摄像头图像处理指南

华北电力大学(保定)
智能车俱乐部
2019 年 10 月 15 日 第一版

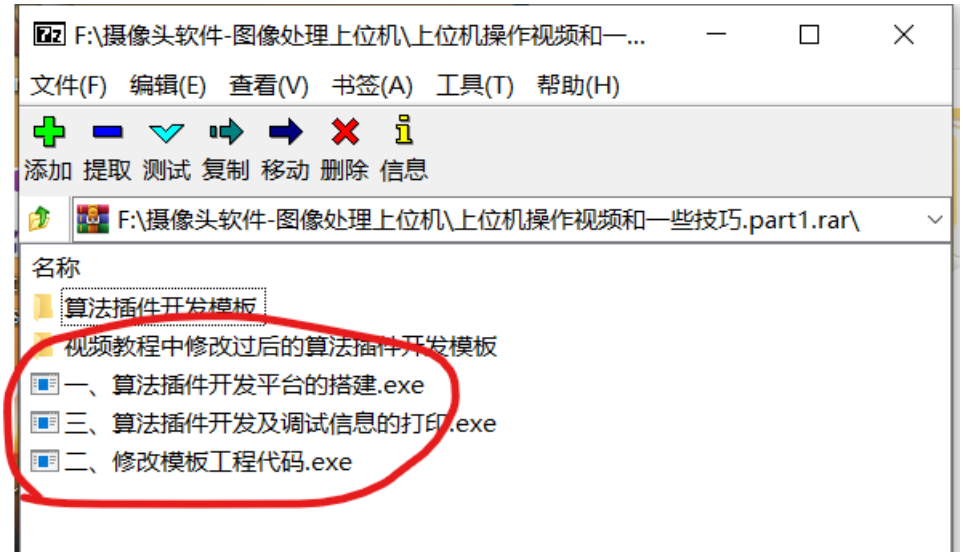
一． 软件的使用与基本知识

关于软件的安装与使用，请先观看所提供的安装包中所提供的视频：

上位机操作视频和一些技巧.part1.rar

上位机操作视频和一些技巧.part2.rar

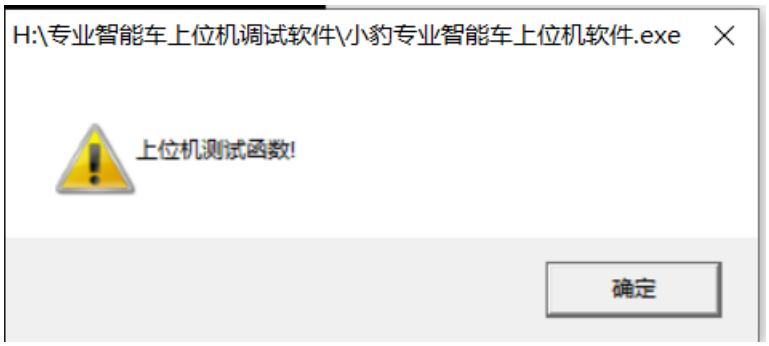
解压后打开.exe 文件可观看。



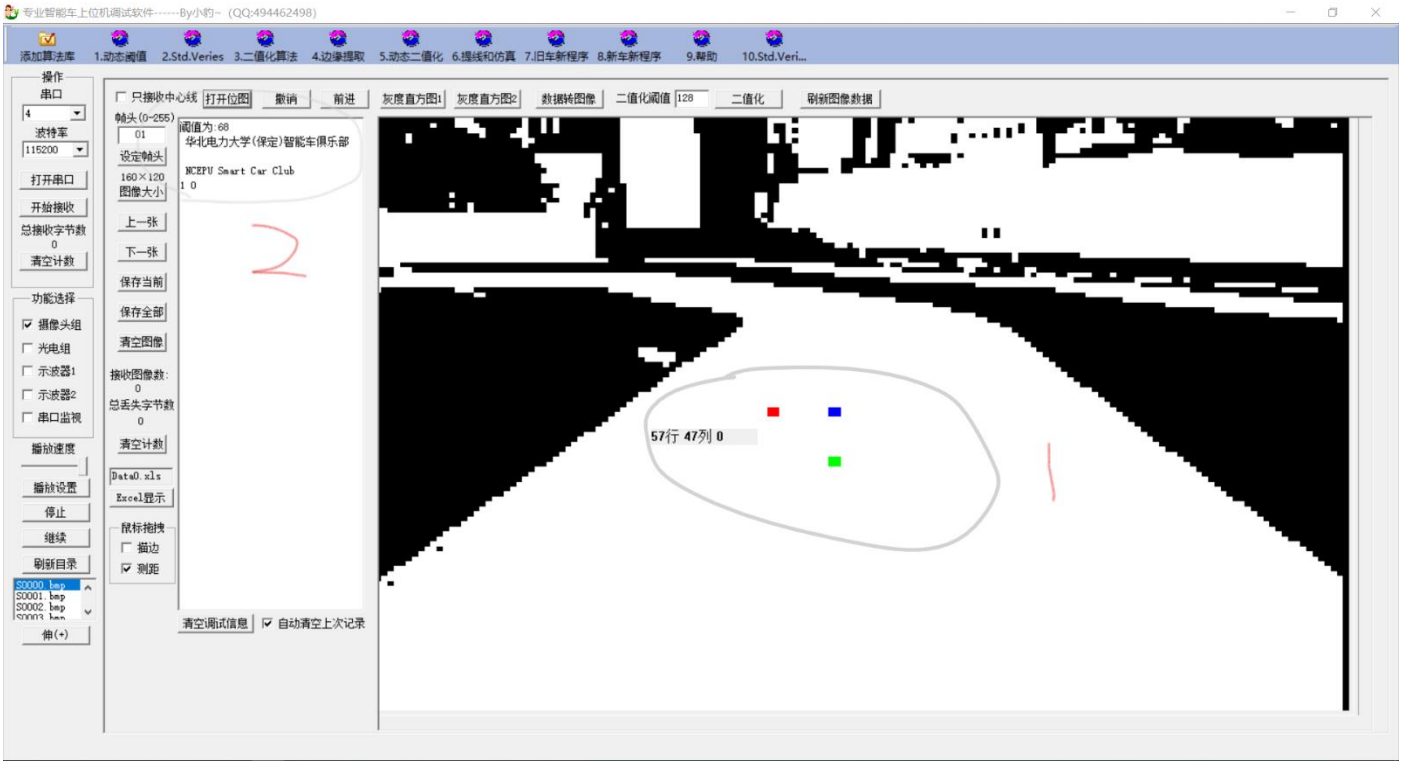
下面提供一个函数用于测试与了解上位机的使用方法：

```
void ProgTest(){
    AfxMessageBox("上位机测试函数!");      //弹窗显示信息(1)
    img[60][65]=RED;img[59][65]=RED;    //标记图像上点的颜色(2)
    img[60][66]=RED;img[59][66]=RED;

    img[60][75]=BLUE;img[59][75]=BLUE;
    img[60][76]=BLUE;img[59][76]=BLUE;

    img[70][75]=GREEN;img[69][75]=GREEN;
    img[70][76]=GREEN;img[69][76]=GREEN;

    str.Format("\r\n 华北电力大学(保定)智能车俱乐部  \r\n");//窗口打印字符(3)
    PrintDebug(str);

    str.Format("\r\n NCEPU Smart Car Club \r\n");
    PrintDebug(str);

    str.Format("%d ",(img[70][75]==GREEN));  //判断点的颜色并打印判断结果(4)
    PrintDebug(str);

    str.Format("%d ",(img[70][75]==BLACK));
    PrintDebug(str);
}
```

上页的函数中，**弹窗显示信息(1)**的效果见下图：



所标记的图像颜色见下图的 1 号位置：



其中，程序中所使用的 RED，BLUE，GREEN 等关键字均通过宏定义来实现：

```
#define IMG_ROWS 120      //图像的总行数，即高
#define IMG_COLS 160      //图像的总列数，即宽

#define WHITE    255      //白色的图象值
#define BLACK    0        //黑色的图象值
#define RED      128      //红色的图象值
#define BLUE     254      //蓝色的图象值
#define GREEN    100      //绿色的图象值
#define GRAY     64       //灰色的图象值

#define u8      unsigned char   //为后期编程中的方便,进行关键字简化
#define u16     unsigned short

#define uint8_t unsigned char
#define uint16_t unsigned short
```

```
#define uint32_t unsigned int
#define int16_t short int
#define int32_t int

unsigned char ImageData[IMG_ROWS][IMG_COLS];//定义图像数据的数组

#define img ImageData  //为后期编程中的方便,进行关键字简化
```

程序中进行的字符串打印均在 2 号位置出现。

二. 起始点的搜索

下面介绍图像处理部分的内容。图像处理部分的主要工作,是根据摄像头所回传的图像,通过图像处理程序提取图像中的左右边界线,通过左右边界线进行舵机与电机的控制。一种较为经典的方法,就是使用左右边界线拟合出中线,而中线是由图像上每一行上的一个点组成的。我们根据每一行上的点到图像中心(IMG_COLS/2)的偏差加权平均得到综合偏差进行控制。

我们一般对舵机使用 PD(比例-微分)控制器进行控制。在有编码器进行速度反馈的前提下,对电机使用 PID(比例-积分-微分控制器进行控制)。

而要搜索边界线,就需要对边界线的起始点进行搜索。我们首先对存储左右边界线的空间进行定义:

```
struct DIV
{
    int hang[240];
    int lie[240];
};

DIV left,right;
```

其中 struct 定义了结构体,其代表了一种数据结构,其中包含 hang,lie 两个成员。然后我们将这种数据结构实例化为 left,right 两对象,分别代表左右边界线。在对其进行使用前,我们需要对其进行初始化。

```
void InitData()
{
    int pin;
    for(pin=0;pin<240;pin++)
    {
        left.hang[pin]=254;left.lie[pin]=254;
        right.hang[pin]=254;right.lie[pin]=254;
```

```
    }
}
```

程序中用 254 代表无效数据，故将数据全部初始化为 254。其代表左右边界线均为空。使用如下函数进行左边界线的扫描：

```
void LeftStartFind()
{
    str.Format("\r\nvoid LeftStartFind():\r\n");
    PrintDebug(str);
    int hang,lie;
    for(hang=IMG_ROWS-1;hang>15 && left.hang[0]==254;hang--)
    {
        for(lie=0;lie<IMG_COLS-40 && left.lie[0]==254;lie++)
        {
            if(img[hang][lie]==BLACK && img[hang][lie+1]==BLACK)
            {
                if(img[hang][lie+2]==WHITE && img[hang][lie+3]==WHITE)
                {
                    left.hang[0]=hang;
                    left.lie[0]=lie+1;
                    img[hang][lie+1]=RED;
                }
            }
        }
    }
    str.Format("左起:(%d,%d)\r\n",left.hang[0],left.lie[0]);
    PrintDebug(str);
}
```

该函数主要靠两层的 for 循环实现功能，第一层循环将行(hang)变量从图像最下部(IMG_ROWS-1)到图像上部第 15 行循环。第二层循环将列(lie)变量从图像最左边 0 到最右边第 120 列(IMG_COLS-40)循环，判断某一个点是否满足左边线(黑黑白白)的条件。若满足条件，则向存储左边线的存储单元中写入有效数据，控制循环跳出，打印数据后该函数退出。

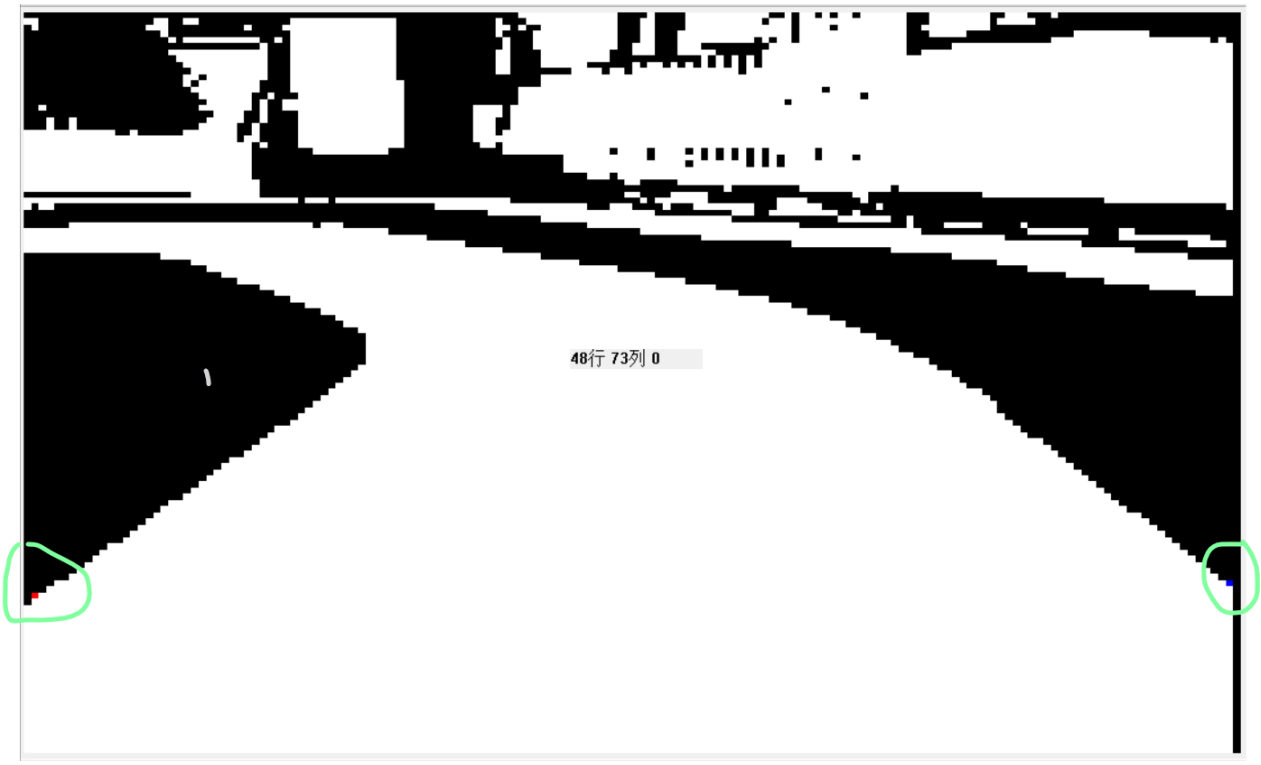
寻找右边线起始点的函数可以根据左边线起始点的函数修改得到：

```
    for( 行变量:图像最下部->图像上部某一行(自行选择) ) //若 c>b
    {                                                  // for(a:b->c) 等价于 for(a=b;a<c;a++)
        for( 列变量:图像最右部->图像左部某一列(自行选择) )
        {
            if( 从左至右 白白黑黑 )
            {
                写入存储右边线存储区的第一个存储单元;
                if( 右边线存储区的第一个存储单元非空 )//right.hang[0]!=254&&&right.lie[0]!=254
```

```
                {
                    return;
                }
            }
        }
    }
```

※上面部分所提供的程序是以**伪代码**的形式呈现的，下面绝大部分程序都将以伪代码的形式呈现。

以上程序效果如下：



左边线用红点表示，右边线用蓝点表示。

三. 左右边线起始点的修正

按照上面的程序，搜索出来的边线不一定完全正确，所以需要对上面程序所处理得到的左右边线坐标进行综合判断，可能需要去掉其中的某个点。这里仅阐述一种思路。当左右边线起始点均正确时，两点连接形成的直线所经过的点均为白色，因为两者连线所经过的均为赛道。上面的图像就符合这一规则。但是当程序所得出的结果不正确时，就可能不符合上面的规则。图像如下：

上面图像(S0056)中,左边界线的扫描出现了问题。在弯道上左边线不一定存在,其可能不在摄像头视野范围内。摄像头余越矮或度数越小越可能出现这种情况。为了解决这一问题,我们在程序中遍历两起始点连接线上的点,根据线段上黑色点的个数来区分这一情况。实现这一功能的伪代码如下:

```
if(左右起始点至少有一点不存在)
{
    退出;
}
声明并初始化斜率为 0;
声明初始化上点行,上点列为无效值(254);
声明初始化下点行,下点列为无效值(254);
if(右起始点行<左起始点行)    //定义变量,将左右两个起始点中 hang 较小与较大的分别存入
{
    上点行=右起始点行;上点列=右起始点列;
    下点行=左起始点行;下点列=左起始点列;
}
else
{
    上点行=左起始点行;上点列=左起始点列;
    下点行=右起始点行;下点列=右起始点列;
}

if( 上点行 不等于 下点行 )    //计算左右起始点连线的斜率
{
    斜率=(下点列-上点列)/(下点行-上点行);
}
else
{
    退出;
```

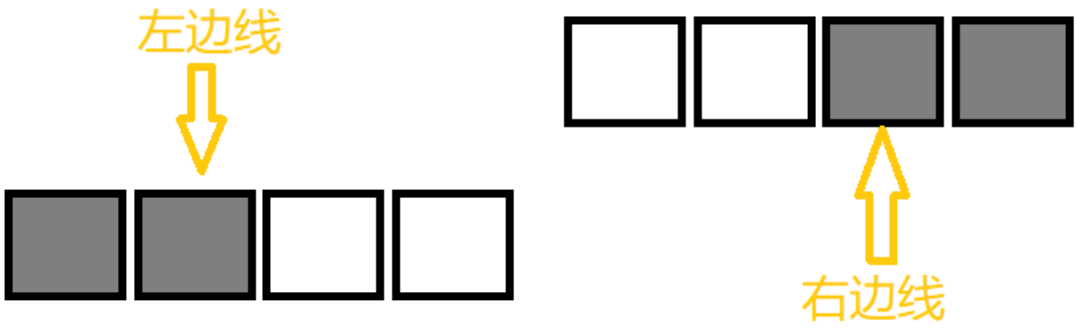
```
    }

    声明行,列,pin,黑色点数;
    黑色点数=0;

    for(pin=0,行=下点行;行>上点行;行--,pin++)  //遍历左右起始点连线上的点
    {
        列=(int)(下点列-斜率*pin);
        if(行>-1 && 行<IMG_ROWS && 列>-1 && 列<IMG_COLS)
        {
            if( 图像上坐标为(行,列)的点为黑色 )
            {
                黑色点数++;
            }
        }
    }

    if(黑色点数大于 35)   //若连线上黑色点较多,则去除更靠近图像上部的起始点
    {
        if(左起始点行<右起始点行)
        {
            左起始点行, 左起始点列赋值为无效值(254);
        }
        else if(左起始点行>右起始点行)
        {
            右起始点行, 右起始点列赋值为无效值(254);
        }
    }
```

程序达到的效果如下:

可以看见所找到的错误的左边线起始点已经被去除。

四. 边线扫描-跳变沿法

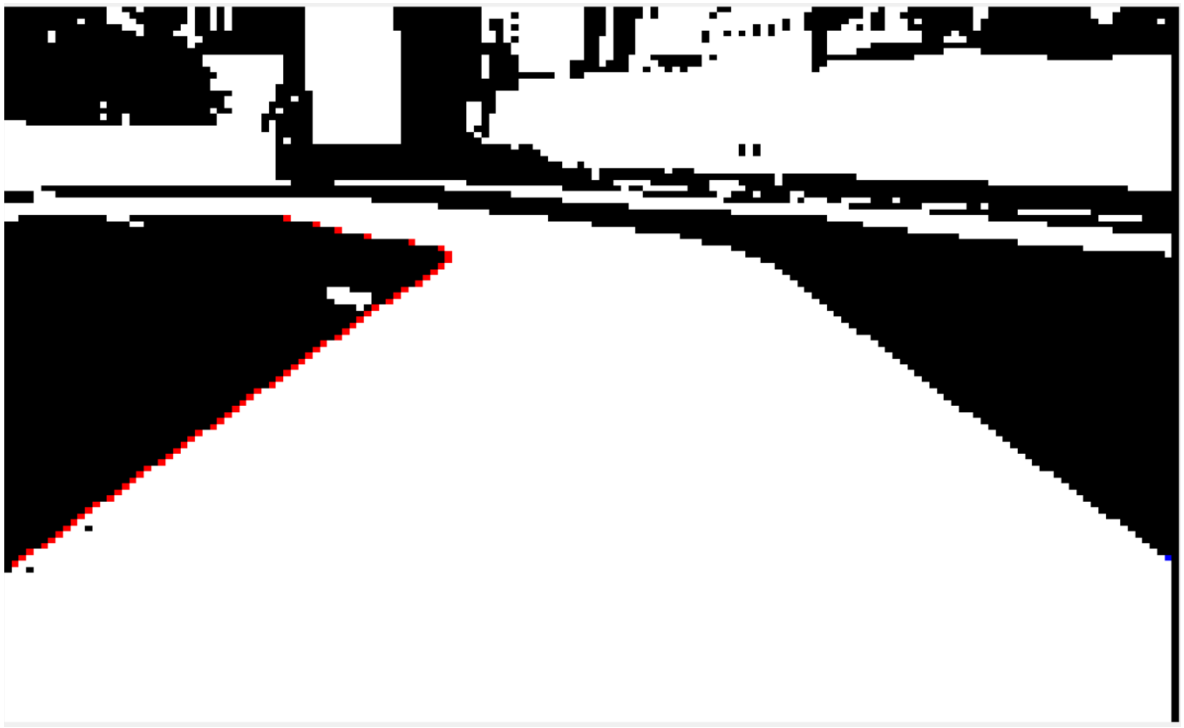
前面的程序处理完成后，得到了左右边界线的起始点。下面以这一起始点为起点，扫描左右边界线。跳变沿是指边界线的定义方式，即从图像左边至右边，黑黑白白为左边线，白白黑黑为右边线。见下图所示：



以左边线为例，扫描跳变沿左边线的方法如下：

```
定义行,列;
定义列最小值,列最大值;
定义 pin 并初始化为 1;
定义 find 标志位并初始化为 0;
if( 左边线起始点 不存在)
{
    退出;
}
for( 行 : ( left.hang[0]-1 ) -> 5 )
{
    find 标志位置为 0;
    列最小值 = left.lie[pin-1]-10;    //定义扫描范围：
    列最大值 = left.lie[pin-1]+10;    //上一个边线点左右十个点之间
    for( 列 : 列最小值 -> 列最大值 )
    {
        if( IMG_COLS-5 > 列 > -1 )
        {
           if( 点(行,列)为黑色 && 点(行,列+1)为黑色 && 点(行,列+2)为白色 && 点(行,列+3)为白色 )
            {   left.hang[pin] = 行;
                left.lie[pin] = 列+1;
                img[行][列+1] = RED;
                pin++;
                find 标志位置为 1;
                break;
            }
        }
    }
    if(find 标志位为 0)
    {break;}
}
```

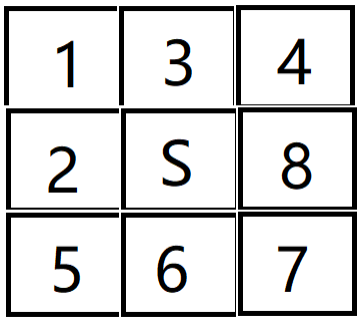
得到的效果如下:



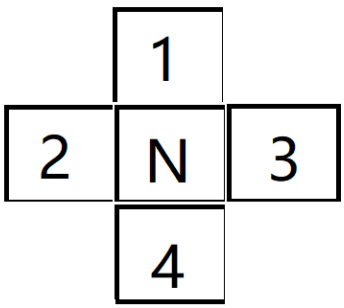
程序思路如下:根据左边线第一个点,下一个左边界线点在上一个左边界线点的上一行,列在上一个点的正负 10 的范围内产生。直到在此范围内找不到符合这一条件的点,则结束程序。

五. 边线扫描-九宫格法

下面以右边界线为例,换一种方法扫描边界线。该种方法较跳变沿法相比,扫描出的边界线更多。但是这种方法扫描出的边界线每一行可能有多个边界线上的点,后续程序中可能需要特殊处理。



图一



图二

程序思路如下:图一中的 S 为上一个右边界线点,以 S 为中心画九宫格,下一个右边

线从九宫格除掉中心的的八个点种产生，这种产生是需要考虑优先级的。即我们依次扫描图一中的 1->2->…->8 点，一旦发现其中**某一点本身为黑点，且该点上下左右四个点中至少有一个为白点**，则确定该点为下一个右边界点，然后重复该过程，直到某一点周围八个点都不符合边界线规则，扫描终止。

边界线扫描规则即上一段红色加粗部分，见图二。即若 N 点为黑色，且 1，2，3，4 点中至少有一个点为白色，则该点为边界点。

程序表述如下：

```
定义 pin;
定义 hang,lie;
定义 hang_temp,lie_temp;
定义 white_counter;
for(pin:1->240)
{
    if( right.hang[pin-1]==254 || right.lie[pin-1]==254 )
    {跳出循环;}
    for(hang,lie 按图一中顺序依次赋值)
    {
        if( 坐标为(hang,lie)的点为黑色 )
        {
            white_counter 赋值为 0;
            for(定义 hang_temp,lie_temp 按图二顺序依次赋值)
            {
                if( 坐标为(定义 hang_temp,lie_temp)的点为白色 )
                {
                    white_counter++;
                }
            }
        }
        if(white_counter 大于 0)
        {
            right.hang[pin]=hang;
            right.lie[pin]=lie;
            img[hang][lie]=BLUE;
            break;
        }
    }
}
```

程序所得结果如下：

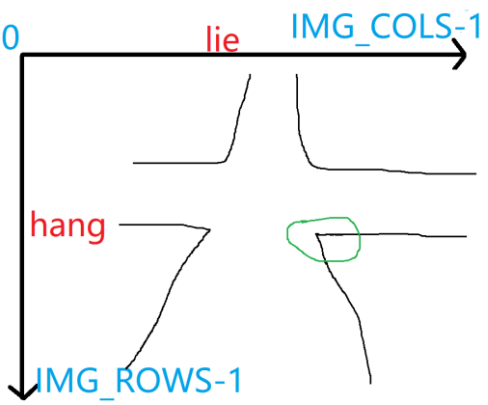
该图像上左边线是通过跳变沿扫描法得出的，右边线是通过九宫格扫描法得出的。其主要效果上的区别在于图中绿色圈中的部分，可见一行中有多个边界点。

在编写该程序时应注意，已经扫描确定的边界点应将其赋值为黑色以外的颜色，否则可能时程序进入死循环，见上面程序中的突出部分。

六. 十字处理-寻找转折点

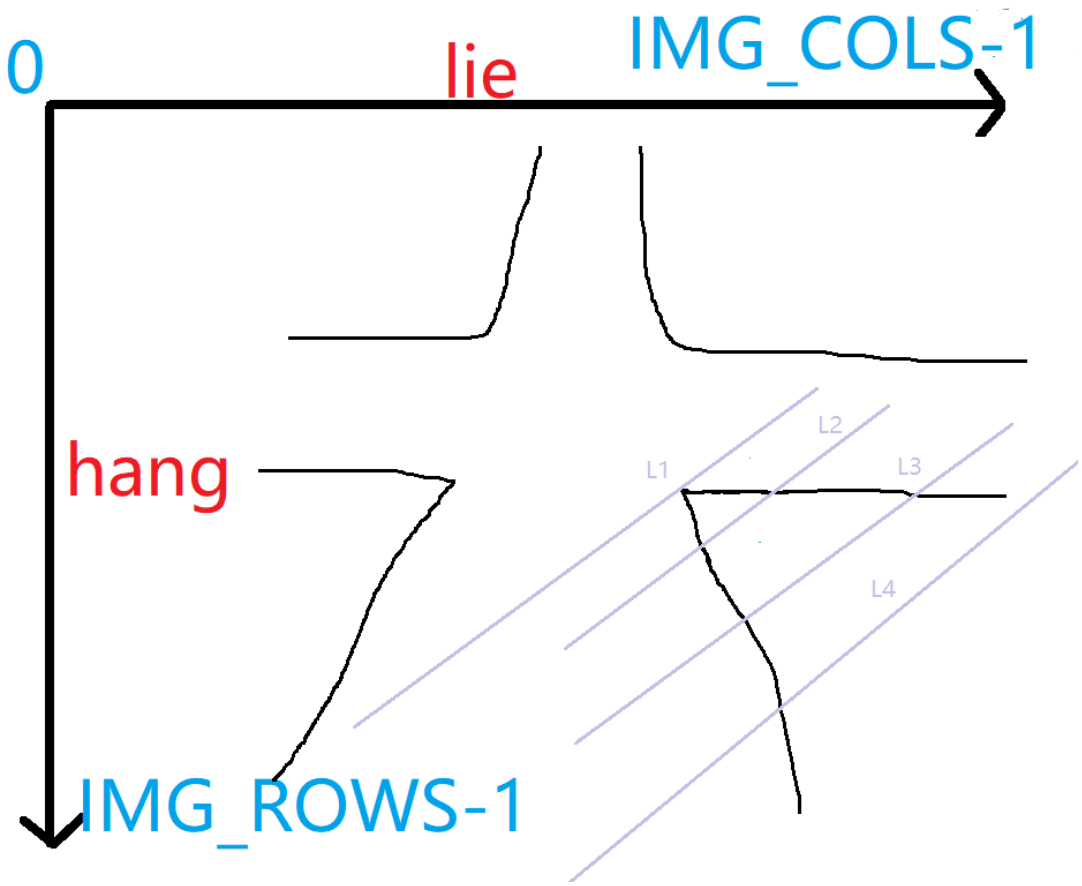
经过上面的处理，边线信息的坐标(x，y)已经存在结构体中，后面需要做的就是根据结构体里的信息找到补线的起始点，即边线上的转折点。

以右边线为例，见右图。图中存在一十字，程序的目标是寻找绿色圆圈圈出的转折点。经过前面程序的处理，右边线的信息已存在与数组中。在程序中对右边线进行遍历，过遍历到的点做直线，直线方程为：



$$hang=-k*lie+b$$

所做的直线如下图：



图中直线 L1，L2，L3，L4 都符合方程：hang=-k*lie+b,其中(hang,lie)为右边线上的点,k 为自行给定的值。有了上述三个变量,就可以求出式子中的 b。

b=hang+k*lie

可以在图中看出,L1 是过转折点的直线,且 L1 直线所求出的 b 是右边线上所有点最小的。于是找转折点的问题转化为了求 min(b)的问题。b 为直线在 hang 轴上的截距。

实现该部分的伪代码如下：

```
声明 pin;
声明直线斜率初始化为 0.5;
声明截距最小值初始化为 100000;
声明截距最小值对应 pin 初始化为 254;
for(pin:0->240)
{
    if(坐标(right.hang[pin],right.lie[pin])未越界)
    {
        if( 直线斜率*right.lie[pin]+right.hang[pin] 小于截距最小值)
        {
            截距最小值=直线斜率*right.lie[pin]+right.hang[pin];
            截距最小值对应 pin=pin;
        }
    }
```

```
        else
        {
            break;
        }
    }
```

该部分效果如下：

按照如上部分程序寻找出的拐点坐标为(55,121)，已在图像上标为红色。

七. 十字处理-补线

前面的程序中已经找到了十字下方补线的起点，补线还需要找到补线的终点。还是以右边线为例，为了找到终点，从起点开始向右上方扫描，找到十字的上半部分。然后再用前面所提到的九宫格法进行扫描，扫描出十字上半部分的边线，再利用上面寻找折点的办法扫描得出终止点。

这里所使用的方法，都是前面所使用过的。以十字下部的转折点开始，向右上方移动寻找扫描起始点的伪代码如下：

```
    定义移动斜率并初始化为 0.5;
    定义行,列;
    if(转折点不存在)
    {
        退出;
    }
    for(行:转折点行->转折点行-30)
```

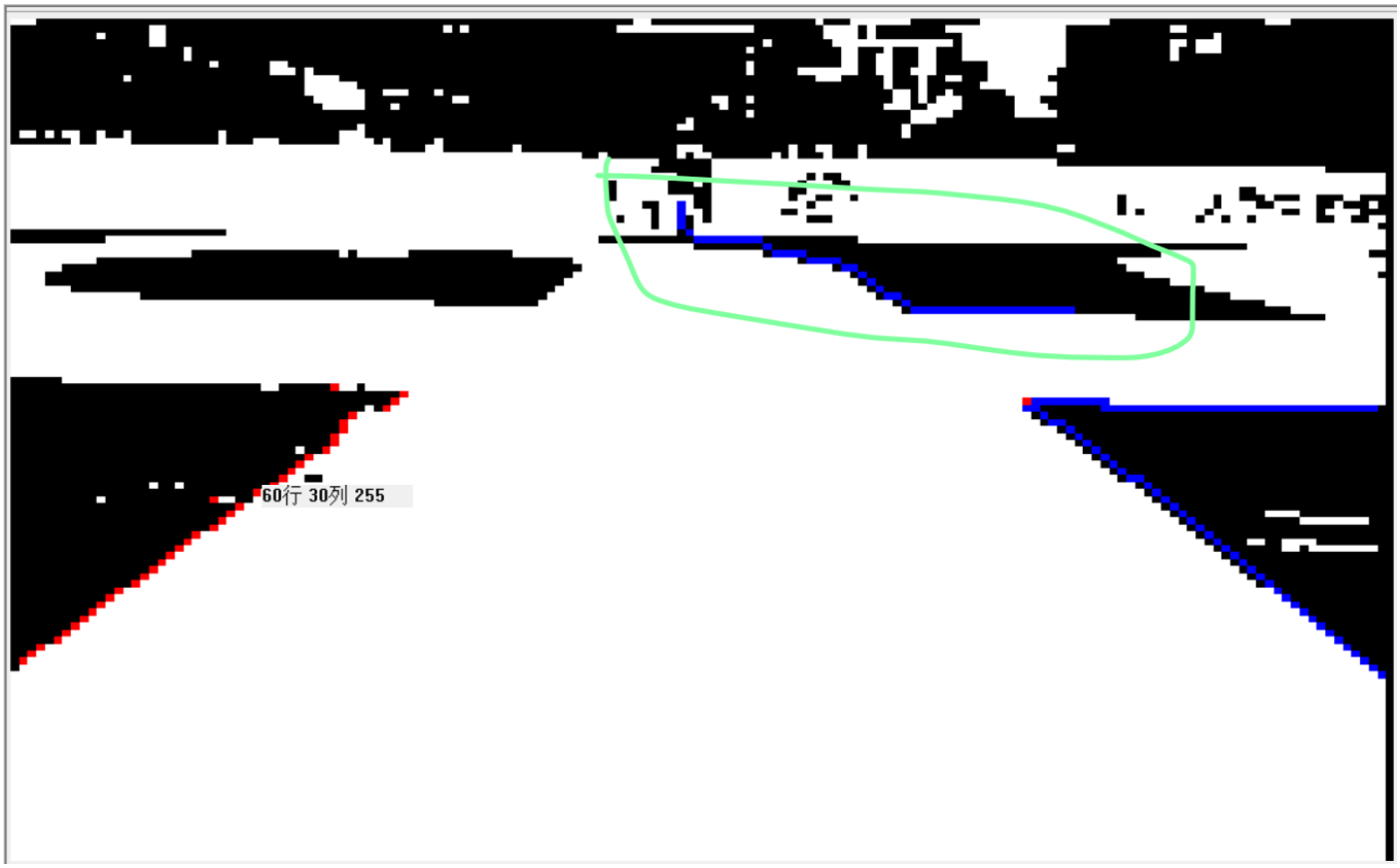
```
    {
        if(行小于 4)
        {
            break;
        }
        列=转折点列+移动斜率*(转折点行-行);
        if( (行,列)和(行-1,列)未越界 )
        {
            if( (行,列)和(行-1,列)均为黑色 )
            {
                break;
            }
        }
    }
```

上述部分程序得出的效果如下:

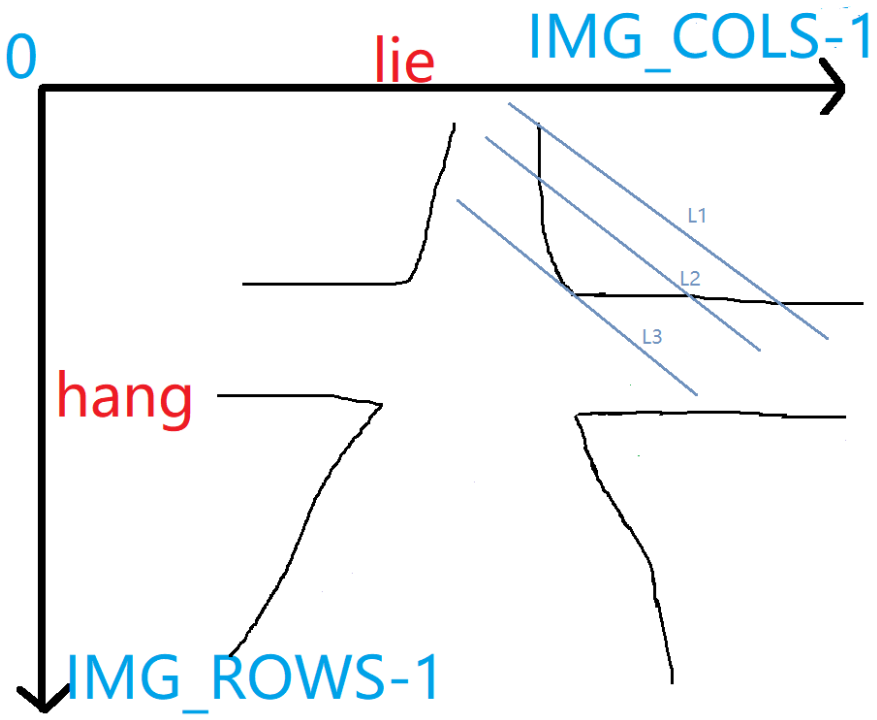
得出十字上半部分的继续扫描点为(41,123),上面伪代码中的(行,列)的运行轨迹是图中的绿色部分,最后找到的结果为图中的红色点。下面将九宫格法的程序进行更改,用于扫描十字上半部分的右边线。

主要是需要更改上一已知边界点周围八个点的优先级,使其适应赛道的特征。还需要对部分变量名进行修改,与之前的程序对接。

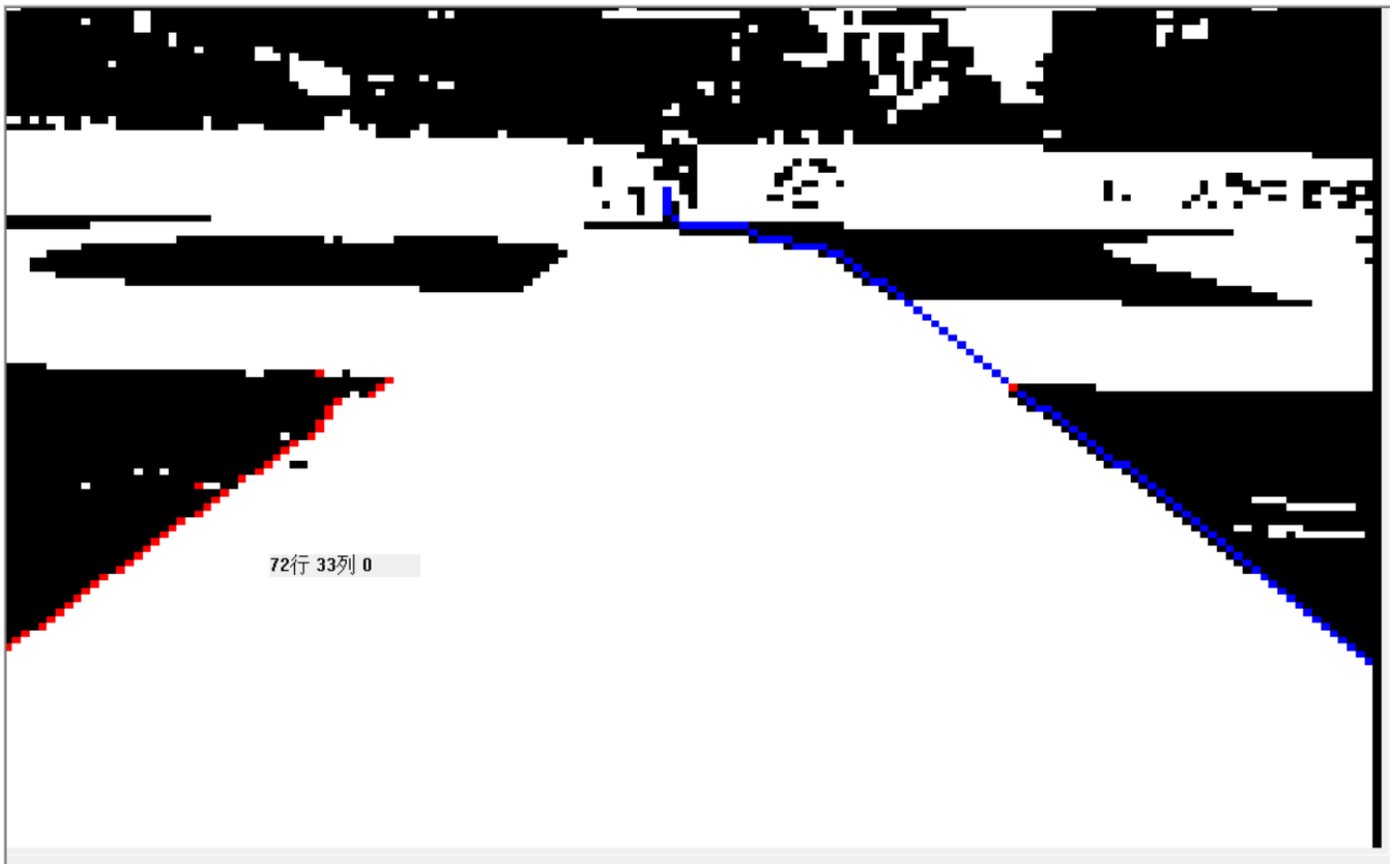
程序修改后所得的效果见下图：



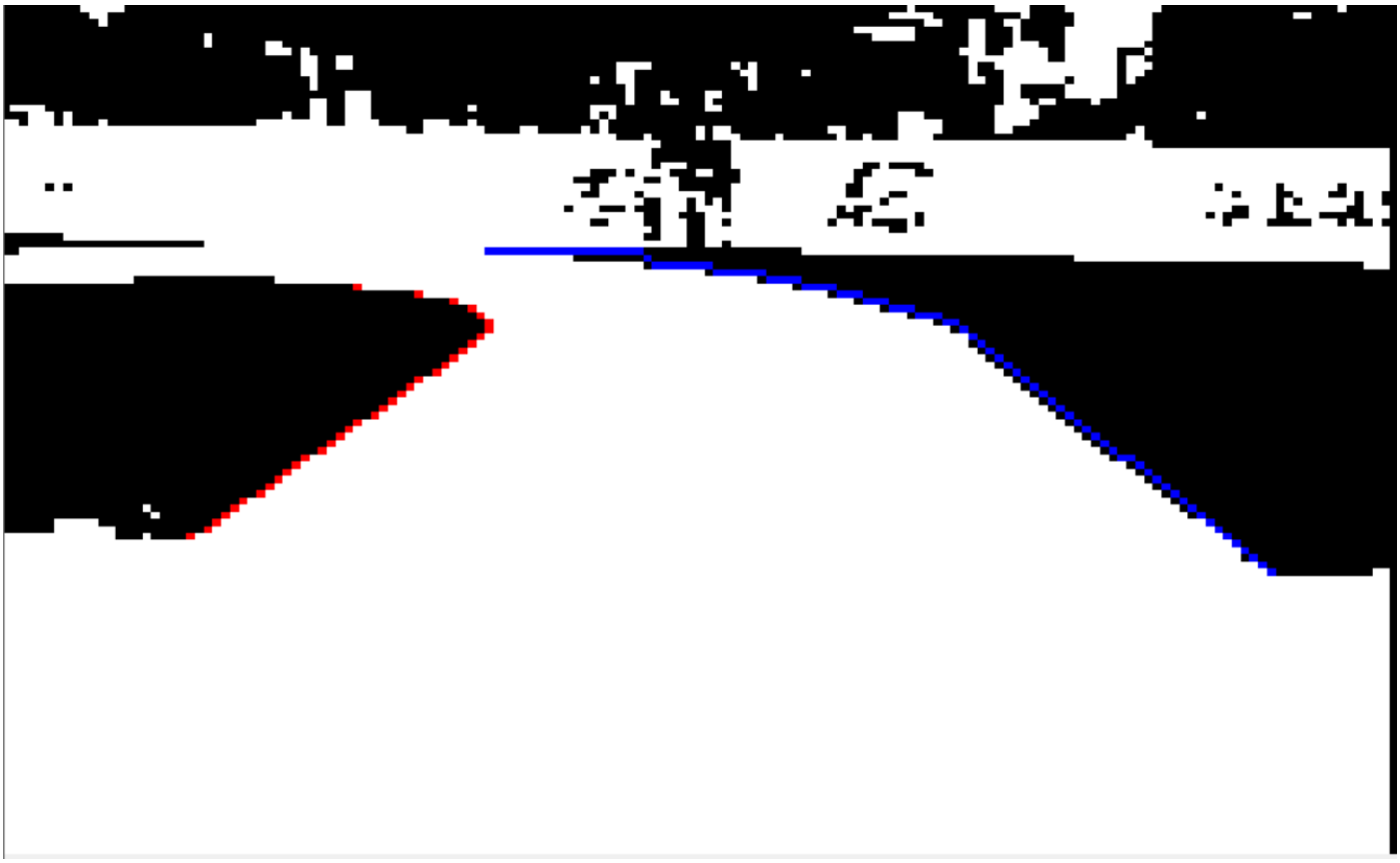
上图中绿色圈中的部分就是扫描得出的边线。使用前面提到的搜索转折点的方法扫描补线终止点，也需要对程序进行变通，主要是要修改 b 的计算式还有对直线斜率进行修改。



将补线终点找到后，进行连线。最终效果如下：



对于十字的处理，还存在下部分十字在摄像头视野之外的情况，这里不再讲述处理方法，需要自行设计方法进行处理。



八. 偏差量拟合-偏差计算与最小二乘拟合

上面的程序已经得到两条边线，而舵机与电机的控制需要根据偏差进行。例程中的方法使根据边线得到中线，中线与图像分界线逐行做差，然后根据每一行加权平均得到偏差，自行对照例程理解相关过程。具体的调试过程需要具体问题具体分析，例程仅作参考，可以使用自己认为效果最好的偏差拟合方法。

最小二乘拟合是一种非常有效的线段拟合方法，其可以将曲线近似成为一条直线，并且得出直线的斜率与截距。之所以使用这种方法，是应为使用最小二乘法拟合图像中线得到的斜率，可以较偏差更超前的反应赛道状况。使用最小二乘法拟合边线，可以反映出赛道的曲折情况，用于区分直道与弯道。最小二乘法代码如下：

```
float x=0,y=0,x2=0,xy=0;
int counter=0,pin;
for(pin=IMG_ROWS-1;pin>-1;pin--)
{
    if( middle_lie[pin]>-1 && middle_lie[pin]<IMG_COLS )
    {
        x=x+(IMG_ROWS-1-pin);
        y=y+(IMG_COLS-1-middle_lie[pin]);
        counter++;
        x2=x2+(IMG_ROWS-1-pin)*(IMG_ROWS-1-pin);
        xy=xy+(IMG_ROWS-1-pin)*(IMG_COLS-1-middle_lie[pin]);
    }

}
if(counter>10)
{
    xy=xy/counter;
    x=x/counter;
    y=y/counter;
    x2=x2/counter;
    k_middle=(xy-x*y)/(x2-x*x);
    b_middle=y-k_middle*x;
}
```

上面程序中的 k_middle 与 b_middle 为拟合出直线的斜率与截距。