于某些语言中的事件函数。无论是鼠标移动还是点击，系统都会调用该函数，同时将鼠标的动作作为参数传递给函数。而系统的调用是有固定格式的，因此回调函数的函数头必须符合系统的规定，即参数的类型、数目、顺序都是固定的。OpenCV 鼠标输入的回调函数头格式为：void mouse(int mouseevent, int x, int y, int flags, void* param)，当然各个参数的名字可以自定。

在这个回调函数中，第一个参数用于接收鼠标的动作，由一组常量中的一个描述，如移动指针、按下左键、双击中间、松开右键等等动作。第二第三个参数表示鼠标指针的位置。第四个参数则表示鼠标以及扩展键的状态，由一组常量表示左键处于按下、Shift 键处于按下等等状态，联合第一个参数可以表示出按着 Ctrl 键并且用左键拖动等动作。第五个参数用于鼠标操作比较复杂的程序，课程中无需使用，但也必须定义出来。具体的常量值请参考

写好回调函数后，要将函数名提供给系统，系统才能调用。具体的方法是使用 cvSetMouseCallback(window, on_mouse)函数，第一个参数指定一个窗口，第二个参数指定回调函数名。调用这个函数后，当鼠标在指定窗口移动或点击时，就会调用回调函数进行处理，而对于未指定的窗口则忽略鼠标动作。通过多次调用这个函数，可以给多个窗口指定不同的回调函数，也可以指定为同一个回调函数。

下面这段程序演示了简单的鼠标操作，即在窗口中左键拖动

```
IplImage* img;                  //原始图像
IplImage* img1;                 //临时图像
int sx, sy;                     //拖动的起点
void mouse(int mouseevent, int x, int y, int flags, void* param)
{
    if (mouseevent == CV_EVENT_LBUTTONDOWN)      //按下左键
    {
        sx = x; sy = y;                          //记录当前位置，作为拖动的起点
    }
    if ((mouseevent == CV_EVENT_MOUSEMOVE) && (flags & CV_EVENT_FLAG_LBUTTON))
        //左键按下的状态移动鼠标，即左键拖动。用&是由于flags可能是组合状态
    {
        cvCopyImage(img, img1);                  //复制原始图像
        BwImage dat(img1);
        int up = (sy < y) ? sy : y;              //计算当前左键拖出的矩形的范围
        int down = (sy < y) ? y : sy;
        int left = (sx < x) ? sx : x;
        int right = (sx < x) ? x : sx;
        for (int i = up; i <= down; i++)
            for (int j = left; j <= right; j++)
                dat[i][j] = 255 - dat[i][j];     //将范围内的图像灰度反转
        cvShowImage("win1", img1);               //显示修改过的临时图像
    }
    if (mouseevent == CV_EVENT_LBUTTONUP)        //松开左键
        cvShowImage("win1", img);                //恢复显示原始图像
}
int main()
{
    img = cvLoadImage("e:\\kc\\5cm.jpg", 0);     //以单通道方式打开图像
    img1 = cvCreateImage(cvGetSize(img), 8, 1); //新建一幅同样大的临时图像
    cvNamedWindow("win1");                       //创建窗口
    cvShowImage("win1", img);                    //显示图像
    cvSetMouseCallback("win1", mouse);           //设置回调函数
    cvWaitKey();                                 //等待按任意键结束程序
    return 0;
}
```

滑块

接下来介绍滑块，滑块可以用拖动的方式连续改变某个变量的数值。配合一定的处理，可以实时看到修改后的结果。后面提到的很多参数都是用滑块进行动态调节，拖动滑块直至达到某种效果。

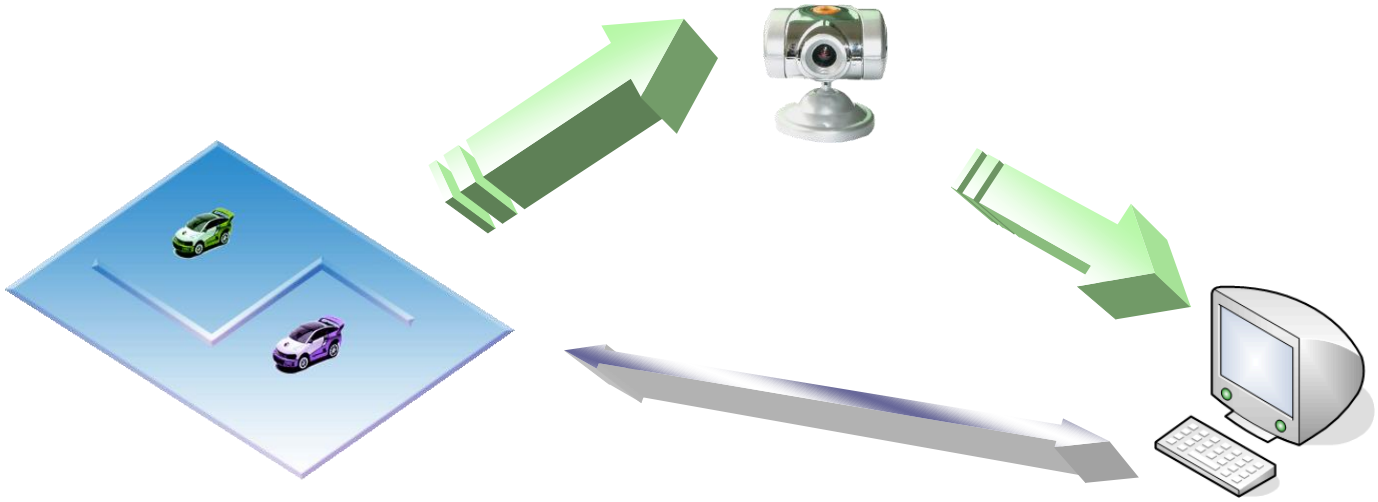
程序中使用 cvCreateTrackbar(trackbar_name, window_name, value, count, on_change)函数创建滑块。第一个参数指定滑块的名称，第二个参数指定滑块所在的窗口，第三个参数指定被滑块调整的变量，第四个参数为滑块的最大值。最后一个是回调函数，当调整滑块的值时被系统调用，如果没有回调函数，可以设置为 NULL。

滑块有多种用法，一种是直接读取被滑块调整的变量；另一种是间接读取滑块的值，通过 cvGetTrackbarPos 函数，指定窗口名和滑块名后返回滑块的值。最后一种就是通过回调函数，回调函数的函数头为 void trackbar(int value)，从参数中可以读取到滑块的当前值。

必须说明的是，要想使用鼠标与滑块，程序中不可缺少 cvWaitKey。因为只有在运行至 cvWaitKey，而使程序暂停的一段时间内，系统才能调用各种鼠标以及滑块的回调函数。例如上面鼠标输入的演示程序，当程序最后运行到 cvWaitKey 并无限等待按键时，鼠标操作才有效。因此如果在一段程序中没有 cvWaitKey 语句，则运行在这一部分的时候无法响应鼠标与滑块的输入。这也包括循环，看似很短的循环体，如果循环次数很多，并且没有 cvWaitKey，那么可能很长时间都无法响应鼠标与滑块输入。因此务必在需要及时响应的地方加入 cvWaitKey(1)这样的语句，而且几乎不会影响程序效率。

课程项目

本课程的题目有两种，小车走迷宫与小车走黑线。无论哪种，基本目的都是控制小车按照某一条线路行进。控制系统的示意图如下所示



摄像头拍摄整个地图；由计算机进行图像处理，确定小车的位置、方向等状态，并与计划的路线进行比较，决定继续前进还是修正姿态，通过有线或无线通信向小车发送指令；小车根据指令运动。

因此项目主要涉及的内容包括：图像采集、预处理、计算路径、跟踪姿态、小车控制、发送信号、接收信号、电机转动控制。计算机负责前 6 项，单片机负责后 2 项。

图像采集

项目中所有的图像都是从摄像头获取的，因此首先了解一下从摄像头获取图像的方法。根据前面介绍的 HighGUI 模块，主要使用 cvCreateCameraCapture 以及 cvQueryFrame 两个函数。

下面是一个最简单的例子：

```
int main()
{
    IplImage* img;                  //定义一个图像指针，后面将指向获取的图像
    cvNamedWindow("win1");          //新建一个窗口用于显示图像
    CvCapture* cam = cvCreateCameraCapture(-1); //初始化摄像头
    if (!cam) return 0;             //初始化失败的话退出程序
    img = cvQueryFrame(cam);        //获取一帧图像，并且用img指向它
    cvShowImage("win1", img);       //将图像显示在之前创建的窗口上
    cvWaitKey();                    //等待按下任意键继续
    cvReleaseCapture(&cam);         //释放摄像头，关闭程序前要释放各种资源
    return 0;
}
```

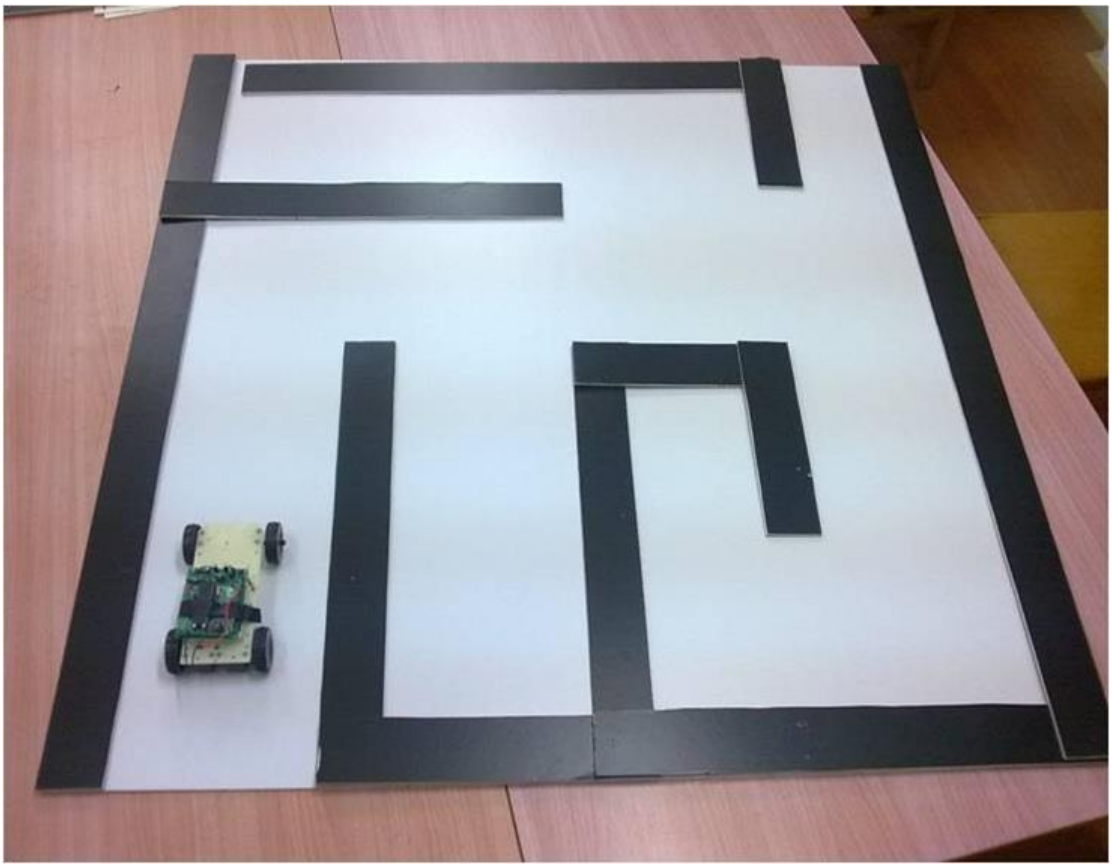
上面的程序打开摄像头，并且获取一帧图像，然后按下任意键退出程序。获取到图像之后可以进行各种操作，上例只是将原始图像显示出来。如果想获得连续图像，显然就是使用循环语句，不停地进行获取——处理——获取，如下所示：

```
int main()
{
    IplImage* img;                      //定义一个图像指针，后面将指向获取的图像
    cvNamedWindow("win1");              //新建一个窗口用于显示图像
    CvCapture* cam = cvCreateCameraCapture(-1); //初始化摄像头
    if (!cam) return 0;                 //初始化失败的话退出程序
    while (1)                           //无限循环
    {
        img = cvQueryFrame(cam);        //获取一帧图像，并且用img指向它
        cvShowImage("win1", img);       //将图像显示在之前创建的窗口上
        if (cvWaitKey(1)>=0) break;     //等待1毫秒，如果期间按下任意键则退出循环
    }
    cvReleaseCapture(&cam);             //释放摄像头，关闭程序前要释放各种资源
    return 0;
}
```

这里需要注意的是 if (cvWaitKey(1)>=0) break。如果缺少这句话，一方面无法正常退出循环，从而变成死循环；另一方面，系统还没来得及在窗口上显示一幅图像，程序已经又循环了一次，系统又被要求显示下一幅图像，从而根本无法显示图像。而 cvWaitKey(1) 一方面给系统一定时间显示图像，另一方面给用户一定的时间退出循环。

预处理——透视变换

得到原始图像之后，首先遇到并且要解决的是视角问题。由于摄像头无法摆在地图的正上方，因此从摄像头的视角获取的图像，都有着明显的透视效果。例如下图，原本是方形的迷宫，却呈现出梯形。



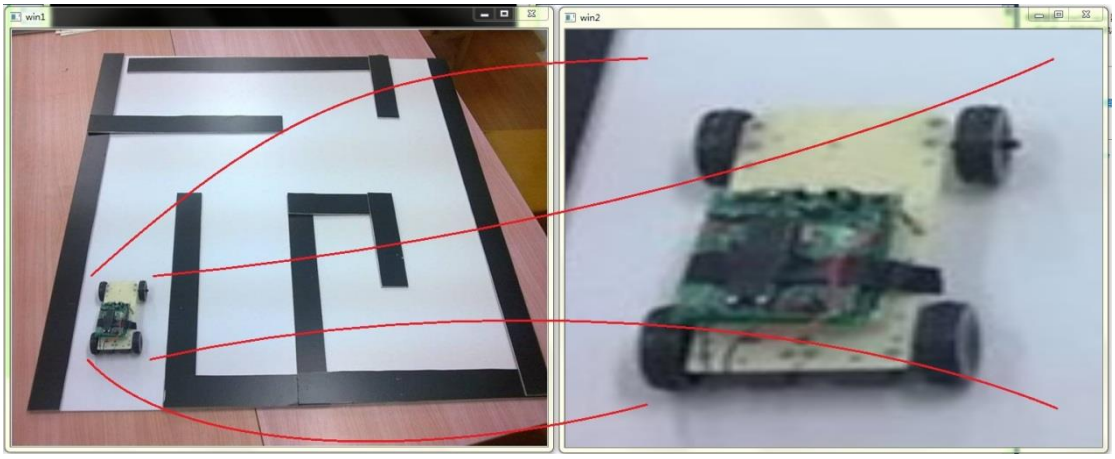
虽然人眼能够轻易适应这种视角，但是用程序进行平面图像处理的时候，梯形比矩形麻烦得多，因此首先要消除透视效果。

OpenCV 对于图像的几何变换，在 CV 模块中提供了一组函数，最灵活的是 WarpPerspective 以及 GetPerspectiveTransform，通过这两个函数，可以对图像进行任意变形。具体的变形方法是选择四个原始图像中的点，然后再指定这四个点的新位置，则整张图像将根据这四个点的相对位移进行变换。简单的演示程序如下：

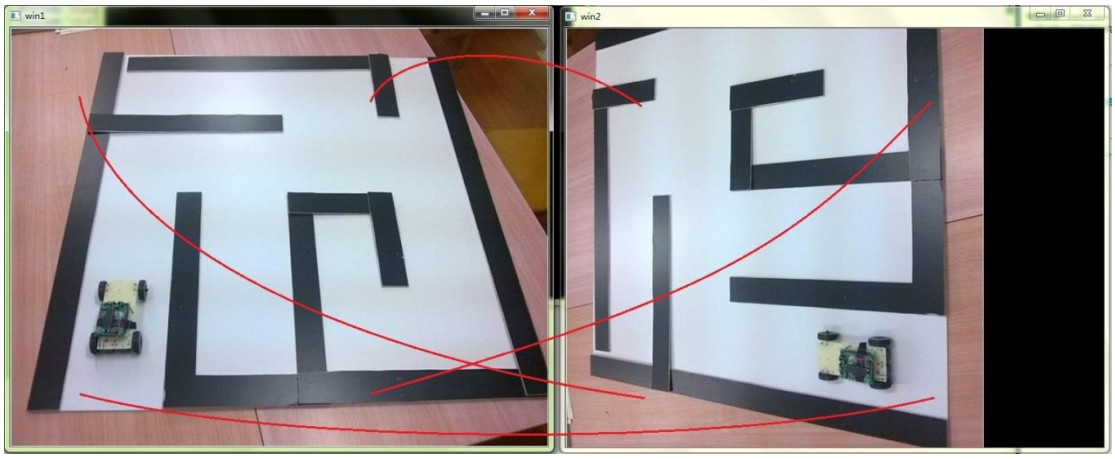
```
int main()
{
    IplImage* img = cvLoadImage("E:\\kc\\trans10.jpg");  //原始图像
    IplImage* transimg = cvCreateImage(cvSize(400, 400), IPL_DEPTH_8U, 3);
                    //创建一个400*400的24位彩色图像，保存变换结果
    CvMat* transmat = cvCreateMat(3, 3, CV_32FC1);       //创建一个3*3的单通道32位
浮点矩阵保存变换数据

    CvPoint2D32f originpoints[4];                //保存四个点的原始坐标
    CvPoint2D32f newpoints[4];                   //保存这四个点的新坐标
    originpoints[0] = cvPoint2D32f(139, 43);
    newpoints[0] = cvPoint2D32f(20, 20);
    originpoints[1] = cvPoint2D32f(585, 49);
    newpoints[1] = cvPoint2D32f(380, 20);
    originpoints[2] = cvPoint2D32f(47, 499);
    newpoints[2] = cvPoint2D32f(20, 380);
    originpoints[3] = cvPoint2D32f(720, 476);
    newpoints[3] = cvPoint2D32f(380, 380);

    cvGetPerspectiveTransform(originpoints, newpoints, transmat);      //根据四个点
计算变换矩阵
    cvWarpPerspective(img, transimg, transmat); //根据变换矩阵计算图像的变换

    cvNamedWindow("win1");
    cvShowImage("win1", img);                    //显示原始图像
    cvNamedWindow("win2");
    cvShowImage("win2", transimg);               //显示变换后的图像
    cvWaitKey();
    return 0;
}
```

下面演示几种变换的结果：

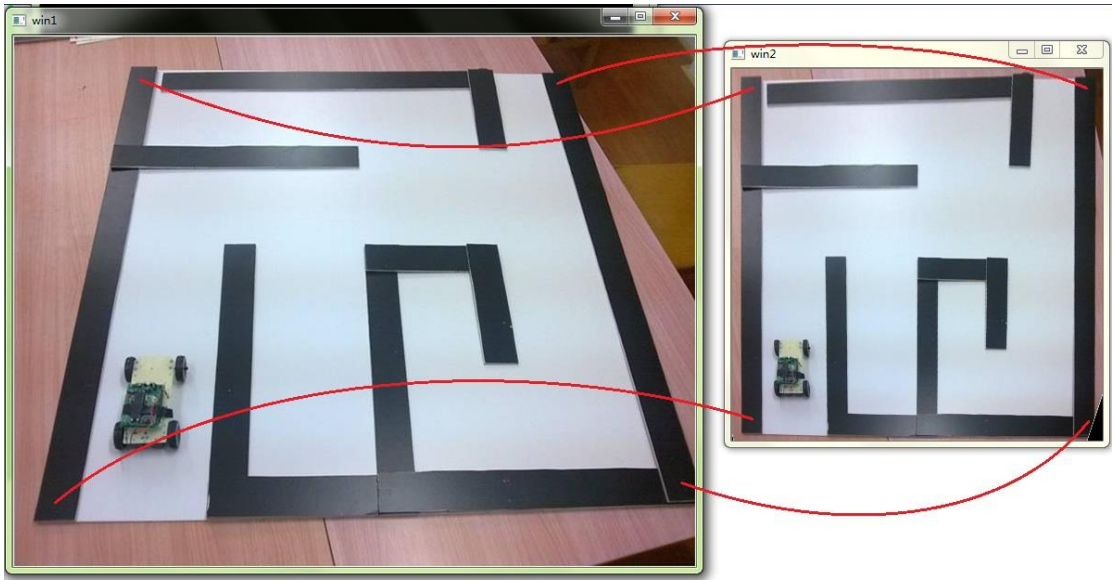
通过拉伸四个点的距离，可以放大图像



通过旋转四个点的位置，可以旋转整幅图像。



将地图的四个顶点的新位置指定为图像的四个顶点，就可以将地图旋转放大到刚好填满图像，而且修正了透视效果。



实际使用时，由于每次摄像头摆放的位置都不同，因此一般用鼠标输入的方法直接在原始图像中选择四个点，而四个点的新位置可以在程序中固定为图像的四个角。

计算路径

二值化

经过透视变换之后，我们得到了很规则的地图。但是由于光照不均匀、材质颜色不均匀、摄像头的随机杂色“噪音”等等因素，这个图像对于控制程序来说还是过于复杂。

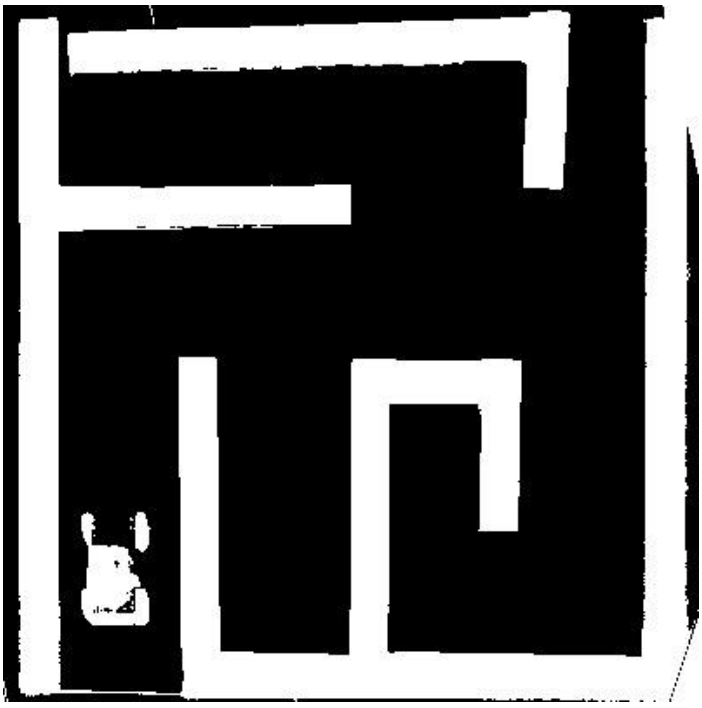
要计算路径，无论是走迷宫还是走黑线，首先要知道哪里能走。对于走迷宫的题目，只有黑线画出的迷宫边界不能通过，其他部分都是畅行无阻的；而走黑线的题目正好相反，只能沿着黑线前进，在黑线以外的地方都是禁区。

因此我们首先要做的是“确定能走的区域”。最简单的方法无非是将地图上的点分为两类，能走与不能走。这正好可以用二值图像表示，比如用 1 表示图像中能走的像素，用 0

表示不能走的部分，这种方法称作图像二值化。而根据前面的分析，两个题目的共同点都是要找到黑线，然后根据题目要求将其标记为能走或者不能走的区域。则任务进一步明确为“找黑线”。

最理想的情况，图像中黑线的像素都是(0,0,0)，即纯黑色。但这种情况过于理想，实际中仅仅由于反光影响，图像中部分黑线的颜色就不会是纯黑。因此可以放宽一些条件，比如三个通道都小于 50（这个数字称为阈值），即允许稍微偏离黑色。将黑线区域标记为白色，得到如下结果：

对比原图像，发现很多黑线区域没有被包括进来，因此不断扩大选择范围进行试验。这时就用到了滑块，直接拖动就可以调整阈值。直到将三个分量均小于 150 的像素作为黑线区域，得到了如下的实验结果：



可以看出黑线基本被完全标记出来，但是也有少量的干扰。比如小车也被包括进来，因此计算路径的时候先不要摆放小车。还有一些其它问题，就需要根据实际情况选择一种或者综合多种方法。比如黑线的判断条件，不仅限制上界，也同时限制其下界；透视变换的时候，尽量把图像用地图填满，而不要露出地图以外的部分；考虑光线或者镜头有一些偏色，或者“黑线”用的不是黑色材料，就需要单独调整每个通道的阈值等等。

具体实现通过 OpenCV 提供的 cvInRangeS 函数，参数中上界与下界都是 CvScalar 类型（可通过 CV_RGB(r,g,b)构造），而参数中保存结果所用的图像必须是与原图像相同大小的单通道图像。

走迷宫

找黑线的任务完成之后，就要根据题目要求计算路径。

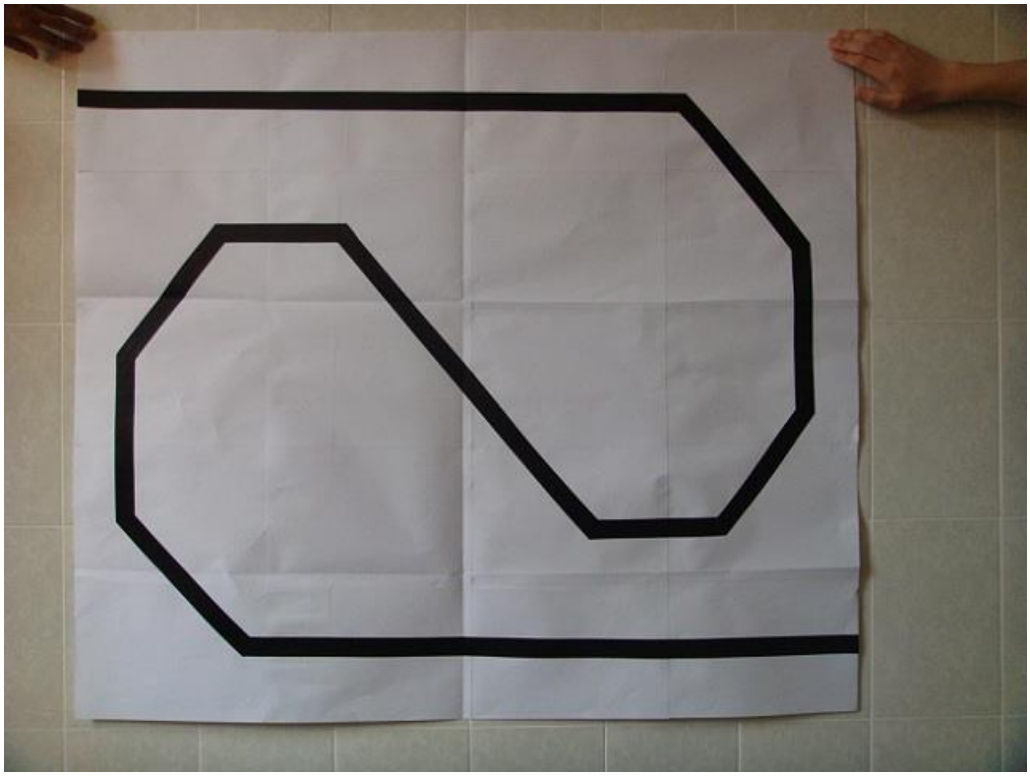
对于走迷宫的题目来说，如果以前接触过类似的程序问题会容易很多。基本思想是将实际的迷宫抽象成二维数组，每个元素代表迷宫中的一格，如果某格是墙壁，则将-1 存入该元素，若不是墙壁，则将 0 存入。最后用任意一种搜索算法找出路径即可。

为了处理简便，在制作迷宫的时候就要尽量将墙壁设计为水平与垂直的，确保所有道路的宽度是一格的整倍数（上图的迷宫就不易处理，道路宽度不一致）。然后判断相邻两格的中心连线上有没有墙壁即可得到迷宫的结构。

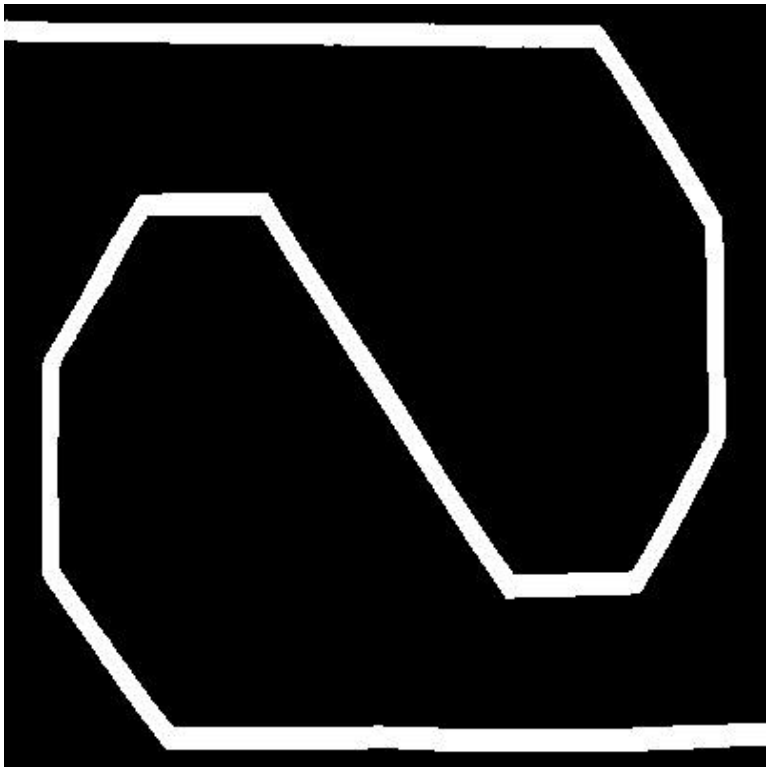
上面只是处理的方法之一，更多方法可以参考课程网站上历年的设计报告。

走黑线

对于走黑线的任务，一般是先将每条直线段确定下来。例如有下面的原始图像

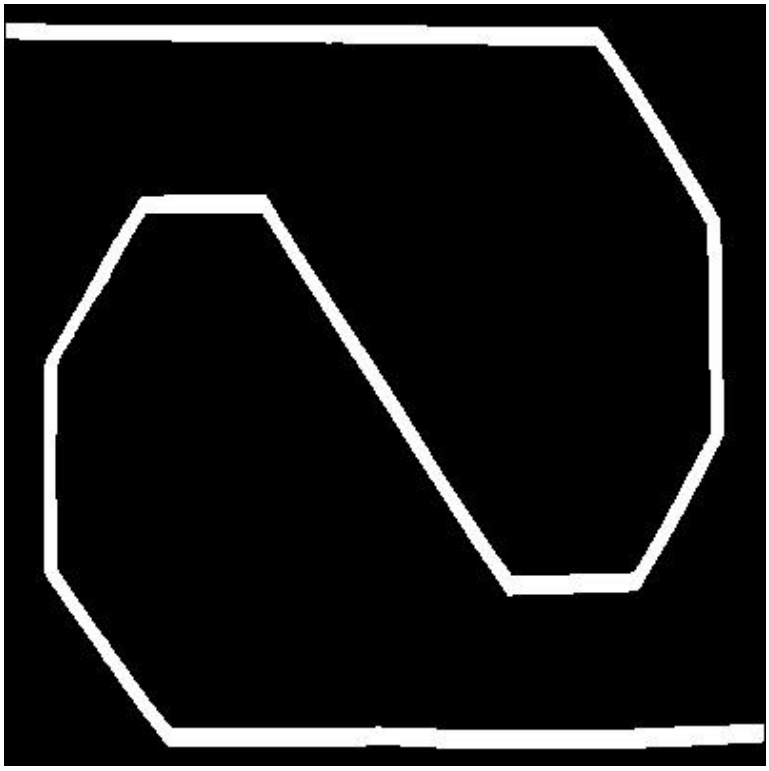


经过透视变换和二值化之后，将黑线区域标记了出来

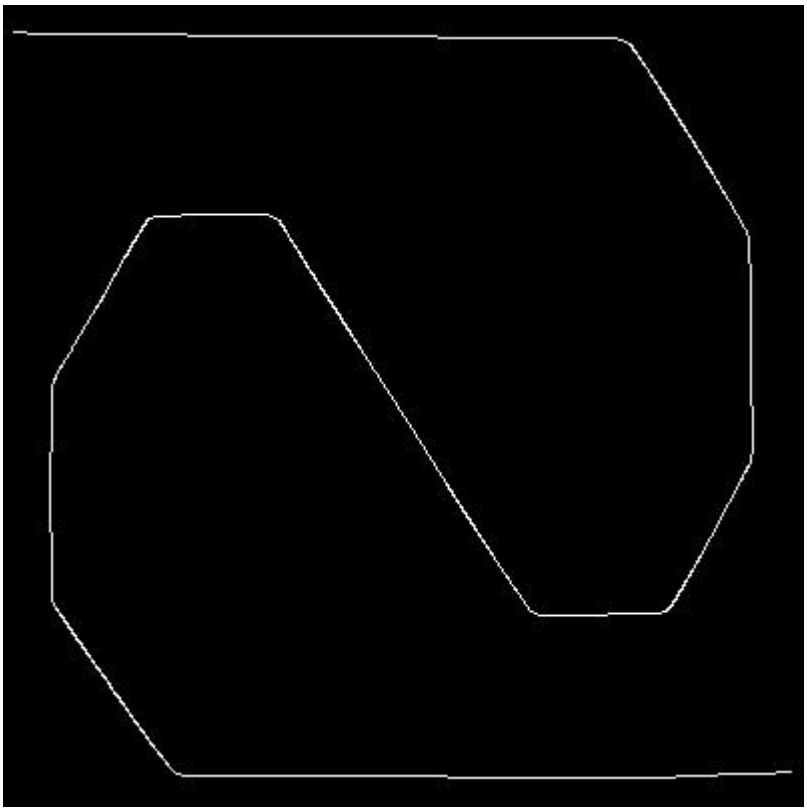


细化

二值化后的图像可以应用一类图像处理方法，称作形态学图像处理。细化就是其中的一种，细化在直观上就是让图像变细，但仍然保持原来的连接性。细化是一个迭代的过程，每次只能将图像消去“一圈”，如下图是进行 1 次细化的结果



可以看出 1 次细化与原图差别并不大。下面进行了 6 次细化



可以看到图像已经只剩主干，所有线段的宽度都是 1 像素。进行更多次的细化，结果不会再发生变化，因为所有线段都只有 1 个像素，不可能在保持连接的情况下去除任何一个像素了。

OpenCV 并没有提供细化函数，因此这里提供一个根据 T. Y. Zhang 等人在论文 A fast parallel algorithm for thinning digital patterns 中提出的细化算法编写的函数，可以直接用在程序里调用。参数中的输入输出图像都必须是 8 位单通道灰度图像，第三个参数指定细化的次数。

```
void cvThin (IplImage* src, IplImage* dst, int iterations = 1) {
    cvCopyImage(src, dst);
    BwImage dstdat(dst);
    IplImage* t_image = cvCloneImage(src);
    BwImage t_dat(t_image);
    for (int n = 0; n < iterations; n++)
        for (int s = 0; s <= 1; s++) {
            cvCopyImage(dst, t_image);
            for (int i = 0; i < src->height; i++)
                for (int j = 0; j < src->width; j++)
                    if (t_dat[i][j]) {
                        int a = 0, b = 0;
                        int d[8][2] = {{-1, 0}, {-1, 1}, {0, 1}, {1, 1},
                                       {1, 0}, {1, -1}, {0, -1}, {-1, -1}};
                        int p[8];
                        p[0] = (i == 0) ? 0 : t_dat[i-1][j];
                        for (int k = 1; k <= 8; k++) {
                            if (i+d[k%8][0] < 0 || i+d[k%8][0] >= src->height ||
                                j+d[k%8][1] < 0 || j+d[k%8][1] >= src->width)
                                p[k%8] = 0;
                            else p[k%8] = t_dat[ i+d[k%8][0] ][ j+d[k%8][1] ];
                            if (p[k%8]) {
                                b++;
                                if (!p[k-1]) a++;
                            }
                        }
                        if (b >= 2 && b <= 6 && a == 1)
                            if (!s && !(p[2] && p[4] && (p[0] || p[6])))
                                dstdat[i][j] = 0;
                            else if (s && !(p[0] && p[6] && (p[2] || p[4])))
                                dstdat[i][j] = 0;
                    }
        }
    cvReleaseImage(&t_image);
}
```

Hough 变换

通过细化得出主干路线之后，用 Hough 变换在图像中寻找直线。Hough 变换的基本原理是这样的：

假设图像中有一条直线 $y=Ax+B$，则直线上某一点(x_0, y_0)必符合该方程，即 $y_0=Ax_0+B$。我们要找出这条直线，但是穿过这一点的直线可以有无数条，它们都满足 $y_0=ax_0+b$，将其换一

种形式，变为 $b=-x_0a+y_0$。这个方程可以看作 ab 坐标系下的一条直线，直线上的任意一个点都代表了一组 a、b 的值，将这组值带入 y=ax+b 就构成 xy 坐标系下的一条直线，而且是一条穿过(x_0, y_0)点的直线。再考虑 y=Ax+B 上的另一点(x_1, y_1)，同样的将穿过它的所有直线用 $b=-x_1a+y_1$ 表示，得到了 ab 坐标系下的另一条直线，则这两条 ab 坐标系下的直线必有一个交点，而且焦点必为(A,B)。因为只有 y=Ax+B 这条直线同时穿过了(x_0, y_0)与(x_0, y_0)，因此只有(A,B)同时符合 $b=-x_0a+y_0$ 与 $b=-x_1a+y_1$。

根据这个基本原理，如果图像中只有一条直线，则直线上每个点都对应 ab 坐标系下的一条直线，而且所有的直线相交于同一点，找到这个交点，就可以得到图像中直线的参数，从而找出这条直线。

如果图像中的直线多于一条，就会出现两种交点。第一种交点如上所述，是由位于同一条直线上的点对应的 ab 坐标系直线构成的交点。另一种则是由不在同一条直线上的点对应的 ab 坐标系直线构成的交点。这是由于两点定一线，所以图像中任意两点都会在 ab 坐标系中产生一个焦点，从而得到一条直线。但是这条直线是否真的在图像中存在，就要看有没有位于该直线的第三个点、第四个点或者更多，也就是看 ab 坐标系中相交于该焦点的直线数目，如果大于某一阈值，就认为这条直线确实在图像中存在。

用类似的原理，也可以找出图像中的圆、正弦曲线等等任何能写出方程的图形。

CV 模块提供了用 Hough 变换找直线的函数 cvHoughLines2。下面是一段示例代码。

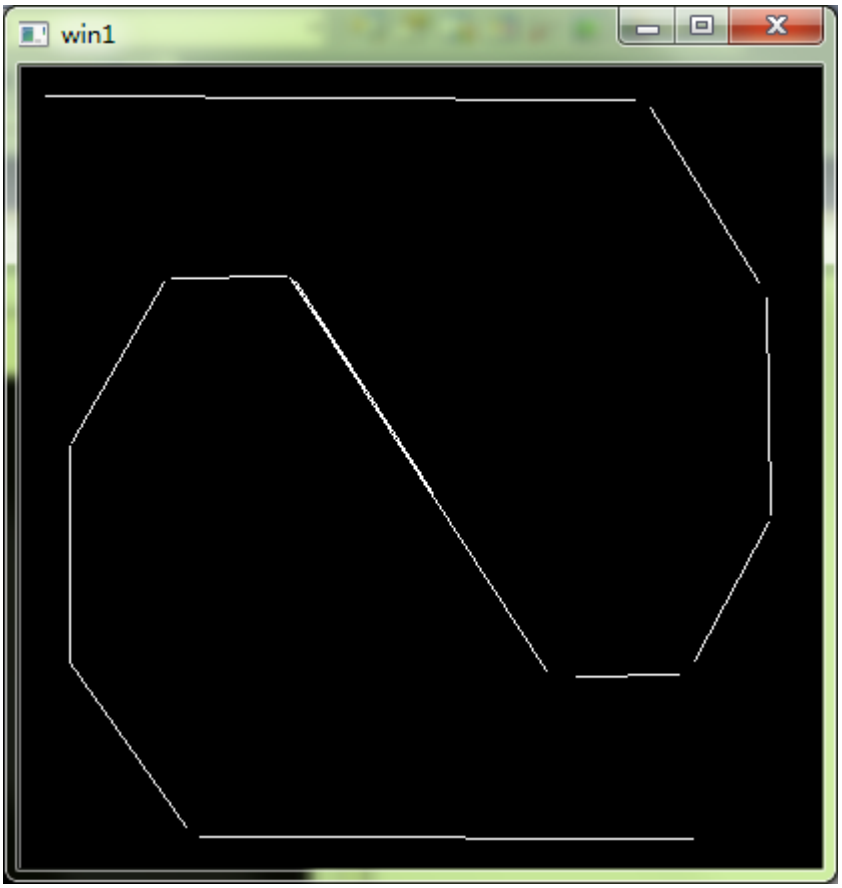
```
int main()
{
    IplImage* img = cvLoadImage("E:\\kc\\line.bmp", 0);       //以单通道方式打开图像，返回单通道灰度图
    cvNamedWindow("win1");
    CvMemStorage* storage = cvCreateMemStorage();             //创建一片内存区域存储线段数据
    CvSeq* lines = cvHoughLines2(img, storage, CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 30, 40, 40); //Hough变换找直线
    IplImage* img1 = cvCreateImage(cvGetSize(img), 8, 1);     //新建一幅同样大的图像，用于画出找到的直线段
    cvSetZero(img1);                                          //填充为黑色
    for(int i = 0; i < lines->total; i++)
    {
        CvPoint* line = (CvPoint*) cvGetSeqElem(lines, i);    //读取第i条线段的两个端点
        cvLine( img1, line[0], line[1], cvScalar(255));       //用白色画出这条线段
    }
    cvShowImage("win1", img1);
    cvWaitKey();
    return 0;
}
```

cvHoughLines2 函数的第一个参数是输入图像，必须是单通道图像。第二个参数是存储数据的内存区域，要先用 cvCreateMemStorage 函数构造。第三个参数在课程中通常用 CV_HOUGH_PROBABILISTIC，这种变换可以直接得到线段的两个端点。第四个和第五个参数设定精度，一般不用变。第六个参数是前面提到过的判断一条直线是否确实存在的阈值，需要

通过实验选择一个合适的值。第七个参数用于限定最短的线段长度，不到这个长度的线段将被忽略。最后一个参数表示线段间隔的最大值，位于同一直线的两条线段，如果它们的距离小于这个参数，将被连接成为一个线段。

通常最后三个参数都是通过实验选择的，基本要求就是尽量找到所有线段，并且找出的线段都是实际存在的。

将之前细化至 1 像素的黑线图像作为输入，上面的演示程序得到的结果如下：



可以看到线段都正确的找到了。但有一些瑕疵，比如有些线段没有严格的首尾相接，有时找到了重复的线段（上图中部）。因此在找到线段后要清理重复线段，基本思想就是判断是否一条线段的两个端点都几乎位于另一条线段上。

指定某一线段的某一端点为起点，则这条线段就是路径的第一步。然后距离该线段另一端点最近的线段为路径的第二步。以此类推，计算出完整的路径。

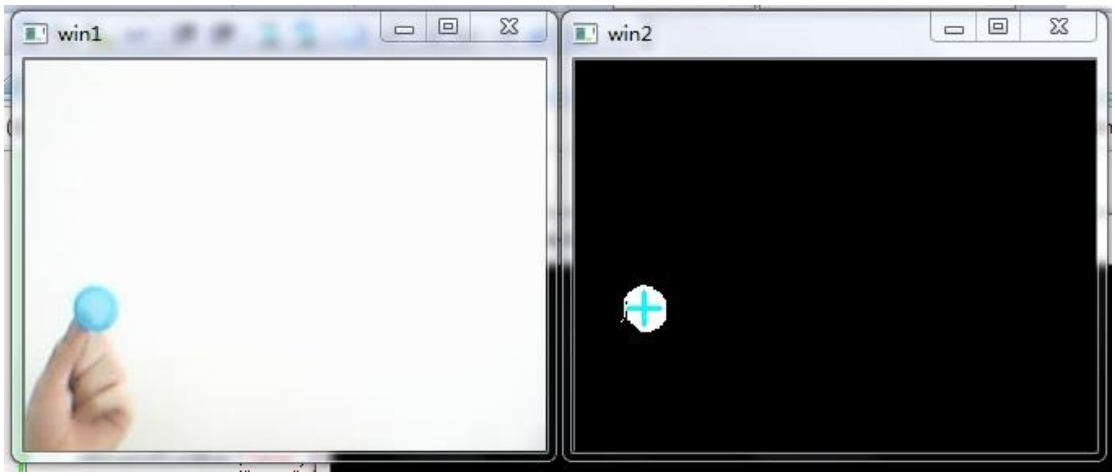
跟踪姿态

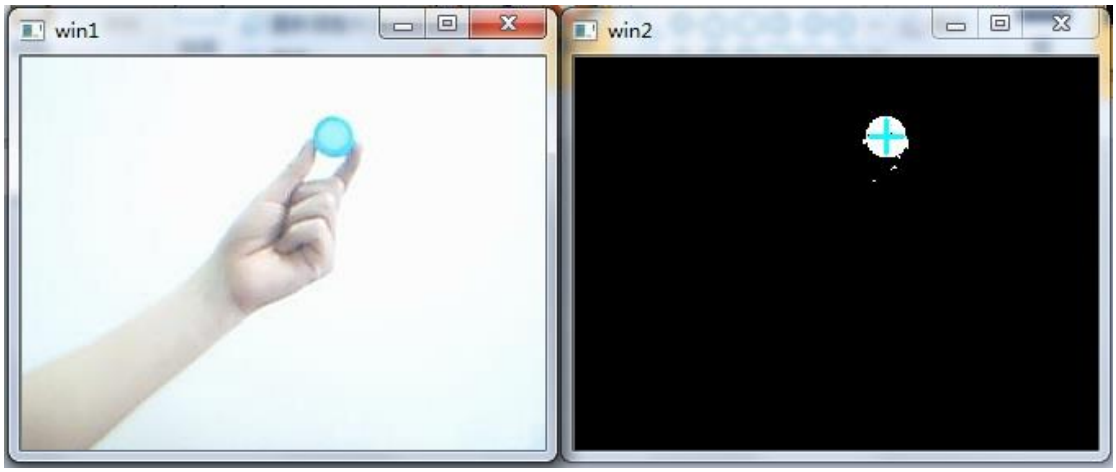
确定路径之后，就可以将小车放在地图的起点，进行实时控制。要进行控制，必须获取小车当前的姿态，具体包括小车的位置与方向。给小车装上某种颜色的外壳，一方面美观，另一方面将小车用颜色标记出来，便于获取姿态。

为了获取姿态，首先要找出小车。找到小车之后，小车的位置自然已知，进而可以确定小车的方向。而小车被某种颜色的外壳覆盖，因此问题简化为在地图中找到某种颜色区域。

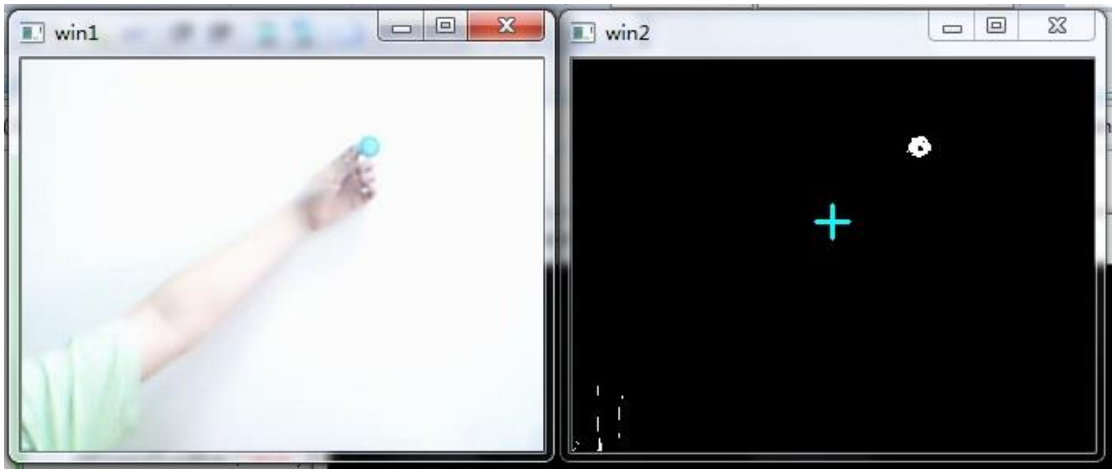
二值化

找颜色区域的方法有很多种，例如之前用二值化的方法找黑线，这里也可以用二值化的方法将具有某种颜色的像素全部标记出来，然后对所有的标记像素坐标求平均值，就得到了标记区域的中心坐标。这种方法代码简单，而且效果不错。

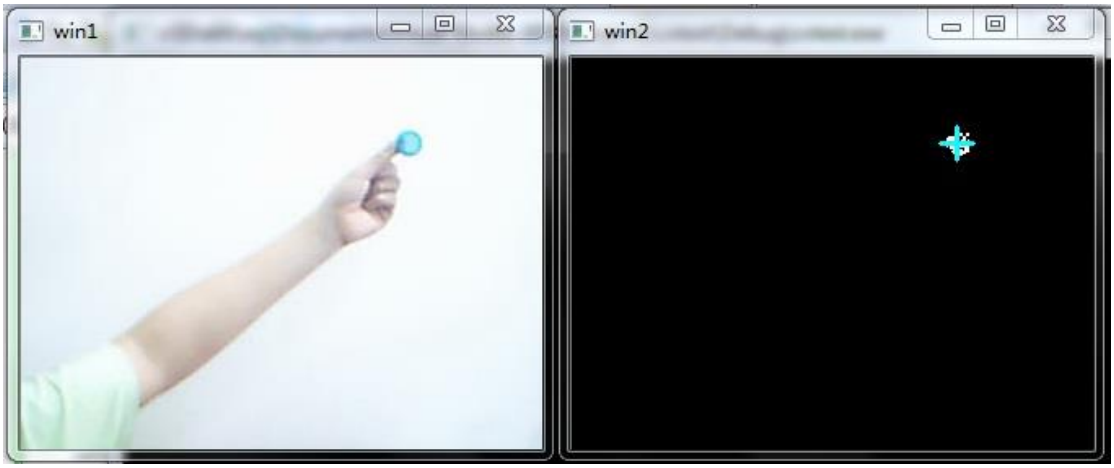




然而当目标区域相对较小，而噪声远离目标中心的时候，噪声会把平均中心明显的拉向一边，例如下图。



有时在 HSV 空间进行二值化，可以避免一些噪声。CV 模块提供了颜色空间的转换函数 cvCvtColor，前两个参数是输入和输出图像，第三个是转换类型，例如从 RGB 空间转换到 HSV 空间，第三个参数就是 CV_BGR2HSV（这里不使用 CV_RGB2HSV 是由于通常情况下原始图像三个通道的顺序为 B、G、R），H 的范围是[0,180]，乘以 2 就对应于色轮上的角度值，S 与 V 都是[0,255]。下图就是在 HSV 空间进行的二值化结果。



使用这种方法，需要非常准确的确定二值化所用的阈值以尽可能避免噪声的干扰。然而阈值过于精确，当光照改变或者遇到镜头污点时，会错误的将目标排除在外；而阈值范围过大，又会包括很多噪声。

目标跟踪

下面介绍一种目标跟踪的方法，这种方法抗干扰能力较强，而且非常稳定，但是原理比较复杂。

首先要对目标采样。为了适应实际环境中的检测，应该将目标放在实际的环境中，在地图的中心以及角落用摄像头拍摄几张目标的图像。实际运行的时候应该尽量保持这种光照、阴影，并且使用同一个摄像头。拍摄图像后，用任意的图像编辑工具将目标从每幅图像中提取出来，合并至一个文件。要确保提取出的每个像素都是目标，合并时也不要留空。这样通过多个样本来近似的代表目标。如下图所示，是把圆形目标从四幅图像中提取合并的结果。

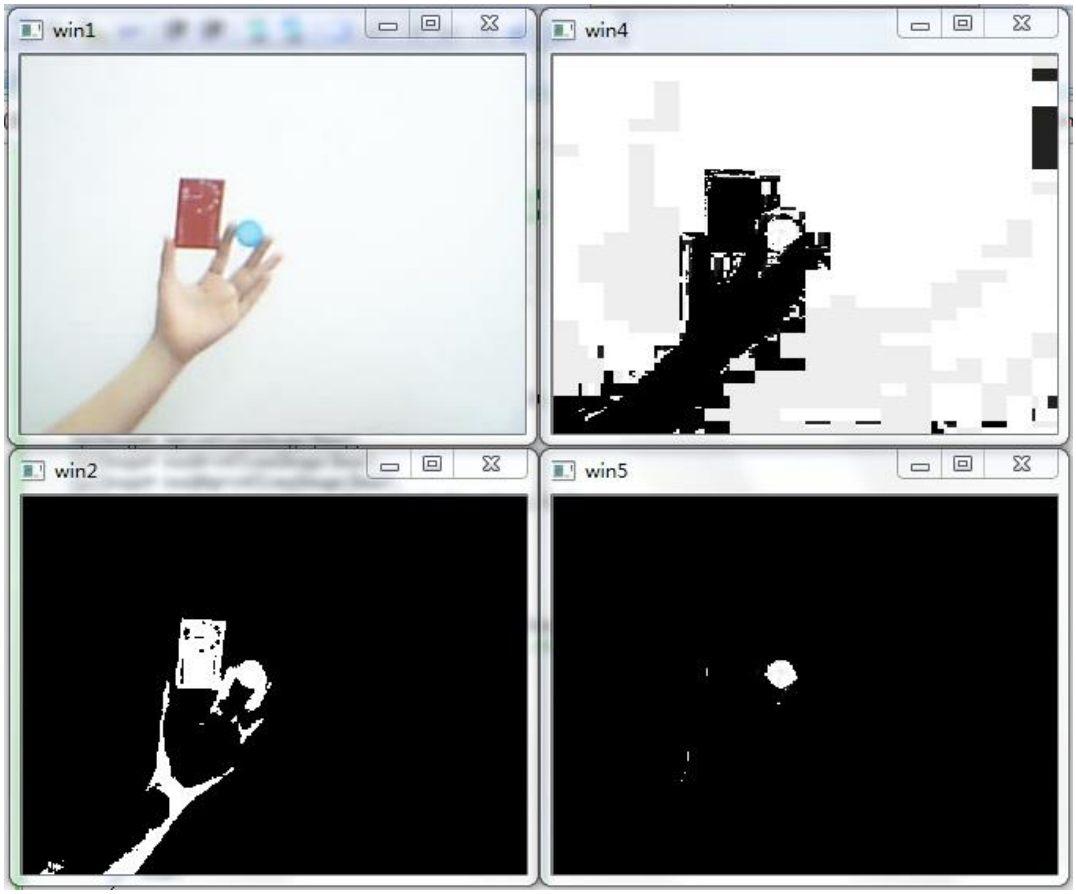
接下来计算样本的直方图。直方图是从单通道灰度图计算出的，因此对于彩色图像要选择一个通道，通常选择色度通道能够获得比较好的效果。下面就是计算色度通道直方图的演示程序

```
int main()
{
    IplImage* cap = cvLoadImage("e:\\kc\\bluecap.bmp");     //目标图像
    IplImage* caphsv = cvCreateImage(cvGetSize(cap), 8, 3); //HSV空间的目标图像
    IplImage* caphue = cvCreateImage(cvGetSize(cap), 8, 1); //目标的色度通道
    cvCvtColor(cap, caphsv, CV_BGR2HSV);           //目标从RGB空间转换至HSV空间
    cvSplit(caphsv, caphue, NULL, NULL, NULL);     //只提取目标的色度通道
    int bins = 30;              //指定直方图的精细程度（横轴等分为多少区间）
    float range[] = {0, 180};   //指定直方图横轴的取值范围，色度通道最大值就是180
    float* ranges[] = {range};  //因为cvCreateHist的参数必须是float**
    CvHistogram* hist = cvCreateHist(1, &bins, CV_HIST_ARRAY, range);//创建直方图
    cvCalcHist(&caphue, hist);  //计算目标的色度通道的直方图
}
```

直方图是一维函数，又是从二维图像计算出来的，因此可以看作是二维到一维的一种映射，或者称为投影。投影之后，直方图就没有空间信息，而包含了各个灰度值在原图像中所占的比例，即各个灰度出现的概率。

给出一幅灰度图像以及一个直方图，可以从直方图中得出图像中每个像素的灰度的概率。将这个概率值放大到[0,255]，就得到了一幅表示概率密度的灰度图，每个像素的灰度表示这个像素在直方图中的概率。这样从一维直方图得到了二位的灰度图，称为直方图的反向投影。

CV 模块提供了反向投影函数 cvCalcBackProject，由于之前计算的是目标色度通道的直方图，因此反向投影的输入也要用色度通道。下图 win1 是原始图像，win4 是反向投影的结果。



可以明显的看出反向投影效果。先不看背景部分，由于目标的样本图像中几乎都是蓝色，因此手的颜色以及卡片的红色出现的概率很小，反向投影图像中的灰度就很小，显示出来就非常暗。而目标的蓝色出现概率很大，显示出来很亮，接近白色。背景由于是白色，饱和度很低，而且由于镜头的偏色，导致色度接近于目标的蓝色，因此通过反向投影得到了较大的概率。

为了滤掉背景的影响，可以在 RGB 空间上对原始图像做粗略的二值化。对于上面的例子，简单的限制一下某个分量的最大值就可以去除绝大部分白色背景。从 win1 中得到 win2 的二值图像。在这个图像中，只粗略消去了背景，而卡片、手以及目标依然存在。

将 win4 与 win2 进行“逻辑与”的运算，可以得到两个图像的公共部分，即目标。结果显示在 win5 窗口中。逻辑与在 Cxcore 模块中提供了 cvAnd 函数。

到此为止，得到了类似于之前直接用二值化得到的结果。但是这种方法只需要使用粗略的阈值就能得到准确的结果。虽然 win5 的图像中，目标周围仍有少量噪声，下面将进一步处理。

CV 模块提供了 cvCamShift 函数用于对象跟踪，输入至该函数的第一个参数就是上面 win5 窗口显示的图像，第二个参数指出图像的初始搜索区域。CamShift 算法将在初始区域附近搜索显著的目标，直到满足某个停止条件。第三个参数就用于指定停止条件，可以用 cvTermCriteria(type, max_iter, epsilon)函数构造。停止条件有两种，type 为 1 是当迭代次数超过 max_iter 后停止，而 type 为 2 是当两次迭代结果的差异小于 epsilon 后停止。也可以同时限制两种条件，type 为 3，既保证结果准确，又防止消耗过多时间。

第四个参数是函数的返回值，CvConnectedComp 结构中的 rect 成员给出了最终的跟踪区域，从给出的值可以计算出这个区域的中心坐标，即为目标中心。同时这个区域可以作为下一次跟踪的初始区域，即函数的第二个参数，从而实现了连续跟踪。

第五个参数可选，同样是函数的返回值。可以更加准确的得到目标的姿态，CvBox2D 结构的 center 成员直接给出了目标的中心，angle 成员给出了目标的方向。而函数的第四个参数只包括目标的外接矩形，没有方向。

通常用鼠标拖动的方法指定最初的搜索区域，上述方法就可以完全自动的跟踪目标，得到任意时刻的目标位置甚至方向。然而有时 cvCamShift 函数返回的方向不够准确，如果认为误差过大影响判断，可在小车的前后安置两个比较小的标记，使用不同的颜色加以区分，分别进行跟踪，通过两个标记的中心坐标可以准确计算出小车的方向。

小车控制

到此为止，得到了小车的位置与方向。将位置与路径进行比较，可以判断是否偏离路线，如果偏离则进行矫正。转弯时将小车方向与目标方向进行比较，能够判断转向是否完成，完成之后就可以继续前进。实时控制的大致原理就是如此，具体实现与题目以及路径的计算方式密切相关，也是课程的主要内容之一，请大家展开创意，自行设计。

附录

参考资料

http://eelab.sjtu.edu.cn/kc

课程网站，可以下载课件、资料、工具，也可以找到往年的小组演示、报告以及问题讨论。

http://www.opencv.org.cn

OpenCV 的中文网站，提供 OpenCV 的下载，以及自行编译安装的方法。大部分函数的中文介绍，需要使用函数的时候可以先查一下各个参数的含义以及类型。

http://opencv.willowgarage.com/

OpenCV 的英文网站，中文网站的内容都是翻译这里的。这里还有更多函数的介绍。

《学习 OpenCV》、《Learning OpenCV》

同一本书的中英文版，章节安排的很好，循序渐进，内容也非常充实。图书馆有几本中文版，Google 可以在线阅读英文版，网络上也有中文英文电子版的下载。

提示

1. 用 cvCreateCameraCapture 打开摄像头后，记得在程序结束时用 cvReleaseCapture 释放摄像头。否则会出现程序运行完了，却没有自动关闭的情况，虽然不是什么大问题。
2. 程序尽量按照功能分成各个模块，并单独为每个模块设计调试程序。数据用参数传递也好，用全局变量也好，总之尽量将不同的功能分开。各个模块通过基本测试后，再由主程序调用。当出现问题时，可以记录每个模块的输入输出，确定问题发生的模块。然后再用相同的输入在模块内单步调试，找出问题的根源。
3. 有可能被连续调用的回调函数中不要出现 cvWaitKey 函数，主要指的是处理滑块以及鼠标拖动的回调函数。拖动滑块时，滑块的值会连续变化，从而连续调用回调函数。假设滑块回调函数 F 中出现 cvWaitKey，当调整滑块位置时，F 被调用。而运行至 F 中的 cvWaitKey 时，又调整了滑块位置，则 F 又被调用。第二次调用的 F 结束后，第一次调用的 F 会从 cvWaitKey 继续运行，可能导致第二次的结果被覆盖。没有 cvWaitKey，就确保了第一次调用结束后，再进行第二次调用。但这样就要求回调函数必须能迅速完成，函数尽量简洁，不要包含复杂的循环或处理等等，否则可能导致响应不及时（滑块不流畅、鼠标很卡等等）。因为鼠标输入是由一个回调函数处理所有情况，所以这里指的是处理拖动的那部分程序要简洁，其他部分复杂一些不会有太大影响。
4. 尽量不要在循环中申请内存，这是程序设计的基本原则，在 OpenCV 中就是不要在

循环中创建图像。而创建图像主要是三个函数——cvCreateImage、cvCloneImage、cvLoadImage。因为这些函数将申请一块内存用于存放图像，而在循环中这些函数将被多次调用，从而消耗大量内存。所以尽量在循环外面创建图像，而在循环中反复利用创建好的图像，比如 cvSetZero 可以清空图像，cvCopyImage 可以复制图像内容。如果必须创建图像，则应该在使用后通过 cvReleaseImage 函数及时销毁，但是每次创建销毁都要花费时间，可能影响程序效率。

5. 迷宫问题在很多数据结构或者算法书中有详细的分析，程序上最简单的就是递归的深度优先搜索。如果存在多条路径，可以进一步选择最短路径，或者转弯最少的路径等等。只要严格按照方格制作迷宫，图像处理非常简单，很容易就能获得迷宫的结构，因此指导书中对这个题目的介绍很短。
6. 有些函数是以宏定义的，比如 CV_RGB 其实是 cvScalar 等等。如果不清楚其参数类型，可以输入函数后按 F12，或者右键选择转到定义，查看函数的定义。如果对函数或者参数不理解，先省略各种可选参数，再进一步通过名称猜测，或者在网上搜索，进行多次试验，就可以发现规律。
7. 遇到无法解决的问题时，可以参考以往的课程网站是否有解决方案。如果没有，请在网站留言，说明详细情况