雷达站开源

雷达站

技术方向	开源部分	开源技术点	编写负责人
算法方案	算法全部	识别、定位	梁俊玮 杨彬

1.背景

22赛季我们的雷达站是借鉴上交的方案，其策略是：在目标识别上使用神经网络识别机器人，再配合雷达深度图对目标进行定位。由于分类方案有所不同，所以我们在定位机器人上并没有如同上交一样，使用深度图中装甲板部分来计算机器人深度，而是使用k聚类算法利用整个机器人的深度图计算机器人的深度，这样可以避免z轴突变的处理以及混战中因为遮挡而导致的深度无法获取。除此之外，在出现丢帧时，上交直接使用上一帧来代替本帧，而我们则使用了目标追踪算法来解决丢帧问题。23赛季相比22赛季变动不大，其中变化最大的是去除了UI部分，因为赛规禁用了云台手电脑显示雷达电脑的图像。其他变动大多是在代码架构方面。

2.功能

- 识别、分类、定位图中的敌我机器人。
- 获取并分析裁判系统信息，并加以处理利用。
- 以恰当的方式将视野、战略信息、决策信息反馈给操作手。

3.效果展示

参考网盘链接压缩包中名为“效果展示”的视频

链接：https://pan.baidu.com/s/1HrjQNzdx1k9zInmWbc8xXg?pwd=e37k

提取码：e37k

3.1依赖工具及环境

1. 依赖工具：autoware、vscode、pycharm
2. 软件环境：Ubuntu20.04、cuda11.1、cudnn8.0.5、tensorrt7.2.3.4
3. 硬件环境：主机（显卡为RTX 2070）、ouster激光雷达、502c迈德威视相机

3.2编译、安装方式

在以上软硬件环境下git clone源码，在SRRS文件夹中创建build文件，进入build文件打开终端，分别执行：

```
cmake ..
 make
 ./Radarstation
```

即可运行代码

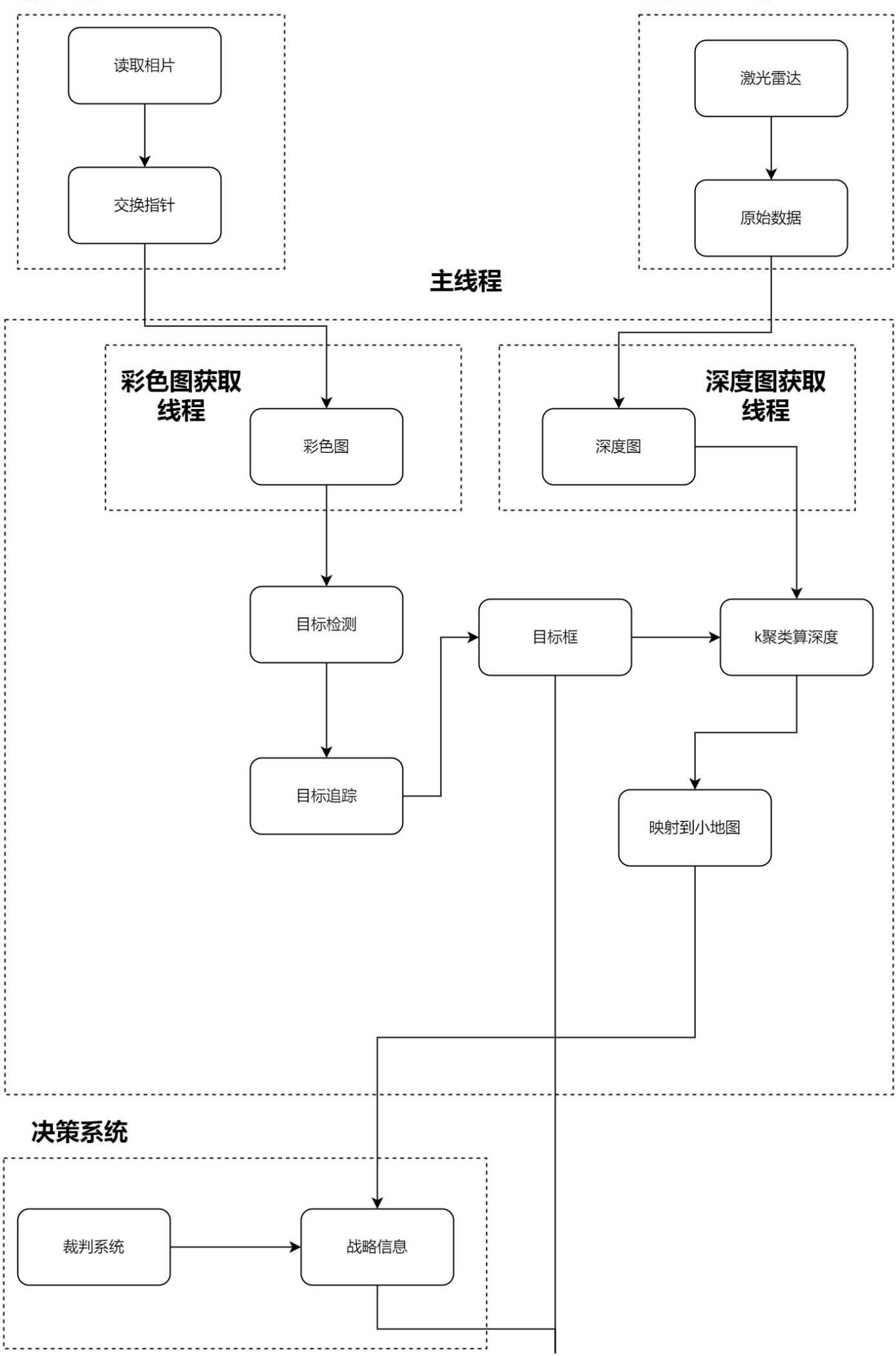
3.3文件架构及用途

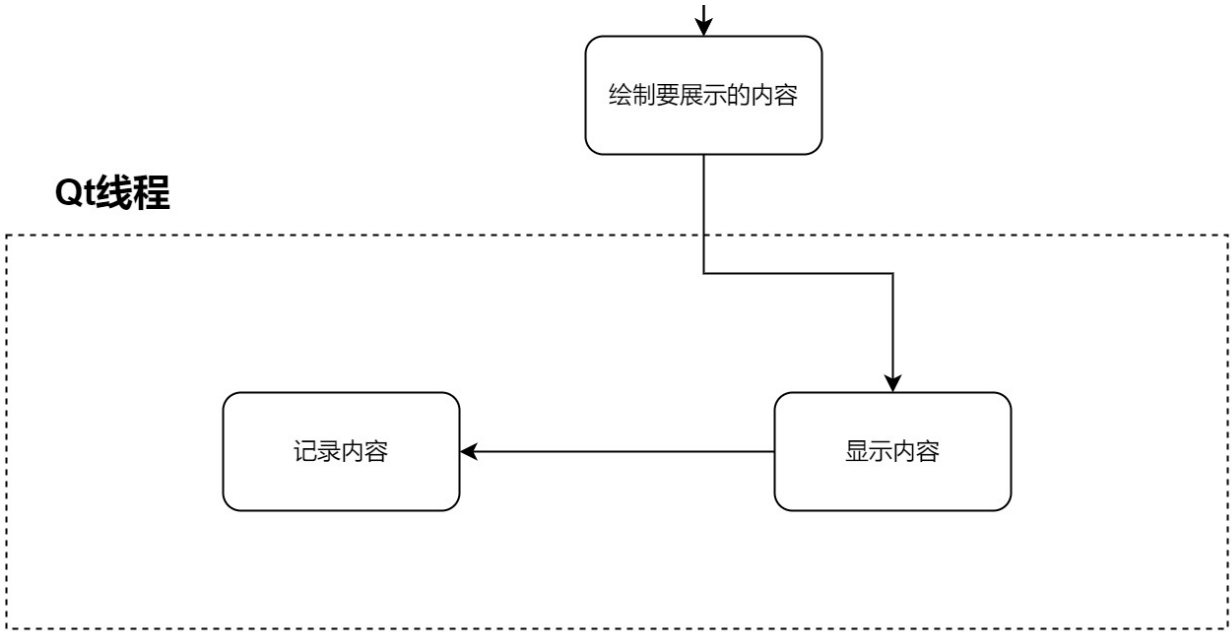
SRRS

- readme.md（说明文件如何运行）
- Setting_file（存放需要使用的.yaml文件，方便参数的修改）
- common（包含功能函数的底层函数）
- modules（包含可直接调用的功能函数）
- CMakeLists.txt
- main.cpp
- 调试流程.md

3.4软件与硬件的系统框图、数据流图

相机线程 **激光雷达线程**


读取相片
交换指针
激光雷达
原始数据
主线程
彩色图获取线程
彩色图
深度图获取线程
深度图
目标检测
目标追踪
目标框
k聚类算深度
映射到小地图
决策系统
裁判系统
战略信息



3.5原理与理论支持

3.5.1整体思路

我们的雷达站设置了多个独立运行的模块以及一条主线。独立模块有：相机模块(时刻获取图像)、激光雷达模块(时刻扫描获取深度信息)、深度图制作模块(时刻获取深度信息并制作深度图)、UI模块(时刻获取信息与图像并作展示，不过23赛季中云台手电脑被禁用，因此UI模块没有发挥作用)。

主线即主线程，当它到达某一步，就会从对应的模块获取信息，然后用于算法计算。它的大体流程是：同时读取图像与深度图，进行目标检测、分类与追踪，随后定位所有目标在小地图上的位置，然后综合这些位置与裁判系统的信息，输入行为树进行决策，随后信息展示在UI上。

3.5.2目标检测（将机器人识别出来）

我们使用的目标检测网络是YOLOX-s，使用的数据集为DJI ROCO并在该数据集中加入了自己制作的哨兵数据集，在官方提供的预训练权重的基础上训练了100个epoch，只检测car这个类(所有地面机器人)，其数据结果和实际效果如下表和下图所示。

Class	IOU	mAP	Recall
Car	0.50	0.99	0.999
Car	0.50：0.95	0.846	0.873

由于使用的是DJI ROCO的高清数据集进行训练，与我们工业相机获取的图像差距还是比较大，因此目标检测的实际结果并没有想象中那么好，特别是仰视或者俯视的视角下，即使在眼前，也可能检测不出来。

后来我们发现了训练图像的分辨率与相机分辨率一致时，效果很好，因此我们对DJI ROCO数据集进行裁剪，取车辆密度最大的区域进行裁剪，使训练图像的分辨率与相机分辨率一致。使用这种技巧后，雷达站的识别能力大大提高。

除此之外，为了缩短算法所需时长，我们使用TensorRT对推理进行加速，并部署到C++上。实际效果能达到平均3ms推理一张图。具体的过程为：将pth权重转为trt权重，随后序列化得到engine文件，比赛时反序列化engine，随后可以进行推理。

3.5.3目标分类（检测机器人上装甲板的类别）

目标分类实际是将目标检测得到的ROI区域裁剪出来作为输入，再次进行YOLOX-s检测，只不过这次检测的是装甲板。一个ROI区域通常会检测到两到三个装甲板，我们以置信度最大的那个装甲板上的数字代表这个机器人的数字。这样做的考虑是：由于机器人会进行小陀螺，在高曝光条件下，装甲板可能会模糊，因此选取置信度高的装甲板是为了防止受到由于运动模糊造成的误差的影响。

但实际上我们会将检测到的装甲板都记录下来，当后面发现有两台机器人的数字一样时，我们会进行比较，让某一个机器人选取置信度排名第二的装甲板作为代表。

在训练时，我们将DJI ROCO数据集中所有的机器人裁剪出来，得到若干张小图，以此进行训练。但由于不符合训练图像的分辨率与相机分辨率一致，再因为目标太小，目标分类在实战中，对远处目标和混战目标的准确度不是特别好。

我们将DJI ROCO数据集以8：2的比例分成训练集与验证集。在训练后我们使用验证集进行实验，测试在不同置信度阈值下分类得准确率，得到的结果如下表：

置信度阈值	分类准确率
0.25	81.7%
0.10	91.7%
0.01	95.1%

通过观察实验结果我们发现，随着置信度阈值的降低，分类准确率进一步提高。我们也查看了实验过程中的数据，发现分类错误的原因，99%是因为检测不到装甲板，而非检测不准确，所以我们可以大胆地设置较低的置信度阈值。

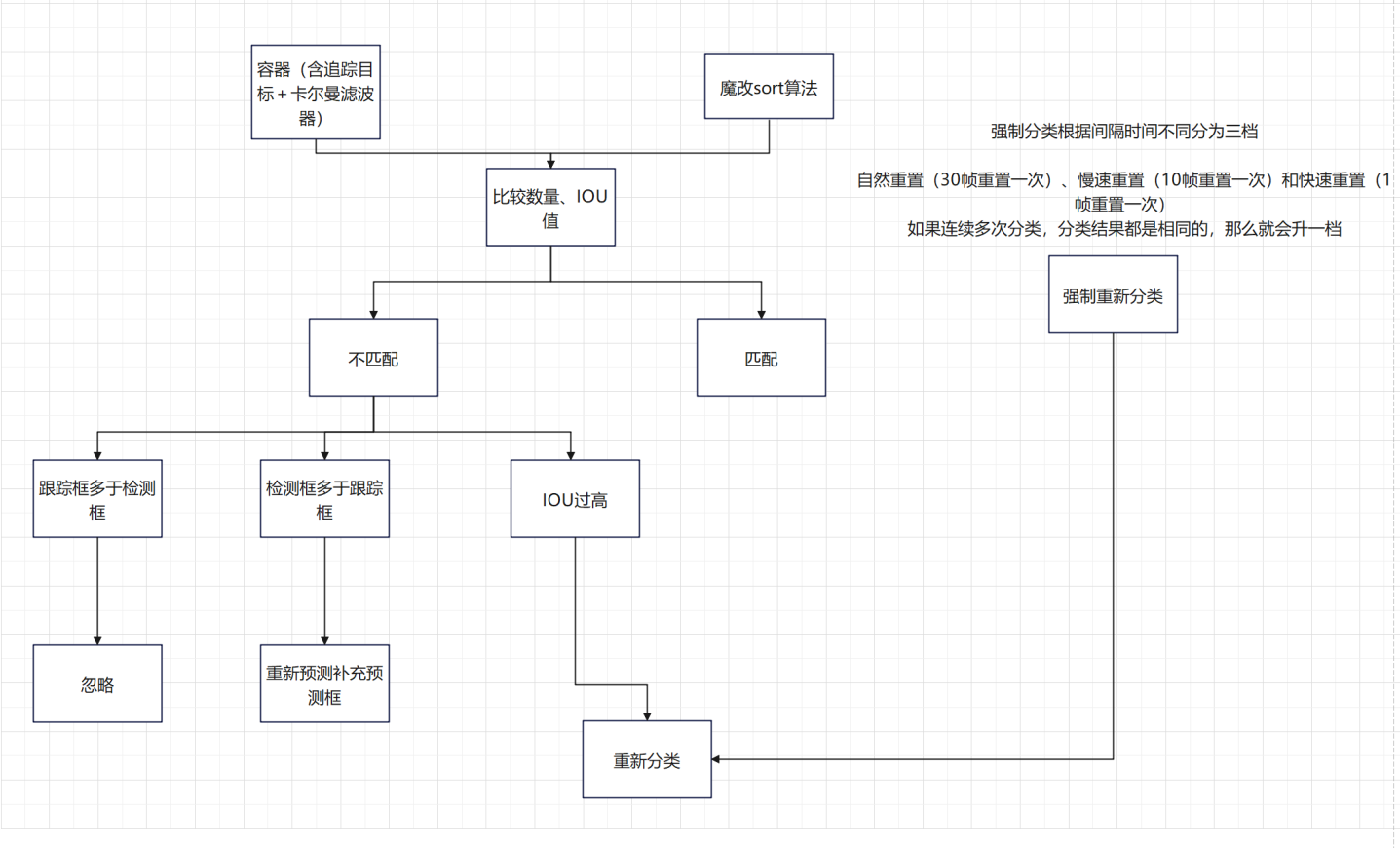
目标分类同样使用TensorRT进行部署，虽然速度还是3ms一帧，但由于有多少个ROI就要进行多少次检测，随着目标的个数增加，分类带来的时间消耗不是我们能够承受的，因此我们使用了目标追踪。

3.5.4目标追踪（追踪机器人装甲板）

为了解决目标分类耗时过久以及解决小陀螺等操作带来的运动模糊等问题，我们使用了目标追踪。

我们使用魔改之后的SORT算法而非deepsort算法，是因为魔改SORT算法可以令程序运行耗时较短，同时魔改SORT算法更加贴合当前环境，更易于实现。原本的SORT算法的大致原理是：算法维护一个容器，用以存储被追踪的目标以及其对应的卡尔曼滤波器。每帧的开始时，SORT算法会调用目标检测算法，获得图中的目标检测框，随后将检测框与已存在于容器中的跟踪框进行匹配。而当SORT算法判断出本帧检测到的物体就是上一帧存在的物体后，我便可以不对此机器人进行目标分类，而沿用上一帧的结果即可。但原本的算法容易出现ID switch和分类错误但一直追踪成功而导致的分类死锁问题。为了解决这两个问题，我们对原来的算法进行了魔改，核心的思路是在触发某些条件时，对目标进行重新的分类。设置了两种条件进行重新分类。第一是当检测到两个机器人IOU大于某个阈值时（即机器人正在交汇）会对两个目标进行重新分类，第二是当追踪器连续追踪一段时间后进行强制重新分类，根据目标初次分类结果将目标分为自然重置（30帧重置一次）、慢速重置（10帧重置一次）和快速重置（1帧重置一次）共三挡。初次分类时分类器给出的评分越高(说明分类器信任这个分类结果)则重置时间越慢。但由于相机性能问题，越远的目标分类概率会降低，但它们的分类结构有些是正确的，如果频繁重新分类会浪费算力，因此我们还设置了升档机制，如果连续多次分类，分类结果都是相同的，那么就会升一档。这样做可以使得分类算法分类的次数减少，提高帧率。

具体追踪流程如下：



实际效果如下图所示：

对于残影物体能够锁定：

对于被障碍物阻挡的物体也可以若干帧的锁定：

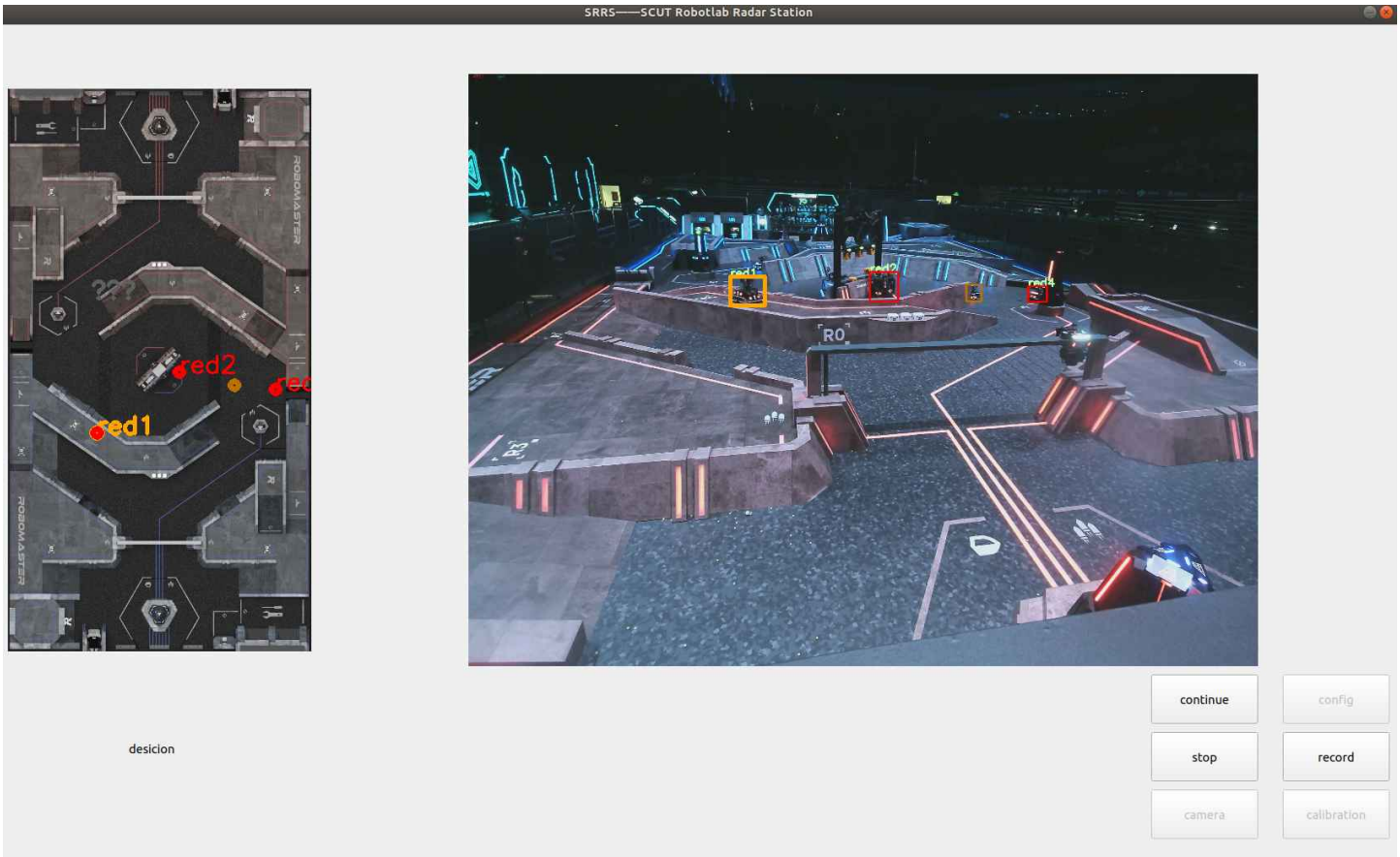
3.5.5小地图映射（将检测到的机器人映射到小地图上）

3.5.5.1三层透视变换法

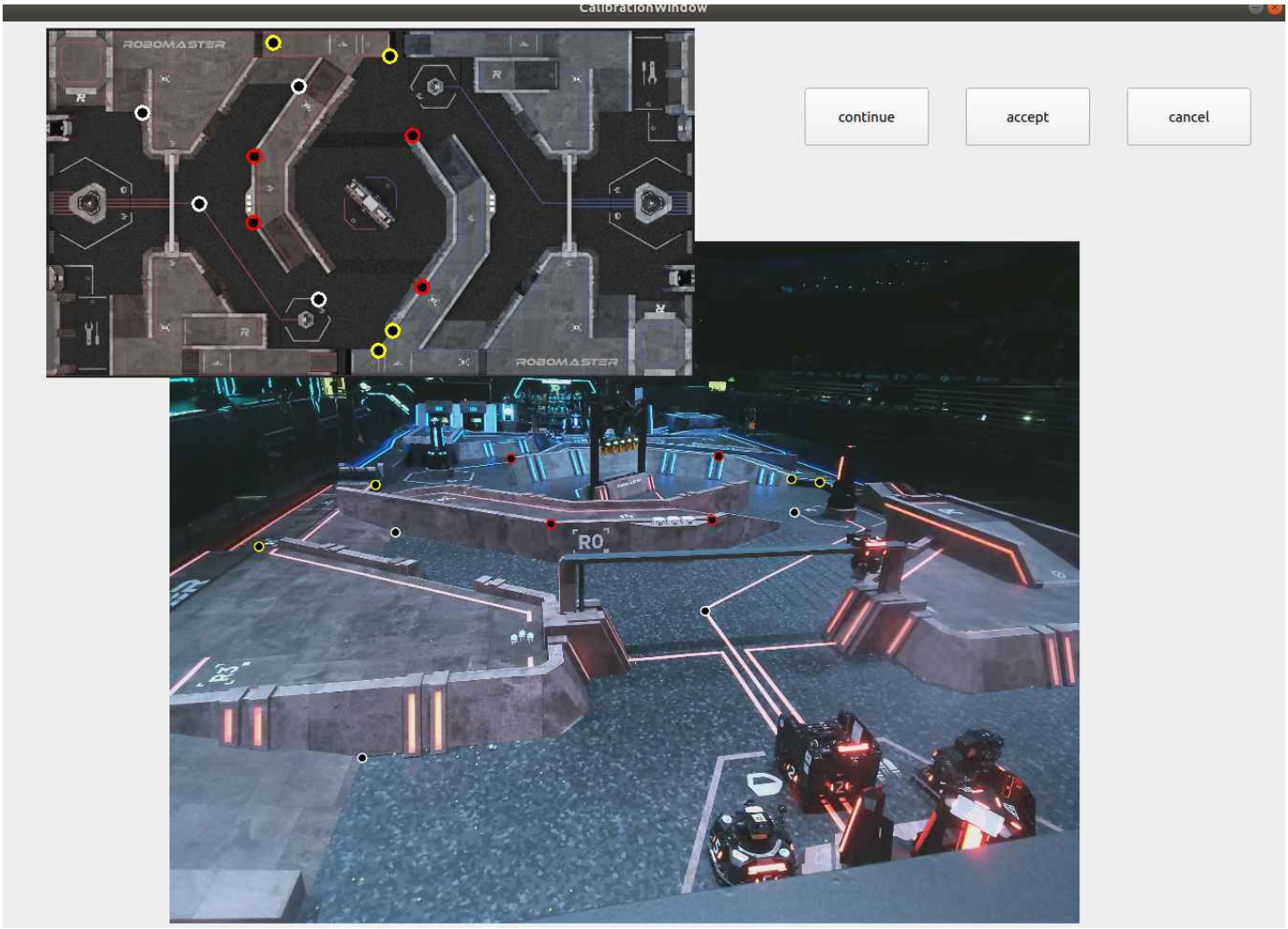
我们的小地图映射方案有两个，在v1.0版本中，我们使用的是三层透视变换方案。

众所周知透视变换会损失高度信息，如果选取地面上四点作透视变换，那么高地上的目标就会映射错误。因此我们把赛场分割为三层（地面，中层，高地层），计算三层透视变换矩阵。对于一个物体，我们先使用高地层的变换矩阵进行计算，如果目标映射后落入地图上高地的区域，那么我们就确定这个物体是高地层的物体。如果不落入高地区域，那么再使用中层变换矩阵，查看是否落在中层区域。如还不是中层区域，则认为其处于地面层，使用地面层变换矩阵得到其在小地图上的坐标。

这种方法对于己方半场的目标的映射效果很好，但对于敌方半场的映射效果误差很大，而且物体在上坡时或者靠在高地边缘时，映射也会不稳定。下图为映射效果图：



而且这个方法还有一个很大的问题，就是它在比赛开始前要标定3*4=12个点，赛场上时间紧张，很难拿出这么多时间进行标定，所以这个方法在后期作为我们的备用映射方法，当激光雷达故障时使用。下图为需要标定的点：



3.5.5.2激光雷达k聚类映射法

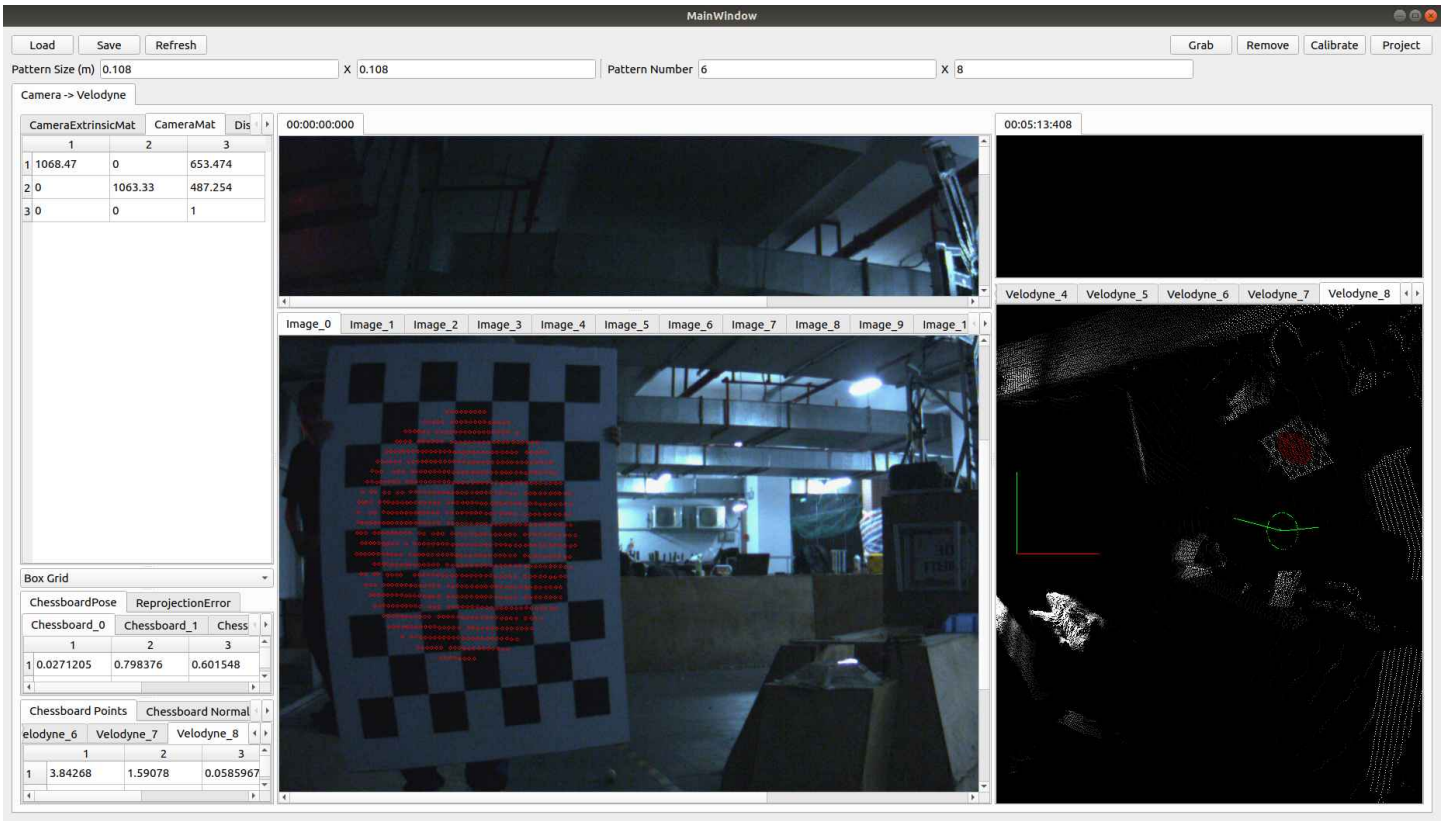
第二个方法我们借鉴上交的方案，即深度图与彩色图对齐。但上交是利用装甲板框内的深度信息，而由于我们只有机器人的检测框，没有装甲板的检测框，因此我们做了些许创新，我们使用k聚类来处理机器人检测框内的深度信息。之所以要进行改动，是因为机器人检测框内存在着背景信息，如果我们直接使用如上交方案的计算平均值，会出现极大的误差。

1. 深度图制作

在说深度信息处理前，我们先来说深度图的制作，根据下面公式我们把点云数据从激光雷达坐标系下投影到像素坐标系下。

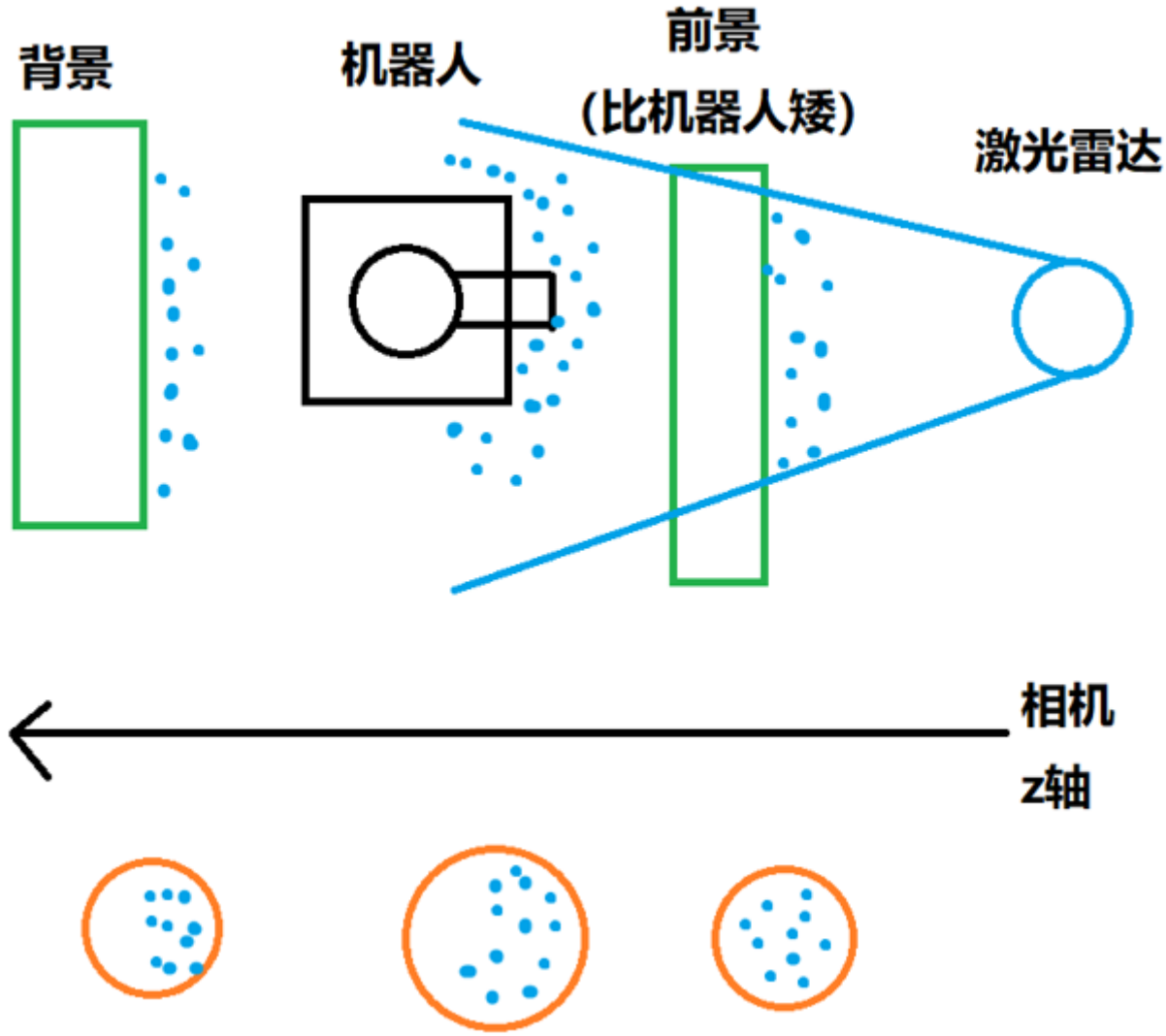
$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & 0 & u_0 & 0 \\ 0 & a_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0_3^T & 1 \end{bmatrix} \begin{bmatrix} X_l \\ Y_l \\ Z_l \\ 1 \end{bmatrix} = M_1 M_2 \begin{bmatrix} X_l \\ Y_l \\ Z_l \\ 1 \end{bmatrix}$$

其中 (X_l, Y_l, Z_l) 为点云在激光雷达坐标系下的笛卡尔坐标，其中 M_1 为相机的内参矩阵， M_2 为由相机到激光雷达的外参矩阵。我们使用Autoware的标定工具包，根据棋盘标定法，联合标定激光雷达与单目相机，从而获得内参与外参矩阵：



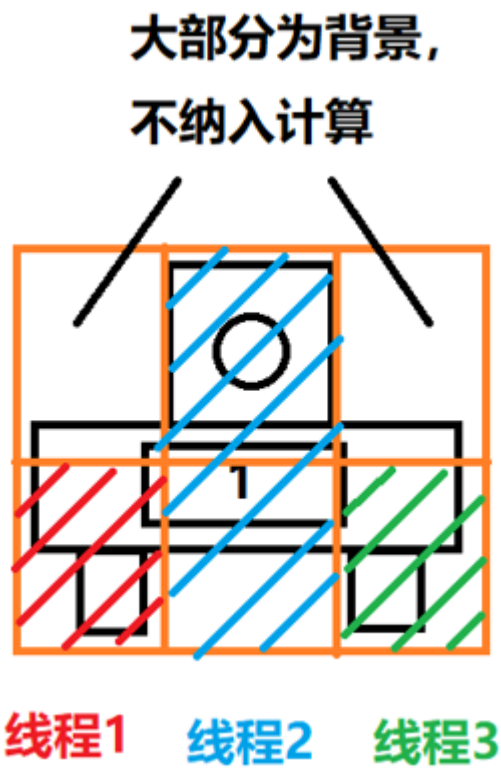
而 (u, v) 为点云投影到图像后的像素坐标。我们会新建一个与相机图像长宽一样的矩阵，并将计算出来的深度信息 Z_c 存储到对应的像素中，如果有多个点云的像素坐标一致，则取 Z_c 小的那个。随后我们便能得到一张深度图。对于每一帧的检测，都会用该帧的信息制作一张深度图。深度图效果如下（为了便于理解，我们按照深度信息大小对圆点进行了缩放，实际深度图需要肉眼仔细分辨才有此效果）：

2. **深度信息处理**



我们设计如上图的模型：激光束从机器人前方打过去，会碰到机器人前方的障碍物(掩体、台阶)，然后才打到机器人身上，随后有些光束没打到机器人，但会打到机器人身后的背景。因此激光雷达获得的点云应该分为三堆，分别在障碍物、机器人、背景的附近，因此可以把该问题抽象为一维的聚类问题。

因此我们使用K聚类处理深度信息。我们令K为3，分别代表前景(机器人前方矮小的障碍物)、中景(机器人)、背景的深度值(机器人背后的障碍物)。首先计算检测框内点云的深度平均值mean_range，以此值为中景的初值(因为大部分点云还是在机器人附近的)，前景与背景的初值为中景初值+1.5与中景初值-1.5(单位为m，1.5是参考制作规范中机器人的限制尺寸得到的经验值)。



为了进一步优化算法速度，我们将机器人的检测框均匀地分为六个格子，其中左上角与右上角的格子大部分为背景信息，可以不纳入计算，另外的四个格子可以分成三个线程单独进行聚类计算，最后再汇总结果。而算法迭代结束的条件为：当迭代次数超过一定次数或者前后两次结果之差低于0.001停止迭代。

我们取中景的结果作为这个机器人的深度代表值 $Z_{\text{represent}}$ ，代入下文公式计算得到机器人世界坐标。

但在实战中，大多数时候机器人前方没有障碍物，所以前景和中景的代表值很接近，甚至有时候中景聚类代表值与背景聚类代表值都代表了背景，只有前景聚类代表值才代表机器人的深度值。这个时候就会产生错误，因此我们可以将公式改为深度值的加权平均，权重由点云数量决定。分别令前景、中景、背景的聚类代表值为front_range、middle_range、back_range，而属于前景簇、中景簇、背景簇的点云数据为front_num、middle_num、back_num。那么加权公式如下：

---

all_num=front_num+middle_num+back_num

z_represent=
(front_range*front_num+middle_range*middle_num+back_range*back_num)/all_num

---

由于点云大多是打到机器人身上，背景比较少，再加上我们独特的深度信息获取逻辑，可以说95%以上的都是前景的点云，这样做使输出结果更稳定。

随后我们使用以下公式，就能获得机器人的世界坐标 (X_w, Y_w, Z_w) ，取世界坐标的 (X_w, Y_w) ，就能获得小地图上的位置。

$$Z_{\text{represent}}\begin{bmatrix} a \\ b \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & 0 & u_0 & 0 \\ 0 & a_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0_3^T & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = M_1 M_3 \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

其中 (a,b) 为物体检测框的中点的像素坐标， $Z_{\text{represent}}$ 由上文的K聚类获得，而 M_1 和上条公式一样，是指内参矩阵，而 M_3 则是指由相机坐标系到世界坐标系的外参矩阵，通过开场标定并使用PNP算法得到。具体标定的点如下图黑点：

实际的定位效果很好，受到遮挡时仍然能稳定识别与定位：参考网盘链接压缩包中名为“定位效果”的视频：

链接：https://pan.baidu.com/s/1HrjQNzdx1k9zInmWbc8xXg?pwd=e37k

提取码：e37k

3.5.6决策

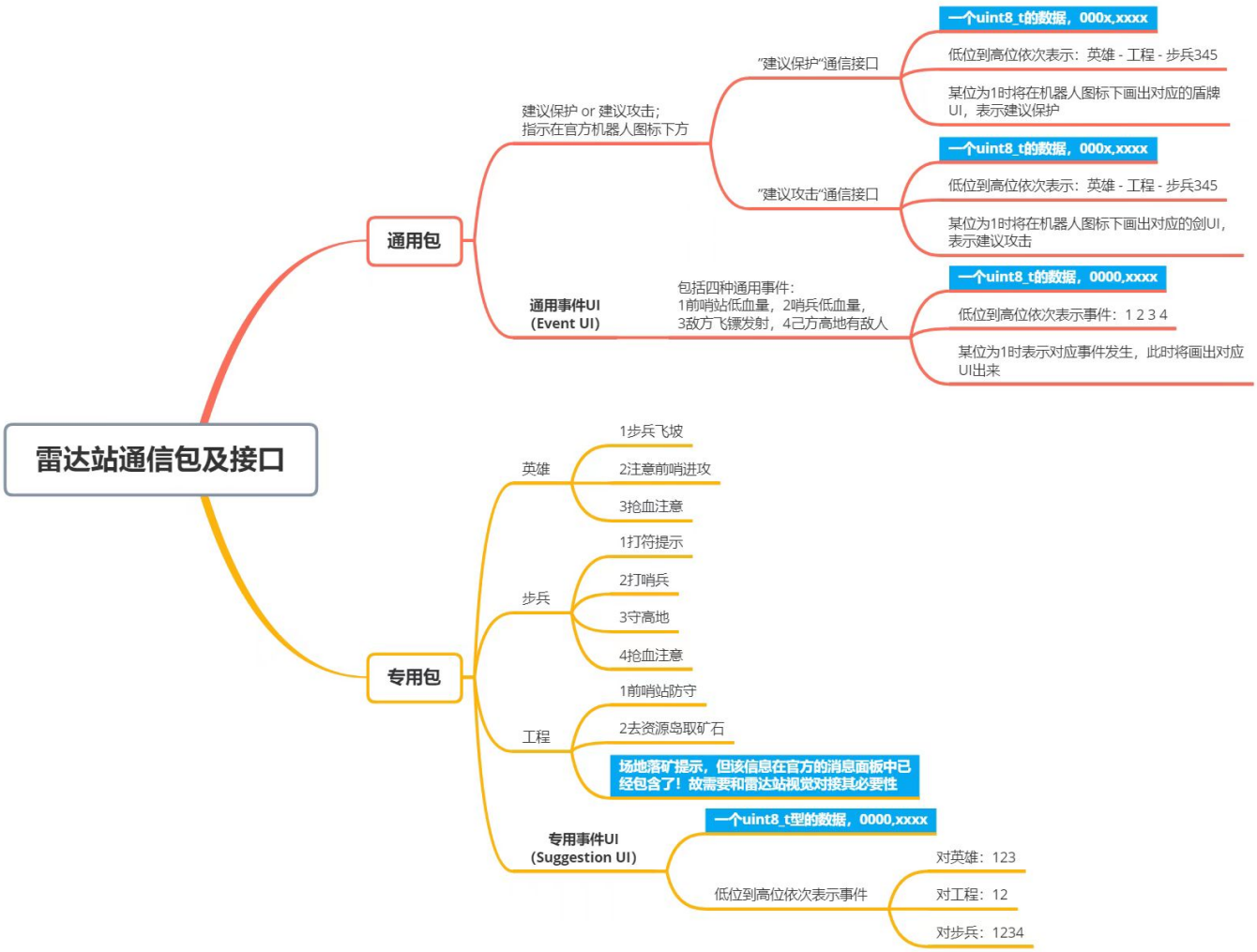
3.5.6.1决策信息来源

决策信息主要来自两方面，敌我位置信息与裁判系统信息(时间、单位血量、场地交互信息)

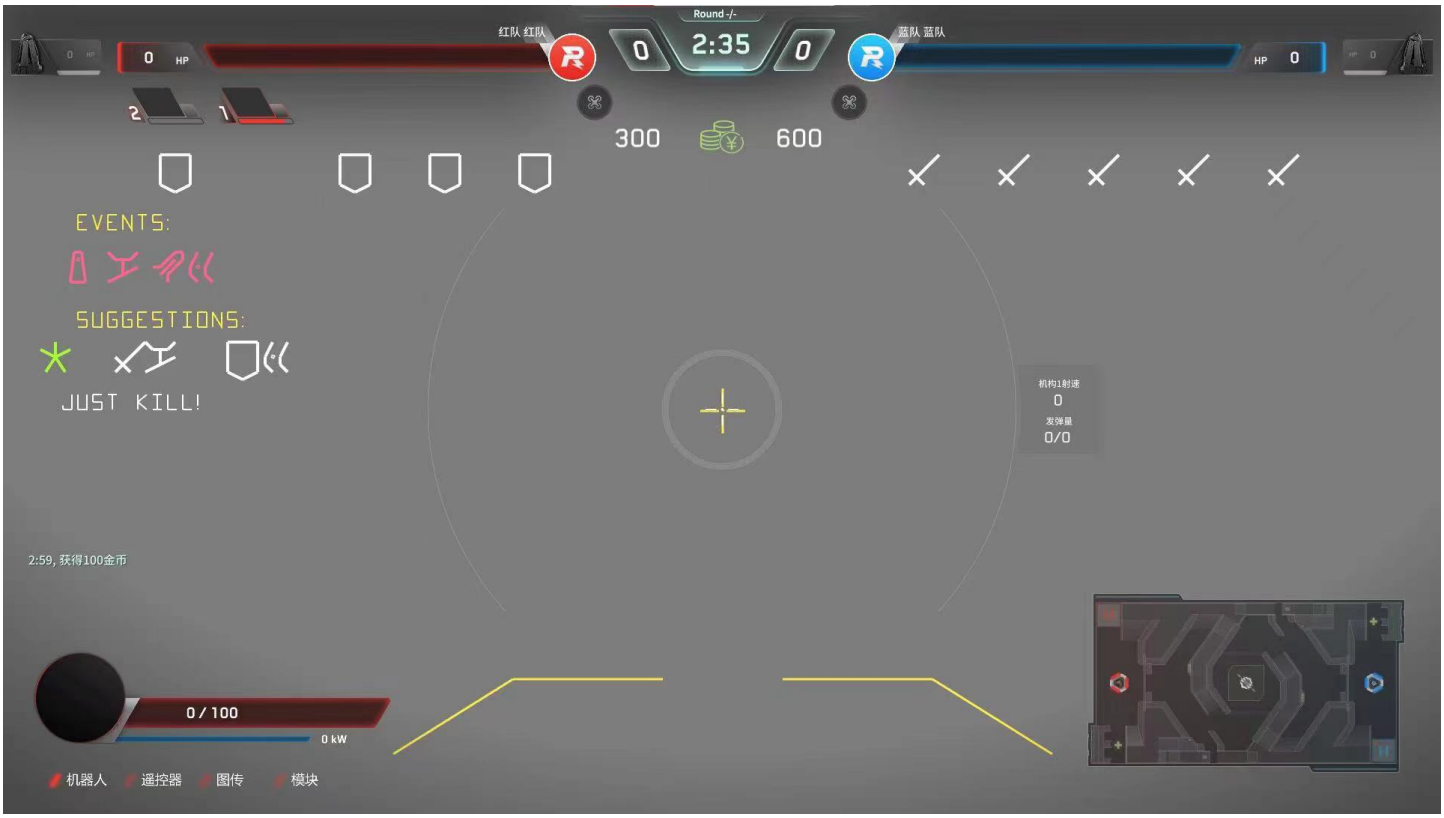
3.5.6.2决策方法

将得到的信息输入行为树，便可以更新各种标志位，在后续过程中，如果判断标志位为真，则执行对应的反应。反应分为两种，一种是通过裁判系统将策略发送到操作手端并在画面上画出来，第二种是直接在雷达的UI上显示。

雷达的通讯包有：



而操作手画面上的UI有（左侧的EVENTS和SUGGESTIONS和上方的盾牌与剑）：



3.5.6.3决策原则

我们的决策大多是基于时间，比如说落矿倒计时，首次大小能量机关的开启倒计时，前哨无敌时间解除的提醒。这些提醒与倒计时，基本只要比赛时间到便可以触发。

但从裁判系统读取比赛时间并没有这么简单，在测试中我们发现，偶尔读取不到比赛时间。因此我们内置了一个同步的计时器，当读取不到裁判系统时间时继续计时，不耽误行为树的决策。因为内置计时器会随时间产生误差，所以每次读取到裁判系统时间时，会对内置计时器进行修正。

第二个原则是基于单位血量，比如说当我方单位血量低迷时，在UI上会框住它的标签，在操作手页面上也会有护盾的标志；当对方单位血量低时，UI上会框住它的标签，且操作手页面有剑的标志；当建筑血量低时，操作手页面会有抢血的标志，UI上会有状态显示；哨兵阵亡(血量为0)时基地虚拟护盾消失。

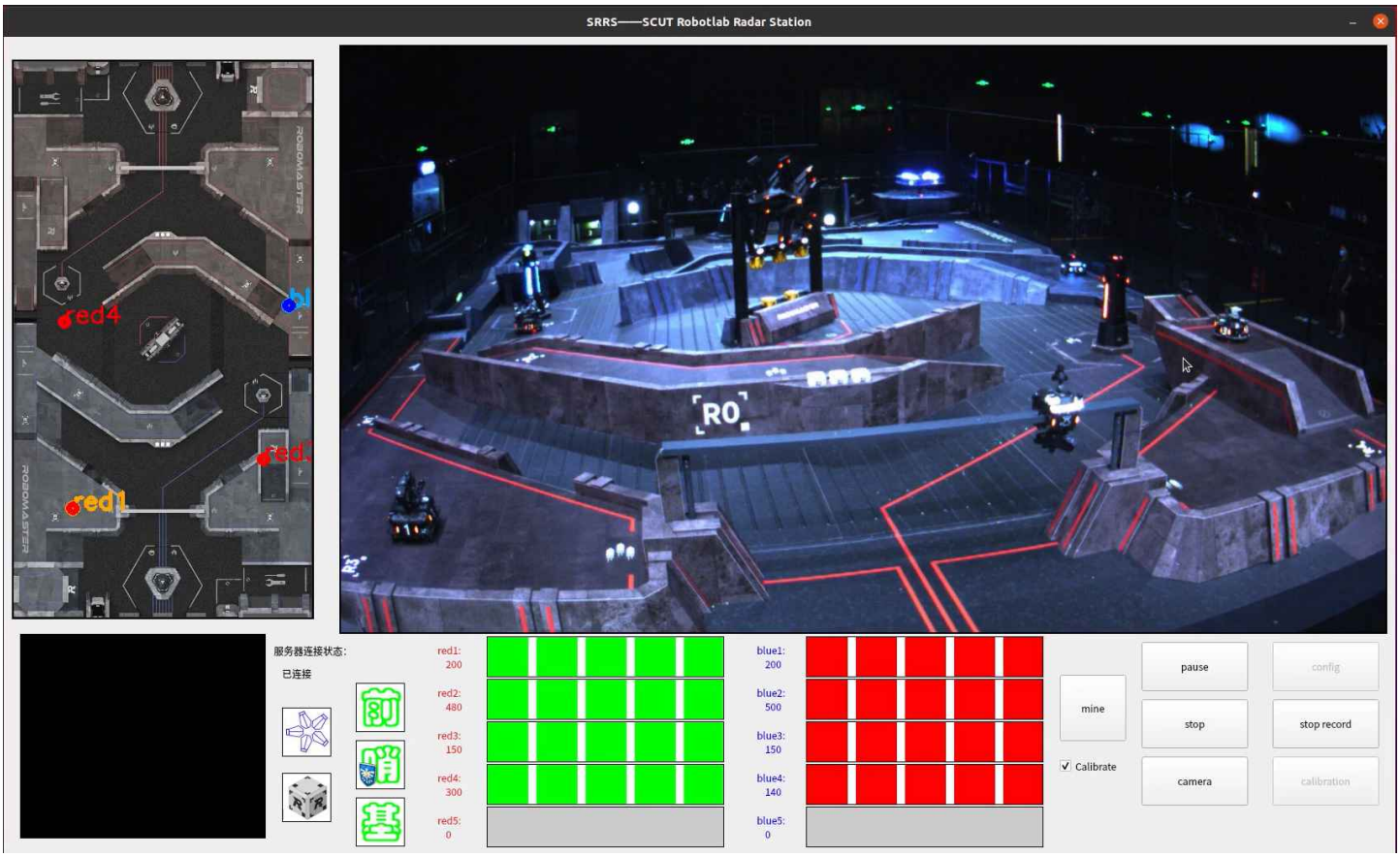
第三个原则是位置原则，通过检测目标在小地图上的位置是否属于某个区域，来判断目标是否入侵或者是否飞坡，从而提出警示。但实际上该功能并未实装，只存在代码中。因为前面的检测的精度尚未满足要求，我们持保守的意见。

实际上决策功能发挥的作用并不多，云台手基本只使用倒计时功能。

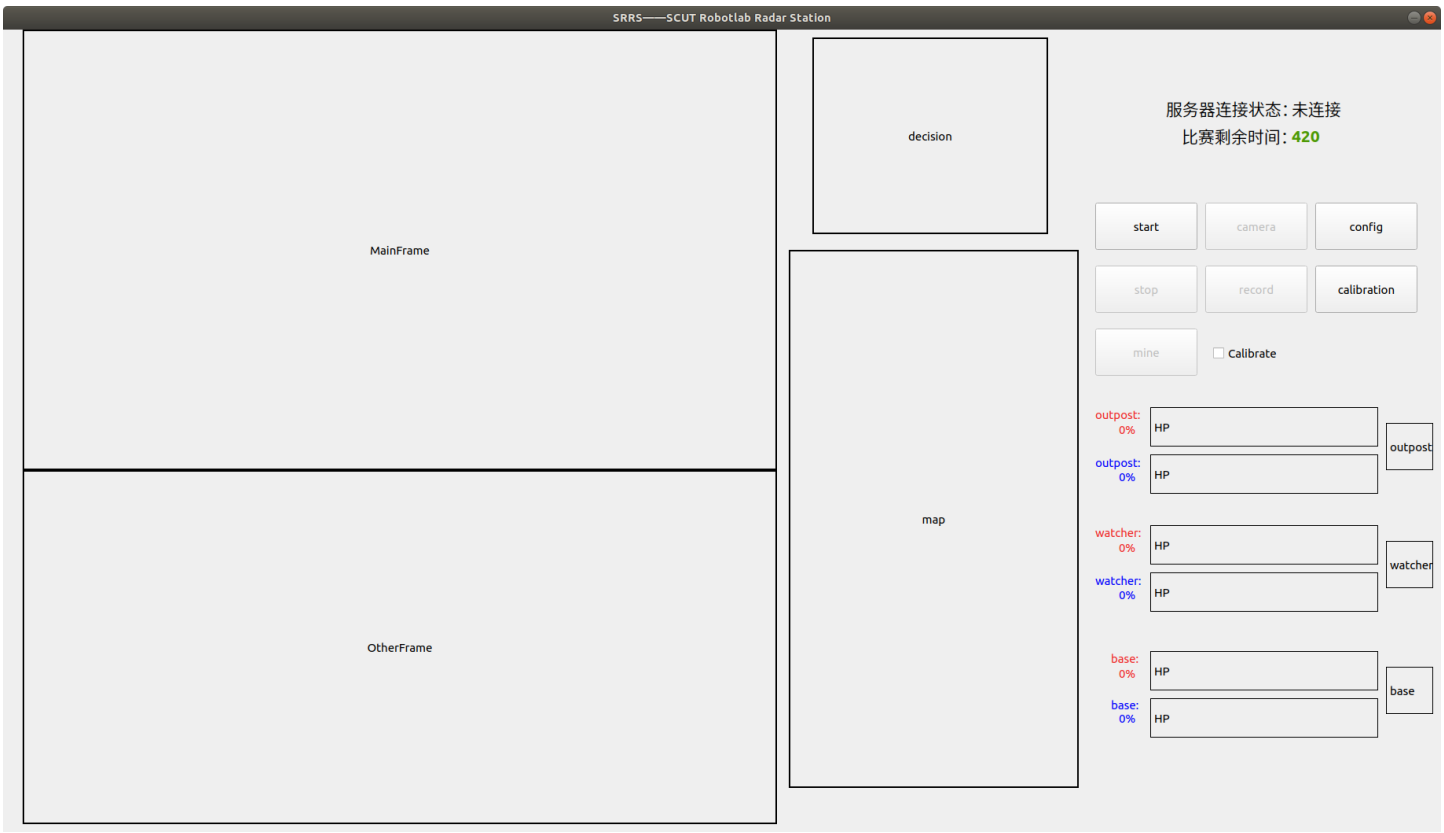
3.5.7UI设计

UI设计方面，我们使用了QT5，在本赛季，一共开发了两个版本的UI。两个版本大体上都分为主页面、配置页面、相机调整页面和标定页面。页面之间的切换主要靠按钮实现。他们的差别在后文描述。

1.主页面



第一代UI

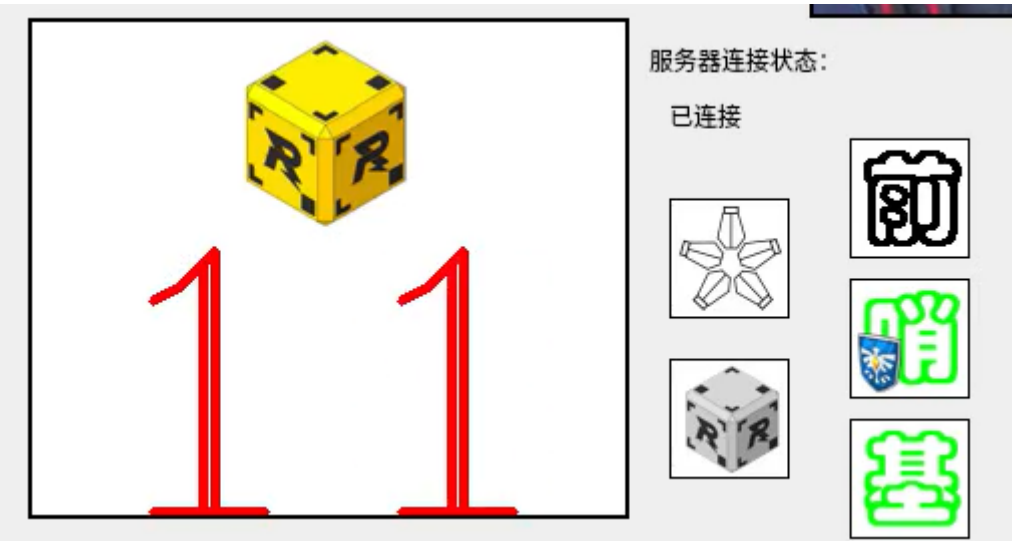


第二代UI

主页面包括主相机画面和小地图的展示，血量实时展示，矿石、神符、基地血量、前哨站血量、哨兵血量的状态显示，落矿、打符倒计时提醒以及落矿区域放大窗口。

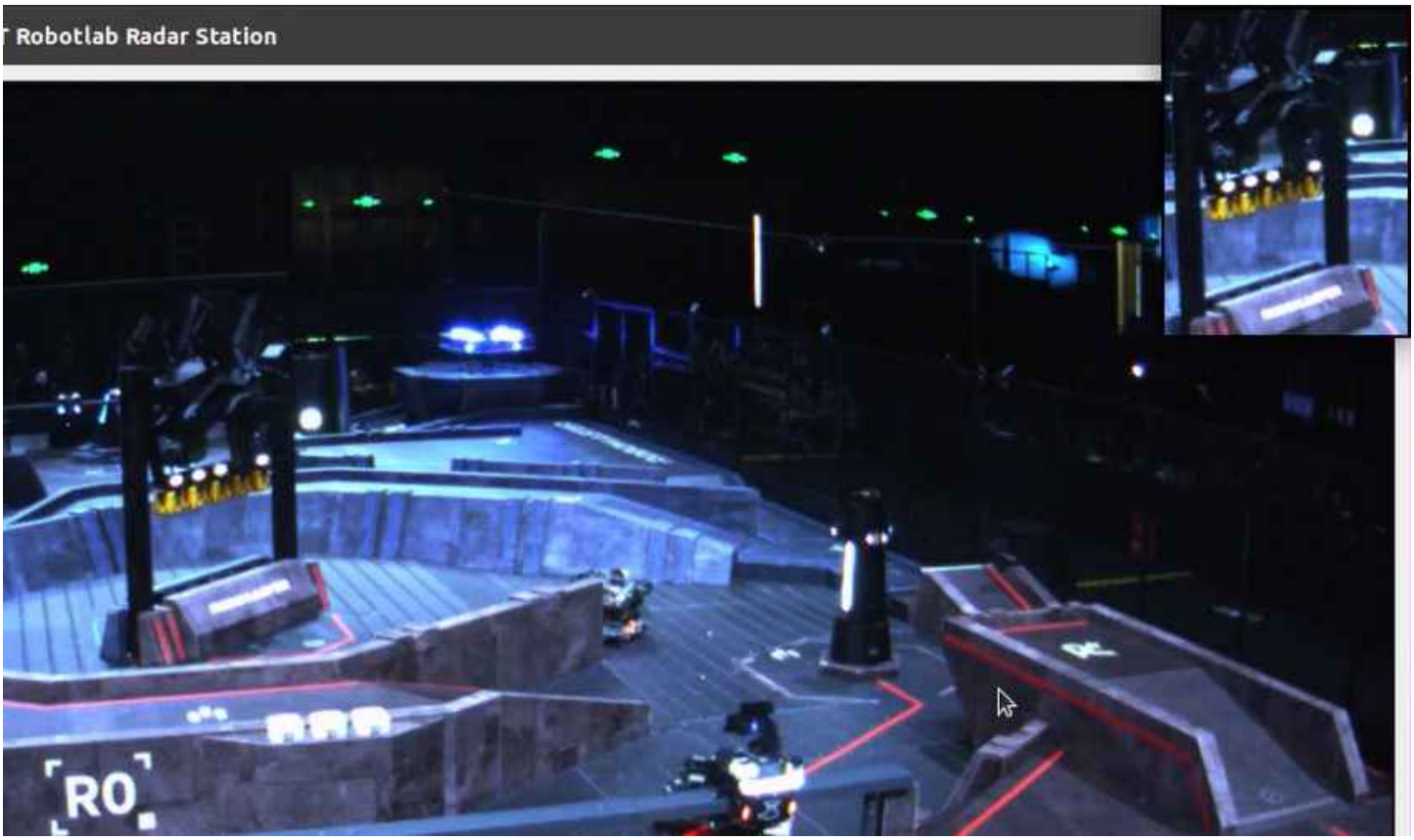
其中第一版的血量实时显示针对的是敌我地面单位，当血量最低时，还会用框框住，提醒云台手。但赛后云台手反应很少留意这个血量问题（因为裁判系统反应得有些迟钝，时效性不高，且操作手会自己注意血量），所以在第二版UI中我们改成了显示哨兵与建筑的血量，时刻反应赛场胜败的关键。

落矿、打符倒计时提醒是云台手最喜欢的功能，它出现在决策窗口，效果如下图。因为非常有用，所以第二版中保留了此功能。

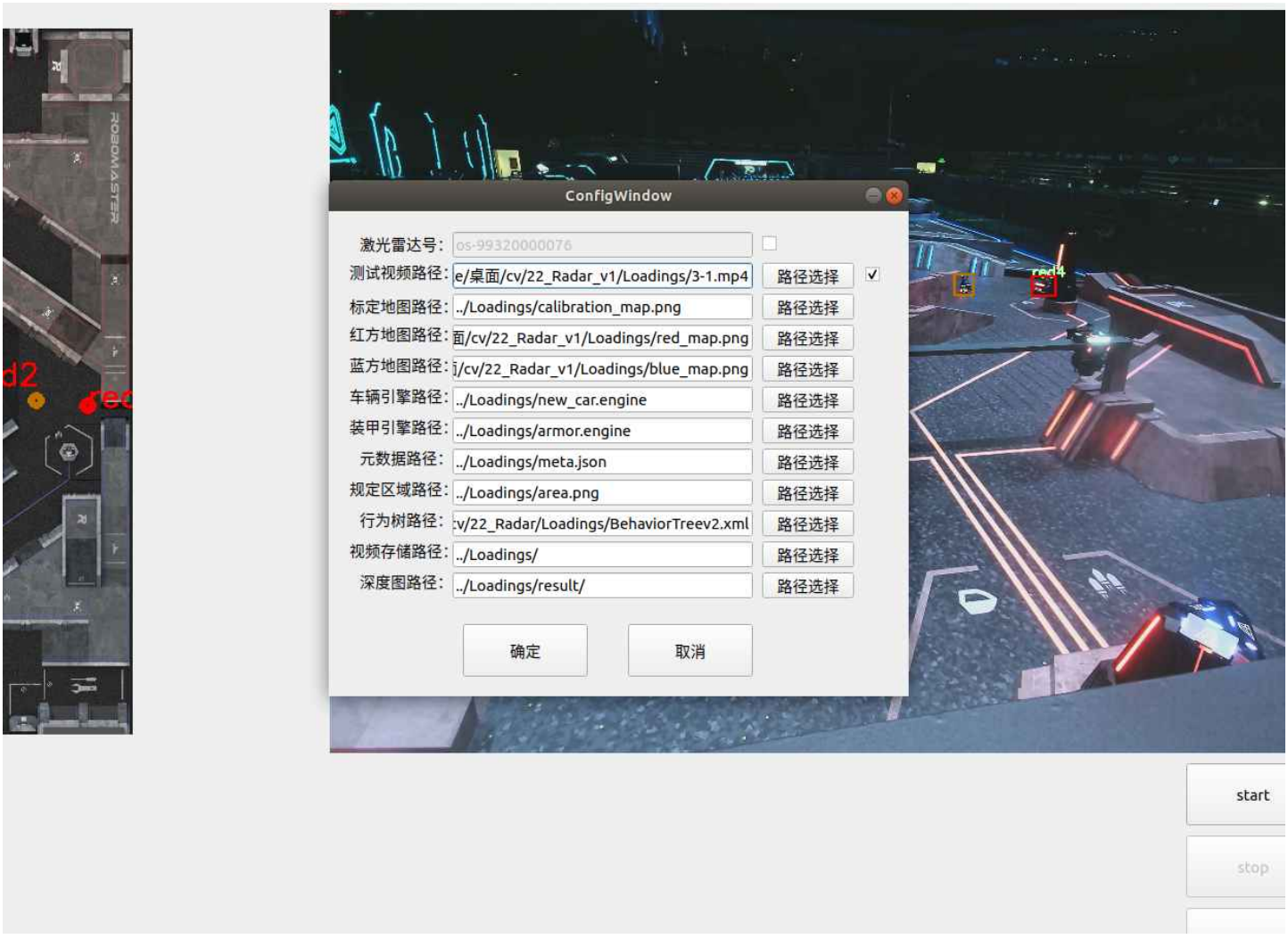


第一版时，矿石、神符、基地血量、前哨站血量、哨兵血量的状态显示在决策窗口右边，如上图所示，他们会根据实际情况显示不同的颜色或者亮起，一些有防御增益或者虚拟护盾的单位，其图片中还会有一个盾牌的图案。第二版时，由于矿石和神符的状态显示过于鸡肋，我们删除了这两个显示。

落矿区域放大窗口其实就是在右上角开一个窗口，让云台手能够更清楚地看到矿区在闪哪一个灯光，落矿前几秒才会自动打开，落矿后就会自动关闭，不影响画面。

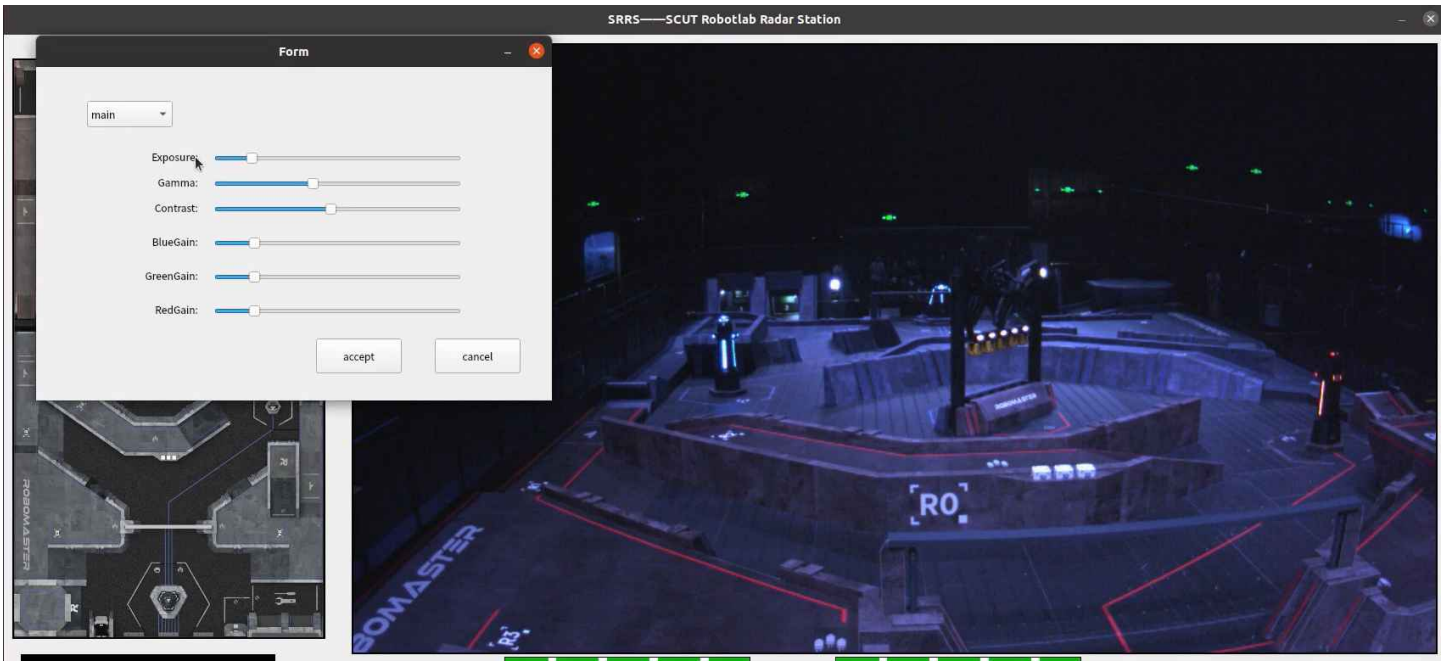

Robotlab Radar Station

2.配置页面


ConfigWindow
激光雷达号:
os-99320000076
测试视频路径:
e/桌面/cv/22_Radar_v1/Loadings/3-1.mp4
路径选择
标定地图路径:
../Loadings/calibration_map.png
路径选择
红方地图路径:
面/cv/22_Radar_v1/Loadings/red_map.png
路径选择
蓝方地图路径:
j/cv/22_Radar_v1/Loadings/blue_map.png
路径选择
车辆引擎路径:
../Loadings/new_car.engine
路径选择
装甲引擎路径:
../Loadings/armor.engine
路径选择
元数据路径:
../Loadings/meta.json
路径选择
规定区域路径:
../Loadings/area.png
路径选择
行为树路径:
v/22_Radar/Loadings/BehaviorTreev2.xml
路径选择
视频存储路径:
../Loadings/
路径选择
深度图路径:
../Loadings/result/
路径选择
确定
取消
start
stop

3.相机调整页面

相机调整页面主要是定义了若干个滑块，用于在场上实时调整曝光、对比度、gamma等参数。第二版的UI新增了可以多个相机可调的功能。由于飞镖观测相机只在飞镖发射时记录画面，并不会在主页面显示，所以为了调节它，在打开调节页面后，它会在右下角弹出一个弹窗便于观察调节效果。

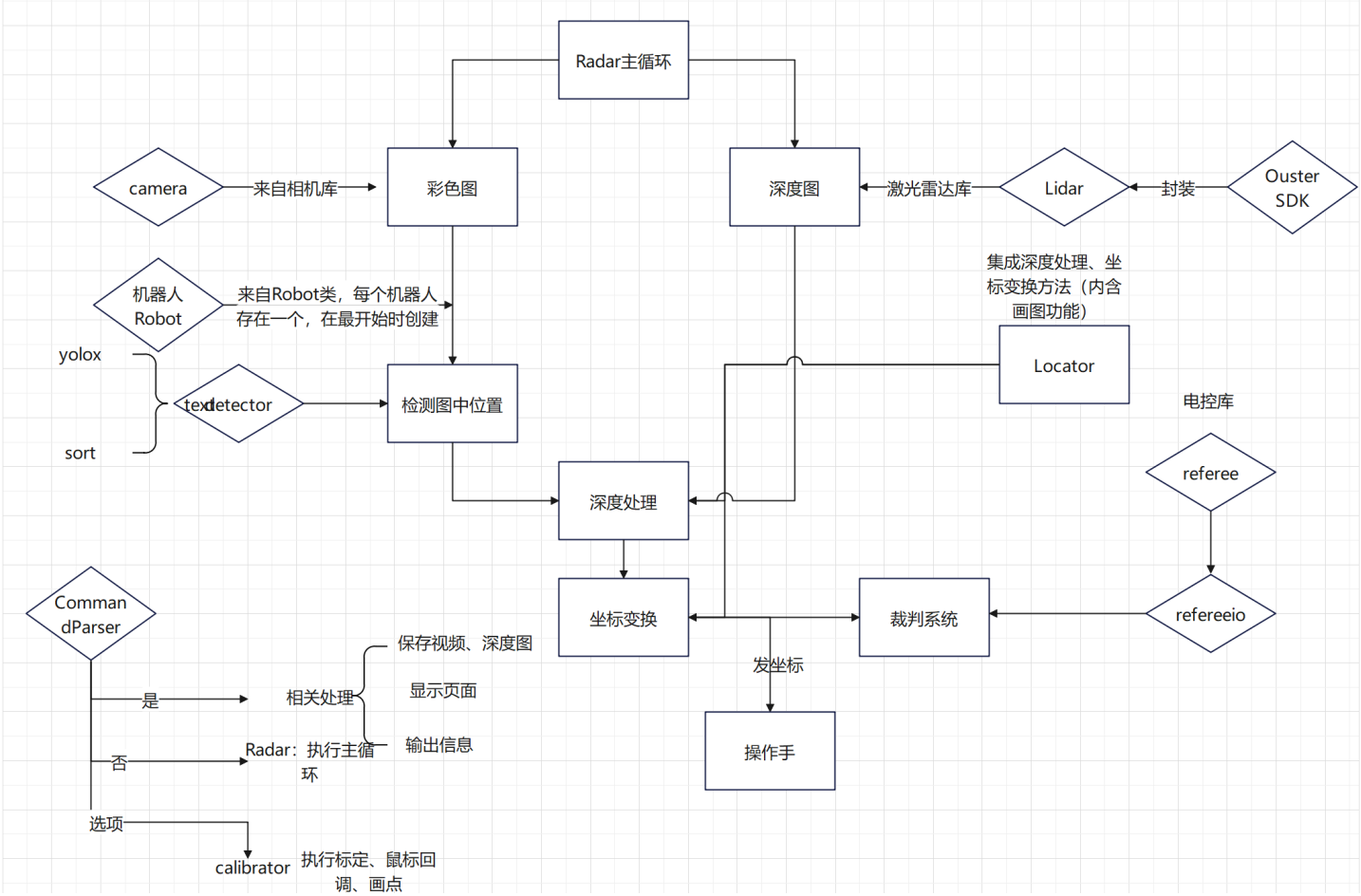


4.标定页面

标定页面用于标定场上若干个点，用于初始化外参矩阵。得到的点通过PNP计算出外参矩阵。

我们不仅把UI定义为赛场上展示的窗口，同时还把它定义为调试用的上位机，所以UI的页面切换与按钮互动的逻辑非常严谨，不会造成冲突问题。到了赛季中后期，基本使用UI进行调试，因为它内置的功能很多。

3.6软件架构



3.7未来优化方向

- 更换与调试深度学习网络模型，探索更多部署方案，以追求更快更精准的性能
- 现方案的标定比较粗糙，无法达到较好的标定精度，之后将深入标定原理，探索对软件环境依赖更小、精度更高、稳定性更强的标定方案。
- 现方案较依赖目标识别的效果，识别的效果压制了定位的发挥，之后将探索更多传感器的方案，以消除该影响

3.8源码

https://github.com/Courteous121/SRRS