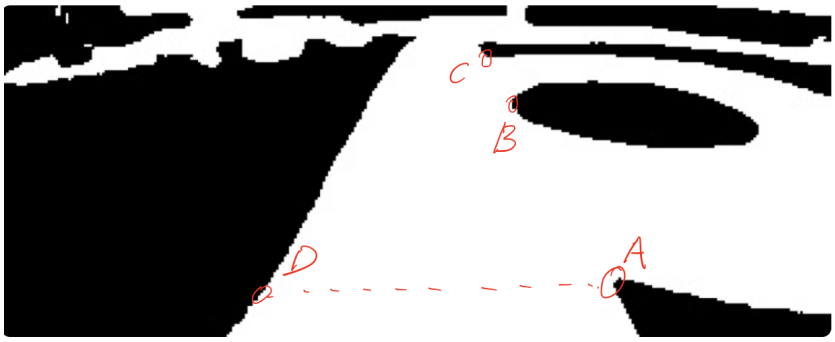
环岛思路

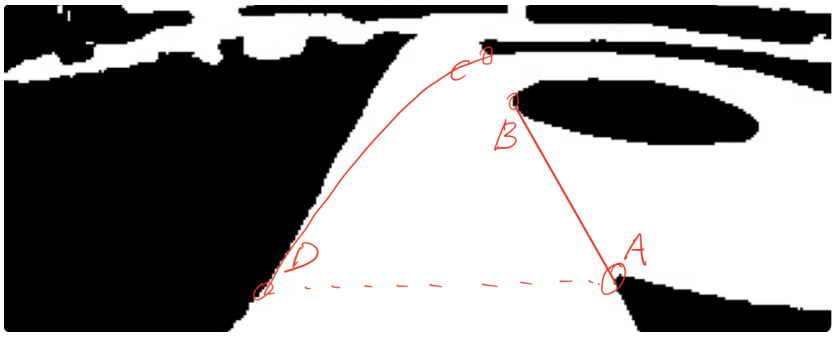
环岛的两条基本思路：连续标志位和找点补线。

首先先定义几个环岛的特殊点以及一些特殊位置：



1. 环岛A点：近处外圆和赛道相交的点
2. 环岛B点：内圆和赛道相切的点
3. 环岛C点：远处外圆和赛道相交的点
4. 环岛D点：与A点处于同一行的另一侧赛道边界点。
5. 环岛出口：A点、B点之间那块区域
6. 环岛入口：B点、C点之间那块区域

这四个点还可以有另一种定义方式：

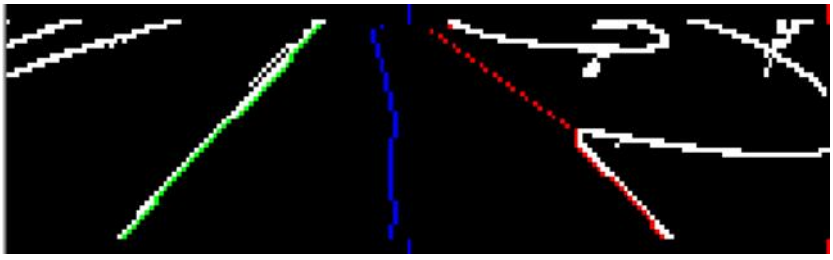


即AB连线为拟合的赛道边界线，DC连线为拟合的赛道边界线。同时把图中两条线称为AB边界线和DC边界线。

环岛状态划分

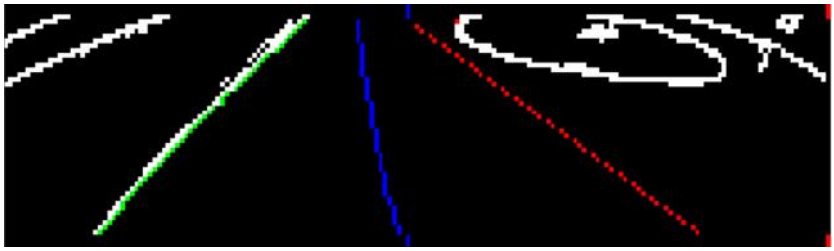
环岛需要根据不同的图像情况划分为不同的状态，每个状态给一个标志位，同时每个状态要检测当前图像是否满足下一状态以完成状态的转换。当把一系列的状态走完一遍之后，环岛也就跑完了。从算法鲁棒性上来说，状态越少稳定性越高，同时某一状态的算法不仅能处理自己状态的图像，还要尽可能的能处理其他状态的图像。我的算法里把环岛划分成了7个状态，算是划分的比较细的了。

环岛状态1及其补线



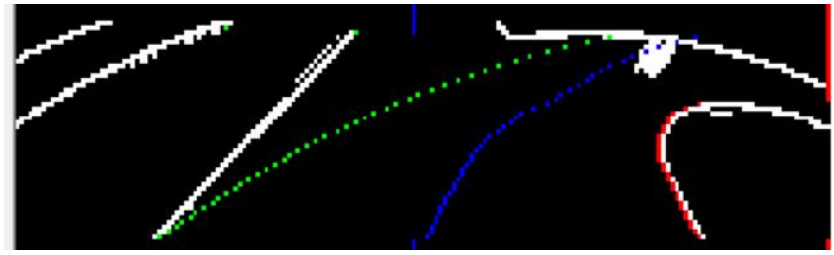
这是在直道上发现前方有环岛的时候的补线。可以发现图中仅存在环岛A点和环岛D点，而环岛B点和环岛C点还没进入摄像头的视野范围内。这时DC边界线不用拟合，直接采用直道边缘。而AB边界线需要拟合，但是没有环岛B点，那这时候就人为设定一个环岛B点，然后连接AB形成AB边界线。

环岛状态2及其补线



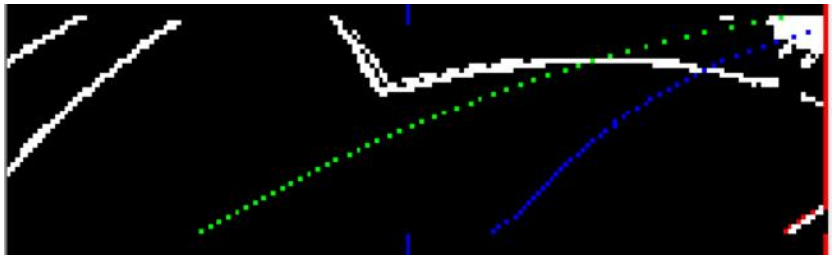
这幅图像大概是车头马上要进入环岛出口的时候。在这幅图中会发现环岛的四个特殊点都不存在。但其实这幅图的处理非常简单，由于环岛的特殊性，这时左边界一定是一条完整的直线，直接平移左边界作为右边界就可以了。

环岛状态3及其补线



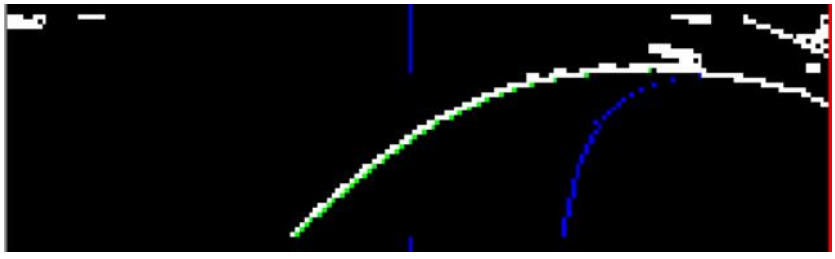
这幅图像是车在环岛B点时候，在这个地方，意味着车应该要进环岛了，因此补出来的线应该是一条向环岛内侧偏的曲线。这种情况下环岛AB边界线可以直接使用内圆的边界线。环岛C点由于比较难搜，而且很容易搜到错误的点，因此这时的C点也可以选择人为确定而不直接搜索，而D点则直接选用左边底行的边界点。然后用曲线连接DC两点即完成了补线。

环岛状态4及其补线

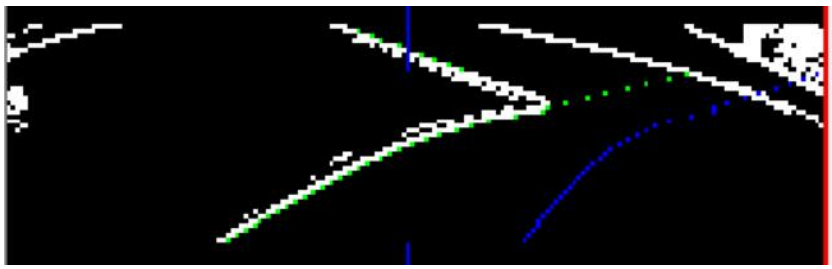


状态4与状态3的区别主要是状态4左边会丢边界点。这个状态主要是确保车能进入环岛，因此只要对补线给出一个趋势即可。因此图中D点可以设定固定的值，C点也可以设定固定的值，然后连接DC即可得到DC边界线。而对于AB边界线，如果有就采用，没有就当做丢边处理。

环岛状态5及其补线

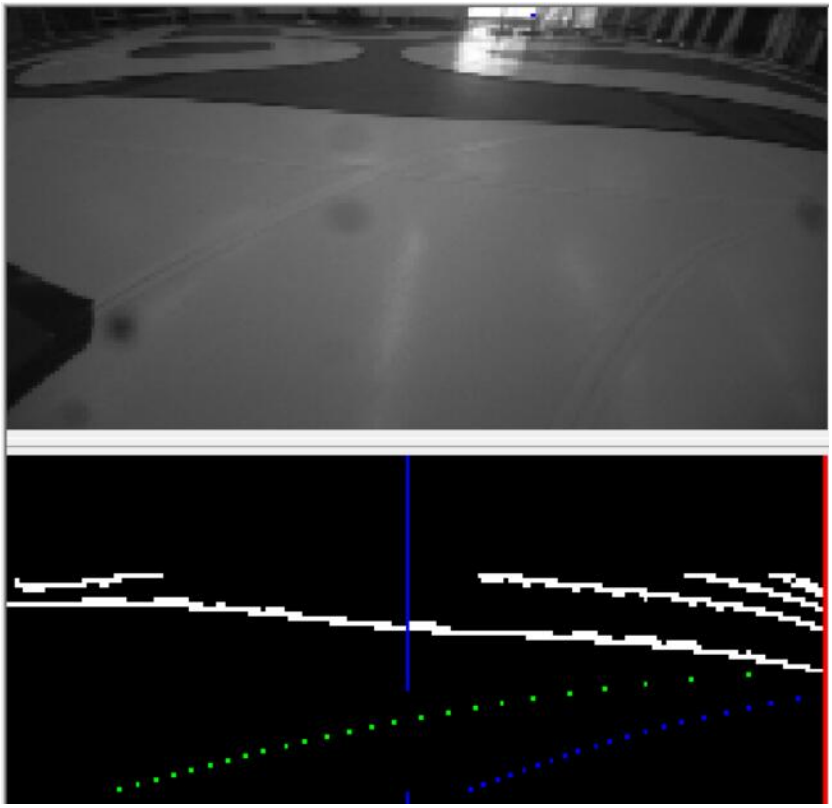


这个时候车已经差不多进入环岛了，在环岛中的图像和普通的弯道一样。但在这个状态下需要不断的检测是否到达环岛出口。



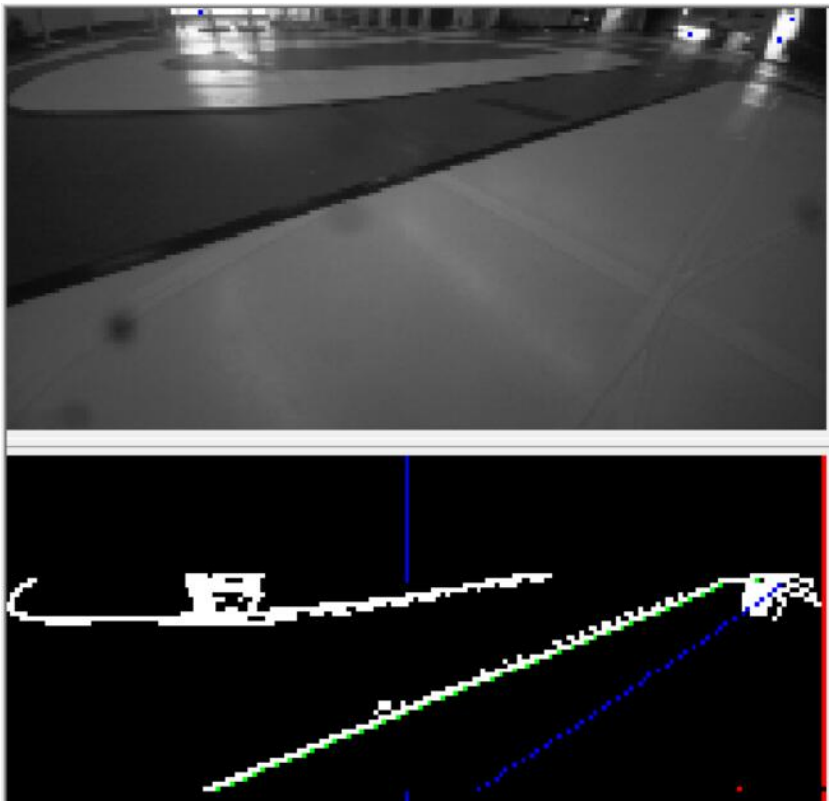
上面这幅图就是环岛出口，和斜十字情况非常类似，因此也可采用斜十字的判别方法判断是否出环。

环岛状态6及其补线



可以看出环岛出口处的图像基本没有什么可以操作的空间，就是白茫茫的一片赛道，因此对于这个状态，同样人为给定环岛D点坐标和环岛C点坐标然后连线得出DC边界线即可。

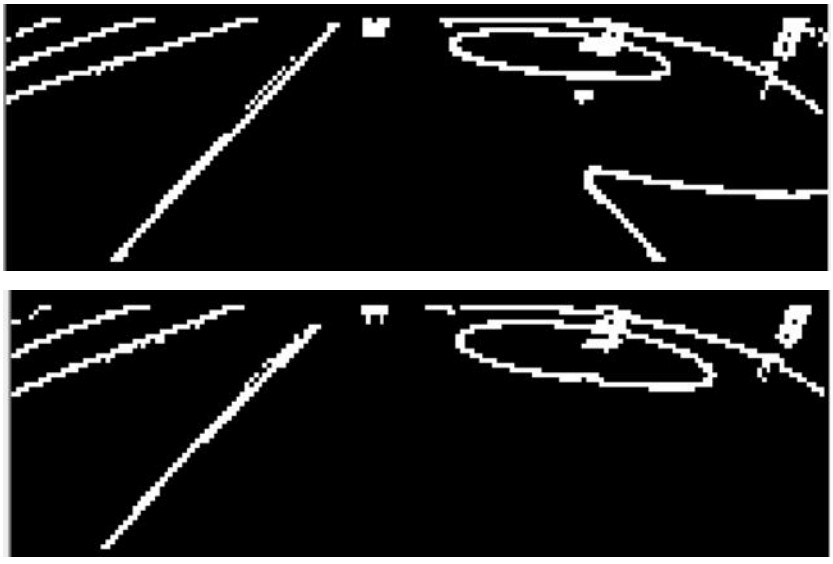
环岛状态7及其补线



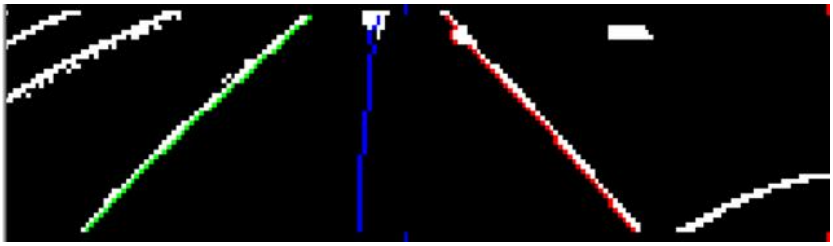
这个状态基本上就是车要回到直道上的时候了，与状态6的区别就是底行左边界不丢边了。这个状态的补线完全可以直接平移左边直道的边界，因为右边的情况对于大环岛小环岛以及车辆姿态等情况会出现各种非常复杂并且难以统一处理的情况。

环岛的识别

环岛最重要的部分就是如何正确识别环岛了，这决定着能否成功进入环岛。环岛的识别我一般放在环岛状态1和状态2，两种情况都有可能。



出环识别



这幅图基本就可以认为是出环了，特点就是赛道已经变回了正常的直道了。虽然这个特点很明显，但是在实际的过程中我发现识别这样一种情况非常复杂，于是对于出环我并没有使用图像中的信息，而是使用了车辆上的编码器。通过对编码器采集到值的积分可以近似得到车的运动距离，在进入环岛状态7的时候，就开始对车的运动距离进行积分，在这个积分值达到一定的阈值的时候，结束环岛状态。

算法鲁棒性提升

由于环岛还有一个明显的特点就是车会在环岛里转一圈出来，于是可以利用车上的陀螺仪，对偏航角速度进行积分。通过这个积分值也可以在一定程度上辅助判断当前是属于环岛的哪个状态，或者强制改变环岛状态。

基本补图思路

1. 边缘提取

Canny算法?

Canny算法包含如下几个步骤

1. 使用高斯滤波器
2. 计算梯度幅值和方向
3. 非极大值抑制
4. 双阈值检测
5. 抑制孤立弱边缘

Canny算法参考资料

但是把全部的步骤应用到一幅图像上有点过于浪费了，于是我提取了其中比较重要的两个方法来对图像进行边缘提取，其实也相当于在原来的边缘提取的基础上，加上了一个双阈值检测。

经过对大量的图像进行试验我发现，采用如下的简化后的Canny算法可以取得较好的效果。

1. ~~使用高斯滤波器~~
2. 计算梯度幅值~~和方向~~
3. ~~非极大值抑制~~
4. 双阈值检测
5. 抑制孤立弱边缘

即仅保留了计算梯度幅值和双阈值检测。

在这个算法中的边界提取算法我采用的是Robert算子，整个算法大概流程就是当我在计算Robert算子的结果是，根据其梯度值的大小然后利用预先设定的高低阈值将其区分成三种情况存入一个与原图像一样大的数组中。三种情况分别记为：

```
enum EageType
{
    NO, HIGH_TH, LOW_TH
    //非边界点，高梯度值边界点，低梯度值边界点
};
```

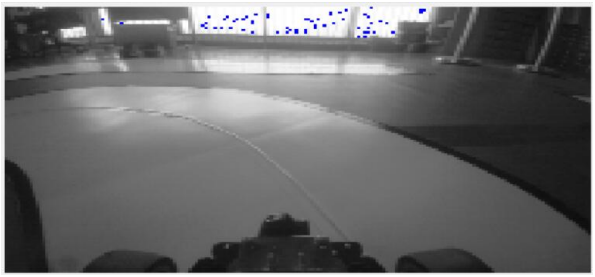
其中高梯度值边界点要满足其梯度值大于高阈值，低梯度值边界点满足其梯度值小于高阈值但是大于低阈值。然后将所有的高梯度值边界点视为真正的边界点，而低梯度值边界点的点需要满足其至少与一个高梯度值边界点相连才将其视为边界点。

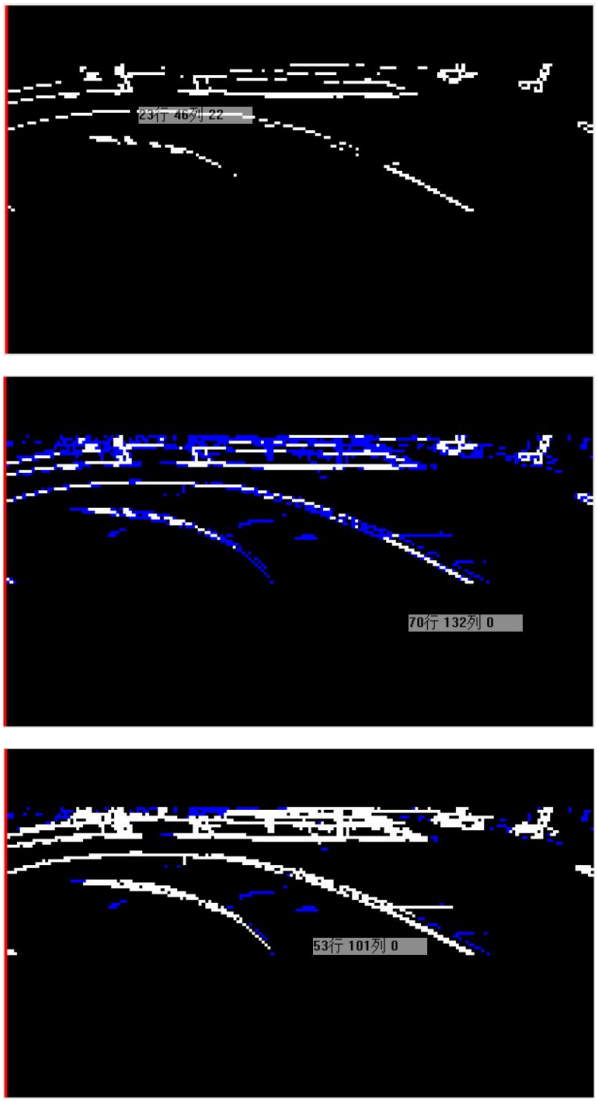
下面是边缘提取的算法和双阈值检测的算法

```
//边缘检测算子
HighThreshold = 40;
LowThreshold = 16;
int GetEageThreshold(int row, int col)
{
    col = MAX(col, LEFT_EAGE + 1);
    unsigned char* p = ImageData[0] + (row - 1) * IMG_COL + col - 1;
    signed int sum_1 = 0, sum_2 = 0;
    sum_1 += *p++; sum_2 += *p++;
    p += (IMG_COL - 2);
    sum_2 -= *p++; sum_1 -= *p++;
    if (sum_1 < 0)sum_1 = -sum_1;
    if (sum_2 < 0)sum_2 = -sum_2;
    sum_1 += sum_2;
    if (sum_1 > HighThreshold)return HIGH_TH;
    else if (sum_1 > LowThreshold)return LOW_TH;
    else return NO;
}
//抑制孤立弱边缘
void PressEage(unsigned char* source)
{
    for (int i = UP_EAGE; i <= DOWN_EAGE; i++)
    {
        unsigned char* p = (source + IMG_COL * i + 1);
        for (int j = 1; j < IMG_COL - 1; j++)
        {
            if (LOW_TH == *p)
            {
                if (IsStrongEage(p))
                    * p = HIGH_TH;
            }
            p++;
        }
    }
}
```

下面是效果图:
四幅图分别为

1. 原始图像
2. 没有双阈值检测处理
3. 白色为高梯度值边界点，蓝色为低梯度值边界点
4. 将所有与高梯度值相连接的低梯度值边界点记为真正的边界点。





从第四幅图中可以看出这种方法有效的将原本没有被检测出来的边界点检测出来了，同时又不会将某些非边界点误判为边界点。

全图边缘提取

由于采用高低阈值方法的特点，需要对整张图提取边缘，因此我在原来的图像数组之外重新创建了一个相同的二维数组来存放提取边缘的结果。即在这个数组中，只有0和非0两种元素，也代表着非边界和边界点两种情况。而在后面的处理中基本上都是针对这个数组进行的。

2. 寻找边缘算法（边界跟踪）

类似于一种沿着边界向上爬的感觉。目的是为了缩小边界搜索的范围，提高算法的效率。

基本想法就是利用边界的连续性，当你已经知道了当前一行的边界点的列值的时候，那么接下来要搜索的这一行的列值的大概范围也应该确定了。假设第n行的列记为x， 那么第n+1行的列值应该在x-d，x+d这个范围内，而这个d值一般就是跳变的阈值了。

如下为一段搜索左边界的代码：

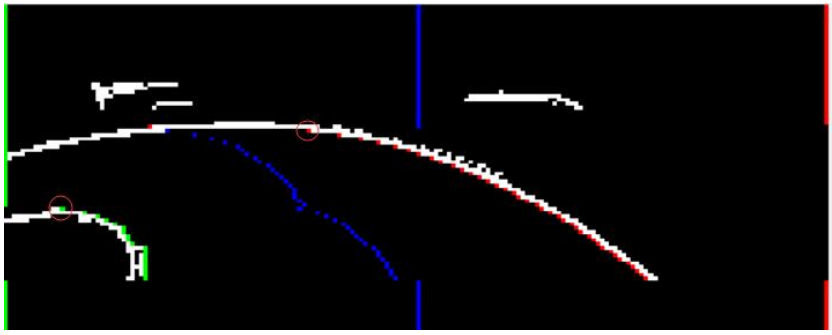
```
//=====================================================================//
//  @brief  :       搜左边线
//  @param  :       row行数 col列数
//  @return :       边界点所在列
//  @note   :       void
```

```
//=====================================================================//
int GetLL(int row, int col)
{
    int TmpCol;
    if (row == DOWN_EAGE - 1)
        col += 8;
    else
        col += 2;
    if (col > RIGHT_EAGE) col = RIGHT_EAGE;
    if (!IsEage(row, col))      //判断该点是否为边界点
        TmpCol = SearchLeftEage(row, col);
    else
        TmpCol = SearchRightNoEage(row, col) - 1;
    return TmpCol;
}
```

该段代码的输入为搜索的起始点，它首先将起始点的列坐标向赛道内移动2个像素点，然后对这个点做一个判断，判断是否为边界点，然后根据结果选择向左边搜还是向右边搜。一般如果该点不是边界点，那么一般要向赛道外面搜索，如果该点是边界点，那么一般向赛道内搜索。

搜索的停止条件我设为当前行和下一行的边界点的列坐标的差值大于某个阈值作为停止条件。并同时记录下来最后一个边界点。

就像下图中红圈圈标注的点就是最后的边界点。



3. 首行边界点的寻找

可以发现我们的搜索算法都是基于能够正确找到第一行的边界点，因此第一行边界点的正确性就决定了我们补图的成功与否。所以对于第一行，我们要尽量找到一个较好的方法能够以较高的正确率确定左右边界。

中间向两边搜索

从图像的中间列开始，往左搜左边界点，往右搜右边界点。这种方法的前提是起始列一定要在赛道内。

首行搜线遇到的第一个问题

一般情况下，大部分图像都能用这种方法找到正确的左右边界。但对于下面这幅图，似乎就没那么容易找到正确的左右边界了。

造成这种情况的原因是在对原始灰度图像边缘提取的时候存在某些高光和阴影的交界使得该部分的梯度值也很大。

对第一个问题的改进

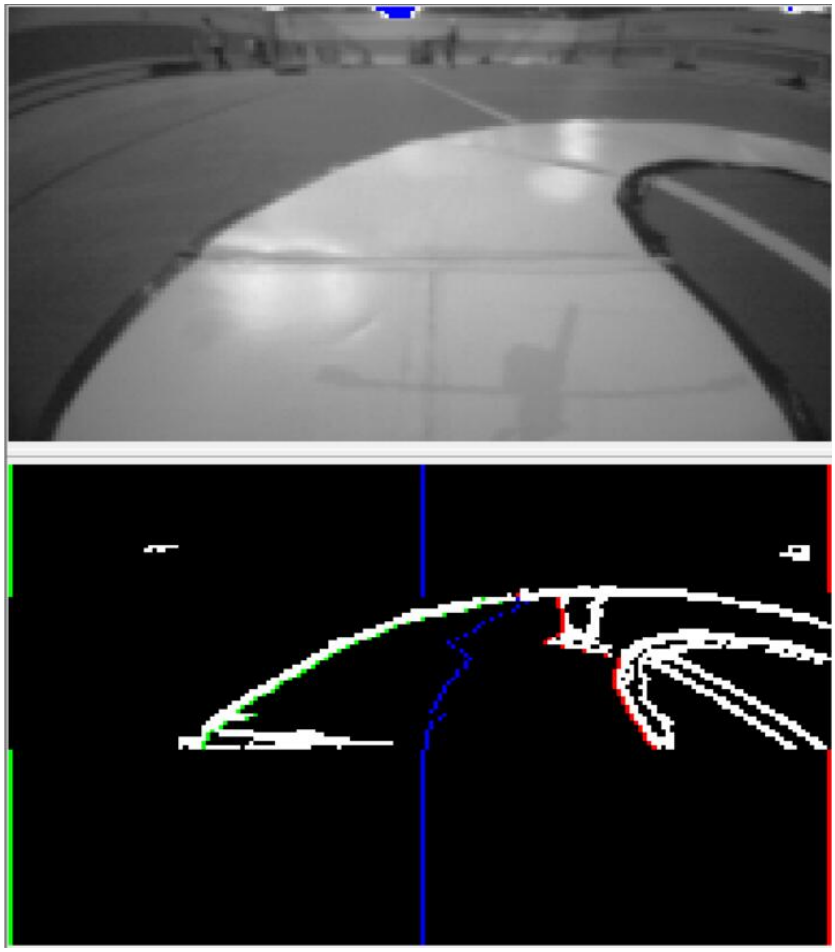
这是一种类似于灰度二值化的思想。由于我主要想处理的是第一行的例外情况，因此我首先对第一行计算一个灰度平均值，即：

$$gray\_ave = \frac{1}{IMG\_COL}\sum_{n=1}^{IMG\_COL} ImageData[DOWN\_EAGE][n]$$

接着我在第一行搜索的过程中判断边界点的时候，不是简单的值为HIGH_EAGE就认为是边界，而是要首先满足计算该点梯度值的四个点的灰度值至少有一个小于上面算出的灰度平均值。即：

$$GrayMatrix:\begin{bmatrix} Gray_{11} & Gray_{12} \\ Gray_{21} & Gray_{22} \end{bmatrix},\quad EdgeMatrix:\begin{bmatrix} * & * \\ * & HIGH\_EAGE \end{bmatrix}$$
$$s.t.\quad Gray_{11} < gray\_ave \quad or \quad Gray_{12} < gray\_ave$$
$$or \quad Gray_{21} < gray\_ave \quad or \quad Gray_{22} < gray\_ave$$

只有满足上述条件的点才被认为是第一行的边界点。

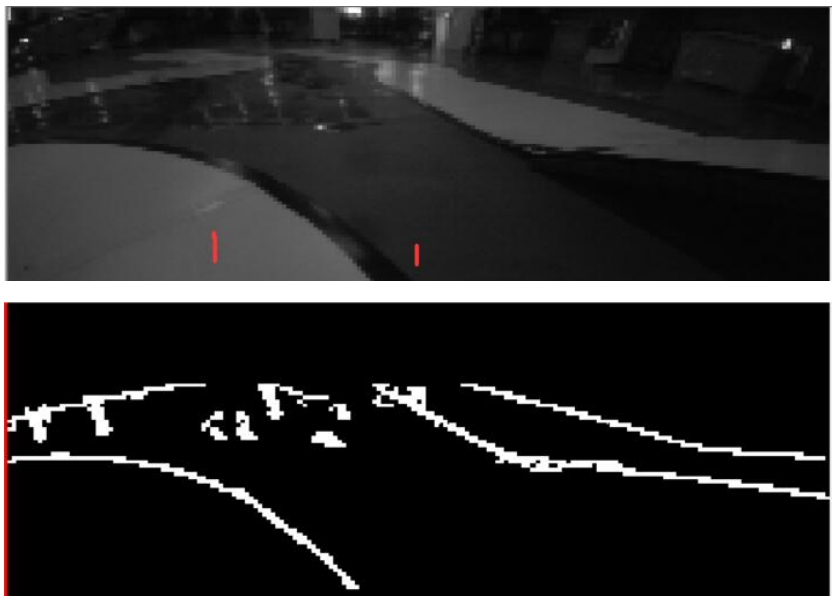


从上图中可以看出，左边成功的找到了正确的边界点，即使有白色的点在影响它。这种想法是基于在一行之中，赛道内的点的灰度值一般来说再低也不会低于这一行的灰度均值，而赛道两旁的黑胶带的灰度值一般来说会低于灰度均值。利用这种特性可以很好的区分开由于高光阴影造成的误判边界点。

首行寻线遇到的第二个问题

在一些较极端的情况下，我们的车可能会偏离赛道太远，容易使得图像的中间一列已经不再赛道内了，而这个时候如果我们还是按照从中间往两边搜左右边界的方法的话，显然是错误的了。这个时候我们就要根据图像内的信息来重新确定我们搜线的起始列，即不再从中间列开始搜索，而是从正确的属于赛道内部的点开始搜索左右边界。接下来就是如何通过图像来确定这个起始列，因为它有可能会偏向图像左边（车过于靠近右边赛道），也有可能会偏向图像右边（车过于靠近左边赛道）。

中间列可能有时候会处于赛道外，但是中间的中间，即四等分点则基本上不会处于对应方向的赛道的外面，因此修正后的起始列相对于中间列的偏移量就确定了。



对于上面这幅图，很明显它的中间列（红色竖线）已经在赛道外了，但是左边的四等分点（红色竖线）依旧处在赛道内，因此从这一列开始搜索就能确保是从赛道内往两边搜索了。而接下来就该确定这个四等分点到底是在左边还是在右边了。

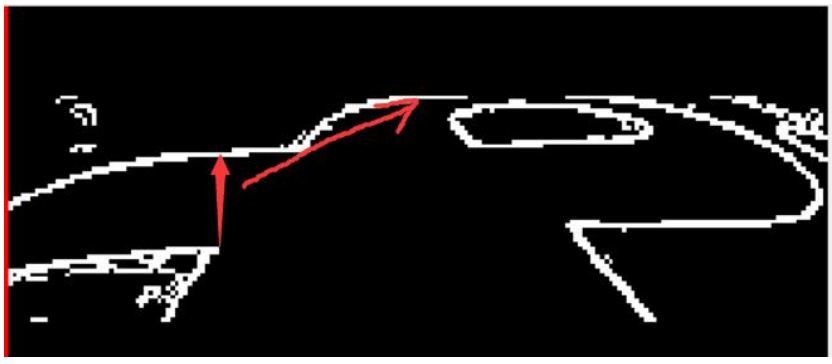
观察上面这幅图像可以看出一个很明显的特点，如果我把图像从中间分开，那么左边的灰度均值一定会比右边的高，同时左边的灰度均值一般会大于前面算出来的整行的灰度均值，而右边的一般会小于整行的灰度均值。这样我们就只需统计一下两边的灰度均值然后做个比较就可以确定到底是左四等分点还是右四等分点了。

```
伪代码
if left_gray > gray_ave and right_gray > gray_ave
    start_col = middle_col
elif left_gray > right_gray
    start_col = middle_col - 40
else
    start_col = middle_col + 40
```

3. 十字补线

十字主要就是找四个角点，然后连线，一般来说下面的两个角点通过前面的边界搜索算法加跳变检测就能判断出来，主要问题是在搜索上面的两个角点。

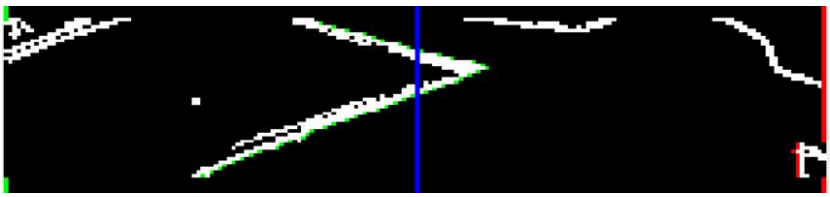
十字补线



通过前面的寻边界，能够确定左下角那个点，然后以这个点为起始点，先向上搜，直到搜到一个边界点，即上图中竖直方向箭头所指的那个点。然后以这个点为起点，再斜着搜索，同样是以当前点的坐标与下一个点的坐标距离之差大于预先设定的阈值作为搜索停止的条件。这样就能准确找出上方角点，然后连接上下两个角点即可完成对空缺部分赛道的补线。

斜十字补线

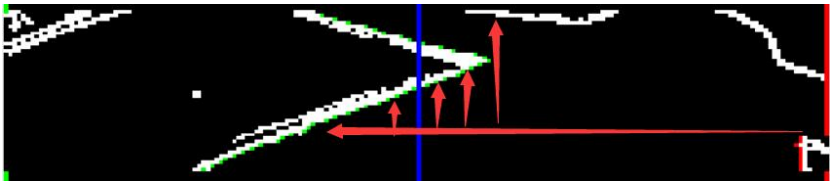
斜十字有别于平十字的一点就是可能会出现在角点处没有跳变。而对于上面两个角点的基本搜索思路和平十字一样。



这幅图像的左边角点就没有因为跳变而终止搜索，但是很显然，在那个地方也确实没有跳变。因此在每次搜索完左右两边边界的时候，还要再加入一个操作，以检验是否有上面这样的情况出现。

方法1

竖向搜索寻找跳变：



按照图中箭头的方向，先根据右边的点所在的行确定左边起始点所在的坐标，然后以这个坐标为起点从下往上搜，将搜到的点和上一次的点作差，直到出现大的跳变为止。

方法2

观察左边这条曲线可以得知，拐点相当于这条曲线的最大值，因此对于左边界数组求最大值的数组索引同样也可以求得拐点坐标。