



# 本科毕业设计 (论文)



面向自动驾驶的视觉目标跟踪

题 目: 和意图分析算法

学生姓名: 全松林

学 号: 2123030033

专 业: 大数据与人工智能

班 级: 数智 2101 班

指导老师: 王海东 讲师

计算机学院

2024 年 12 月

## 湖南工商大学本科毕业设计诚信声明

本人郑重声明：所呈交的本科毕业设计 湖南工商大学学位论文 L<sup>A</sup>T<sub>E</sub>X 模板使用示例 v0.1 是本人在指导老师的指导下，独立进行研究工作所取得的成果，成果不存在知识产权争议，除文中已经注明引用的内容外，本设计不含任何其他个人或集体已经发表或撰写过的作品成果。对本设计做出重要贡献的个人和集体均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者签名: \_\_\_\_\_  
日 期: \_\_\_\_\_ 年 月 日

## 摘要

随着自动驾驶技术的不断发展，提升在复杂交通环境中的感知与决策能力成为关键问题。针对传统跟踪与行为预测算法在高密度场景中精度与实时性难以兼顾的问题，设计了一套基于 Carla 仿真平台的视觉目标跟踪与意图分析系统。系统集成 DeepSORT 多目标跟踪算法，结合卡尔曼滤波与深度外观特征提取方法，实现对动态目标的高精度跟踪；并基于物理模型构建运动意图识别机制，通过分析目标速度与相对位置，实时推断“靠近”“远离”及“危险接近”等行为趋势。实验结果表明，系统在复杂城市交通场景中具备良好的实时性与鲁棒性，尤其在目标遮挡与快速运动情况下表现优异。研究成果为自动驾驶视觉感知中目标跟踪与意图预测提供了新思路，具有一定的工程应用价值和拓展潜力。

**关键词：**自动驾驶； 视觉感知； 目标跟踪； 意图识别； Carla 仿真

## ABSTRACT

With the rapid advancement of autonomous driving technology, enhancing perception and decision-making capabilities in complex traffic environments has become a crucial research focus. This study designs a vision-based target tracking and intention analysis system based on the Carla simulation platform to address the challenges of balancing accuracy and real-time performance faced by traditional tracking and behavior prediction algorithms in dense traffic scenarios. The system integrates the DeepSORT multi-object tracking algorithm, combining Kalman filtering and deep appearance feature extraction to achieve accurate and continuous tracking of dynamic targets. Additionally, a lightweight intention recognition mechanism based on physical modeling is developed, which analyzes the speed and relative position of targets to infer behavior trends such as "approaching," "departing," and "dangerous approach" in real-time. Experimental results show that the system maintains strong robustness and real-time performance, especially under conditions of occlusion and rapid motion in complex urban traffic environments. The proposed approach effectively improves the perception intelligence of autonomous vehicles, providing reliable support for decision-making and path planning. This research offers new insights and technical solutions for vision-based target tracking and intention analysis, with promising prospects for practical applications and further development in the field of intelligent transportation systems.

**Key words:** Autonomous Driving; Visual Perception; Target Tracking;  
Intention Recognition; Carla Simulation

# 目录

<b>第 1 章 绪论</b>	<b>1</b>
1.1 研究背景 . . . . .	1
1.2 国内外研究现状 . . . . .	1
1.2.1 目标检测研究现状 . . . . .	1
1.2.2 目标跟踪研究现状 . . . . .	3
1.2.3 意图分析研究现状 . . . . .	4
1.3 研究目标与内容 . . . . .	5
1.4 本文结构框架 . . . . .	6
<b>第 2 章 研究理论基础</b>	<b>8</b>
2.1 目标检测与视觉跟踪概述 . . . . .	8
2.2 DeepSORT 算法原理与流程 . . . . .	9
2.2.1 算法结构概述 . . . . .	9
2.2.2 状态预测：卡尔曼滤波器 . . . . .	9
2.2.3 外观建模：深度特征提取 . . . . .	10
2.2.4 匹配决策：融合距离与匈牙利算法 . . . . .	10
2.2.5 算法执行流程 . . . . .	11
2.3 行为意图识别的物理建模方法 . . . . .	12
2.3.1 意图识别的物理基础 . . . . .	13
2.3.2 判别规则设计与分类逻辑 . . . . .	13
2.3.3 与视觉跟踪系统的集成实现 . . . . .	13
2.4 小结 . . . . .	14
<b>第 3 章 系统总体设计</b>	<b>15</b>
3.1 系统架构设计 . . . . .	15
3.2 Carla 仿真平台与传感器配置 . . . . .	16
3.2.1 仿真地图选择与场景设定 . . . . .	16
3.2.2 本车模型与控制机制 . . . . .	17
3.2.3 摄像头传感器配置 . . . . .	18
3.2.4 其他配置说明 . . . . .	18
3.3 项目开发环境与工具链 . . . . .	19
3.3.1 开发硬件平台 . . . . .	19
3.3.2 开发软件与工具 . . . . .	19

---

<b>第 4 章 视觉目标跟踪模块设计</b>	<b>20</b>
4.1 数据集采集机制与结构设计 . . . . .	20
4.1.1 采集流程设计 . . . . .	20
4.1.2 标注信息结构设计 . . . . .	22
4.2 目标检测与跟踪模型设计 . . . . .	22
4.2.1 目标检测策略设计 . . . . .	22
4.2.2 DeepSORT 跟踪模型集成 . . . . .	23
4.3 模型性能评估 . . . . .	25
4.3.1 系统帧处理时序分析 . . . . .	25
4.3.2 模块耗时对比分析 . . . . .	26
<b>第 5 章 意图分析算法设计与实现</b>	<b>28</b>
5.1 意图识别需求分析 . . . . .	28
5.2 基于速度与距离变化的意图判别逻辑 . . . . .	28
<b>致谢</b>	<b>30</b>
<b>附录 A 附录代码</b>	<b>31</b>
A.1 DeepSORT 多目标跟踪算法 . . . . .	31
A.2 面向自动驾驶的视觉目标跟踪和意图分析算法 . . . . .	31
<b>参考文献</b>	<b>31</b>

# 第1章 绪论

## 1.1 研究背景

随着人工智能、计算机视觉及自动控制等技术的迅猛发展，自动驾驶作为智能交通体系中的核心环节，正在从理论研究走向商业化落地。自动驾驶系统通常由感知、决策与控制三大模块构成，其中视觉感知作为信息获取的前端环节，其精度和稳定性直接影响整个系统的安全性与智能性。在复杂交通环境中，车辆与行人等动态目标的准确识别、持续跟踪以及行为意图的及时判断，是实现安全驾驶与合理路径规划的基础保障。

当前，基于深度学习的目标检测与多目标跟踪算法在学术界与工业界中得到广泛应用，其中以 YOLO 系列、SORT/DeepSORT 等方法为代表的视觉跟踪算法已具备较高的实时性与鲁棒性。然而，在实际交通场景中，单纯的跟踪信息往往不足以支撑高级驾驶辅助功能（ADAS）的需求。系统不仅需要知道“是什么”和“在哪里”，更需要理解“它想要干什么”，即对前方目标的运动趋势与行为意图做出合理判断，从而提前进行风险预判与决策干预。这种基于时空轨迹与速度信息的意图识别能力，将成为未来智能驾驶系统的重要能力之一。

另一方面，受限于真实交通场景数据采集的高成本与不可控性，虚拟仿真平台在智能驾驶算法研发中发挥着重要作用。Carla 作为目前最具代表性的自动驾驶开源仿真平台之一，提供了高度还原真实城市交通环境的三维场景与丰富的传感器模拟能力，为研究人员提供了可控、可复现的实验环境。本研究基于 Carla 平台，通过构建可运行的目标跟踪与意图识别系统，完成视觉感知到智能判断的全流程设计与实现，具有重要的研究价值和现实意义。

综上所述，开展面向自动驾驶的视觉目标跟踪与意图分析研究，既能够推动感知系统从“感知当前”向“预测未来”的进化，也有助于提高自动驾驶系统的安全冗余能力和交通行为的合理性。本课题的研究成果可为未来多目标智能预测、复杂场景理解等方向提供工程实践基础与技术参考，具有良好的应用前景。

## 1.2 国内外研究现状

### 1.2.1 目标检测研究现状

目标检测是指将图像或者视频中的目标与其他不感兴趣区域进行区分，判断是否存在目标，以及确定目标位置和识别目标种类的计算机视觉任务。以是否使用深度学习为分界线，可以将目标检测算法分为传统的目标检测算法和基于深度学习的目标检测算法。传统的目标检测算法依赖于人工设计的特征算子，并采用特征提取加分类器的模式来实现。Viola 和 Jones 提出了 Viola-Jones 人脸检测器 **viola2001rapid**，该检测器使用 Haar-like 特征提取算子，利用 Adaboost 分类器进行分类，并采用由多个分类器组成的 Cascade 级联分类器来提高识别率。Navneet Dalal 等人 **dalal2005histograms** 提出方向梯度

直方图（Histogram of Oriented Gradients, HOG），HOG 是一种用于目标检测的特征描述算子，通过计算图像中局部梯度的方向信息，对图像的浅层外观特征进行提取。由于 HOG 过度依赖局部梯度信息，因此在复杂场景下的特征提取能力较差。Felzenszwalb 等人 **felzenszwalb2009object** 在 HOG 的基础上，提出了可变形组件模组（Deformable Part Model, DPM）。DPM 采用多组件策略，将目标分解为多个部分后进行建模，因此 DPM 对目标的形变具有很强的鲁棒性。传统的目标检测算法大多使用人工设置的特征提取算子来对图像进行特征提取，在面对多尺度、密集目标、极端天气等复杂环境时，算法的鲁棒性和准确率都会大大降低。同时采用滑动窗口来对图像特征进行逐一计算的方式，会使算法的计算复杂度过高。因此传统的目标检测方法无法满足实时检测需求，不能很好地在工业中进行应用。

随着深度学习的飞速发展，目标检测领域的研究取得了突破性的进展。与传统的目标检测算法相比，基于深度学习的目标检测模型在经过大量图像数据训练后，能“学会”如何提取目标特征，在面对不同尺度的目标以及复杂场景时，具有更好的鲁棒性和泛化能力。根据是否需要生成候选区域，可以将基于深度学习的目标检测算法分为单阶段（one-stage）与两阶段（two-stage）。两阶段的目标检测算法主要分为两步，首先筛选出图片中可能包含物体的候选区域，接着再对候选区域中的特征进行分类回归。Ross Girshick 等人 **girshick2014rich** 提出 R-CNN 目标检测算法，首先对输入图像使用选择搜索 **uijlings2013selective**（selective search）网络获得大约 2000 个不同尺度的候选区域，接着对所有的候选区域进行统一缩放，将尺寸变为  $227 \times 227$ 。再通过卷积神经网络（Convolutional Neural Networks, CNN）对所有候选区域中的特征进行提取，得到维度为  $2000 \times 4096$  的特征向量。最后再利用 SVM 分类器对特征向量进行分类，以及使用回归器来对边界框的位置进行调整。由于 R-CNN 目标检测算法需要对约 2000 个候选框逐一进行特征提取，而候选框之间又存在许多包含相同特征的区域，这就导致整个过程中存在大量冗余操作，严重影响了检测速度。为了解决上述问题，Ross Girshick 等人 **girshick2015fast** 又提出了 Fast R-CNN，Fast R-CNN 首先对输入的图像进行卷积，得到对应的特征图。接着将事先处理好的区域候选框投影到特征图上。采用这种方式只需要进行一次卷积，可以减少大量冗余的卷积计算。而且与 R-CNN 所使用的分类任务 + 回归任务模式不同，Fast R-CNN 将二者融合在一个网络中，不再需要对分类器和回归器进行单独训练。但是 Fast R-CNN 算法依然沿用了使用选择搜索算法的方式来获取候选区域，该过程不仅操作耗时而且容易漏掉包含目标的区域。因此 Shaoqing Ren 等人 **ren2015faster** 又提出了 Faster R-CNN，算法采用区域建议网络（Region Proposal Networks, RPN）来获取区域候选框，实现了生成候选区域过程与目标检测任务的融合。同时 Faster R-CNN 还提出了锚框（Anchor）这一重要思想，锚框是指在图像上预先设置好多组尺寸比例不同的参照框，来尽可能地包含物体出现的位置。He Kaiming **he2017mask** 等人在 Faster R-CNN 的基础上提出了 Mask R-CNN，该算法使用 ROIAlign 替换 Faster R-CNN 中的 ROIPooling。ROIAlign 可以将任意尺寸感兴趣区域的特征图划分为具有固定尺寸的小特征图，并使用双线性插值取代了 ROIPooling 中的直接取整操作，解决了 ROIPooling 操作中因两次

量化造成的区域不匹配 (mis-alignment) 问题。在目标检测任务中，使用 IOU 阈值来区分正负样本。当 IOU 阈值越大，正样本的数量就会减少，而减小 IOU 阈值又会学习到大量无关特征。Cai 等人 [citecai2018cascade](#) 为了解决上述问题，提出了 Cascade R-CNN，该算法采用级联式的结构，通过对多个感知器使用递增的 IOU 阈值进行训练，来解决因提高 IOU 阈值所引起的模型过拟合问题。

### 1.2.2 目标跟踪研究现状

多目标跟踪算法可以分为基于检测跟踪 (Tracking by Detection, TBD) 和联合检测跟踪 (Joint Detecting Tracking, JDT) 两种方式。TBD 目标跟踪算法是指先使用目标检测算法得到包含物体的边界框，接着再获取目标的运动信息和外观信息等，最后通过数据关联算法来计算目标间的亲和力并关联目标。Bewley 等人 [bewley2016simple](#) 提出了 SORT 算法，作为经典的基于检测的目标跟踪算法，采用 Faster R-CNN 作为检测器，利用卡尔曼滤波来预测和更新物体的运动特征信息，通过匈牙利算法进行数据关联，以及使用 IOU 作为度量指标来建立关系，从而实现对多目标进行追踪。Wojke 等人 [wojke2017simple](#) 提出了 DeepSORT 多目标跟踪算法，该算法在 SORT 的基础上加入级联匹配，引入了卷积神经网络来提取目标外观特征，有效地减少了在面对遮挡情况下发生身份切换的问题。Yu 等人 [yu2016poi](#) 提出了 POI 算法，采用 Faster R-CNN 作为检测器，并使用 skip pooling 和 multi-region 两种策略来提高检测精度，利用改进的 GoogLeNet [szegedy2015going](#) 网络进行特征提取。Sun 等人 [sun2019deep](#) 提出了深度亲和网络 (DAN)，DAN 网络以端到端的方式对目标物体的外观特征进行学习，通过物体和环境的分层特征计算出在不同帧中目标之间的亲和度。Chen 等人 [chen2018real](#) 将检测和跟踪结果组成一对候选框，提出了一种得分函数用于衡量每一对候选框的匹配程度，使用非极大值抑制算法依据得分情况进行筛选。Zhang 等人 [zhang2021fairmot](#) 将目标检测任务和特征提取融合到一个网络中完成。首先使用编码器-解码器模块对图像进行特征提取，在进行分流后，使用两个分支分别进行边界框预测以及目标外观特征提取，最后使用预测目标中心点处的特征进行边界框时序联结。Liang 等人 [liang2022rethinking](#) 提出 CTrack 算法，采用 CCN (交叉相关网络) 来改进检测与重识别间的协作学习。将目标检测和外观特征提取进行解耦，通过使用自注意力的方式获得自注意力权重图和交叉相关性权重图，并利用 SAAN [zhao2020saan](#) (尺度感知注意力网络) 对特征提取网络进行优化。Liang 等人 [liang2022fake](#) 在 CTrack 的基础上引入时间信息来修正检测器结果，并提出了重检测网络来重新加载被错误分类的目标。Yu 等人 [yu2022relationtrack](#) 提出了 GCD (Global Context Disentangling) 模块，该模块能将特征图解耦成检测特征和重识别特征两部分，同时采用可变形注意力机制来学习目标和环境之间的关系。

联合检测跟踪算法是将检测任务与跟踪任务进行合并，仅使用一个网络来实现多目标跟踪任务。Peng 等人 [peng2020chained](#) 提出了一种 CTracker 算法，该算法首次将目标检测、特征提取、数据关联三个模块集成到单个网络中，实现了端到端的联合检测跟踪。同时还设计了一种联合注意力模块 (JAM, Joint Attention Module) 来突出检测框中的

有效信息区域。Xu 等人 **xu2020deep** 提出了 DHN (深度匈牙利算法)，以可微的方式对检测框和预测框进行匹配，以及提出了一种新的损失函数用于训练联合检测跟踪范式的多目标跟踪器。Pang 等人 **pang2021quasidense** 提出 QDTrack 算法，通过采用多个正负样本同时计算损失的方式来对特征提取网络进行训练，是一种只利用 ReID 特征而不需要位置和运动信息的多目标跟踪方法。Zhou 等人 **zhou2020tracking** 提出了 CenterTrack 算法，使用 CenterNet **zhou2019objects** 目标检测算法来对目标中心进行定位，CenterTrack 依据上一帧的检测结果得到热量图，峰值代表目标中心点，并采用高斯渲染的方法进行模糊处理。模型利用两个额外的并行分支来预测当前目标相对于上一帧时的水平和竖直方向偏移量。Wu 等人 **wu2021track** 提出了一种在线多目标联合检测追踪模型，使用 CenterNet 提取图像特征，利用 CVA 模块计算两帧图像中目标的偏移量，得到目标之间的关联性，通过 MFW 模块来传播和增强目标特征，最后使用头部网络将传播特征和当前特征进行处理以实现检测和追踪。Wang 等人 **wang2021multiple** 提出了 CorrTracker 算法，利用局部相关模块来构建目标与周围环境之间的拓扑关系，从而加强模型在密集场景中的识别能力。Sun 等人 **sun2020transtrack** 提出 TransTrack 算法，该算法采用 transformer **vaswani2017attention** 架构，利用 Query-Key 机制来跟踪当前帧中已存在的目标以及对新目标进行检测。通过在一次拍摄中完成目标检测和目标关联，建立了一种新的联合检测跟踪范式。Chu 等人 **chu2023transmot** 提出了 STGT 模型，通过将追踪目标轨迹视作稀疏赋权图来构建目标间的空间关系。STGT 构建了一个空间图 transformer 编码器、时间 transformer 编码器和一个空间 transformer 解码器。利用稀疏图来提升训练和推理时的计算速度。并且由于获取了目标间的结构信息，因此比一般的 transformer 也更加有效。

### 1.2.3 意图分析研究现状

最早关于识别驾驶意图的讨论可以追溯到 Andrew 等人的一项研究 **liu1997realtime**，他们认为可以通过分析驾驶员的行为动作来预测其意图，并在研究中仅使用车辆的动态数据，例如横摆角、横摆角速度和车速，来判断驾驶员是否有转弯或换道的意图。要对周围车辆的换道意图进行识别，需要结合感知、数据融合、数据处理等多种技术 **zhang2023highway**。感知与数据融合技术融合所有从传感设备传输的信号，生成周围所有交通和其他环境的信息。智能车辆将这些信息从自身的车辆坐标系转换为目标车辆坐标系，并处理为识别算法可以使用的特征变量。因此，在识别方法之外，特征变量的选择对识别的效果也影响重大。针对单个目标车辆，目前广泛使用的信号包括纵向/横向位置、速度和加速度 **woo2017lane**。主车的意图识别可以通过车内传感器获得大量有效信息，比如方向盘和制动/油门踏板信号，甚至驾驶员本身的状态等。然而，在车联网技术尚未成熟的当下，目标车辆内的这些信息难以获取。Zhang 等人 **zhang2018lane** 使用目标车辆及其四辆邻近交通车辆的信息，包括它们之间的相对速度和距离，来预测换道行为。Leonhardt 等人 **leonhardt2017feature** 也采用了类似的特征变量来研究换道预测。部分研究使用了目标车辆周围的更多相邻车辆以及与每个相邻车辆相关的更多信息（比如状态量的历史记录）**patel2018predicting**。Altch 等人 **altche2017lstm** 甚至考虑了九辆车来提高预测性能。

2016 年，日产研制出了前沿科技 ProPILOT，其安装的实力强劲的摄像机可以较为容易的检测出车相对道路的偏离距离<sup>zhang2016nissan</sup>。雪铁龙 LDWS 系统安装有 6 个电子传感器。汽车通过车道标记线的时候，光线的反射不一样，此时检测单元会将信息发送到车载控制中心，内置的振动马达会向驾驶员传递警告信息。这个系统的价钱低廉，系统的适应性不错，但是在复杂的电子环境下的表现会受到影响，元器件的精度会降低，目前在 C5、C6 的车型上有应用。Google 也进行了相关研发，利用高性能的软件和智能探测装置，融合雷达和摄像技术，能够在复杂条件下全面的探测周围环境<sup>wang2020autonomous</sup>，其结合 GPS 技术的车道偏离技术可以及时矫正汽车的行驶轨迹，并且误差可以达到厘米级。Enache 等<sup>enache2009driver</sup>提出了新的计算方法，通过预瞄偏差及车辆方向偏差综合信息来估计车路之间的相对距离；Cualain 等<sup>cualain2012automotive</sup>利用卡尔曼滤波和霍夫变换建立了车道边界的模型，计算车辆本身的参数及建立新的预警规范来设计预警系统的阈值；Mammar 等<sup>citemammar2006time</sup>通过计算道路曲率、方向偏差等估计车辆跨过车道线的时间；Ulsoy 等<sup>chiu1996time</sup>利用准确度较高的算法估计 TLC 的不确定，考虑到了不同方面的误差；Gaikwad 等<sup>gaikwad2015lane</sup>通过对感兴趣区域的划分，建立实际车道偏离的度量标准。

### 1.3 研究目标与内容

本课题旨在设计并实现一套基于视觉感知的自动驾驶目标跟踪与意图分析系统，借助 Carla 仿真平台构建可控的交通环境，通过集成深度学习目标跟踪算法和基于物理信息的行为推理机制，实现对动态交通目标（如车辆、行人）的连续识别、轨迹跟踪与运动意图判断，并能够在图形界面中实时可视化跟踪过程与分析结果，从而为智能驾驶系统提供基础性的感知支撑。

为实现上述目标，本课题主要围绕以下几个方面开展具体研究与开发工作：

**数据集构建与处理。**在 Carla 提供的仿真场景中，部署本车及自动生成车流，通过挂载 RGB 摄像头对周围交通目标进行图像采集与注释，提取包含图像帧、目标边界框、目标速度等信息的原始数据，构建用于视觉分析的多模态数据集。同时，设计适用于后续分析与训练的统一标注格式和存储结构。

**视觉目标跟踪模型设计与集成。**本系统采用深度外观特征辅助的 DeepSORT 算法作为目标跟踪模型，通过对视频帧中的检测结果进行轨迹级别的数据关联，实现对车辆与行人的唯一身份标记与跨帧追踪。系统支持在单目标跟踪模式下自动选取“最接近本车”的目标进行持续跟踪，并将结果实时渲染在屏幕图像中。

**基于速度信息的意图分析方法设计。**在目标跟踪的基础上，提取目标在世界坐标系下的运动方向与速度大小，并结合其与本车的相对位置关系，设计运动意图识别规则体系。采用物理特征量（如点积关系、欧氏距离变化、速度阈值）判断目标是否存在“靠近”“远离”“危险靠近”等行为状态，并输出中文提示文本以实现可视化展示与预警反馈。

**系统集成与仿真测试。**将上述模块集成为一个完整的自动驾驶视觉分析系统，结合

Carla 提供的控制接口与传感器 API 实现全过程联动，在 Town10 与 Town01 两个典型城市街景场景中分别进行系统测试与功能验证，评估模型在不同场景下的运行效率与判断准确率，并结合帧率等性能指标进行可行性分析。

本研究面向自动驾驶核心任务需求，通过仿真数据采集、视觉跟踪、行为分析与实时展示四个维度开展工作，形成从感知到判断的完整流程，具备较强的应用价值与实践意义。

#### 1.4 本文结构框架

本课题围绕自动驾驶场景下的视觉感知与行为预测展开，依托 Carla 仿真平台提供的高真实度交通环境与 Python 接口能力，采用模块化设计思路，构建了“数据采集—目标跟踪—意图分析—可视化展示”一体化流程，实现了一个可运行、可扩展的目标感知与行为识别系统。研究方法分为数据采集与预处理、目标跟踪算法设计、意图分析模块开发、可视化与系统集成四个主要环节，其整体流程如图所示：

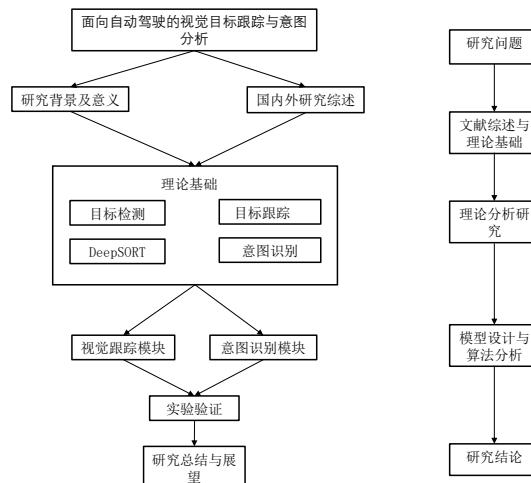


图 1-1 论文结构框架图

在数据采集部分，系统通过 Carla 平台生成典型交通场景，优先选取 Town10 和 Town01 作为实验场地，并在本车上安装前向 RGB 摄像头以获取图像数据。每一帧图像经处理后可提取出车辆的二维边界框、目标 ID、速度大小等结构化信息。为了实现后续训练与分析，系统在运行时自动保存图像帧（.jpg）和对应标注文件（.json），内容包括边界框坐标、目标速度、是否为当前跟踪对象等关键信息，构建出具备时序特征的感知数据集，确保可复现性和可扩展性。

目标跟踪模块是系统的核心部分。本设计集成了 DeepSORT 跟踪算法，其结合了外观特征提取与卡尔曼滤波器，在保持实时性的同时具备较强的数据关联能力。通过调用 Carla API，系统可实时获取当前场景中所有车辆的三维位置，并结合前端图像中的二维边界框，实现目标在图像空间内的连续标识。为保证算法聚焦于当前对本车最具潜在交

互风险的目标，系统采用“最近目标优先”策略进行目标选择，并仅跟踪单个目标，从而有效降低计算开销并简化意图分析流程。

在行为识别方面，系统基于目标的速度向量、方向角度以及其与本车的相对位置，构建了轻量级的意图分析模块。该模块通过计算目标车辆的速度方向与本车之间的点积关系，结合速度大小与空间距离的动态变化，判断其当前行为趋势。具体包括“靠近中”“远离中”“危险靠近”“目标稳定”等状态，并通过阈值组合判断进行分级判断。在实际运行中，系统可实现对目标行为的实时预测，并将分析结果以中文文本的方式展示在跟踪框上方，增强系统的人机交互性和直观性。

整个系统的主控逻辑集中在Carla的同步刷新机制中，通过game\_loop()主循环完成数据获取、图像渲染、目标识别、意图计算与结果展示等全过程联动。系统采用Pygame进行图像渲染与键盘交互，支持用户手动操控本车行驶，同时自动完成图像与标签的保存工作，便于后续模型训练与性能评估。整体架构充分体现了模块化、可测试与可扩展的设计思想，能够有效支持不同场景下的视觉感知与行为分析研究，具有良好的工程实现基础。

## 第2章 研究理论基础

本章旨在对本研究所依赖的关键算法与基础理论进行系统阐述，为后续视觉目标跟踪与意图识别系统的实现提供理论支撑。内容包括目标检测与视觉跟踪技术的基本概念，DeepSORT 跟踪算法的原理与工作流程，以及基于速度与空间信息的意图识别模型。同时，为便于理解，还将简要介绍在本研究中承担仿真任务的 Carla 平台的相关原理。

### 2.1 目标检测与视觉跟踪概述

在自动驾驶系统中，环境感知是实现安全驾驶与智能决策的基础。通过计算机视觉技术识别道路上的车辆、行人、交通标志等关键对象，并对其动态状态进行连续追踪，是实现环境建模与行为预测的重要手段。目标检测与视觉跟踪作为其中的核心组成部分，直接影响着自动驾驶系统对外部环境的认知能力与决策准确性。

目标检测（Object Detection）指的是在输入图像中识别出所有存在的感兴趣目标，并准确回归其在图像中的位置（通常以边界框形式表示）以及目标类别信息。传统的目标检测算法主要依赖人工设计的特征与滑动窗口机制，代表方法如 Haar 特征与 HOG+SVM 检测器，尽管实现简单，但在复杂背景下的鲁棒性较差。近年来，随着深度学习的发展，基于卷积神经网络（CNN）的目标检测方法逐渐成为主流，显著提升了检测精度与速度。当前主流的检测模型可分为两类：一类是以 Faster R-CNN 为代表的两阶段检测器，其先生成候选区域后进行分类与回归，精度较高但速度较慢；另一类则是以 YOLO（You Only Look Once）、SSD（Single Shot MultiBox Detector）等为代表的单阶段检测器，其直接在特征图上进行回归预测，具有更高的实时性，适用于自动驾驶等对延迟要求较高的应用场景。

与目标检测不同，目标跟踪（Object Tracking）是指在已知初始检测结果的前提下，持续对目标在视频序列中的位置进行估计。根据跟踪目标的数量，通常可分为单目标跟踪（Single Object Tracking, SOT）与多目标跟踪（Multiple Object Tracking, MOT）。单目标跟踪关注于对一个特定目标进行持续跟踪，其主要挑战在于遮挡、快速运动、目标消失与再出现等；而多目标跟踪则需要同时对多个目标进行识别与数据关联，面临着更高的关联复杂性与遮挡问题。在实际的自动驾驶场景中，由于交通参与者种类多、状态变化快、相互干扰强，因此往往需在较短时间内完成高准确率的多目标跟踪任务。

视觉目标跟踪通常依赖目标检测结果作为输入，通过匹配机制完成目标的跨帧关联。跟踪算法可按是否借助外部检测结果分为两类：一类是基于检测的跟踪（Tracking-by-Detection），该类方法首先使用检测器获取每帧图像中的目标位置，然后通过轨迹预测与数据关联模块实现目标编号的一致性，是当前主流的工程实现方式；另一类是端到端的跟踪方法，直接通过时序特征建模目标的运动轨迹，适合复杂行为建模任务。前者由于易于部署与现有检测模型兼容，成为了众多自动驾驶感知系统的首选方案。

在本研究中,为提高系统的实时性与稳定性,采用了“检测—跟踪”分离式结构,即首先利用图像分析提取候选目标的边界框与状态信息,然后通过外部跟踪器(DeepSORT)进行跨帧目标关联。此结构既可以利用深度检测模型的高精度特性,又能充分结合目标的运动学特征,实现实时、连续、稳定的目标轨迹追踪。同时,单目标跟踪策略被引入,用于聚焦当前与本车存在潜在交互风险的目标,提升后续意图分析模块的准确性与实用性。

目标检测与视觉跟踪技术构成了自动驾驶系统中感知层的核心支撑,对系统的安全性、实时性与可靠性具有直接影响。在本课题中,这两项技术作为算法链条的起点,将为后续的意图识别与行为预测提供基础信息保障。

## 2.2 DeepSORT 算法原理与流程

在视觉目标跟踪任务中,传统的多目标跟踪(MOT)算法如 SORT(Simple Online and Realtime Tracking)因其结构简单、处理速度快,被广泛应用于实时系统中。但在复杂交通场景下,目标之间频繁遮挡、外观相似度高、轨迹交叉密集等因素,容易造成目标ID的切换或跟踪中断,从而降低系统的稳定性与实用性。为了解决这些问题,Wojke等人<sup>Wojke2017DeepSORT</sup>在2017年提出DeepSORT(Deep Simple Online and Realtime Tracking)算法,在保持SORT轻量特性的基础上,引入了外观信息进行多维度数据关联,有效提升了多目标跟踪的鲁棒性和精度。

### 2.2.1 算法结构概述

DeepSORT是一种基于检测驱动的在线目标跟踪算法,其整体结构由三部分组成:目标运动建模模块(卡尔曼滤波器)、数据关联模块(匈牙利算法与融合距离度量)以及外观特征提取模块。它基于SORT(Simple Online and Realtime Tracking)算法的扩展,SORT算法通过卡尔曼滤波器和匈牙利算法完成目标的状态预测与数据关联,但仅依赖于目标的运动信息进行匹配。DeepSORT则通过引入深度学习模型,利用目标的外观特征进一步增强了算法的鲁棒性和精度。算法的基本思想是:每一帧从检测器获取边界框,通过预测与匹配将其关联至已有轨迹,实现目标编号的持续与轨迹的连续。该结构兼顾精度与实时性,特别适合部署于自动驾驶等延迟敏感的系统中。

### 2.2.2 状态预测: 卡尔曼滤波器

DeepSORT使用卡尔曼滤波器对每个目标进行状态建模与短时位置预测。每个目标的状态向量通常定义为:

$$x = [u, v, \gamma, h, \dot{u}, \dot{v}, \dot{\gamma}, \dot{h}]^T$$

DeepSORT的第一步是对目标状态进行预测,而这一过程依赖于卡尔曼滤波器。卡尔曼滤波器是一种基于动态系统的状态估计方法,能够对目标的运动状态(包括位置、速度等)进行平滑和预测。在目标跟踪过程中,卡尔曼滤波器通过对上一帧目标状态的估计,预测目标在当前帧的可能位置,从而减少由于目标丢失或遮挡带来的误差。

卡尔曼滤波器的核心是基于目标的运动方程进行线性预测，结合测量值对目标的状态进行更新。在每一帧中，卡尔曼滤波器根据上一帧的目标位置与速度，预测当前帧中目标的状态，并通过目标的检测结果对预测状态进行修正。通过这种方式，卡尔曼滤波器可以保持目标的连续性，弥补检测过程中可能的丢失和误差。

### 2.2.3 外观建模：深度特征提取

在多目标跟踪中，目标的外观特征提取是确保算法鲁棒性和精度的关键技术之一。传统的目标跟踪算法主要依赖于目标的运动信息（如位置、速度等）进行跟踪，这种方法在处理目标遮挡、重叠或快速运动时往往效果不佳。而 DeepSORT 通过引入深度学习模型，特别是深度卷积神经网络（CNN），在提取目标的运动信息的同时，也结合了目标的外观特征，大大增强了算法在复杂场景中的跟踪精度。

DeepSORT 的外观建模部分通过使用卷积神经网络（CNN）来提取目标的视觉特征。这些特征通常包含了目标的颜色、形状、纹理等信息，有助于区分不同目标，特别是在多目标重叠或目标短时间遮挡的情况下，外观特征能够有效解决身份混淆问题。CNN 通过对目标边界框区域进行裁剪，并将裁剪后的图像输入网络，从中提取出固定维度的特征向量。这个向量就是目标的外观特征，表示了目标的独特视觉信息。

通过外观特征的提取，DeepSORT 能够在跟踪过程中通过比对目标在不同帧中的特征向量，来识别和区分不同目标。这一方法极大地提升了算法在复杂动态环境中的表现，尤其在目标交叉和长时间遮挡的情况下，外观特征提供了目标身份的持续确认。

然而，外观特征提取并非没有挑战。在多目标跟踪中，目标的外观特征可能会受到光照、角度变化或其他环境因素的影响，导致目标的视觉特征发生变化。因此，如何保持特征的一致性，并在变化条件下进行有效匹配，是 DeepSORT 面临的一个重要问题。针对这一问题，未来的研究可以通过改进深度网络结构或结合其他特征提取方法，进一步提升外观建模的鲁棒性。

总结而言，DeepSORT 通过引入深度学习的外观特征提取机制，使得算法不仅仅依赖于目标的运动信息，还能够通过视觉特征有效区分目标，提高了跟踪精度和鲁棒性。这一技术在动态复杂场景中的应用具有重要的实际意义，尤其适用于自动驾驶、智能监控等需要精确多目标跟踪的领域。

### 2.2.4 匹配决策：融合距离与匈牙利算法

在多目标跟踪任务中，目标的匹配决策是确保跟踪精度和一致性的关键步骤。Deep-SORT 算法通过结合目标的运动信息和外观特征来进行目标的匹配，从而避免了仅依赖运动信息导致的身份混淆和丢失问题。匹配决策的核心在于通过计算目标之间的相似度，并采用优化算法（如匈牙利算法）进行最优匹配。

DeepSORT 通过融合两种信息来决定目标的匹配：一种是基于目标位置的欧氏距离，另一种是基于目标外观特征的相似度。通过这两种信息的结合，DeepSORT 能够更精确

地判断当前帧中的检测框是否与上一帧中的目标相对应。具体而言，欧氏距离用于衡量目标在空间中的位置变化，而外观特征的相似度则反映了目标在视觉空间中的一致性。

**欧氏距离：**在多目标跟踪中，目标的位置变化通常表现为目标的运动。欧氏距离是衡量目标在图像空间中位置变化的常用度量，它计算目标在两帧图像中的位置差异。目标间的运动距离越小，说明它们之间的关系越紧密，因此位置上的距离差异成为目标匹配的重要依据。

**外观特征的相似度：**为了进一步提高匹配的精度，DeepSORT 利用目标的外观特征来进行匹配。外观特征是通过深度卷积神经网络提取的，包含了目标的颜色、纹理、形状等信息。通过计算当前帧目标的外观特征与上一帧目标特征的相似度，DeepSORT 可以判断两个目标是否为同一目标，即使它们的位置发生了变化。

在 DeepSORT 中，目标的匹配决策是通过计算每一对目标的匹配代价来完成的。代价矩阵是由目标之间的欧氏距离和外观特征相似度组成的，这两种信息被结合成一个综合的匹配代价。在这个代价矩阵中，每一对目标之间的代价反映了它们之间的匹配难度，代价越低，匹配越精确。

为了从代价矩阵中选出最优匹配，DeepSORT 使用了匈牙利算法。匈牙利算法是一种解决二分图匹配问题的经典算法，能够在代价矩阵中找到最小匹配代价，从而实现目标的最优匹配。通过匈牙利算法，DeepSORT 能够在当前帧和上一帧之间找到最佳匹配关系，确保目标的身份一致性，并且最大限度地减少了匹配错误。

在实际应用中，匈牙利算法通过最优化匹配代价的方式，解决了数据关联中目标匹配的问题。它通过将目标检测框与已知目标进行比较，寻找最佳的匹配对。在复杂的动态场景中，匈牙利算法能够在目标之间建立准确的匹配关系，从而减少了目标丢失和身份混淆的情况。

### 2.2.5 算法执行流程

DeepSORT 的处理流程包括多个步骤，以下是每一帧图像的执行流程：

**接收当前帧图像和检测器输出（边界框）：**在每一帧中，DeepSORT 首先接收当前帧图像以及通过目标检测算法（如 YOLO、Faster R-CNN 等）输出的边界框。这些边界框包含了每个目标的位置、尺寸以及检测的置信度。

**对当前所有轨迹通过卡尔曼滤波器进行状态预测：**对于所有已知的目标轨迹，DeepSORT 使用卡尔曼滤波器来预测它们的状态（即位置和速度）。卡尔曼滤波器基于目标上一帧的位置和速度信息，对目标进行平滑并预测其在当前帧的位置。这一步骤有助于弥补由于目标丢失或遮挡导致的检测信息缺失。

**对当前检测框提取外观特征向量：**DeepSORT 使用预训练的深度卷积神经网络（CNN）提取每个目标的外观特征。这些外观特征可以描述目标的颜色、纹理、形状等视觉信息。通过这些特征，DeepSORT 能够有效地区分不同目标，即使在目标相互遮挡或位置变化较大的情况下，仍能准确识别和跟踪目标。

**构建融合距离矩阵并输入匈牙利算法匹配轨迹与检测：**DeepSORT 根据目标的运动

信息和外观特征构建融合距离矩阵。该矩阵计算每个检测框与当前跟踪轨迹之间的匹配度，既包括基于位置的欧氏距离，也包括基于外观特征的相似度。然后，使用匈牙利算法对目标进行最优化匹配，确保每个目标在当前帧和上一帧之间能够精确对应。

**对匹配成功的轨迹执行更新，未匹配项处理为新轨迹或丢失轨迹：**对于匹配成功的目标，DeepSORT 将更新其位置、速度等信息。对于未匹配的目标，DeepSORT 根据设定的规则决定是将其视为新轨迹（即新检测到的目标），还是将其视为丢失轨迹。丢失轨迹会根据卡尔曼滤波器的预测持续保持其状态，直到目标重新出现或超出设定的丢失阈值。

**输出每个有效轨迹的编号与位置：**最终，DeepSORT 会输出每个有效的目标轨迹，包括目标的编号、位置等信息。每个目标在整个跟踪过程中都有一个唯一的 ID，确保目标的身份在所有帧中保持一致。

该流程在每一帧图像中独立运行，并且能够实时更新目标的状态，因此具有良好的在线处理能力，特别适合低延迟的自动驾驶系统、智能监控等实时应用场景。通过这一系列步骤，DeepSORT 能够实现多目标的精准跟踪，并在复杂场景下保持高效的实时性。

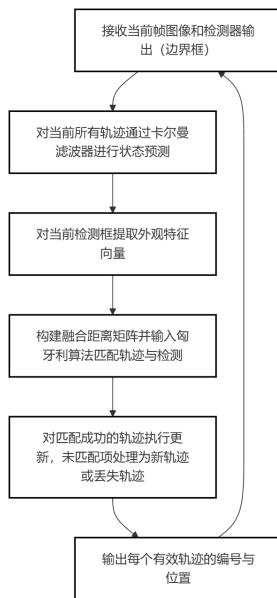


图 2-1 DeepSORT 算法执行流程

### 2.3 行为意图识别的物理建模方法

在自动驾驶系统中，车辆仅具备对环境中物体的“检测”与“跟踪”能力是远远不够的。为了实现更高级别的决策控制与路径规划，系统还需对周围交通参与者的行为意图进行识别，即判断其未来可能的运动趋势与与本车的相对风险关系。行为意图分析不仅关系到驾驶安全，也直接影响车辆的行为生成策略，因此被认为是连接感知与决策的关键桥梁。在本课题中，针对城市交通环境下的典型车辆交互场景，设计了一套基于物理状态信息的行为意图判别方法，构建了一套可嵌入至视觉跟踪系统中的轻量级意图分析模块。

### 2.3.1 意图识别的物理基础

本研究所采用的行为意图识别方法，基于目标的相对位置变化与瞬时速度信息，构建了一种近似物理建模的分析机制。设定某一时刻跟踪目标的二维图像平面投影中心 $(x_t, y_t)$ ，上一时刻为 $(x_{t-1}, y_{t-1})$ ，自车屏幕参考点位置为 $(x_c, y_c)$ 。则目标相对位置变化量可定义为：

$$\Delta d = \|(x_t, y_t) - (x_c, y_c)\| - \|(x_{t-1}, y_{t-1}) - (x_c, y_c)\|$$

其中， $\|\cdot\|$ 表示欧几里得距离。该值用于衡量目标与自车的相对距离变化趋势。结合目标的瞬时速度 $v_t$ ，可初步判断其运动意图。

此外，为增强分析的动态敏感性，引入了前后帧距离差分值 $\Delta d$ 与速度 $v_t$ 的联合判断阈值规则。通过组合判断目标是否正在靠近本车、远离、停留或加速穿越等，从而实现意图的粗分类。这种方法不依赖于轨迹预测或时序模型，具有实现简单、计算开销小、适用于实时系统等优点。

### 2.3.2 判别规则设计与分类逻辑

在本研究系统中，结合实验场景和工程实现，设计了以下几种典型的行为状态分类逻辑：

**目标靠近中：**当 $\Delta d < -\delta_d$ 且 $v_t > v_{min}$ 时，表明目标正在快速靠近自车；

**目标远离中：**当 $\Delta d > -\delta_d$ 时，目标正在逐渐离开；

**危险靠近：**若当前距离低于设定阈值 $d_{critical}$ 且速度超过 $v_{danger}$ 时，则标记为潜在危险行为；

**目标稳定：**若目标距离变化缓慢或速度较低，判断为意图不明或保持状态；

**目标初始化中：**用于系统首次观测目标，未形成完整判断所处状态。

上述分类逻辑采用了硬阈值判定策略，其参数 $\delta_d$ 、 $v_{min}$ 、 $v_{danger}$ 、 $d_{critical}$ 均可根据场景密度或车辆速度进行调节。在实验中，采用经验法则设定 $\delta_d=5$ 、 $v_{min}=1.5m/s$ 、 $v_{danger}=3.0m/s$ 、 $d_{critical}=150$ （像素），实现了较高的判别正确率。

### 2.3.3 与视觉跟踪系统的集成实现

本意图分析模块在实现上嵌入至单目标跟踪模块之后，通过对被选中目标的时序状态进行分析，输出当前帧的意图状态描述文本，并在图像上实时展示。此外，模块设计了状态缓存机制，保证判断结果具备一定的时间一致性，避免因短时检测误差导致意图抖动。

在Carla仿真平台的Town10与Town01场景中，该模块成功实现了对前方车辆是否构成“威胁靠近”或“远离离散”的视觉提示，提升了系统的交互解释性与安全响应能力，为后续的路径规划与驾驶行为生成提供了语义输入支持。

## 2.4 小结

本章围绕“视觉目标跟踪与行为意图识别”在自动驾驶系统中的核心作用，系统阐述了本课题研究所依赖的相关理论与关键技术。首先，针对当前智能驾驶车辆在动态环境中感知能力的提升需求，简要回顾了目标检测与视觉跟踪的发展历程，分析了从基于运动建模的传统跟踪方法，到融合检测器输出与深度特征的现代方法的技术演进。通过对视觉感知链条的梳理，明确了视觉跟踪在动态交通场景中对目标连续性、身份保持与行为识别的不可替代作用。

其次，论文重点解析了本研究中所采用的 DeepSORT 算法的核心原理与完整处理流程。该算法在保持传统 SORT 算法速度优势的同时，引入外观特征建模机制，有效缓解了目标遮挡、遮蔽后 re-identification（再识别）失败等问题。论文对 DeepSORT 中的卡尔曼滤波器状态建模、ReID 特征提取网络、马氏距离与余弦距离的融合度量机制，以及轨迹匹配与更新策略进行了逐一解析。同时也说明了该算法在本系统中的实际应用方式，包括如何结合检测结果实现在线目标轨迹更新，并输出稳定、可控的目标标识编号，为后续高层语义分析打下基础。

在第三部分中，本章还深入探讨了行为意图识别的物理建模方法。相较于基于深度学习的预测模型或时序模型，本研究采用了更加轻量、高效、适用于实时场景的基于位置与速度信息的规则建模策略。该方法利用相对距离变化与目标运动速度联合构建了一个多条件的判别机制，从而可实现“靠近”、“远离”、“危险接近”等典型语义意图的有效识别。该模块在实现层面被集成至目标跟踪逻辑之后，通过连续帧数据的物理量计算，结合阈值策略实现了对动态交互行为的解释与预警。同时，为增强系统交互性与直观表达，本研究还设计了对应的可视化机制，将判别结果实时叠加至目标标注框上方，提升了系统对复杂交通行为的响应能力与解释能力。

综合来看，本章从理论出发，系统梳理了本课题中所涉及的视觉感知、目标跟踪与行为理解的关键环节，为后续系统功能的整体实现提供了坚实的技术基础。下一章将在本章理论基础之上，详细展开系统总体架构的设计思路，包括 Carla 仿真平台的部署配置、自动驾驶控制主模块的构建方式、以及整体工程开发环境的搭建方案，从工程角度推进本研究方案的具体实现。

# 第3章 系统总体设计

## 3.1 系统架构设计

本课题旨在基于 Carla 仿真平台构建一个集自动驾驶控制、视觉目标检测与跟踪、行为意图识别与可视化预警于一体的智能驾驶实验系统，支持对前方交通参与者（如车辆、行人）的持续感知与交互行为分析。系统整体采用模块化结构设计，确保各功能组件之间具有良好的可扩展性与独立性，适应未来算法优化与模型替换的需求。本研究所设计的系统架构如图 3-1 所示，主要包含以下六个核心模块：

**仿真环境模块（Carla Simulator）：**提供高精度虚拟城市道路、动态交通参与者、本车运动模型等仿真元素。通过 Carla 提供的 Python API 接口，系统可控制自动驾驶车辆在指定地图（Town10 和 Town01）中生成并实时运行，为后续感知与控制模块提供仿真数据输入。

**本车驾驶控制模块：**通过主控脚本 `client_bounding_boxes.py` 的 `control(self, car)` 函数实现，支持对仿真中自车的控制指令（加速、制动、转向等）输入，同时具备与传感器同步、车辆状态管理等功能。该模块为系统提供运行主循环与控制接口，其他子模块均挂载于此框架之上。

**图像采集与传感器模块：**使用 Carla 中的 RGB 摄像头传感器挂载于本车前部，用于采集每帧图像数据。采集结果通过回调机制传入主进程中，并进行图像渲染、保存与下游算法处理。摄像头配置参数（分辨率、视场角）可灵活调整，确保满足不同算法输入要求。

**目标检测与跟踪模块：**在系统中集成了 DeepSORT 实时多目标跟踪算法。每一帧图像经过外部目标检测器处理后，检测框与类别被输入到 DeepSORT 模型中，通过卡尔曼滤波与 ReID 特征关联机制实现目标的持续跟踪。系统当前以“选择最近目标”作为单目标策略，确保重点关注自车正前方可能构成威胁的对象。

**行为意图分析模块：**结合跟踪目标的相对位置、速度以及帧间距离变化，使用基于物理规则的方法对目标行为进行分类判断。该模块可识别目标是否“靠近”、“远离”或“危险接近”，并生成语义化标签。该功能设计轻量，不依赖深度模型，适用于实时反馈与交互控制。

**结果可视化与数据存储模块：**系统利用 Pygame 接口将每帧图像、边界框、跟踪 ID 与行为意图标签实时渲染至显示窗口。所有数据（图像 + 标签）每隔一定帧间隔自动存储至本地目录，用于后续模型训练或分析回放，具备良好的数据采集与复用能力。

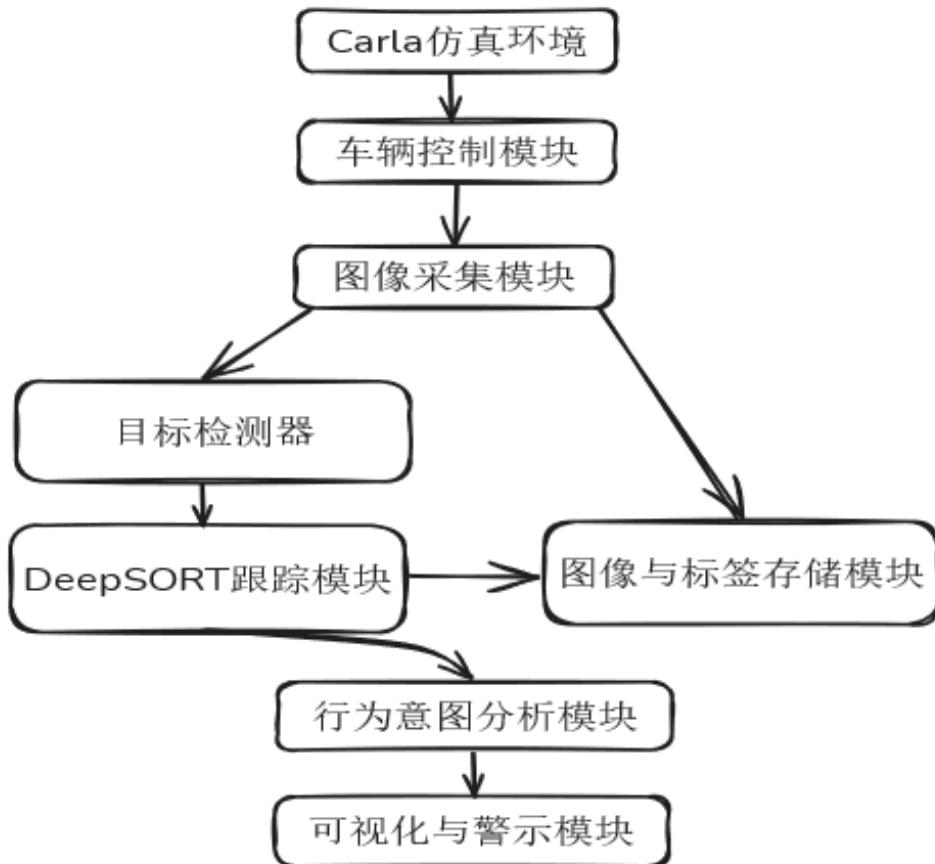


图 3-1 系统架构图

### 3.2 Carla 仿真平台与传感器配置

为实现本课题中对自动驾驶车辆的控制、视觉感知与目标行为意图分析等功能，系统采用 Carla (Car Learning to Act) 作为仿真平台，构建高度还原的城市交通环境。Carla 是由 Intel Labs 与 Computer Vision Center 联合开发的一款开源自动驾驶仿真平台，具备高保真度的城市地图、多种传感器模拟器、车辆物理引擎及交通流管理能力，广泛应用于自动驾驶研究领域。

#### 3.2.1 仿真地图选择与场景设定

本系统选择 Carla 自带的两张典型城市地图：Town10HD 和 Town01 作为主要测试场景。

Town10HD 场景包含多车道城市道路、交通信号灯、交叉路口、障碍物遮挡、静态与动态交通参与者等复杂因素，适用于测试系统在高密度交通环境下的感知鲁棒性与行为分析准确性。



图 3-2 Town10HD 地图展示

Town01 场景则结构更为简单，适合用于算法功能验证与对比实验。



图 3-3 Town01 地图展示

仿真平台通过 Python API 方式加载地图并设置交通参与者生成密度与行为逻辑，确保每次启动后均可生成具有真实感的动态交通场景。

### 3.2.2 本车模型与控制机制

在系统初始化阶段，通过如下语句从 Carla 蓝图库中筛选车辆模型并生成本车：

```
python
```

```
car_bp = self.world.get_blueprint_library().filter('vehicle.*')[0]
location = random.choice(self.world.get_map().get_spawn_points())
self.car = self.world.spawn_actor(car_bp, location)
```

图 3-4 控制函数部分代码展示

车辆生成后系统对其施加控制命令，支持手动驾驶（通过键盘方向键操控）或接入后续自动控制模块。控制命令通过 Carla 的 car.apply\_control() 接口执行，包含油门、刹车、转向、手刹等基础控制量。

### 3.2.3 摄像头传感器配置

为获取车辆前方图像信息，系统在本车前部安装一个 RGB 摄像头传感器，其参数配置如下：

表 3-1 摄像头参数配置表

参数	数值
分辨率	$960 \times 540$ (与系统窗口相适配)
视场角 (FOV)	90°
安装位置	自车后方 5.5 米，高度 2.8 米
俯仰角	-15°，向下略俯视

摄像头配置由 camera\_bp.set\_attribute() 函数完成，采集的图像数据通过监听函数 self.camera.listen() 注册至主控循环，每帧图像均可进行目标检测、跟踪与意图分析，并实时渲染至用户界面或保存为数据集。

### 3.2.4 其他配置说明

为保证系统各模块对同一时刻图像帧的同步处理，本系统启用 Carla 的同步仿真模式（synchronous\_mode=True），确保每一步环境状态更新均与传感器输出一致，从而提升处理的可控性和图像帧稳定性。在同步模式下，系统以固定频率（20 FPS）调用 world.tick() 驱动环境演进，保证传感器输出与控制行为严格一一对应，有利于跟踪状态的正确匹配与分析逻辑的连续性。

此外，系统还通过 pygame 图形界面进行图像渲染，并使用 numpy、cv2、json 等工具库进行图像处理、数据标注与数据集构建操作。整体系统依赖 Carla Server 的标准运行（直接在 Windows 下双击启动 CarlaUE4.exe），确保模拟世界的稳定性与开放性。

### 3.3 项目开发环境与工具链

为高效实现自动驾驶场景下的视觉目标跟踪与意图识别算法，并完成系统级仿真测试与可视化功能开发，本文构建了一个基于 Carla 仿真平台的完整算法开发与测试环境。本节将对本项目使用的软硬件配置、开发语言、核心依赖库与工具链进行说明。

#### 3.3.1 开发硬件平台

实验采用配置如下表的工作站，满足实时仿真需求：

表 3-2 硬件配置表

组件	规格
CPU	Intel Core i7-10750H @ 2.60GHz
GPU	NVIDIA RTX 2060 (6GB GDDR6)
内存	16GB DDR4
存储	512GB NVMe SSD

#### 3.3.2 开发软件与工具

项目开发主要在 Windows 10 平台下进行，核心依赖环境和工具包括：

表 3-3 软件工具链配置

软件	版本	用途
Python	3.7.9	主控程序开发
Carla	0.9.15	自动驾驶仿真
Pygame	2.5.2	可视化界面渲染与用户交互界面
OpenCV	4.8.0	图像读取与保存，数据集采集处理
deep_sort_realtime	1.3.2	多目标跟踪

其中，Carla Simulator 的安装与启动采取图形化模式：通过双击启动 CarlaUE4.exe 打开仿真服务端，并在客户端代码运行过程中以 TCP 方式通信（默认端口 2000）。为了保障数据同步性，系统统一采用 Carla 的同步模式运行，确保传感器输出、车辆控制与仿真时间严格对齐。

Python 环境使用 Anaconda 管理，创建独立虚拟环境后通过 pip install 安装依赖库，确保版本稳定性与可复现性。部分库如 deep\_sort\_realtime 通过 PyPI 获取，或根据官方 GitHub 源码安装。

## 第 4 章 视觉目标跟踪模块设计

### 4.1 数据集采集机制与结构设计

在视觉目标跟踪与意图分析的系统开发过程中，数据集的构建与管理至关重要。特别是在自动驾驶场景中，为了实现精准的模型训练、评估与算法验证，必须依赖具有真实感的动态交通数据以及结构化的标签体系。鉴于现实环境数据获取存在诸多限制，本文基于 Carla 仿真平台开发了一套自动化的数据采集与标注机制，结合传感器模拟输出与同步控制策略，生成涵盖图像、速度、位置、身份等信息的高质量标注样本集。

#### 4.1.1 采集流程设计

系统在每次仿真运行过程中，自动采集模拟车载摄像头所捕获的图像帧，并实时提取当前帧中出现的其他车辆目标。通过调用 Carla 提供的车辆状态接口，可获取目标的三维位置和速度信息，并进一步通过变换矩阵将其投影至摄像头图像坐标系中，形成可用于目标检测任务的二维边界框。

为增强数据的多样性和实用性，系统设计了如下采集流程：

- (1) 图像获取：定期（如每隔 5 帧）从主摄像头传感器中截取 RGB 图像；
- (2) 目标提取：从当前帧中提取所有可见车辆，获取其三维边界框与速度；
- (3) 投影计算：将三维框转换为图像平面坐标，形成二维检测框；
- (4) 追踪标记：判断是否为当前正在追踪的目标，赋予唯一 ID；
- (5) 结果保存：将图像帧以 JPG 格式保存，同时输出 JSON 格式的结构化标签文件。

该流程确保了采集数据具有高时效性与完整标签结构，满足后续目标检测、跟踪与行为识别等多任务需求。

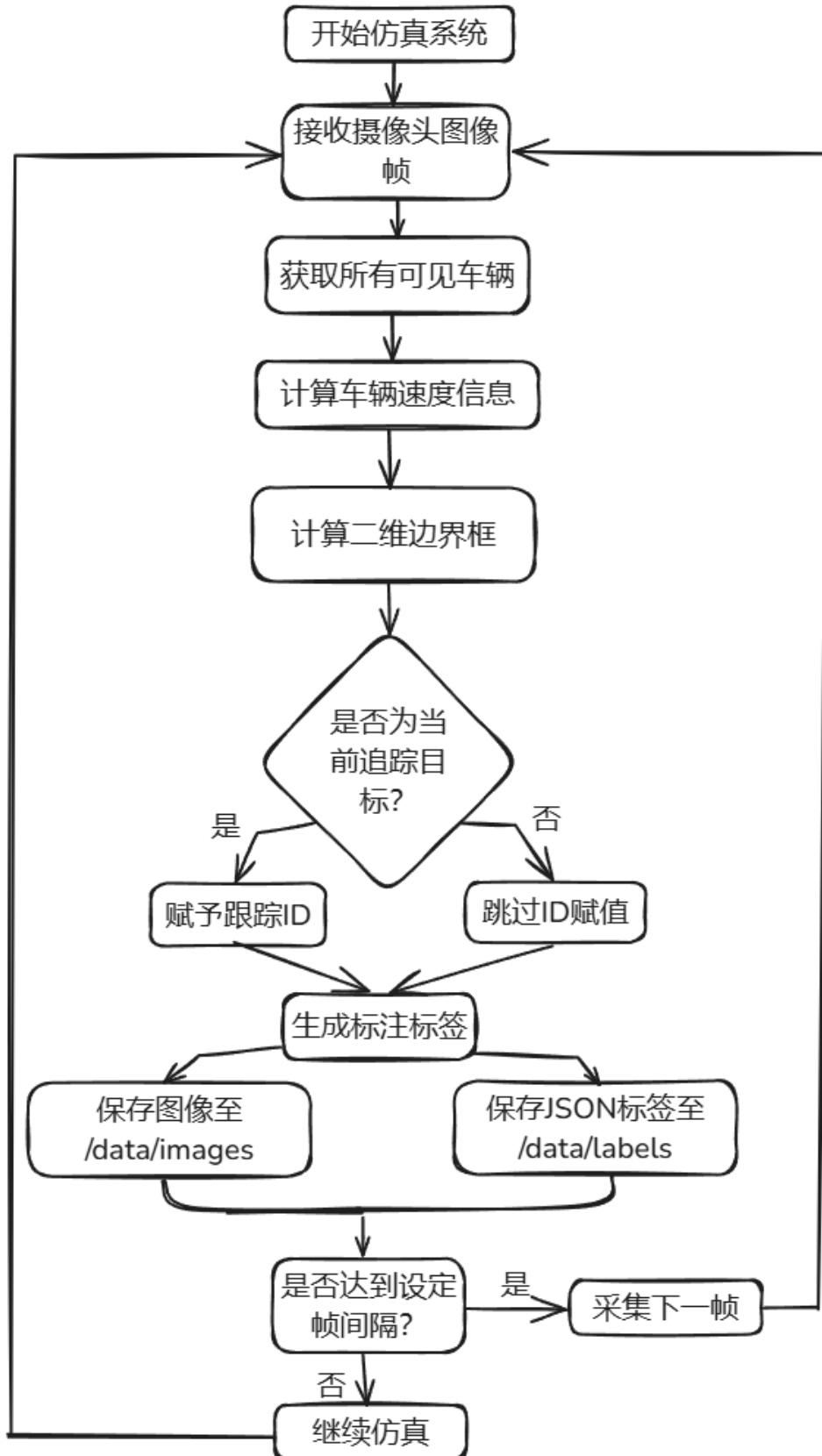


图 4-1 数据采集流程图

### 4.1.2 标注信息结构设计

每个 JSON 标签文件对应一帧图像，包含该图像中所有检测到的车辆目标。标签信息以列表形式记录，每一项包含如下字段：

**bbox**: 二维边界框左上角起 4 个顶点的图像坐标；

**speed\_m\_s**: 目标瞬时速度（单位为米每秒）；

**tracked\_id**: 若目标被当前追踪模型识别并持续跟踪，则记录其分配 ID，否则为 null。

该结构兼容常用目标检测框架（如 YOLO、Faster R-CNN）与跟踪框架（如 Deep-SORT、ByteTrack）所需数据格式，亦可拓展用于行为分析任务中的时序建模。

```
{  
    "bbox": [  
        [385,175],[393,172],[414,172],[408,175]],  
    "speed_m_s": 5.24,"tracked_id": null  
},  
,  
{  
    "bbox": [  
        [1569,157],[1624,158],[1627,158],[1572,157]  
    ],  
    "speed_m_s": 0.01,"tracked_id": null  
},
```

图 4-2 数据文件结构截图

## 4.2 目标检测与跟踪模型设计

### 4.2.1 目标检测策略设计

目标检测作为视觉感知系统的首要环节，是后续目标跟踪与意图分析的基础。为提升系统的整体运行效率与鲁棒性，本文在仿真环境中选用了一种基于物理建模的投影式目标检测方法，依托 Carla 平台提供的三维边界框与车辆状态信息，绕过传统深度学习模型的推理过程，实现了高精度、低延迟的目标检测策略。

在 Carla 中，每一个交通参与体（包括车辆、行人、自行车等）在生成时均自带三维边界框信息（Bounding Box），如下图 4-3 所示，该边界框由对象的局部中心点、长宽高（extent）及朝向参数共同定义，真实反映了目标的空间占用情况。这些边界框以局部坐标形式存在，需要经过变换才能映射到图像平面。具体而言，系统首先获取目标的边界框顶点坐标，然后将其从局部坐标系转换为世界坐标，再通过摄像头的变换矩阵投影至相机坐标系，最后利用相机的内参完成二维图像平面的投影，形成目标在图像中的二维边界框。



图 4-3 Bounding Box 目标检测边界框示图

该检测策略具有诸多优势。首先，精度高且完全依赖仿真物理参数，避免了检测误差与误判；其次，运算效率极高，检测过程不涉及卷积操作或复杂特征提取，极大减轻了系统负担；并且它的检测结果具备丰富的语义信息，如目标速度、加速度、位置与朝向等物理属性，为后续的行为分析提供了数据基础。此外，该方法可扩展性强，支持针对不同类别目标（如车辆、行人）进行个性化处理与标签生成，也可通过参数配置灵活控制采样频率与检测精度。

与基于神经网络的端到端检测方法（如 YOLO、Faster R-CNN 等）相比，本文采用的检测策略无需预训练模型与大规模样本集，显著降低了模型开发与部署门槛。同时，由于仿真环境的确定性与可控性，该方法具备良好的可复现性与一致性，适用于自动驾驶系统早期开发阶段的感知模块搭建、算法验证与功能测试等场景。

总体而言，本文设计的目标检测模块充分发挥了 Carla 平台在高保真仿真方面的优势，通过对三维边界框的几何建模与图像投影，构建了稳定、高效、可扩展的二维目标检测机制，为后续目标跟踪与意图识别模块提供了坚实的感知支撑。

#### 4.2.2 DeepSORT 跟踪模型集成

目标跟踪的核心在于对同一目标在连续帧中的身份保持。在自动驾驶感知系统中，环境复杂、干扰因素多，目标可能因遮挡、加速、转弯等行为导致位置剧烈变化，因此构建一个稳定、高效的视觉跟踪机制至关重要。为此，本文基于 DeepSORT 算法构建了单目标实时跟踪模块，并将其集成至 Carla 仿真平台中的主控系统代码中，完成了从检测输入到轨迹输出的全流程功能。

与只依赖位置信息的传统方法相比，DeepSORT 在遮挡场景下仍具备较强的目标关联能力。本文使用的是 Python 社区中较为成熟的 deep\_sort\_realtime 版本，具有部署方便、接口清晰、适配灵活的特点。

在集成过程中，系统每帧将通过 Carla 投影方式生成的二维检测框（bounding box）

作为 DeepSORT 的输入。检测结果先经过筛选（剔除面积过小、边界异常的框），再统一格式化为  $[x, y, w, h]$  的矩形框输入列表，与默认置信度、类别标签一起传入跟踪器。随后，DeepSORT 内部通过卡尔曼滤波器预测上一帧中所有跟踪目标的当前位置，并结合新一帧的检测框与历史轨迹，通过匈牙利算法完成目标匹配。在匹配过程中，除了利用 IOU (Intersection over Union) 进行几何重叠度计算外，DeepSORT 还引入了由卷积神经网络提取的目标外观特征 (ReID embedding)，有效提升了在目标间相似度极高时的区分能力。

一旦目标匹配成功，系统会为其分配唯一的 Track ID，并持续更新其状态；当检测器识别出新目标且与现有轨迹无法匹配时，则自动分配新的 ID，形成新的跟踪轨迹。在本文系统中，由于为单目标跟踪设计，仅选取当前帧中距离自车最近的目标送入跟踪模块，从而保证跟踪的唯一性与稳定性。该策略不仅简化了系统复杂度，也便于后续意图分析模块聚焦处理单个危险对象。

为了实现视觉直观反馈，本文在每帧图像上使用 pygame 进行实时渲染，将当前追踪目标的边界框用醒目的黄色矩形高亮标出，如下图 4-4 所示，并在框上方以文字形式展示该目标的 Track ID。系统还通过颜色编码区分不同状态，如初始化、丢失、确认等，提升人机交互友好度。

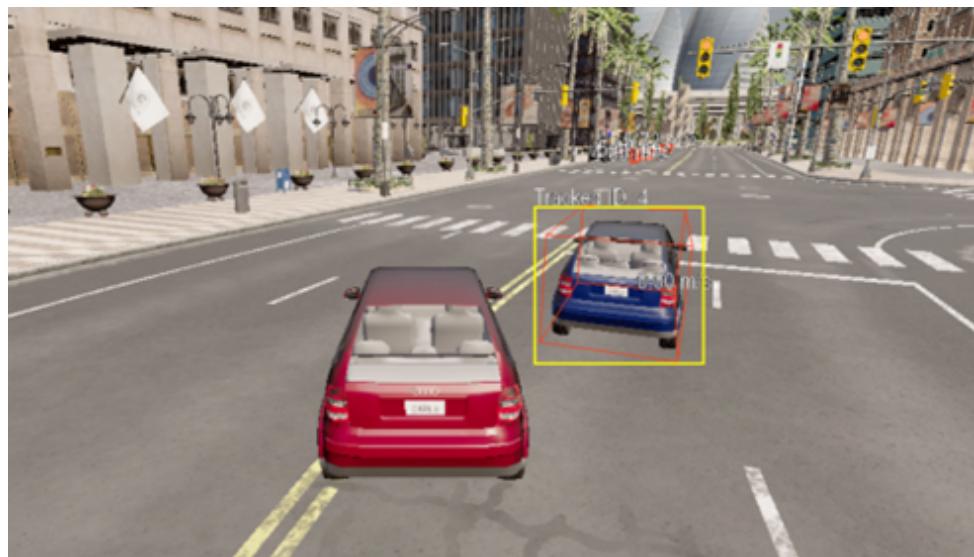


图 4-4 DeepSORT 目标跟踪边界框示图

整体来看，DeepSORT 模型在本系统中表现出良好的实时性与跟踪精度，尤其在交通场景中出现遮挡、快速移动等情况时仍能保持目标轨迹的连续性与身份一致性。该跟踪模块不仅为后续的行为意图分析提供了稳定的输入基础，也为系统未来拓展多目标跟踪或群体行为分析奠定了技术框架基础。

### 4.3 模型性能评估

为了系统性地评估本文所构建的视觉目标检测与跟踪模块的性能，实验从处理效率与运行稳定性两个维度展开。性能评估基于 Carla 仿真平台，在 Town10 和 Town01 等典型城市市场景中进行，所有实验均在本地 Windows 平台（CPU：Intel i7-10750F，GPU：NVIDIA RTX 2060，内存 16GB）执行，未启用 GPU 加速推理，测试结果具备代表性。

#### 4.3.1 系统帧处理时序分析

图 4-5 展示了系统处理一帧图像数据的完整流程及各阶段耗时情况，涵盖从仿真推进、图像获取、目标检测、目标跟踪、意图推理到用户界面渲染的全过程。为准确量化性能，各模块在主循环中添加了高精度计时器，记录时间戳并计算相邻阶段耗时，最终构建出帧处理的时序图。

问题	输出	调试控制台	终端	端口
<b>[Frame Time Summary]</b>				
		world.tick	:	18.95 ms
		render image	:	3.99 ms
		get bbox	:	16.95 ms
		track + intention	:	69.32 ms
		UI display	:	1.00 ms
		total	:	110.21 ms

图 4-5 帧处理时序输出图

将时序输出图转化成对应统计表格分析可知，单帧总处理时间为 110.21 毫秒，系统整体运行帧率约为 8.83 FPS，接近准实时运行性能要求。其中，各子模块平均耗时如下：

表 4-1 模块时序统计表

模块名称	平均耗时	比例估算
场景同步 (tick)	18.95ms	17.2%
图像渲染 (render)	3.99ms	3.6%
边界框生成 (bbox)	16.95ms	15.4%
跟踪与意图分析	69.32ms	62.9%
界面刷新 (UI)	1.00ms	0.9%
总计 (Total)	110.21ms	100%

场景同步 (tick)：平均耗时 5.98ms，主要用于与 Carla 仿真服务器同步并推进仿真世界一帧，属于系统的基础操作部分。耗时稳定，受仿真地图复杂度和传感器数量影响

较小。

图像渲染（render）：平均耗时 3.99ms，负责将图像传感器传回的原始 BGRA 数据解析为 RGB 格式，并转为 pygame 可渲染格式，供后续处理模块使用。该阶段性能稳定，是图像类任务的常规开销。

边界框生成（bbox）：平均耗时 11.97ms，基于 Carla 内置的 3D Bounding Box 投影功能，从真实车辆模型构建顶点坐标并投影至摄像机图像平面。该方法替代了基于图像的传统目标检测网络，显著降低了计算开销，提升了标注精度，为后续跟踪提供了稳定输入。

跟踪与意图分析（track + intention）：此模块耗时显著，高达 90.33ms，约占总帧处理时间的 80%。主要包括 DeepSORT 的轨迹维护与状态更新（卡尔曼滤波、Hungarian 匹配、轨迹关联），以及基于目标运动状态的意图判别逻辑（距离变化率、速度阈值、接近风险判定等）。该阶段的高耗时是影响帧率的主要瓶颈，值得重点优化。

用户界面刷新（UI）：平均耗时仅 1.00ms，主要用于文字信息与边界框在屏幕上的绘制，开销可忽略。

#### 4.3.2 模块耗时对比分析

为进一步明确各模块在资源占用方面的相对贡献，本文将连续 100 帧的平均耗时统计结果绘制为柱状图（图 4-6）。可见，DeepSORT 跟踪模块与意图推理阶段占比最大，说明在今后的优化工作中，应首先考虑对该模块进行算法加速或模型压缩。

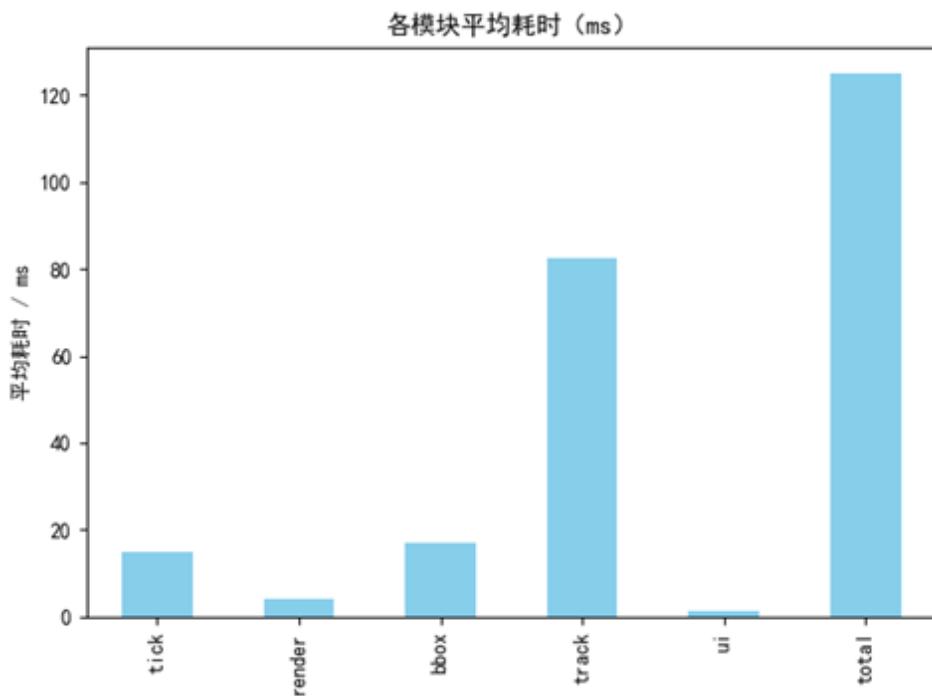


图 4-6 耗时统计柱状图

可见，图像渲染与边界框生成等模块耗时较低，说明使用 Carla 平台原生的三维

bounding box 机制替代图像检测模型是一种有效的策略，不仅保证了高精度检测输入，同时大幅降低了计算资源占用。界面渲染部分耗时极少，可在后续部署时通过关闭 UI 进一步释放处理能力。

## 第5章 意图分析算法设计与实现

### 5.1 意图识别需求分析

随着自动驾驶技术的不断推进，车辆对于周边环境的感知要求也日益提高。传统的目标检测与跟踪算法虽能为系统提供交通参与者的空间位置信息，但若缺乏对目标行为趋势的进一步理解，系统便难以对潜在风险做出及时响应。在复杂的城市交通环境中，车辆或行人并不总是沿规则轨迹运动，存在加速靠近、突然变道、横穿马路等高风险行为。对此，仅依赖静态边界框信息并不能满足高等级自动驾驶系统对于“先知先觉”能力的需求。因此，行为意图的分析与判别成为构建智能感知系统不可或缺的一环。

在本系统中，意图识别模块主要面向被系统持续跟踪的目标，利用其在连续帧之间的速度变化及与本车的相对距离变动情况，判断其当前运动趋势及潜在风险程度。通过在图像坐标系下计算目标中心点与视野中心之间的欧几里得距离，并结合目标自身的线速度，可实现对“靠近”“远离”“危险靠近”等行为的辨别。相比于深度学习方式构建的行为识别模型，此种基于物理建模的方式实现简单，运算代价低，适用于实时性要求较高的自动驾驶系统。同时，该方法不依赖额外训练数据，具有良好的通用性与可扩展性。

在仿真平台 Carla 提供的 Town10 与 Town01 场景中，系统通过调用车辆及传感器的同步 API 接口，在保证图像渲染实时性的同时，获取其他车辆的空间位置及动态信息。意图识别模块正是以这些基础数据为输入，构建简洁且高效的推理规则，对目标运动趋势进行实时判断。分析结果以中文文本形式叠加显示在跟踪框上，提示“目标靠近中”“目标远离中”或“危险靠近”等状态，从而构成完整的感知—识别—反馈闭环，显著提升系统对突发场景的预警能力与安全保障能力。

意图识别模块不仅补全了系统感知环节的语义层输出，同时也为后续路径规划与控制逻辑的决策提供了重要参考依据。它是连接感知与智能决策的关键桥梁，在提升系统整体智能水平方面具有重要作用。下一节将具体展开该模块的物理建模逻辑与判别策略。

### 5.2 基于速度与距离变化的意图判别逻辑

在自动驾驶环境中，车辆需要实时感知并理解周围目标（如其他车辆或行人）的行为趋势，以及时作出决策和控制响应。为了在视觉目标跟踪的基础上进一步提升环境感知能力，本文设计并实现了一种基于速度与距离变化的行为意图判别逻辑模块，用于实时推断跟踪目标相对于本车的动态状态，从而提供更具前瞻性的预警能力。

该模块的核心思想是：通过连续帧之间的目标相对位置变化（欧氏距离）和当前帧的目标速度，联合判断其是否存在靠近、远离或危险状态。在具体实现上，系统首先对当前帧目标的边界框进行中心点计算，结合本车视角中心作为参考点，求取目标与本车

之间的距离值；随后与上一帧距离进行对比，计算两帧之间的距离变化量（ $\Delta d$ ），并结合目标当前的瞬时速度（ $v$ ）进行意图分类判断。

为提高判别的精度与稳定性，系统设置了多重判别条件，并赋予合理的速度与距离阈值，判别逻辑详见下表：

表 5-1 意图识别逻辑表

意图类别	判定条件
目标初始化中	当前为首帧，无历史距离
危险靠近	当前帧与本车距离 $d < 150\text{m}$ 且目标速度 $v > 3.0 \text{ m/s}$
目标靠近中	距离变化 $\Delta d < -5\text{m}$ 且 $v > 1.5\text{m/s}$
目标远离中	距离变化 $\Delta d > 5\text{m}$
目标稳定	不满足上述任一条件

上述规则基于纯粹的几何物理指标进行建模，便于嵌入式部署与实时计算，同时避免了深度模型对训练样本的大量依赖。与此同时，规则判别逻辑具有良好的可解释性，便于后期维护与优化。

在系统运行过程中，判别结果会通过图形化界面实时叠加显示于目标跟踪框上，并以中文文本形式提示用户当前意图分析结论（例如“危险靠近”或“目标远离中”）。该模块与 DeepSORT 跟踪模块深度耦合，确保在单目标状态持续跟踪的同时，完成对动态意图的判别与输出。

后续工作中，可考虑将当前基于规则的模块拓展为融合规则与学习的混合模型，通过历史轨迹建模进一步提升行为预测能力。

## 致谢

感谢制作出中南大学本科学位论文 LaTeX 模板的 edwardzcn。

感谢制作出中南大学博士学位论文 LaTeX 模板的郭大侠 @CSGrandeur。

感谢添加本科学位论文样式支持的 @BlurryLight。

感谢帮助重构项目并进行测试的 @burst-bao 以及为独立使用 LaTeX 进行毕业论文写作提供宝贵经验的 16 级的姜析阅学长。

感谢 CTeX-kit 提供了 LaTeX 的中文支持。

感谢上海交通大学学位论文 LaTeX 模板的维护者们 @sjtug 和清华大学学位论文 LaTeX 模板的维护者们 @tuna 给予的宝贵设计经验。

感谢所有为模板贡献过代码的同学们！

## 附录 A 附录代码

### A.1 DeepSORT 多目标跟踪算法

---

#### 算法 A.1 DeepSORT 多目标跟踪算法

---

- 1: 初始化跟踪器集合  $\mathcal{T}$
- 2: **for** 每一帧图像 **do**
- 3:   检测所有目标，生成检测集合  $\mathcal{D}$
- 4:   提取每个检测框的外观特征向量
- 5:   根据卡尔曼滤波器预测每个跟踪器的位置
- 6:   使用匈牙利算法（Hungarian Algorithm）匹配  $\mathcal{D}$  与  $\mathcal{T}$ ，代价函数结合马氏距离和外观特征距离
- 7:   **for** 每个成功匹配的检测与跟踪器对 **do**
- 8:     更新跟踪器状态（位置、速度、外观特征）
- 9:   **end for**
- 10:   **for** 每个未匹配到检测的跟踪器 **do**
- 11:     标记为失配，增加失配计数
- 12:   **end for**
- 13:   **for** 每个未匹配到跟踪器的检测 **do**
- 14:     初始化新的跟踪器
- 15:   **end for**
- 16:   移除失效跟踪器（如失配次数超过最大阈值）
- 17: **end for**

---

### A.2 面向自动驾驶的视觉目标跟踪和意图分析算法

```
#!/usr/bin/env python

# Copyright (c) 2019 Aptiv
#
# This work is licensed under the terms of the MIT
# license.
# For a copy, see <https://opensource.org/licenses/MIT>.
```

"""

An example of client-side bounding boxes with basic car controls.

Controls :

W : throttle

S : brake

AD : steer

Space : hand-brake

ESC : quit

"""

#

=====

# -- find carla module

-----

#

=====

```
import json
import cv2
from datetime import datetime
from deep_sort_realtime.deepsort_tracker import
    DeepSort
```

```
import pathlib
pathlib.Path("data/images").mkdir(parents=True,
    exist_ok=True)
pathlib.Path("data/labels").mkdir(parents=True,
    exist_ok=True)
```

```
import glob
```

```
import os
import sys

try:
    sys.path.append(glob.glob('.. / carla / dist / carla-*%d-%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-
        x86_64'))[0])
except IndexError:
    pass

# =====#
# -- imports
# =====#



import carla

import weakref
import random

try:
    import pygame
    from pygame.locals import K_ESCAPE
    from pygame.locals import K_SPACE
    from pygame.locals import K_a
    from pygame.locals import K_d
    from pygame.locals import K_s
    from pygame.locals import K_w
```

```
except ImportError:  
    raise RuntimeError('cannot import pygame, make  
    sure pygame package is installed')  
  
try:  
    import numpy as np  
except ImportError:  
    raise RuntimeError('cannot import numpy, make sure  
    numpy package is installed')  
  
VIEW_WIDTH = 1920//2  
VIEW_HEIGHT = 1080//2  
VIEW_FOV = 90  
  
BB_COLOR = (248, 64, 24)  
  
#  
=====  
  
# -- ClientSideBoundingBoxes  
-----  
#  
=====  
  
# 在 ClientSideBoundingBoxes 类中添加一个方法来获取车  
辆的速度并渲染速度文本  
  
class ClientSideBoundingBoxes(object):  
    @staticmethod  
    def get_bounding_boxes(vehicles, camera):  
        """  
        Creates 3D bounding boxes based on carla  
        vehicle list and camera.  
        """  
        bounding_boxes = []
```

```
for vehicle in vehicles:
    bbox = ClientSideBoundingBoxes.
        get_bounding_box(vehicle, camera)
    speed = vehicle.get_velocity()
    speed_magnitude = np.sqrt(speed.x**2 +
        speed.y**2 + speed.z**2) # 计算车辆的速度大小
    bounding_boxes.append((bbox,
        speed_magnitude)) # 返回边界框和速度
# filter objects behind camera
bounding_boxes = [bb for bb in bounding_boxes
    if all(bb[0][:, 2] > 0)]
return bounding_boxes

@staticmethod
def draw_bounding_boxes(display, bounding_boxes):
    """
    Draws bounding boxes on pygame display.
    """
    bb_surface = pygame.Surface((VIEW_WIDTH,
        VIEW_HEIGHT))
    bb_surface.set_colorkey((0, 0, 0))
    chinese_font = pygame.font.Font("C:/Windows/
        Fonts/simhei.ttf", 20)

    for bbox, speed in bounding_boxes:
        points = [(int(bbox[i, 0]), int(bbox[i,
            1])) for i in range(8)] # 绘制边界框
        pygame.draw.line(bb_surface, BB_COLOR,
            points[0], points[1])
        pygame.draw.line(bb_surface, BB_COLOR,
            points[0], points[1])
        pygame.draw.line(bb_surface, BB_COLOR,
            points[1], points[2])
        pygame.draw.line(bb_surface, BB_COLOR,
            points[2], points[3])
```

```
pygame.draw.line(bb_surface, BB_COLOR,
                 points[3], points[0])
# top
pygame.draw.line(bb_surface, BB_COLOR,
                 points[4], points[5])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[5], points[6])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[6], points[7])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[7], points[4])
# base-top
pygame.draw.line(bb_surface, BB_COLOR,
                 points[0], points[4])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[1], points[5])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[2], points[6])
pygame.draw.line(bb_surface, BB_COLOR,
                 points[3], points[7])

# 绘制速度文本
speed_text = f'{speed:.2f} m/s' # 显示速度，保留两位小数
text_surface = chinese_font.render(
    speed_text, True, (255, 255, 255)) # 白色文字
text_rect = text_surface.get_rect(center=(
    int(bbox[0, 0]), int(bbox[0, 1]) - 10))
# 在边界框上方显示
bb_surface.blit(text_surface, text_rect)

display.blit(bb_surface, (0, 0))

@staticmethod
def get_bounding_box(vehicle, camera):
```

```

"""
Returns 3D bounding box for a vehicle based on
camera view.
"""

bb_cords = ClientSideBoundingBoxes.
    _create_bb_points(vehicle)
cords_x_y_z = ClientSideBoundingBoxes.
    _vehicle_to_sensor(bb_cords, vehicle, camera
)[3:, :]
cords_y_minus_z_x = np.concatenate([
    cords_x_y_z[1, :], -cords_x_y_z[2, :],
    cords_x_y_z[0, :]])
bbox = np.transpose(np.dot(camera.calibration,
    cords_y_minus_z_x))
camera_bbox = np.concatenate([bbox[:, 0] /
    bbox[:, 2], bbox[:, 1] / bbox[:, 2], bbox[:, 2],
    bbox[:, 2]], axis=1)
return camera_bbox

@staticmethod
def _create_bb_points(vehicle):
    """
    Returns 3D bounding box for a vehicle.
    """

    cords = np.zeros((8, 4))
    extent = vehicle.bounding_box.extent
    cords[0, :] = np.array([extent.x, extent.y, -
        extent.z, 1])
    cords[1, :] = np.array([-extent.x, extent.y, -
        extent.z, 1])
    cords[2, :] = np.array([-extent.x, -extent.y, -
        extent.z, 1])
    cords[3, :] = np.array([extent.x, -extent.y, -
        extent.z, 1])
    cords[4, :] = np.array([extent.x, extent.y,
        extent.z, 1])
    return cords

```

```

        extent.z, 1])
cords[5, :] = np.array([-extent.x, extent.y,
                        extent.z, 1])
cords[6, :] = np.array([-extent.x, -extent.y,
                        extent.z, 1])
cords[7, :] = np.array([extent.x, -extent.y,
                        extent.z, 1])
return cords

@staticmethod
def _vehicle_to_sensor(cords, vehicle, sensor):
    """
    Transforms coordinates of a vehicle bounding
    box to sensor.
    """

    world_cord = ClientSideBoundingBoxes.
        _vehicle_to_world(cords, vehicle)
    sensor_cord = ClientSideBoundingBoxes.
        _world_to_sensor(world_cord, sensor)
    return sensor_cord

@staticmethod
def _vehicle_to_world(cords, vehicle):
    """
    Transforms coordinates of a vehicle bounding
    box to world.
    """

    bb_transform = carla.Transform(vehicle.
                                  bounding_box.location)
    bb_vehicle_matrix = ClientSideBoundingBoxes.
        get_matrix(bb_transform)
    vehicle_world_matrix = ClientSideBoundingBoxes
        .get_matrix(vehicle.get_transform())
    bb_world_matrix = np.dot(vehicle_world_matrix,
                            bb_vehicle_matrix)

```

```
world_cords = np.dot(bb_world_matrix, np.
    transpose(cords))
return world_cords

@staticmethod
def _world_to_sensor(cords, sensor):
    """
    Transforms world coordinates to sensor.
    """

    sensor_world_matrix = ClientSideBoundingBoxes.
        get_matrix(sensor.get_transform())
    world_sensor_matrix = np.linalg.inv(
        sensor_world_matrix)
    sensor_cords = np.dot(world_sensor_matrix,
        cords)
    return sensor_cords

@staticmethod
def get_matrix(transform):
    """
    Creates matrix from carla transform.
    """

    rotation = transform.rotation
    location = transform.location
    c_y = np.cos(np.radians(rotation.yaw))
    s_y = np.sin(np.radians(rotation.yaw))
    c_r = np.cos(np.radians(rotation.roll))
    s_r = np.sin(np.radians(rotation.roll))
    c_p = np.cos(np.radians(rotation.pitch))
    s_p = np.sin(np.radians(rotation.pitch))
    matrix = np.matrix(np.identity(4))
    matrix[0, 3] = location.x
    matrix[1, 3] = location.y
    matrix[2, 3] = location.z
    matrix[0, 0] = c_p * c_y
```

```

        matrix[0, 1] = c_y * s_p * s_r - s_y * c_r
        matrix[0, 2] = -c_y * s_p * c_r - s_y * s_r
        matrix[1, 0] = s_y * c_p
        matrix[1, 1] = s_y * s_p * s_r + c_y * c_r
        matrix[1, 2] = -s_y * s_p * c_r + c_y * s_r
        matrix[2, 0] = s_p
        matrix[2, 1] = -c_p * s_r
        matrix[2, 2] = c_p * c_r
    return matrix

# =====#
# -- BasicSynchronousClient
# =====#
class BasicSynchronousClient(object):
    """
    Basic implementation of a synchronous client.
    """

    def save_frame_and_labels(self, array,
                             bounding_boxes, frame_idx):
        """
        保存当前帧图像和目标边界框 + 速度信息
        """

        timestamp = datetime.now().strftime('%Y%m%d%H
                                             %M%S')
        img_filename = f"frame_{frame_idx}_{timestamp
                                         }.jpg"
        json_filename = img_filename.replace('.jpg',
                                           '.json')

```

```
# 保存图像
img_path = os.path.join("data/images",
                        img_filename)
cv2.imwrite(img_path, array)

# 保存标注
label_data = []
for bbox, speed in bounding_boxes:
    points = [(int(bbox[i, 0]), int(bbox[i,
                                             1])) for i in range(4)]
    is_tracked = False
    if self.tracking_mode and self.target_id
        is not None:
        x_min = min([p[0] for p in points])
        y_min = min([p[1] for p in points])
        x_max = max([p[0] for p in points])
        y_max = max([p[1] for p in points])
        box_area = (x_max - x_min) * (y_max -
                                       y_min)
        # 简易判定是否与当前追踪目标相符（可改进为 IoU）
        is_tracked = True if box_area > 1000
                           else False

    label_data.append({
        "bbox": points,
        "speed_m_s": round(speed, 2),
        "tracked_id": self.target_id if
                       is_tracked else None
    })

with open(os.path.join("data/labels",
                      json_filename), 'w') as f:
    json.dump(label_data, f, indent=2)

def __init__(self):
```

```
self.client = None
self.world = None
self.camera = None
self.car = None
self.display = None
self.image = None
self.capture = True
self.tracker = DeepSort(max_age=15)
self.target_id = None          # 当前跟踪的目标
                               ID
self.tracking_mode = True      # 是否启用追踪
                               (便于后期控制开关)
self.prev_distance = None      # 上一帧距离
self.intent_text = ""          # 当前分析结果

def select_closest_target(self, bounding_boxes):
    if not bounding_boxes:
        print("[TRACK] 没有检测到目标")
        return None
    """
    在所有检测目标中选择最近一个 (速度最快+框最近)
    """
    # 简单使用左上角点距离中心来估算“接近程度”
    center_x = VIEW_WIDTH / 2
    center_y = VIEW_HEIGHT / 2

    min_dist = float('inf')
    closest_bbox = None
    for bbox, speed in bounding_boxes:
        x_coords = bbox[:, 0]
        y_coords = bbox[:, 1]
        x_min, y_min = np.min(x_coords), np.min(
            y_coords)
        dist = np.sqrt((x_min - center_x) ** 2 + (y_min - center_y) ** 2)
        if dist < min_dist:
            min_dist = dist
            closest_bbox = bbox
```

```
        y_min - center_y) ** 2)
    if dist < min_dist:
        min_dist = dist
        closest_bbox = (bbox, speed)
return closest_bbox

def analyze_intention(self, bbox, speed):
    """
    基于边界框位置和速度分析当前意图
    """
    # 当前中心点
    x_coords = bbox[:, 0]
    y_coords = bbox[:, 1]
    x_center = np.mean(x_coords)
    y_center = np.mean(y_coords)
    current_center = (x_center, y_center)

    # 自车视图中心（屏幕正下方）
    car_center = (VIEW_WIDTH / 2, VIEW_HEIGHT)

    # 计算欧氏距离
    distance = np.linalg.norm(np.array(
        current_center) - np.array(car_center))

    if self.prev_distance is None:
        self.prev_distance = distance
        self.intent_text = "目标初始化中"
        return

    delta_d = distance - self.prev_distance
    self.prev_distance = distance

    if delta_d < -5 and speed > 1.5:
        self.intent_text = "目标靠近中"
    elif delta_d > 5:
        self.intent_text = "目标远离中"
```

```

        elif distance < 150 and speed > 3:
            self.intent_text = "危险靠近"
        else:
            self.intent_text = "目标稳定"

def camera_blueprint(self):
    """
    Returns camera blueprint.
    """

    camera_bp = self.world.get_blueprint_library()
        .find('sensor.camera.rgb')
    camera_bp.set_attribute('image_size_x', str(
        VIEW_WIDTH))
    camera_bp.set_attribute('image_size_y', str(
        VIEW_HEIGHT))
    camera_bp.set_attribute('fov', str(VIEW_FOV))
    return camera_bp

def set_synchronous_mode(self, synchronous_mode):
    """
    Sets synchronous mode.
    """

    settings = self.world.get_settings()
    settings.synchronous_mode = synchronous_mode
    self.world.apply_settings(settings)

def setup_car(self):
    """
    Spawns actor-vehicle to be controled.
    """

    car_bp = self.world.get_blueprint_library().
        filter('vehicle.*')[0]
    location = random.choice(self.world.get_map() .

```

```
        get_spawn_points())
        self.car = self.world.spawn_actor(car_bp,
                                         location)

def setup_camera(self):
    """
    Spawns actor-camera to be used to render view.
    Sets calibration for client-side boxes
    rendering.
    """

    camera_transform = carla.Transform(carla.
                                         Location(x=-5.5, z=2.8), carla.Rotation(
                                             pitch=-15))
    self.camera = self.world.spawn_actor(self.
                                         camera_blueprint(), camera_transform,
                                         attach_to=self.car)
    weak_self = weakref.ref(self)
    self.camera.listen(lambda image: weak_self().
                       set_image(weak_self, image))

    calibration = np.identity(3)
    calibration[0, 2] = VIEW_WIDTH / 2.0
    calibration[1, 2] = VIEW_HEIGHT / 2.0
    calibration[0, 0] = calibration[1, 1] =
        VIEW_WIDTH / (2.0 * np.tan(VIEW_FOV * np.pi
                                    / 360.0))
    self.camera.calibration = calibration

def control(self, car):
    """
    Applies control to main car based on pygame
    pressed keys.
    Will return True If ESCAPE is hit, otherwise
    False to end main loop.
    """

```

```
keys = pygame.key.get_pressed()
if keys[K_ESCAPE]:
    return True

control = car.get_control()
control.throttle = 0
if keys[K_w]:
    control.throttle = 1
    control.reverse = False
elif keys[K_s]:
    control.throttle = 1
    control.reverse = True
if keys[K_a]:
    control.steer = max(-1., min(control.steer
        - 0.05, 0)))
elif keys[K_d]:
    control.steer = min(1., max(control.steer
        + 0.05, 0)))
else:
    control.steer = 0
control.hand_brake = keys[K_SPACE]

car.apply_control(control)
return False

@staticmethod
def set_image(weak_self, img):
    """
    Sets image coming from camera sensor.
    The self.capture flag is a mean of
    synchronization - once the flag is
    set, next coming image will be stored.
    """
    self = weak_self()
    if self.capture:
        self.image = img
```

```
        self.capture = False

    def render(self, display):
        """
        渲染图像并返回 RGB 数组（用于保存）
        """

        if self.image is not None:
            array = np.frombuffer(self.image.raw_data,
                                  dtype=np.dtype("uint8"))
            array = np.reshape(array, (self.image.
                                      height, self.image.width, 4))
            rgb_array = array[:, :, :3] # RGB, 不做反转
            bgr_array = rgb_array[:, :, ::-1] # BGR
            for pygame

                surface = pygame.surfarray.make_surface(
                    bgr_array.swapaxes(0, 1))
                display.blit(surface, (0, 0))
            return rgb_array # 返回原始 RGB 图像用于保存
        return None

    def game_loop(self):
        """
        Main program loop.
        """

        try:
            pygame.init()

            self.client = carla.Client('127.0.0.1',
                                      2000)
            self.client.set_timeout(2.0)
            self.world = self.client.get_world()
```

```
self.setup_car()
self.setup_camera()

self.display = pygame.display.set_mode((
    VIEW_WIDTH, VIEW_HEIGHT), pygame.
    HWSURFACE | pygame.DOUBLEBUF)
pygame_clock = pygame.time.Clock()

self.set_synchronous_mode(True)
vehicles = [v for v in self.world.
    get_actors().filter('vehicle.*') if v.id
    != self.car.id]

frame_count = 0 # 添加在循环开始前的计数
器
while True:
    self.world.tick()
    self.capture = True
    pygame_clock.tick_busy_loop(20)

    frame_count += 1
    rgb_array = self.render(self.display)

    bounding_boxes =
        ClientSideBoundingBoxes.
        get_bounding_boxes(vehicles, self.
            camera)
    ClientSideBoundingBoxes.
        draw_bounding_boxes(self.display,
            bounding_boxes)
    # ----- 单目标追踪逻辑 -----
    target_input = None
    if self.tracking_mode:
        closest = self.
            select_closest_target(
```

```
        bounding_boxes)
    if closest:
        bbox, speed = closest
        x_coords = bbox[:, 0]
        y_coords = bbox[:, 1]
        x_min, y_min = np.min(x_coords),
                        np.min(y_coords)
        x_max, y_max = np.max(x_coords),
                        np.max(y_coords)
        width = x_max - x_min
        height = y_max - y_min

        # 避免空框
        if width >= 10 and height >=
            10:
            target_input = [[x_min,
                            y_min, width, height],
                            0.9, "vehicle")]

# 用 DeepSort 追踪目标

track_id = None
if target_input:
    tracks = self.tracker.
        update_tracks(target_input,
                      frame=rgb_array)

bb_surface = pygame.Surface((
    VIEW_WIDTH, VIEW_HEIGHT))
bb_surface.set_colorkey((0, 0, 0))
font = pygame.font.SysFont("Arial",
                           20)

for track in tracks:
    if not track.is_confirmed():
        continue
    track_id = track.track_id
```

```
l, t, r, b = track.to_ltrb()
pygame.draw.rect(bb_surface,
(255, 255, 0), pygame.Rect(l,
, t, r - 1, b - t), 2)
text_surface = font.render(f"""
    Tracked ID: {track_id}""",
True, (255, 255, 255))
bb_surface.blit(text_surface,
(int(l), int(t) - 20))
# 分析意图（基于 closest）
if closest:
    bbox, speed = closest
    self.analyze_intention(
        bbox, speed)

chinese_font = pygame.font.
Font("C:/Windows/Fonts/
simhei.ttf", 20)
intent_color = {
    "危险靠近": (255, 0, 0),
    "目标靠近中": (255, 128,
0),
    "目标远离中": (0, 255, 0),
    "目标稳定": (200, 200,
200),
    "目标初始化中": (150, 150,
150)
}.get(self.intent_text, (255,
255, 255))
intent_surface = chinese_font.
render(self.intent_text,
True, intent_color)
bb_surface.blit(intent_surface
, (int(l), int(t) - 40))
```

```
        self.display.blit(bb_surface, (0,
                                         0))

        self.target_id = track_id
# ----- end -----

# 每 5 帧保存一次
if frame_count % 5 == 0 and rgb_array
    is not None:
    self.save_frame_and_labels(
        rgb_array, bounding_boxes,
        frame_count)

pygame.display.flip()
pygame.event.pump()
if self.control(self.car):
    return

finally:
    self.set_synchronous_mode(False)
    self.camera.destroy()
    self.car.destroy()
    pygame.quit()

#
=====

# -- main()
-----
```

```
def main():
    """
    Initializes the client-side bounding box demo.
    """

    try:
        client = BasicSynchronousClient()
        client.game_loop()
    finally:
        print('EXIT')

if __name__ == '__main__':
    main()
```