循迹 ros 小车

笔者：纸壳小宝

目录：

1. 项目规划

该项目完成了一辆可以在一个由圆锥组成的道路环境下自动行驶的小车。行驶环境和小车如下图所示：

图 1.1：行驶环境

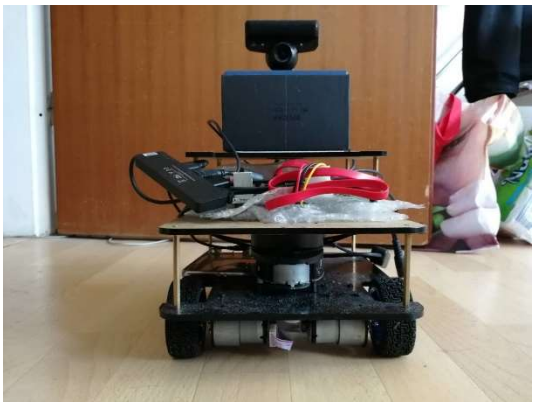

图 1.2：小车

1.1. 硬件

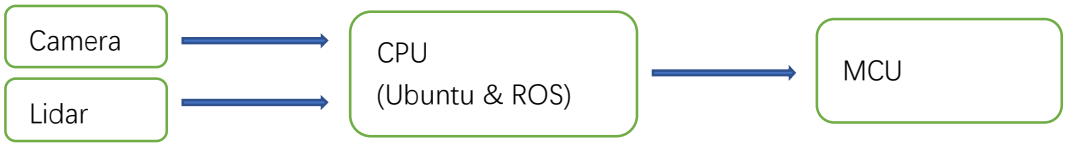

图 1.3：项目框架

- **Camera:** 普通的 usb 摄像头都可以，任务是识别圆锥
- **Lidar:** 我使用的是 2d 的 rplidar，任务是与图像融合，获取圆锥的位置
- **CPU:** 运行 ubuntu 和 ros，任务是处理激光雷达和摄像头数据，实现感知、规划和决策
- **MCU:** 获取 CPU 传来的控制指令
- **行驶环境:** 左边红色圆锥，右边黄色圆锥，终点蓝色圆锥

1.2. CPU 里的程序运行流程

1. 执行启动激光雷达的 ros 节点，发布/scan 话题
2. 运行 lidar_camera 节点，rosrun lidar_camera lidar_camera，程序里的任务：
   - 接收/scan 话题
   - 融合激光雷达和摄像头数据，识别定位圆锥
   - 局部路径规划，发布/cmd_vel 话题
3. 执行 CPU 和 MCU 通讯的 launch 文件

2. 摄像头激光雷达数据融合

具体内容在“激光雷达与摄像头融合”文件夹的 pdf 文档里

3. 圆锥的识别及圆锥位置的确定

3.1. 圆锥的识别

圆锥的识别流程：

- RBG 转 HSV
- 根据目标颜色进行图像二值化
- 对图像进行腐蚀、膨胀和中值滤波
- 找出目标颜色轮廓
- 对轮廓进行多边形拟合
- 找出顶点数在 3 到 10 之间的多边形
- 找出上边沿比下边沿小的多边形，认为它是圆锥

程序中该部分主要运用了 opencv，在 lidar_camera 包里的 cone.cpp 中，另外可以用资料里的 trackbar 程序来找出合适的 HSV 参数，而后放入 cone.cpp 的 inRange 函数中。这里的锥桶识别用的是传统的图像处理，读者也可以尝试深度学习的方法。

3.2. 圆锥位置的确定

1. 找出图中所有红色、黄色和蓝色的多边形并将他们的 ROI 信息（感兴趣区，在图像上的坐标和宽高）存入数组。
2. 对每个 ROI 中的图像执行 3.1 中的算法，依次识别 ROI 中是否有红色、黄色或者蓝色圆锥。
3. 找到圆锥后提取在该圆锥面里的激光雷达数据点，中值滤波后求这些点的平均值作为该圆锥的位置信息。括号内的第一个值是横向距离，第二个值是纵向距离，单位是毫米，程序在 lidar_camera 程序包中的 opencv_lidar.cpp 里，效果如右图所示：

4. 路径规划和决策

4.1. 边界的确定

设红色圆锥为左边界，黄色圆锥为右边界。为了确定左边界，必须先找出所有红色圆锥，在这些圆锥中找出离小车最近的那一个，并将它设为第一个圆锥，再找出离第一个圆锥最近的圆锥，并将它设为第二个，以此类推。右边界同理。伪代码如下：

Algorithm 1 sort cone

Input: allConeU,allConeV
Output: sortConeU,sortConeV

```
u[0] ← mapWidth/2
v[0] ← mapHeight
u ← allConeU
v ← allConeV
for i = 1 → coneAmount − 1 do
    for j = 1 → coneAmount do
        coneDistance ← (u[0] − u[j]) ∗ (u[0] − u[j]) + (v[0] − v[j]) ∗ (v[0] − v[j])
    end for
    for out = 0 → coneAmount − 2 do
        for in = 0 → coneAmount − 2 − out do
            if coneDistance[in] < coneDistance[in+1] then
                swap(u[in + 1], u[in + 2])
                swap(v[in + 1], v[in + 2])
                swap(coneDistance[in], coneDistance[in + 1])
            end if
        end for
    end for
    sortConeU ← u[1]
    sortConeV ← v[1]
end for
```

找到圆锥顺序后从第一个到最后一个依次在圆锥间用 bresenham 算法画线，这样就确定了边界线，并将边界线在另一个实时图像中显示（下图中的 online_map），达到俯视图的效果。找到边界线后就可以通过路径规划算法寻找合适的路径了。路径规划指的是机器人的最优路径规划问题，即依据某个或某些优化准则（如工作代价最小、行走路径最短、行走时间最短等），在工作空间中找到一个从起始状态到目标状态能避开障碍物的最优路径，以下尝试了两种方法。

4.2. 路径规划

- A*算法：

路径规划算法有很多，我最先尝试的是 A*算法，在计算机科学中，A*算法作为 Dijkstra 算法的扩展，因其高效性而被广泛应用于寻路及图的遍历，如星际争霸等游戏中就大量使用。它也是一种静态路网中求解最短路径最有效的直接搜索方法。经过试验发现规划出的路径更倾向于贴近边界，这时小车无法看到另一条边界，会使行驶不太稳定，并不适合局部路径规划，最终没有采用，但程序还留在 lidar_camera 包中，感兴趣的读者可以自行调试，效果如下图所示：

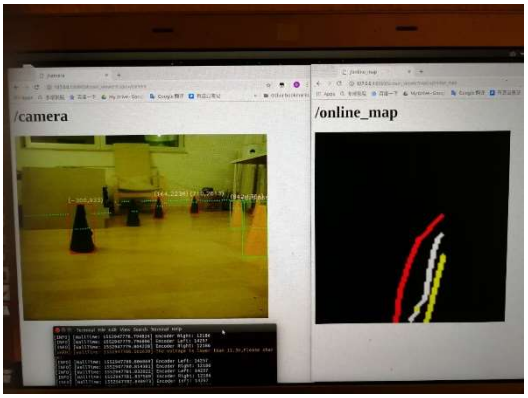

图 4.1：A Star

- 中线算法：

以中线作为路径是个相对折中的方法，这样小车能够更稳定地行驶。为了得到中线，我先将左边界线和右边界线进行了四等分，再计算左右对应等分点的像素位置的平均值，这样就得到了四个中间点，将中间点依次连接即可看作中间线。

因为边界线是由连续的像素点组成，并且像素点是按圆锥的顺序依次连接的，所以我把边界像素点放入了二维向量数组中，数组大小除以四就是等分点的间隔值，将计算结果四舍五入获得整数间隔值，程序在 lidar_camera 包中的 middlelane.cpp 中，效果如下图所示：

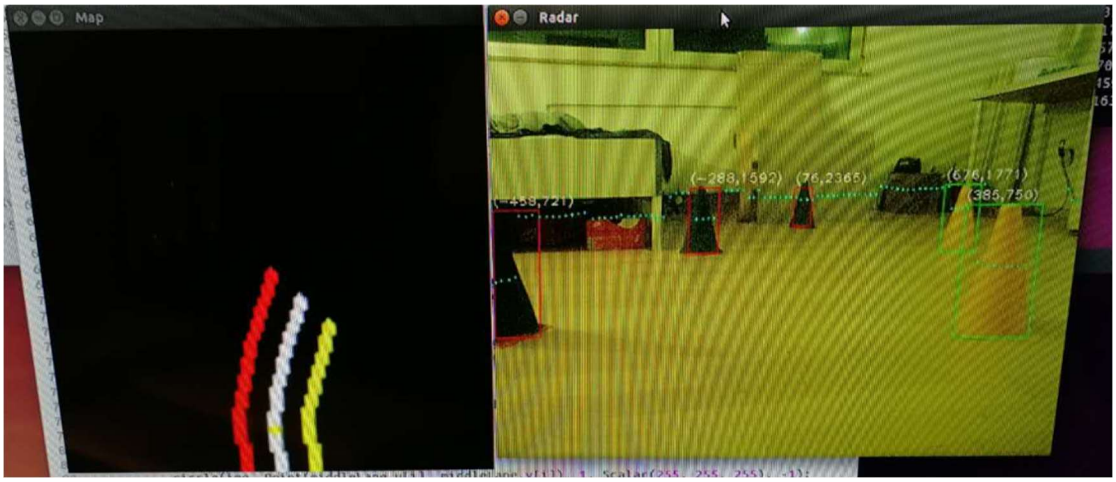

图 4.2：中线

小车在实际行驶中一共会遇到三种情况：

1. 两条边界都可以通过之前的算法正常识别出，则只需要按正常流程计算出中线
2. 只有一条边界，则在程序中事先设定好一个合适的另一边的边界线，再计算中线。
3. 没有边界，这很有可能是圆锥存在，而摄像头因为光线等原因没有正确识别出任何一个圆锥，这时可以使用上一帧的局部地图，如果持续一秒（经验值）都没有发现圆锥则停车。

4.3. 决策

找到中线后需要在中线上设定一个合适距离的局部目标点，将该点的横坐标与车体的横坐标（永远是横轴的一半）做差，这个偏差值乘以一定的比例系数就可以作为要输出的转向值，读者也可以尝试用三角函数求出偏角（如下图）乘以一定的比例系数作为转向角。同时输出固定的速度值，小车即可实现循迹的功能。

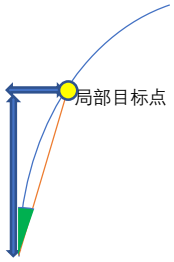

图 4.3：转向角