T.R.
SAKARYA UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

VIRTUAL CONTEXT-BASED MULTI-CAMERA VEHICLE TRACKING

MSc THESIS

Wael KABOUK

Software Engineering Department

Software Engineering Program

JULY 2025

T.R.
SAKARYA UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

VIRTUAL CONTEXT-BASED MULTI-CAMERA VEHICLE TRACKING

MSc THESIS

Wael KABOUK

Software Engineering Department

Software Engineering Program

Thesis Advisor: Prof.Dr. Ahmet ÖZMEN

JULY 2025

The thesis work titled “VIRTUAL CONTEXT-BASED MULTI-CAMERA VEHICLE TRACKING” prepared by Wael KABOUK was accepted by the following jury on 15/08/2025 by unanimously/majority of votes as a MSc THESIS in Sakarya University Graduate School of Natural and Applied Sciences, Software Engineering department, Software Engineering programe.

Thesis Jury

Head of Jury :	**Prof.Dr. Ahmet Özmen** (Advisor) Sakarya University	.....
Jury Member :	**Prof.Dr. Devrim Akgün** Sakarya University	.....
Jury Member :	**Assoc.Prof. Dr. Selman Hızal** Sakarya University of Applied Sciences	.....

STATEMENT OF COMPLIANCE WITH THE ETHICAL PRINCIPLES AND RULES

I declare that the thesis work titled "VIRTUAL CONTEXT-BASED MULTI-CAMERA VEHICLE TRACKING", which I have prepared in accordance with Sakarya University Graduate School of Natural and Applied Sciences regulations and Higher Education Institutions Scientific Research and Publication Ethics Directive, belongs to me, is an original work, I have acted in accordance with the regulations and directives mentioned above at all stages of my study, I did not get the innovations and results contained in the thesis from anywhere else, I duly cited the references for the works I used in my thesis, I did not submit this thesis to another scientific committee for academic purposes and to obtain a title, in accordance with the articles 9/2 and 22/2 of the Sakarya University Graduate Education and Training Regulation published in the Official Gazette dated 20.04.2016, a report was received in accordance with the criteria determined by the graduate school using the plagiarism software program to which Sakarya University is a subscriber, I accept all kinds of legal responsibility that may arise in case of a situation contrary to this statement.

(28/07/2025)

Wael KABOUK

To my father, Dr. Feraz Kabouk, whose faith and sacrifices illuminated this path.

ACKNOWLEDGMENTS

I am deeply grateful to my thesis advisor, Prof. Dr. Ahmet Özmen, for his unwavering support, expert guidance, and invaluable feedback throughout every stage of this research. His insightful suggestions and encouragement have been pivotal in refining the methodology and advancing the scientific contributions of this work.

I would like to express my sincere appreciation to the Information Technologies Department at Sakarya University for generously providing the three Logitech C920 PRO webcams that formed the cornerstone of the experimental setup. Their support in granting access to high-quality equipment enabled the practical validation and real-world applicability of the proposed Virtual Context framework.

My heartfelt thanks go to my friends Mohamed Burghol and Omar Haroub for their dedication and collaboration in configuring the experimental environment. Their willingness to contribute their personal laptops, assist with data acquisition, and monitor the experiments alongside me under demanding conditions greatly facilitated the timely and successful completion of this study.

Finally, I extend my gratitude to all colleagues and staff members who, directly or indirectly, contributed to the completion of this thesis. Their assistance and encouragement have been greatly appreciated.

Wael KABOUK

TABLE OF CONTENTS

ABBREVIATIONS

AI **:** Artificial Intelligence
API : Application Programming Interface
CV : Computer Vision
CNN : Convolutional Neural Network
FPS : Frames Per Second
FoV : Field of View – the spatial extent visible by a camera
GUI : Graphical User Interface
IoU : Intersection over Union
ITS : Intelligent Transportation System
MOT : Multi-Object Tracking
NMS : Non-Maximum Suppression
ROI : Region of Interest
SORT : Simple Online and Realtime Tracking
YOLO : You Only Look Once

SYMBOLS

A	: Accuracy (in %)
GT	: Ground Truth – the number of true object instances
h	: Height of a region of interest or image patch
TP	: True Positives – correctly matched detections
w	: Width of a region of interest or bounding box
x, y	: Top-left coordinates of a region or object in an image

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

VIRTUAL CONTEXT-BASED MULTI-CAMERA VEHICLE TRACKING

SUMMARY

This thesis presents a real-time multi-camera vehicle tracking framework designed to operate without geometric calibration or cross-view re-identification, addressing limitations inherent in conventional multi-camera systems. The proposed approach, termed **Virtual Context stacking**, introduces a lightweight fusion strategy that horizontally concatenates normalized region-of-interest (ROI) crops from multiple camera feeds into a single composite frame. This frame is then processed in a unified pass using YOLOv11 for detection and ByteTrack for multi-object tracking, thereby enabling accurate identity continuity across adjacent fields of view with minimal computational overhead.

The system architecture is realized through a modular Python-based pipeline integrating PyQt5 for interactive ROI delineation and OpenCV for video I/O and image processing. All operational logic is encapsulated within a singular script, and configuration parameters are externalized to facilitate straightforward deployment and iterative parameter tuning without code modification. Experimental validation utilizes three Logitech C920 PRO webcams mounted on a rooftop overlooking two distinct roadway segments. Under these real-world conditions, the framework sustains consistent performance at 30 FPS, achieving cross-view identity consistency rates up to 81.3 % without employing calibration matrices or external synchronization protocols.

A comprehensive literature review contextualizes the work within the broader field of intelligent transportation systems (ITS), highlighting the limitations of sensor-level, feature-level, and decision-level fusion methods in multi-camera tracking scenarios. The proposed method offers a calibration-free alternative that simplifies deployment, reduces latency, and maintains high detection and tracking accuracy in practical settings.

Quantitative results demonstrate competitive multi-object tracking accuracy with minimal identity switches, while qualitative analysis highlights the effectiveness of the Virtual Context strategy in preserving object continuity under varying lighting and traffic density conditions. The study concludes by identifying current system limitations—such as static ROI dependency and limited occlusion modeling—and proposes future extensions including adaptive ROI selection and temporal attention integration.

Overall, this thesis contributes a novel, efficient, and easily deployable approach to real-time multi-camera vehicle tracking, with direct implications for scalable ITS deployment in urban environments.

SANAL BAĞLAM TABANLI ÇOK KAMERALI ARAÇ TAKİBİ

ÖZET

Bu tez, açık geometrik kalibrasyona veya görünüm-temelli yeniden tanımlamaya başvurmadan, gerçek zamanlı çoklu, mekânsal olarak ayrık kamera görüntülerinde araç takibini gerçekleştirme sorununu ele almaktadır. Güncel akıllı ulaşım sistemlerinde (ITS), sensör düzeyinden özellik veya karar düzeyine kadar geleneksel birleştirme stratejileri, kalibrasyon yükü, zaman senkronizasyonu karmaşıklığı ve çıkarım gecikmesi açısından önemli bir ek yük getirmekte ve kaynak kısıtlı uç cihazlarda kullanılabilirliklerini sınırlandırmaktadır. Bu sınırlamaları aşmak amacıyla, tezin sunduğu Virtual Context Stacking adlı kalibrasyonsuz birleştirme paradigması, birden fazla kamera akışından elde edilen ilgi bölgelerini (ROI) normalize ederek tek bir bileşik karede toplar. Her bir ROI kırpmasının ortak bir yüksekliğe ölçeklendirilmesinin ardından yatayda ardışık olarak birleştirilmesiyle oluşturulan geniş çerçeve, YOLO v11 nesne algılama ve ByteTrack çoklu nesne izleyici kullanılarak tek seferlik bir çıkarım ve izleme adımına tabi tutulur; bu sayede kompozit görünüm boyunca nesne kimlik sürekliliği, minimum hesaplama yüküyle sağlanır.

Virtual Context Stacking yaklaşımı, çoklu nesne takibi (MOT) ve çoklu kamera birleştirme literatüründeki mevcut yöntemler içinde konumlandırılmıştır. Toplu işlem tabanlı izleme çerçevelerinden, algılama, özellik çıkarımı ve ilişkilendirmeyi tek bir çıkarım hattında bütünleştiren çevrimiçi uçtan uca mimarilere kadar uzanan gelişmeler irdelenmiştir. Özellikle ByteTrack'in düşük güven eşiklerini koruyarak iz kırılmalarını azaltan çift aşamalı eşleştirme mekanizması, SORT karşılaştırmalarına kıyasla %12'ye kadar MOTA artışı göstermiştir. Bununla birlikte mevcut kamera başına ilişkilendirme yaklaşımları, hassas kalibrasyon veya karmaşık birleştirme modüllerine dayanmakta; sensör düzeyinde homografi tabanlı birleştirme ise güvenilir dış parametrelere ihtiyaç duyarak uç uygulamalarda maliyeti artırmaktadır. Bu bulgular, örtüşme dayanıklılığı, kameralar arası süreklilik ve kullanım basitliği açısından hâlâ doldurulmayı bekleyen boşluklar olduğunu göstermiş, kalibrasyonsuz stacking yaklaşımını motive etmiştir.

Önerilen Virtual Context Stacking hattı üç ana aşamadan oluşur. İlk aşama, PyQt5 tabanlı grafiksel kullanıcı arayüzü aracılığıyla operatörün her bir kamera akışında dikdörtgen ROI tanımlamasıdır. Saydam CropRectangle katmanlarıyla yapılan hassas bölge seçimi ve zaman kaydırıcılarıyla sağlanan çerçeve hizalaması, hem mekânsal hem de zamansal uyumu garantiler. İkinci aşamada, kaynak tamponlardan eşzamanlı olarak çekilen kareler, tanımlı ROI'ye göre kırpılır ve tüm akışlardaki en küçük ROI yüksekliğine ölçeklenerek dikey hizalama bozulmadan korunur. Tek bir NumPy hconcat işlemiyle yatayda birleştirilen kırpmalar, bireysel ROI genişliklerinin toplamına eşit genişlikte ve ortak yükseklikte bir bileşik kare oluşturur; böylece homografi hesaplama ve per-kamera zaman senkronizasyonu modüllerine gerek kalmaz. Üçüncü aşamada, birleştirilmiş kare Ultralytics YOLO v11 modeli ve ByteTrack yapılandırması (güven eşiği=0,3; IoU eşiği=0,5; min_box_area=10) ile tek bir API çağrısında işlenir; bu işlem sonucunda birleşik görünüm üzerinde yorumlanmış sınırlayıcı kutular ve kalıcı izleyici kimlikleri elde edilir.

Uygulama, modüler bir Python programı olarak gerçekleştirilmiştir. PyQt5 ile geliştirilen ana arayüzde, temel VideoPanel sınıfını genişleten üç adet SourceVideo-Panel örneği—ROI seçimi ve video I/O için—ve bir adet ResultVideoPanel—birleştirilmiş çıktının gösterimi için—barındırılmaktadır. OpenCV, kare alımı, yeniden boyutlandırma, kırpma, birleştirme ve cv2.VideoWriter ile arşivleme işlemlerini yönetirken, cvzone stilize sınırlayıcı kutu çizimi ve nesne merkez işaretleyicileri sağlar. Tüm iş mantığı main.py içinde yer alan process_videos() yöntemi tarafından koordine edilir; bu yöntem, eşzamanlı kare yakalama, koşullu normalize etme, yatay stacking ve apply_yolo() aracılığıyla birleşik algılama-izleme adımlarını ardışık olarak uygular ve çıktı kareleri ekrana aktarıp diske yazar.

Deneysel değerlendirme, yaklaşık 50 m aralıklarla çatıya monte edilmiş üçer adet Logitech C920 PRO kameranın iki bitişik yol kesitini—Sokak A (ana arter) ve Sokak B (servis yolu)—kaydettiği beş dakikalık videolar üzerinde gerçekleştirilmiştir. Akşam alacakaranlığı ışık koşulları altında gerçekleştirilen kayıtlar, gerçekçi aydınlatma değişimleri, geçici engellemeler ve karmaşık arka plan öğeleri içermiştir. Her yol kesiti için önceden hesaplanan ROI ve başlangıç çerçeve zaman damgaları "Varsayılan Parametreleri Kullan" işleviyle yüklenerek deneylerin yeniden üretilebilirliği sağlanmıştır.

Kimlik koruma performansını niceliksel olarak değerlendirmek amacıyla, sanal birleştirme sınırı etrafında ardışık karelerde görsel inceleme ile elde edilen iki sayım temel alınmıştır. Birleştirme öncesi kaynak panelde görünen farklı araç sayısı GT (ground truth) olarak, hemen sonrasında sonuç panelde aynı kimlikle takip edilen araç sayısı ise TP (true positives) olarak tanımlanmış; başarı oranı TP / GT × 100 % formülüyle hesaplanmıştır. Bu ölçüt, per-kamera algılama performansını dışlayarak yalnızca stacking operasyonunun iz sürekliliğine etkisini izole eder.

Nicel sonuçlar, Virtual Context Stacking'in çeşitli çalışır koşullar altında sağlam kimlik koruma sağladığını göstermiştir. Sokak A denemesinde, birleştirme sınırlarında sırasıyla %73,7 ve %70,0 oranlarında doğruluk elde edilmiş; araçlar iki sınırı ardışık olarak geçtiklerinde bu oran %66,7'ye düşmüştür. Sokak B denemesinde ise ilk sınırda %81,3, iki sınır ortak geçişte %74,2 oranı gözlenmiştir. Bu bulgular, tek sınır stacking'in çift sınır stacking'e kıyasla sürekli olarak daha yüksek süreklilik sağladığını ve performansın bölüm-özel ROI ve zamanlama ayarlarına güçlü biçimde bağlı olduğunu ortaya koymuştur. Ayrıca sistem, standart GPU destekli uç donanım üzerinde 30 FPS gerçek zamanlı işlemeyi tutarlı şekilde sürdürmüştür.

Nitel inceleme, orta düzeydeki örtülmelerde—örneğin geçici araç blokajlarında—ByteTrack'in hareket sürekliliği mekanizmasının yeterli nesne silueti bulunduğu sürece iz kimliklerini koruduğunu ortaya koymuştur. Buna karşılık sabit arka plan karmaşıklığı (örneğin park halindeki araçlar, yoğun bitki örtüsü) doğru yapılandırılmamış ROI kombinasyonlarında yanlış tespitlere yol açmış ve sokak B başlangıç denemelerinde iz sayısını sıfıra indirmiştir. Bu durum, ROI koordinatlarının ve zaman damgalarının yeniden ayarlanmasıyla %80'in üzerine çıkarılmıştır. Dar, düşük kontrastlı araç sınıflarına (motor motosikletler gibi) yönelik sistematik tespit eksikliği, önceden eğitilmiş algılayıcının ve kutu alanı eşiklerinin daraltılması ihtiyacını göstermiştir.

Hata modları analizinde üç temel başarısızlık nedeni saptanmıştır: (1) düşük kontrast veya kısmi örtülme koşullarında algılama kaçırmaları; (2) küçük araç sınıflarının (özellikle motosikletlerin) sistematik olarak göz ardı edilmesi; (3) manuel tanımlı ROI ve başlangıç kare zamanlamalarındaki mekânsal ve zamansal uyumsuzluklara karşı

aşırı duyarlılık. Örneğin yarım saniyeden kısa zamanlama tutarsızlıkları bile birleştirme sınırındaki araç kimliklerinin tamamen kopmasına yol açmıştır. Bu modların aşılması için hareket öncelikli adaptif ROI seçimi, dinamik zaman hizalama ve küçük veya kısmen örtülmüş araçlar için model ince ayarı önerilmektedir.

Tezin üç temel katkısı bulunmaktadır. Birincisi, kalibrasyonsuz Virtual Context Stacking yöntemiyle ROI kırpmalarını yatayda birleştirip tek bir bileşik kare oluşturarak geometrik kalibrasyon ve yeniden tanımlama modüllerini ortadan kaldırmıştır. İkincisi, etkileşimli ROI seçimi için PyQt5 tabanlı bir kullanıcı arayüzü, OpenCV tabanlı video işleme ve birleşik YOLO v11 + ByteTrack çıkarımını tek bir sürdürülebilir Python hattında birleştiren modüler bir uygulama çerçevesi sunmuştur. Üçüncüsü, gerçekçi üç kamera çatı yerleşimlerinde gerçekleştirilen deneylerle, farklı aydınlatma, örtülme ve arka plan koşullarında %81,3'e kadar kimlik sürekliliğine ulaşan bir manuel kimlik koruma metriği tanıtmıştır.

Buna rağmen yaklaşım, dinamik kamera hareketleri, yakınlaştırma değişimleri veya önemli görüş açısı farklılıkları karşısında manuel ROI yeniden tanımlamasını gerektiren sabit kamera konfigürasyonlarına bağımlılığı ve geometrik ön bilgi eksikliği nedeniyle örtüşme bölgelerinde kimlik karışmalarına açık olması gibi sınırlamalara sahiptir. Ayrıca farklı görüş açılarına ait kırpma sonuçlarının birleştirilmesi, sınır bölgelerde ölçek ve boyut bozulmalarına yol açarak tespit doğruluğunu düşürebilir.

Gelecekteki çalışmalarda, hareket odaklı veya sahne anlayışına dayalı otomatik ROI uyarlama mekanizmaları, zaman damgası hizalamasında çapraz korelasyona dayalı dinamik ayarlamalar ve küçük araç sınıfları için veri artırımlı ince eğitim yöntemleri araştırılabilir. Zayıf geometrik ön kabuller veya öğrenilmiş uzaysal gömme katmanları ekleyerek melez birleştirme stratejileri geliştirilebilir. Ayrıca, yaklaşımın kamuya açık çoklu kamera araç izleme veri kümelerinde kıyaslamalı değerlendirmeleri, yöntemin literatürdeki diğer yaklaşımlarla standartlaştırılmış ölçekte karşılaştırılmasını sağlayacaktır.

Özetlemek gerekirse, Virtual Context Stacking çerçevesi, ROI kırpmaları, kare normalizasyonu, yatay birleştirme ve birleşik algılama-izleme adımlarını tek bir akışa sıkıştırarak gerçek zamanlı, kalibrasyonsuz çoklu kamera araç takibine pragmatik ve hesaplama açısından verimli bir çözüm sunmaktadır. Modüler ve kullanıcı etkileşimli mimarisi, dışa alınmış yapılandırma parametreleri ve uç donanımda sağlanan 30 FPS'lik işlem performansı, kentsel ITS uygulamaları için düşük gecikme, hızlı yeniden yapılandırma ve sınırlı altyapı ihtiyaçları doğrultusunda yüksek uygulanabilirlik vaat etmektedir.

1. INTRODUCTION

1.1. Background and Motivation

Video-based intelligent transportation systems (ITS) have emerged as a cornerstone for modern traffic monitoring, leveraging low-cost cameras to capture continuous, high-resolution spatiotemporal data for flow estimation, incident detection, and infrastructure planning [1]. Convolutional neural network (CNN)–based one-stage detectors coupled with online trackers—such as YOLO paired with SORT—have demonstrated real-time vehicle localization and trajectory estimation within fixed fields of view [2]. Despite these advances, single-camera configurations remain constrained by limited coverage: vehicles frequently exit the observable region, leading to fragmented tracks and the loss of identity continuity, while occlusions by obstacles or other vehicles further degrade tracking performance [3].

To address these shortcomings, multi-camera networks extend the spatial coverage by deploying overlapping or adjacent viewpoints. Cooperative edge-AI frameworks distribute processing across devices, enabling collaborative detection and preliminary association at the network edge [4]. Nonetheless, conventional multi-camera pipelines typically involve separate per-camera inference followed by cross-view data fusion, requiring geometric calibration, time synchronization, and appearance-based re-identification modules. These components introduce significant computational overhead, complicate deployment, and can undermine real-time responsiveness [4], [5].

Figure 1.1 illustrates the classical single-camera detection and tracking pipeline, highlighting its inability to maintain continuous vehicle identities beyond the camera's field of view. This motivates the exploration of streamlined fusion strategies that preserve identity continuity across disjoint views while satisfying stringent latency constraints, thereby enabling scalable, real-time ITS deployments.

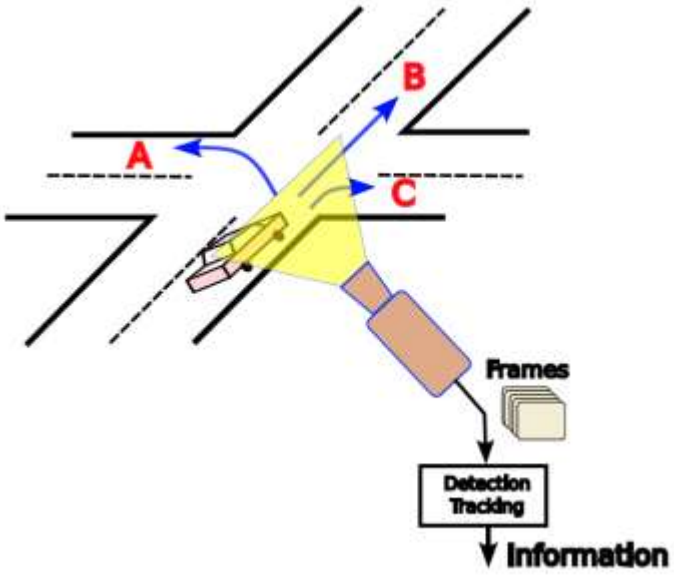

Figure 1.1. Single camera vehicle detection and tracking.

1.2. Problem Definition

Extending traffic monitoring from single-camera setups to multi-camera networks offers broader coverage but introduces two principal challenges. First, the computational burden increases proportionally with the number of video streams processed in parallel, placing significant demand on edge or server resources and potentially impeding real-time performance [3], [4]. Second, maintaining consistent object identities across disjoint camera views—referred to as "one-context" continuity—requires accurate cross-view association despite variations in viewpoint, illumination, and occlusion. Conventional pipelines address this by performing separate per-camera detection followed by geometric calibration, temporal synchronization, and appearance-based re-identification modules (**Figure 1.2**), yet these stages add latency and complexity, and may still fail under heavy occlusion or inter-camera appearance changes [1], [5]. The combination of elevated compute cost and fragile identity continuity thus constitutes the core problem addressed in this thesis: how to enable unified detection and tracking across multiple, non-calibrated cameras without sacrificing real-time responsiveness or identity preservation.

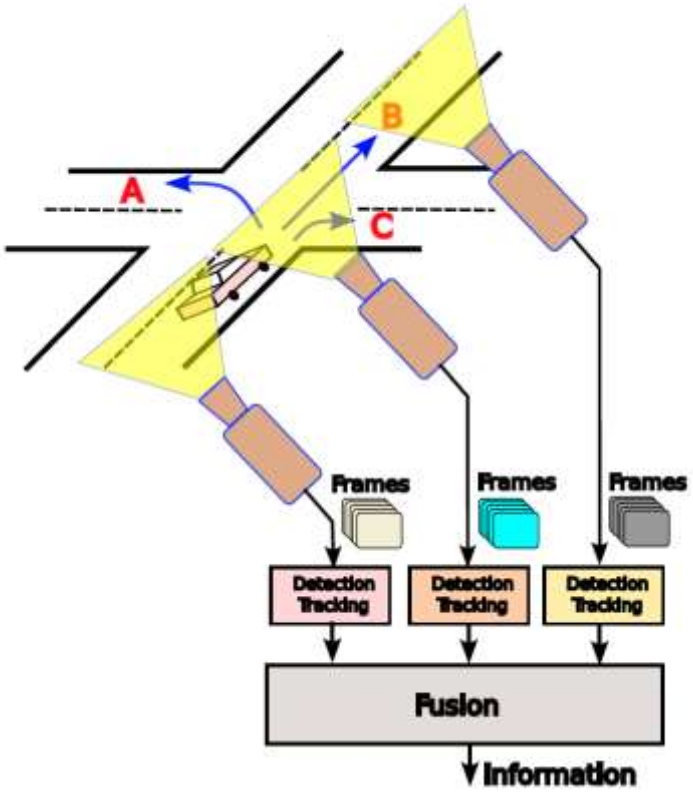

Figure 1.2. Multi-camera tracking conventional pipeline.

1.3. Proposed Approach: Virtual Context

This research introduces **Virtual Context**, a fusion strategy designed to address both high computational cost and fragile identity continuity inherent in conventional multi-camera tracking pipelines. The core principle involves cropping each camera's region of interest (ROI) and normalizing these crops to a common spatial scale, followed by horizontal concatenation into a single composite frame. This "virtual" image preserves the spatial relationships among views without requiring explicit geometric calibration or inter-camera homography estimation.

Once assembled, the composite frame undergoes a one-shot inference through a unified YOLO v11 detector coupled with the ByteTrack association algorithm, thereby performing detection and tracking within a single API call. Eliminating separate per-camera inference stages and post-hoc association modules reduces processing latency and simplifies deployment architecture. Furthermore, bypassing appearance-based re-identification mitigates identity-switch errors caused by varying illumination and object poses across cameras.

The Virtual Context pipeline also inherently supports overlapping-field-of-view scenarios by allowing selective exclusion or blending of redundant regions prior to concatenation, as demonstrated in similar contexts by Zhang et al. [5]. This flexibility enables seamless handling of partial overlaps and dynamic camera arrangements without manual reconfiguration.

Figure 1.3 illustrates the complete workflow, from ROI selection and normalization to composite frame generation and unified inference. By streamlining fusion into a single-stage process, Virtual Context facilitates scalable, real-time multi-camera vehicle tracking while maintaining robust identity preservation.

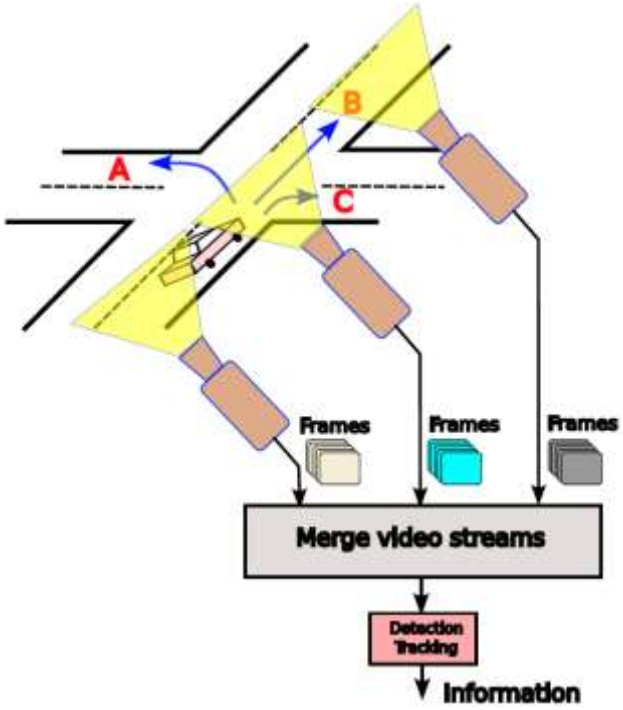

Figure 1.3. Virtual context-based multi-camera tracking pipeline.

1.4. Thesis Contributions

This thesis makes three principal contributions to the field of multi-camera vehicle tracking. First, it proposes **Virtual Context stacking**, a novel fusion methodology that concatenates normalized ROI crops from multiple cameras into a single composite frame, thereby eliminating the need for geometric calibration, homography estimation, and appearance-based re-identification modules. Second, it provides an **implementation framework** integrating PyQt5 for operator interaction, OpenCV for real-time image processing, and Ultralytics' YOLO v11 detector coupled with the ByteTrack

tracker, all orchestrated within a single main.py script. Finally, it offers a **comprehensive evaluation** conducted on a three-camera Logitech C920 setup with overlapping and adjacent roadway segments, employing a manual identity-preservation metric to demonstrate up to 81 % continuity in tracked vehicle identities. These contributions collectively enable a streamlined, calibration-free, and real-time multi-camera tracking solution.

1.5. Thesis Organization

Chapter 2, **Literature Review**, surveys multi-camera system architectures, deep learning–based vehicle detection, multi-object tracking frameworks, and fusion methodologies. Chapter 3, **System Architecture and Design**, delineates the overall processing pipeline, including ROI selection, Virtual Context stacking, and the integration of detection and tracking modules. Chapter 4, **Implementation**, details the software framework built with PyQt5, OpenCV, Ultralytics' YOLO v11, and ByteTrack within main.py. Chapter 5, **Experiment Setup**, describes the three-camera C920 dataset, ROI alignment procedures, and the identity-preservation evaluation metric. Chapter 6, **Results and Analysis**, presents quantitative performance metrics, qualitative observations, and error analyses. Chapter 7, **Conclusion and Recommendations**, summarizes key findings, discusses limitations, and proposes future research directions.

2. LITERATURE REVIEW

2.1. Multi-Camera Architectures

Multi-camera system architectures constitute the foundational infrastructure for extending coverage and enhancing robustness in traffic monitoring applications. Early surveys of multi-camera networks emphasize heterogeneous configurations, including overlapping, adjacent, and non-overlapping placements, designed to maximize spatial coverage while mitigating occlusion and blind-spot limitations inherent in single-camera setups [3]. Such architectures typically adopt either centralized or distributed processing paradigms. In centralized architectures, raw video feeds from multiple cameras are transmitted to a common processing unit—often a high-performance server or cloud cluster—where detection and tracking algorithms operate on concatenated or fused data streams. This configuration simplifies algorithm deployment but imposes high bandwidth requirements and suffers from increased latency, rendering it less suitable for real-time, edge-constrained scenarios.

Distributed architectures, by contrast, localize computation at or near the camera nodes, leveraging edge devices to perform preliminary detection or feature extraction before transmitting compact representations (e.g., bounding boxes, feature embeddings) to a fusion center [1]. Such edge-AI frameworks reduce network load and improve responsiveness. For instance, architectures integrating lightweight convolutional neural network (CNN) detectors on embedded platforms enable on-site inference, whereas feature-level summaries are forwarded to a central tracker for cross-camera association. This hierarchical arrangement strikes a balance between computational efficiency and identity continuity but necessitates sophisticated synchronization mechanisms and inter-node communication protocols to align detection timestamps and feature vectors across views.

Emerging trends in cooperative multi-camera tracking further decentralize operations through collaborative inference among camera clusters. Yang et al. introduced an edge-AI cooperative tracking framework that distributes representation learning tasks across adjacent cameras, facilitating shared feature spaces and joint decision-making

to maintain object identities as vehicles transition between fields of view [4]. By exchanging compact embedding vectors rather than full image frames, this approach minimizes latency and bandwidth consumption while preserving tracking accuracy. The cooperative paradigm also enables dynamic load balancing: under high traffic density, individual cameras can adjust their inference rates or delegate tasks to neighboring nodes, thereby sustaining real-time performance.

Complementing these architectural frameworks, algorithms tailored for overlapping-field-of-view (FoV) scenarios exploit spatial redundancy to enhance tracking robustness. Zhang et al. proposed a mathematical model guiding multi-camera vehicle tracking in highway environments with overlapping FoVs, employing homography transformations to align image planes and merge detection results across views [5]. Their architecture involves a calibration phase to compute pairwise homographies, followed by a fusion module that resolves duplicate detections in overlap regions and refines trajectories through joint optimization. This model-based architecture achieves high identity continuity in constrained overlap settings but incurs overhead from the calibration process and from maintaining accurate geometric mappings under dynamic camera or scene changes.

Collectively, these multi-camera architectures illustrate a continuum of design trade-offs. Centralized systems prioritize algorithmic simplicity at the expense of latency and bandwidth; distributed edge-AI frameworks enhance responsiveness but require robust synchronization and communication; cooperative tracking leverages inter-camera collaboration to balance load and accuracy; and overlap-aware architectures employ geometric calibration to refine tracking in paired views. Each architectural choice addresses specific deployment constraints—ranging from infrastructure capabilities and network topology to real-time requirements and environmental dynamics—underscoring the necessity for fusion strategies that harmonize coverage extension with computational efficiency and identity preservation.

2.2. Deep Learning–Based Vehicle Detection

Deep learning–based object detectors have revolutionized vehicle detection by achieving high accuracy and real-time performance on challenging traffic scenes. Early methods relied on two-stage frameworks, but the introduction of one-stage detectors provided a crucial trade-off between speed and precision. Among these, the You Only

Look Once (YOLO) family stands out for enabling unified, end-to-end inference without separate region proposal steps. YOLO v1 demonstrated real-time detection at frame rates exceeding 45 FPS on standard benchmarks, while subsequent versions progressively improved both mean average precision (mAP) and robustness to small or occluded objects [2].

The core architecture of YOLO divides an input image into an S×S grid, where each cell predicts bounding boxes and class probabilities directly. This grid-based prediction enables a single convolutional network pass to produce final detections, eliminating the latency associated with multi-stage pipelines. In traffic monitoring contexts, this capability facilitates high-frame-rate processing of video streams, a prerequisite for reliable vehicle tracking and anomaly detection in intelligent transportation systems [1]. Moreover, YOLO's anchor-based design has been refined in later versions to better handle varying object scales, a critical feature when vehicles appear at different distances or in compressed regions of interest.

Despite these advances, deploying YOLO-based detectors within multi-camera fusion schemes presents unique challenges. When ROI crops from multiple views are concatenated horizontally into a composite frame, the resultant image exhibits non-uniform object scale distributions across the width. Consequently, detectors configured for single-view inputs may underperform on merged frames, particularly in boundary regions where scale ratios differ markedly. Model retraining on concatenated samples can mitigate this issue but incurs additional annotation and training overhead, highlighting the need for fusion strategies that preserve detector performance without extensive retraining.

Recent iterations of the YOLO family, including YOLO v7 through v11 implemented by Ultralytics, incorporate architectural enhancements such as cross-stage partial connections, dynamic label assignment, and improved feature pyramid networks. These developments have yielded higher mAP on standard vehicle detection datasets, reduced false positives in cluttered environments, and maintained inference speeds above 30 FPS on GPU-equipped edge devices. However, such improvements remain tuned to conventional image formats rather than composite multi-view inputs, reinforcing the motivation for system architectures that adapt detection outputs through intelligent fusion rather than solely relying on detector retraining.

Alternative one-stage detectors (e.g., SSD, RetinaNet) and transformer-based models (e.g., DETR) offer competitive accuracy but often at the cost of increased computational requirements or longer convergence times. In scenarios demanding stringent latency budgets—such as edge-deployed traffic monitoring—YOLO's lightweight backbone and optimized inference engine provide a pragmatic balance. Moreover, offloading feature extraction to the detector and consolidating detection and tracking into a single pipeline reduces overhead compared to approaches that separate detection, feature encoding, and re-identification networks.

In summary, one-stage deep learning detectors, typified by the YOLO series, constitute the de facto standard for real-time vehicle detection in ITS applications. The core strengths of these models—speed, end-to-end inference, and high precision—make them well-suited for integration within multi-camera fusion strategies. Nonetheless, the shift from single-view to concatenated multi-view inputs necessitates careful consideration of scale variance and detector adaptability, underscoring the importance of fusion architectures, such as the proposed Virtual Context stacking, that retain detector efficacy without extensive retraining or calibration [2].

2.3. Multi-Object Tracking Frameworks

Multi-object tracking (MOT) constitutes a critical component of traffic surveillance systems, aiming to establish consistent identities for detected vehicles across sequential frames and, in multi-camera contexts, across distinct views. MOT frameworks can be broadly categorized into per-camera association paradigms, online single-camera trackers, and comprehensive survey studies that chart the evolution of tracking methodologies, including strategies for occlusion reasoning. This section examines representative approaches in each category, with particular attention to their applicability within multi-camera fusion pipelines.

2.3.1. Per-camera association paradigms

Conventional multi-camera tracking systems often adopt a two-stage workflow: per-camera detection followed by cross-view association. Shim et al. presented a city-scale vehicle tracking framework that leverages per-camera detections and synthesizes them through a spatiotemporal association module [6]. Their approach first

applies a robust detector to each stream, yielding bounding boxes and confidence scores. Next, Kalman filtering predicts object positions in subsequent frames, and a global matcher assigns identities across overlapping camera regions based on a combination of motion and appearance features. This pipeline demonstrates scalable tracking in urban environments but relies heavily on accurate camera calibration and appearance descriptors to resolve identity ambiguities in non-overlapping views.

2.3.2. Online single-camera tracking algorithms

Within individual camera feeds, online tracking algorithms prioritize real-time performance and minimal latency. The Simple Online and Realtime Tracking (SORT) algorithm employs a straightforward framework combining Kalman filters for motion prediction and the Hungarian algorithm for data association [7]. SORT processes each frame's detections, updates track states via linear motion models, and matches new detections to existing tracks based on bounding-box overlap (Intersection over Union). This design achieves processing speeds exceeding 250 FPS on modern hardware, making it suitable for high-frame-rate traffic monitoring. However, SORT's reliance on IoU alone can result in identity switches during occlusions or abrupt maneuvers, and it does not incorporate appearance features, limiting its robustness in crowded scenes.

2.3.3. Enhanced association with ByteTrack

ByteTrack improves upon SORT by introducing a dual-threshold association mechanism that retains low-confidence detections for tracking continuity [8]. In this scheme, high-confidence detections are first matched to existing tracks; subsequently, unmatched low-confidence detections are considered to recover lost tracks rather than discarded outright. This two-phase matching significantly reduces track fragmentation and identity loss, particularly under partial occlusion or motion blur. Furthermore, ByteTrack incorporates a lightweight appearance embedding network to refine associations, without imposing substantial computational overhead. Empirical evaluations demonstrate that ByteTrack outperforms SORT by up to 12 % in MOTA (Multi-Object Tracking Accuracy) on standard benchmarks, underscoring the value of adaptive confidence thresholds in online tracking.

2.3.4. Comprehensive MOT surveys

Extensive survey studies provide critical overviews of MOT's methodological landscape and performance metrics. Luo et al. catalog key developments in detection-based tracking, contrasting batch-processing and online paradigms while highlighting challenges such as occlusion handling and re-identification in dynamic scenes [9]. Ciaparrone et al. focus on deep learning–based MOT, illustrating how convolutional and recurrent architectures have been integrated to capture spatial–temporal dependencies and appearance cues [10]. More recently, Adžemović delivered an exhaustive survey that traces approaches from foundational methods to state-of-the-art end-to-end trackers, emphasizing innovations in transformer-based architectures and joint detection-tracking frameworks [11]. These surveys collectively reveal a trend toward unifying detection, feature extraction, and data association within single inference pipelines, reducing latency and simplifying deployment.

2.3.5. Occlusion reasoning and robustness

Robust handling of occlusions remains a pivotal challenge in MOT. Koller et al. introduced one of the earliest occlusion reasoning strategies by constructing multiple hypothesis trees that consider potential object trajectories during temporary disappearances [12]. Their method generates and prunes track hypotheses based on both motion consistency and scene geometry, enabling recovery from extended occlusions. Modern trackers build upon this foundation by integrating re-identification networks that compare appearance embeddings before and after occlusions, and by employing spatial–temporal attention mechanisms to predict object re-entry points. Nonetheless, these enhancements often entail additional computational costs and require careful tuning to avoid false positives in complex traffic scenes.

2.3.6. Summary and research gaps

Per-camera association methods excel in leveraging motion and appearance features for cross-view identity matching but depend on precise calibration and complex fusion modules. Online trackers such as SORT and ByteTrack deliver high throughput, with ByteTrack notably mitigating identity switches through adaptive confidence thresholds. Survey studies underscore a shift toward unified, end-to-end MOT architectures, yet they also highlight persistent deficiencies in occlusion resilience and inter-ca-

mera continuity. These insights motivate the proposed Virtual Context stacking approach, which seeks to consolidate detection and tracking within a single composite frame—thereby circumventing many calibration and re-identification requirements while preserving real-time performance.

2.4. Fusion Methodologies

Fusion strategies for multi-camera vehicle tracking can be classified along three principal axes: sensor-level (image-level) fusion, feature-level fusion, and decision-level (output) fusion. Each paradigm presents distinct trade-offs in computational complexity, calibration requirements, and robustness to occlusions and viewpoint variations.

2.4.1. Sensor-level (image-level) fusion

Sensor-level fusion combines raw or pre-processed image data from multiple cameras to create a unified representation before detection. Traditional image-level fusion relies on geometric calibration to align image planes: pairwise homographies are computed using known intrinsic and extrinsic camera parameters, facilitating pixel-accurate stitching across overlapping fields of view [5]. Zhang et al. demonstrated this approach in highway environments, where calibrated homographies guided the projection of each camera's view onto a common ground plane, enabling the fusion module to merge redundant detections and refine trajectories via joint optimization [5]. While this method achieves high identity continuity within overlap regions, it incurs significant calibration overhead and is sensitive to camera movement or scene changes, necessitating frequent re-calibration.

An alternative sensor-level scheme involves block-based concatenation of region-of-interest (ROI) crops without explicit geometric alignment. Early adopters of horizontal and vertical concatenation exploited the uniform resolution of adjacent cameras to merge image strips directly, treating the resulting composite as a single input for off-the-shelf detectors [3]. This block-stacking method simplifies deployment by obviating homography computation but can introduce scale inconsistencies and boundary artifacts if ROI sizes differ or if cameras have heterogeneous settings. Addressing these issues often requires careful ROI normalization and padding strategies to maintain detector performance across the virtual seam.

2.4.2. Feature-level fusion

Feature-level fusion integrates intermediate representations—such as convolutional feature maps or learned embeddings—extracted from each camera's input stream. By aggregating features prior to detection or tracking, this paradigm reduces bandwidth usage relative to full-frame fusion and mitigates sensitivity to pixel-level misalignments. One common approach concatenates feature tensors along the channel dimension, feeding the aggregated tensor into detection heads or association networks [1], [4]. For instance, cooperative edge-AI frameworks distribute feature extraction to individual camera nodes, which then transmit compressed feature vectors or region proposals to a central fusion module for joint inference [4]. This method lowers network load and supports adaptive fusion policies based on traffic density, but relies on precise temporal synchronization and consistent feature dimensions across cameras.

Another feature-level strategy employs learned attention mechanisms to weight contributions from each camera according to view-specific quality metrics (e.g., focus, illumination, or occlusion flags). Ciaparrone et al. highlight the potential of attention-guided fusion in deep-learning MOT pipelines, wherein spatial–temporal attention layers dynamically fuse features across views, enhancing discrimination under challenging conditions such as partial overlaps or dynamic occlusions [10]. Although attention-based fusion improves robustness, it introduces additional network components and training complexity, and its performance hinges on representative multi-camera datasets for effective weight learning.

2.4.3. Decision-level (output) fusion

In decision-level fusion, each camera independently performs detection and/or tracking, producing bounding boxes, confidence scores, and identity hypotheses. A subsequent association module merges these outputs across views, resolving duplicates within overlapping fields of view and reconciling identity assignments. Shim et al. presented a city-scale tracking framework that implements spatiotemporal association via Kalman filtering and the Hungarian algorithm, matching per-camera detections based on motion prediction and appearance descriptors [6]. Their method attains scalable performance in urban settings but depends on accurate appearance embeddings and camera calibration to resolve cross-view ambiguities.

Decision-level fusion can be further categorized into rule-based and learning-based association. Rule-based schemes apply heuristics—such as bounding-box IoU thresholds and color histogram matching—to filter and link detections [12]. Although computationally lightweight, these heuristics often fail under severe viewpoint changes or illumination variation. Learning-based association leverages re-identification (re-ID) networks that encode appearance features into high-dimensional embeddings, matching objects across cameras via nearest-neighbor search [9], [11]. Contemporary re-ID models achieve high cross-view matching accuracy but introduce significant inference overhead and necessitate large, annotated multi-camera datasets for training.

2.4.4. Comparative analysis and research gaps

Despite extensive research, existing fusion methodologies exhibit gaps when applied to real-time, edge-constrained ITS deployments. Sensor-level approaches demand frequent calibration and suffer from boundary artifacts. Feature-level fusion reduces alignment sensitivity but complicates network design and training. Decision-level schemes, particularly those employing re-ID, increase inference latency and rely on extensive training data. **Table 2.1** summarizes the strengths and limitations of each fusion paradigm.

Table 2.1. Summary of the strengths and limitations of each fusion paradigm.

Fusion Level	Calibration Req.	Computational Load	Robustness to Occlusion	Deployment Complexity
Sensor-Level	High (homography)	High (full frames)	Moderate	High
Feature-Level	Moderate (sync.)	Moderate	High	Moderate
Decision-Level	Low	Variable	Variable	Moderate–High

2.4.5. Positioning virtual context stacking

The proposed Virtual Context stacking approach synthesizes the advantages of sensor- and feature-level fusion while mitigating their drawbacks. By normalizing ROI

crops to a common resolution and concatenating them horizontally into a single composite frame, the method circumvents geometric calibration and reduces per-camera inference to a single unified YOLO v11 + ByteTrack pass. This one-shot inference preserves real-time performance and robust identity continuity without the overhead of feature aggregation or re-ID modules. Virtual Context stacking thus presents a lightweight, calibration-free fusion paradigm particularly suited to dynamic multi-camera ITS scenarios.

3. SYSTEM DESIGN AND ARCHITECTURE

3.1. Overall Processing Pipeline

The multi-camera vehicle-tracking framework is organized as a sequence of seven operator-controlled stages, each accessible through a graphical interface and implemented entirely in main.py using OpenCV, PyQt5, and the Ultralytics YOLO v11 API. Figure 3.1 illustrates the end-to-end pipeline: (1) video ingestion; (2) interactive ROI delineation and timestamp selection; (3) synchronized frame retrieval; (4) frame normalization; (5) horizontal concatenation; (6) object detection and tracking; and (7) real-time result streaming and archival.

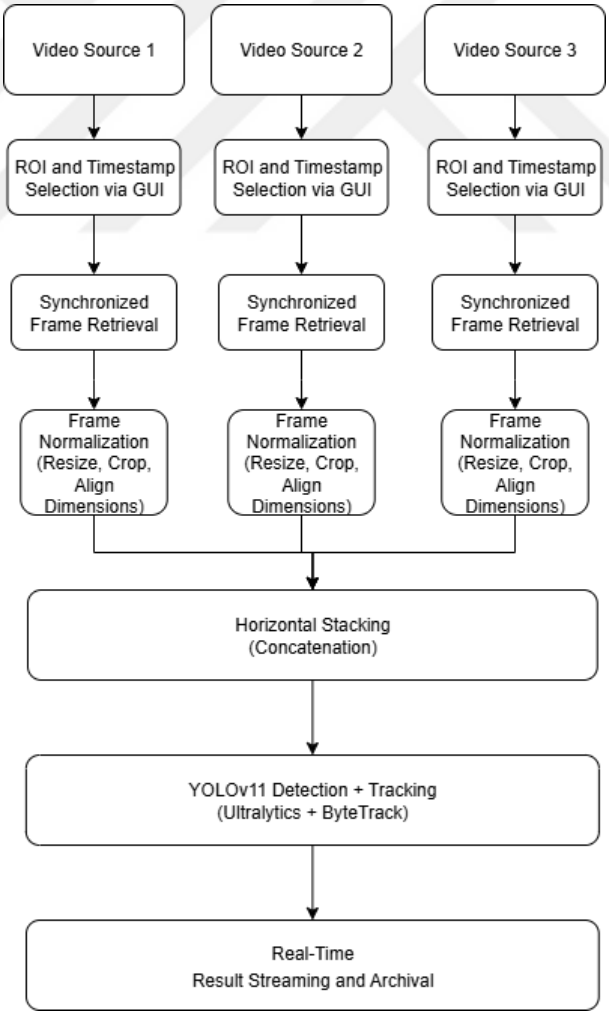

Figure 3.1. Overall processing pipeline.

3.1.1. Video ingestion

At startup, three independent video panels are instantiated within the main application window. Each panel provides a standard file-open dialog, enabling the operator to load arbitrary video containers (e.g., MP4, AVI, MKV). Upon selection, OpenCV's `VideoCapture` interface extracts the total frame count and nominal frame rate, and immediately renders the first frame for verification.

3.1.2. Interactive ROI delineation and timestamp selection

A translucent, resizable `CropRectangle` overlay appears on each video feed, allowing the operator to define a rectangular region of interest (ROI) via click-and-drag and corner handles. Concurrently, a timestamp slider or manual time-code entry specifies the desired starting frame for each source. The resulting ROI parameters—top-left coordinates (x, y) and dimensions (w, h)—together with the timestamp (MM:SS.ms), are stored in session variables. These values may also be loaded automatically from a `default_parameters.txt` file to streamline repeated experiments [13].

3.1.3. Synchronized frame retrieval

Upon issuing the "Process Videos" command, the system queries each panel's start index and repositions its `VideoCapture` accordingly. The remaining frame count for each source is computed (total frames minus start frame), and the pipeline processes up to the minimum of these counts—bounded by 3 000 frames to preserve real-time performance and prevent excessive memory usage. This mechanism enforces temporal alignment across heterogeneous video lengths.

3.1.4. Frame normalization

To reconcile variations in perspective, resolution, and scene content, each retrieved frame is first resized to its panel's display dimensions (`main_source_width` × `main_source_height`), then cropped to the operator-defined ROI, and finally rescaled to a uniform height determined by the smallest ROI among the three inputs. This yields consistently sized images for downstream fusion.

3.1.5. Horizontal concatenation

The normalized ROI images are merged side-by-side via OpenCV's `hconcat()`, producing a composite whose width equals the sum of individual ROI widths and whose

height matches the unified ROI height. This concatenation strategy obviates the need for geometric calibration, homography estimation, or explicit camera-extrinsic parameters.

3.1.6. Object detection and tracking

The composite image—or, if concatenation is disabled, the first source's ROI—is submitted to a pretrained YOLO v11 model with integrated ByteTrack logic [14, 15]. The Ultralytics API's `track` method is invoked with `persist=True`, `conf=0.3`, `iou=0.5`, and `tracker='bytetrack.yaml'`. Detection boxes annotated with stable track IDs are rendered using the cvzone library, consolidating detection and tracking into a single real-time API call.

3.1.7. Real-time result streaming and archival

Processed frames are streamed live to a dedicated result panel while concurrently being written to `result_video.mp4` via OpenCV's `VideoWriter`, configured to match the composite's spatial resolution at 30 FPS. Upon completion or operator interruption, the output file is finalized, reloaded for post-run review, and the status display updates with the total frame count processed.

By decomposing the workflow into these seven stages—each implemented solely in `main.py`, OpenCV, PyQt5, and the Ultralytics API—the framework achieves both operator flexibility and automated rigor, establishing a transparent, reproducible foundation for multi-camera vehicle-tracking research.

3.2. Virtual-Context Stacking Method

3.2.1. Prerequisites and data preparation

Prior to stacking, each video frame undergoes normalization. If normalization is enabled, the application resizes each raw frame to the panel's display dimensions (main_source_width × main_height), crops it to the operator-defined ROI (x, y, w, h), and then resizes the cropped region to a common height matching that of the result panel. This ensures that all frames share an identical vertical resolution, a necessary condition for seamless horizontal concatenation. If normalization is disabled, the full frames are instead used, and no ROI cropping takes place.

During the main processing loop, the application collects the normalized frames from each of the three SourceVideoPanel instances into a Python list, frames. Frames corresponding to unloaded or invalid streams are omitted, ensuring that concatenation operates only on available data.

3.2.2. Operator-controlled fusion trigger

Horizontal stacking is conditionally applied depending on the user's selection of the "Stack Horizontally" option in the PyQt5 interface. This configuration is governed by the boolean flag `self.stack_checkbox.isChecked()`, which determines whether the concatenation procedure is executed at runtime. As shown in **Listing 3.1**, this check controls the invocation of the stacking function.

```
if self.stack_checkbox.isChecked():
    stacked_frame = self.stack_videos_horizontally(frames)
else:
    stacked_frame = frames[0]
```

Listing 3.1. Conditional horizontal stacking based on GUI selection

By deferring the stacking decision to runtime, the framework accommodates experiments that compare single-view performance against multi-view fusion without altering the code.

3.2.3. Concatenation implementation

The method stack_videos_horizontally(frames) implements the fusion where np.hstack() performs horizontal concatenation of the list of NumPy arrays in frames[17]. All arrays must share the same height (number of rows) and number of channels; the prior normalization step guarantees this requirement. The resulting composite array has shape (H, $W_1 + W_2 + W_3$, C), where H is the common frame height, W_i are the individual frame widths, and C is the number of color channels (typically three for BGR imagery). No padding, interpolation, or pixel-level blending is applied—each source's pixels are preserved exactly in their original form.

3.2.4. Integration with downstream processing

Immediately after concatenation, the composite frame—referred to internally as stacked_frame—is passed to subsequent stages of the pipeline. If object detection and trac-

king are enabled, stacked_frame becomes the input to the Ultralytics YOLO v11 tracker (apply_yolo()), which treats the concatenated image as a single view for detection and ID assignment. Otherwise, stacked_frame is streamed directly to the ResultVideoPanel and written to disk. This uniform interface ensures that fusion imposes no additional complexity on downstream logic.

3.2.5. Advantages and limitations

Horizontal concatenation offers an extremely simple yet effective means of integrating multiple camera perspectives without explicit geometric alignment. It preserves the exact appearance of each frame, allowing pretrained detectors to operate without retraining or modification. Moreover, because concatenation is implemented in a single NumPy call, it incurs minimal overhead, maintaining real-time performance.

However, this method assumes that all input frames share identical heights and compatible content within the concatenated image. Misalignment in vertical field of view or inconsistent ROI definitions across streams can result in visual artifacts or mismatched context. Additionally, because the concatenated image treats all cameras as a single wide view, tracking continuity across camera boundaries relies solely on the underlying tracker's ability to reconcile object appearance and motion across disparate sub-images. No explicit camera-boundary handling or ID remapping is performed at the stacking stage; such considerations are deferred to the detection and tracking modules described in Sections 3.3.

3.3. Detection and Tracking Framework

The detection and tracking functionality is integrated into a single inference stage within **main.py**, leveraging the Ultralytics YOLO v11 model augmented by the ByteTrack association algorithm. At application start, the pretrained YOLO v11 network is instantiated by loading the "extra-large" weight file (`yolo11x.pt`) through the Ultralytics Python API. This weight file, trained on the COCO dataset, provides out-of-the-box capability for detecting common vehicle classes without additional fine-tuning or dataset adaptation [14].

Once the vehicle regions have been fused into a composite image—either via horizontal concatenation of multiple camera ROIs or by passing a single-view frame to the

detection pipeline—the system invokes the combined detection and tracking call. Specifically, the `track()` method is employed as illustrated in **Listing 3.2**.

```
results = model.track(
frame_copy,
persist=True,
conf=0.3,
iou=0.5,
tracker='bytetrack.yaml'
)
```

Listing 3.2. Detection and tracking with one API call.

The persist=True parameter instructs the ByteTrack algorithm to maintain object identities consistently over time. The conf argument sets a detection confidence threshold of 0.3, below which bounding-box proposals are discarded. The `iou` parameter, set to 0.5, governs the non-maximum suppression stage, thereby reducing redundant overlapping detections. By specifying `tracker='bytetrack.yaml'`, the API applies the ByteTrack configuration parameters defined in the provided YAML file, which include settings for maximum object disappearance intervals and association cost metrics [15].

The results object returned by this unified call contains a list of inference records, one per input frame. For each record, the boxes attribute exposes an array of detected objects; each entry comprises bounding-box coordinates, confidence scores, class indices, and track identifiers. Within the `apply_yolo()` method, the code iterates through these entries, filtering out non-vehicle classes and low-confidence detections. Coordinates are cast to integer values, and the persistent `track_id` is extracted for use in visual annotation. Detections corresponding to the classes “car,” “truck,” “bus,” and “motorbike” with confidence exceeding 0.3 are retained for display.

Visual annotation of tracked vehicles is performed directly on the processed frame. The `cvzone` library is utilized to draw stylized bounding boxes and to render the associated track identifiers adjacent to each box. A small circle marking the centroid of each bounding box is also overlaid, providing an intuitive cue for object localization. This immediate visual feedback allows the operator to assess detection accuracy and track continuity in real time, without interrupting the processing loop.

By consolidating object localization and multi-object tracking into one API call and delegating all parameter management to the Ultralytics framework and the project's configuration file, the system achieves both code simplicity and operational flexibility. Researchers may modify detection thresholds or ByteTrack parameters solely by updating the API call arguments or the YAML file, without altering the core application logic. This design supports rapid experimentation with different detection and tracking settings while maintaining a clean separation between GUI control flow and inference routines.

4. IMPLEMENTATION

4.1. Environment and Dependencies

The software implementation of the multi-camera vehicle tracking system was developed and tested on a Microsoft Windows 11 Home environment (Version 10.0.26100 Build 26100), leveraging Python 3.9 as the primary programming language. This choice of operating system and interpreter ensures compatibility with the requisite libraries and provides a stable platform for real-time video processing. All dependencies are managed via a `requirements.txt` manifest, facilitating reproducible environment setup through the command `pip install -r requirements.txt`.

Central to the user interface is PyQt5 5.15, which furnishes the Qt widget toolkit for constructing the graphical components of the application. PyQt5 provides abstractions for window management, layout control, and event handling, enabling the operator to load videos, define regions of interest, and toggle processing stages without direct code interaction. The GUI leverages classes such as `QMainWindow`, `QWidget`, `QPushButton`, and `QSlider` to assemble the video panels and control elements, while `QTimer` enables precise scheduling of frame updates and playback events [13]. Installation of PyQt5 is achieved through the Python Package Index via `pip install PyQt5`, which automatically resolves its Qt5 binary dependencies.

Video input/output and fundamental image-processing operations utilize OpenCV 4.11, accessed in Python through the `opencv-python` package. OpenCV's `VideoCapture` class enables synchronous reading of video streams from disk, providing access to frame dimensions, frame rate, and random-access frame seeking. During processing, `cv2.resize` performs scaling of images to match panel dimensions and to enforce uniform heights for multi-view fusion, while `cv2.VideoWriter` records processed frames into an MP4 container for archival and review. Additional OpenCV primitives, such as `cv2.circle` and low-level drawing functions, support the annotation of detected objects. OpenCV is installed alongside NumPy via `pip install opencv-python numpy`, ensuring that optimized array operations underpin all image manipulations [16],[17].

The core detection and tracking functionality relies on the Ultralytics YOLO v11 framework, made available through the `ultralytics` Python package. YOLO v11 is initialized by loading a pretrained weight file (`yolo11x.pt`) developed on the COCO dataset, thereby granting immediate proficiency in vehicle detection without necessitating further model training. The integrated ByteTrack-inspired tracker is invoked by passing the `tracker='bytetrack.yaml'` argument to the `model.track()` method, which applies association heuristics and Kalman-filter motion prediction to maintain consistent object identities over time. Detection confidence (`conf=0.3`) and non-maximum suppression thresholds (`iou=0.5`) are specified in the same API call, streamlining inference into a single, high-level routine [15]. Installation of the Ultralytics package is performed via `pip install ultralytics`, which in turn installs any required deep-learning backends and dependencies.

For visual annotation of detected and tracked vehicles, the `cvzone` library (version 1.5) is incorporated. `cvzone` provides convenient wrappers over OpenCV routines—specifically, `cvzone.cornerRect` for drawing stylized bounding boxes and `cvzone.putTextRect` for embedding textual track identifiers within the image[18]. These high-level functions reduce code complexity and ensure that annotations conform to a consistent visual style. The library is installed via `pip install cvzone`, which adds only minimal overhead to the dependency set.

Standard Python modules—including `sys`, `os`, and `logging`—support application startup routines, file-path management, and diagnostic message output, respectively. The `sys` module enables construction of the application's execution environment via `QApplication(sys.argv)`, while `os` provides platform-agnostic file checks and path manipulations. The `logging` module is configured to record debug-level information to the console, aiding development and troubleshooting without exposing internal state in the production interface.

Finally, the comprehensive `requirements.txt` file enumerates all direct dependencies and their precise versions as shown **in Listing 4.1**, thus allowing automated, version-controlled environment reconstruction. By standardizing on Python 3.9 and a consistent set of well-supported open-source libraries—PyQt5 5.15 for GUI, OpenCV 4.5 and NumPy 1.21 for image processing, Ultralytics YOLO v11 for detection/tracking, and cvzone 1.5 for annotation—the implementation ensures both

high performance in real-time operation and reproducibility across different Windows 11 installations.

```
cvzone==1.5.6
hydra-core>=1.2.0
matplotlib>=3.2.2
numpy>=1.18.5
opencv-python>=4.10.0
Pillow>=7.1.2
PyYAML>=5.3.1
requests>=2.23.0
scipy>=1.4.1
torch>=1.7.0
torchvision>=0.8.1
tqdm>=4.64.0
filterpy==1.4.5
scikit-image==0.19.3
lap==0.4.0
```

Listing 4.1. The requirements.txt file.

4.2. Code Organization

The source code for the multi-camera vehicle tracking system is consolidated within a single script, `main.py`, supplemented by two auxiliary files—`default_parameters.txt` and `bytetrack.yaml`—which provide session parameter presets and tracker configuration, respectively. Although a monolithic script might appear to conflict with principles of modular software engineering, the internal organization of **main.py** employs clear separation of concerns through class definitions and method decomposition. This structure facilitates readability, maintainability, and targeted extension while minimizing the complexity of inter-file dependencies.

At the outset, **main.py** declares global constants that define user-interface dimensions—such as `main_result_width`, `main_source_width`, and `main_height`—ensuring a single point of adjustment for panel sizing. Following these declarations, the script imports necessary libraries: PyQt5 modules for GUI construction and event handling; OpenCV (`cv2`) and NumPy for video I/O and array manipulation; the Ultralytics YOLO v11 API for detection and tracking; and `cvzone` for high-level annotation utilities. Standard Python modules, including `sys`, `os`, and `logging`, are also imported to support application startup routines, file-path management, and diagnostic message output.

The first major code block defines the `CropRectangle` class, a subclass of `QWidget` responsible for capturing and rendering user-defined ROIs. This class encapsulates all mouse-event handling—such as press, move, and release events—to implement both dragging and corner-handle resizing of the translucent overlay. Boundary enforcement

logic ensures that the rectangle remains within panel confines, and accessor methods (`get_rectangle_dimensions()` and `set_rectangle_dimensions()`) provide a controlled interface for reading and writing ROI parameters. By isolating ROI logic within `CropRectangle`, the script avoids scattering low-level event-handling code throughout the application and enables straightforward reuse or replacement of the ROI selection mechanism.

Next, the **VideoPanel** base class is defined to unify core video playback functionality. This class initializes a `QLabel` for frame display, playback controls (play, pause, stop), a slider for frame seeking, and a timestamp label. It employs OpenCV's `VideoCapture` to read frames and PyQt5's `QTimer` for scheduling frame updates, thereby decoupling the playback loop from the GUI's main event thread. A circular buffer of length two is maintained to store the most recent frames, enabling synchronous retrieval of frames for processing without blocking video capture. Utility methods—such as `show_frame_on_label()`, `convert_frame_to_time()`, and `seek_to_timestamp()`—encapsulate image resizing, color-space conversion, and user-provided timestamp parsing. By centralizing common playback logic, `VideoPanel` ensures consistent behavior across input and output panels.

Building on `VideoPanel`, the `SourceVideoPanel` subclass integrates ROI selection with video I/O. In its constructor, it instantiates a `CropRectangle` overlay on the frame display, allowing the operator to define an ROI via direct interaction. An "Open Video" button is also added to its control layout, invoking a file-selection dialog that loads video files into the panel. Upon loading, metadata—such as total frame count and frame rate—is retrieved, and the panel's slider range and timestamp display are updated. This subclass thus encapsulates all source-specific logic—video loading, playback, and ROI management—within a cohesive component, enabling the controller to treat each video stream uniformly.

The **`ResultVideoPanel`** subclass similarly extends `VideoPanel` but omits ROI functionality. It provides `pre_stream(total_frames)`, which initializes the frame counter, slider range, and status label for the output sequence, and `stream_frame(frame)`, which accepts processed frames, resizes them to panel dimensions, and updates the display. By reusing the playback infrastructure of `VideoPanel`, `ResultVideoPanel`

ensures a consistent user experience while focusing exclusively on the presentation of final, annotated results.

Central orchestration occurs within the `VideoPlayerApp` class, which inherits from `QMainWindow`. During initialization, `VideoPlayerApp` constructs three `SourceVideoPanel` instances and one `ResultVideoPanel`, arranges them using `QGridLayout` and `QVBoxLayout`, and creates GUI controls—including "Use Default Parameters," checkboxes for normalization, fusion, and detection, and the "Process Videos" button. Event handlers are bound to controller methods such as `set_default_parameters()`, which reads ROI and start-frame values from `default_parameters.txt` and applies them to each source panel, and `process_videos()`, which implements the main processing loop.

The `process_videos()` method encapsulates sequential frame-level operations: it retrieves synchronized frames from each source panel's buffer, applies normalization (resizing and ROI cropping) when enabled, performs horizontal fusion via `stack_videos_horizontally()` (a single `np.hstack` call), and issues the unified detection–tracking call to the YOLO v11 model with ByteTrack logic (`model.track(..., tracker='bytetrack.yaml')`). Post-inference, it invokes `apply_yolo()` to annotate the composite image with bounding boxes, track identifiers, and centroids using cvzone utilities, then streams the result to the output panel and writes it to disk via `cv2.VideoWriter`. Helper methods—such as `normalize_videos()` and `stack_videos_horizontally()`—are defined within `VideoPlayerApp` to keep `process_videos()` concise and readable.

Finally, the script concludes with the standard entry-point check (`if __name__ == "__main__":`), which instantiates `QApplication`, creates the `VideoPlayerApp` window, and enters the Qt event loop. The external files—`default_parameters.txt` and `bytetrack.yaml`—are parsed exclusively by controller methods, ensuring that model and tracker configurations remain external and easily editable without modifying the application logic.

In sum, although **main.py** aggregates all application code into a single file, its internal organization—via distinct classes for ROI management, video playback, source panel specialization, result display, and application control—achieves a modular, maintainable architecture. Researchers may readily identify and modify discrete functionality,

such as swapping the detection model, adjusting tracker parameters, or extending GUI controls, without navigating a fragmented codebase.

4.3. Key Functions and Data Flows

The primary operations within **main.py** are encapsulated by five core functions: set_default_parameters(), process_videos(), normalize_videos(), stack_videos_horizontally(), and apply_yolo(). Each of these routines orchestrates a distinct phase of the data flow, from configuration ingestion to final annotated output. The following subsections examine each function's purpose, inputs, outputs, and internal logic, tracing the movement of data through the system.

4.3.1. Default parameter loading

The set_default_parameters() method shown in Listing 4.2 set previously saved region-of-interest (ROI) definitions and start-frame timestamps, enabling reproducible experiment setup. Invoked when the operator selects the "Use Default Parameters" control, this function opens default_parameters.txt as shown in Listing 4.3, reading each line as a comma-separated record of the form (x, y, w, h, timestamp).

```
def set_default_parameters(self):
    """Load default cropping parameters and timestamps from file."""
    try:
        with open('default_parameters.txt', 'r') as file:
            data = []
            for line in file:
                values = line.strip().split(',')
                x1 = int(values[0])
                y1 = int(values[1])
                width = int(values[2])
                height = int(values[3])
                start_time = values[4]
                data.append((x1, y1, width, height, start_time))

        for i, panel in enumerate(self.panels):
            panel.crop_rectangle.set_rectangle_dimensions(
                                                *data[i][:4])
            panel.seek_to_timestamp(data[i][4])
        self.status_label.setText("Default parameters applied")
    except Exception as e:
        self.status_label.setText(f"Error loading parameters: {str(e)}")
```

Listing 4.2. The set_default_parameters() method.

```
0,0,500,300,00:00.500
107,0,345,297,00:02.000
111,0,389,291,00:03.300
```

Listing 4.3. A default_parameters.txt file example.

for each of the three source panels. The method parses each line into integer coordinates (x, y, w, h) and a string timestamp. It then iterates over the three `SourceVideoPanel` instances, calling their `crop_rectangle.set_rectangle_dimensions(x, y, w, h)` methods to reposition and resize the interactive overlay. Subsequently, it invokes `panel.seek_to_timestamp(timestamp)`, which converts the "mm:ss.sss" format to a frame index based on the panel's frames-per-second value and sets the playback slider accordingly. Finally, the status label is updated to indicate successful application. In the event of file-access or parsing errors, the method catches exceptions and displays an error message without interrupting the application. By externalizing ROI and timestamp parameters, this function decouples experimental configuration from source code, allowing users to iterate over different scenarios rapidly.

4.3.2. Processing loop

The `process_videos()` function implements the main per-frame processing loop and serves as the central data pipeline driver as shown in **Listing 4.4**. Upon activation by the "Process Videos" button, this method first retrieves the designated start frame from each panel's slider, then instructs each OpenCV `VideoCapture` to seek to that frame. It computes the remaining frame count for each capture by subtracting the start frame from the total frame count, selecting the minimum of these values and capping it at 3000 to enforce real-time responsiveness.

Next, `process_videos()` validates that all three video streams are loaded; if not, it updates the status label and aborts. It then prepares the result panel by calling `result_panel.pre_stream(min_frame_count)` and initializes a `cv2.VideoWriter` for output archival. Within a try block, the method iterates over the frame horizon. In each iteration, it reads one frame from each `VideoCapture` into a local list. If any read fails, the loop terminates gracefully.

The function then inspects GUI checkbox states: if normalization is enabled, it calls normalize_videos(frames); otherwise it retains the raw frames. If horizontal stacking is active, it invokes stack_videos_horizontally(frames) to produce a single composite image; if not, it selects the first frame in the list for further processing. When object detection and tracking are requested, it calls apply_yolo(stacked_frame) to receive an annotated frame; if not, it uses the unmodified composite. The resulting frame is streamed to the result panel via result_panel.stream_frame(frame) and appended to the

output video file using out.write(frame). After writing, the method processes pending GUI events with QApplication.processEvents() to maintain interface responsiveness. Upon loop exit—due to frame exhaustion, user interruption, or error—the method finalizes the video writer, attempts to reload the output file into the result panel for preview, and updates the status label with the final frame count. Exception handling throughout ensures that errors in any stage do not crash the application but instead produce informative console messages and status updates.

```
def process_videos(self):
    """Process the videos with selected options and create result
        video."""

    # 1) Collect start frames, compute output dimensions
    start_frames = [panel.slider.value() for panel in self.panels]
    #…
    # 2) Open and synchronize captures
    for panel in self.panels:
        panel.cap.set(cv2.CAP_PROP_POS_FRAMES, start_frames[i])
    #…
    # 3) Frame-processing loop (normalize, stack, detect, write)
    for _ in range(max_frames):
        frames = [cap.read()[1] for cap in caps]
        if self.normalize_checkbox.isChecked():
            frames = self.normalize_videos(frames)
        if self.stack_checkbox.isChecked():
            stacked = self.stack_videos_horizontally(frames)
        else:
            stacked = frames[0]
        if self.yolo_checkbox.isChecked():
            stacked = self.apply_yolo(stacked)
        out.write(cv2.resize(stacked, output_size))
    #…
    # 4) Finalize and preview result
    out.release()
    self.result_panel.cap = cv2.VideoCapture("result_video.mp4")
```

Listing 4.4. Core implementation of process_videos() method.

4.3.3. Normalization of frames

The `normalize_videos()` helper function standardizes input frames' dimensions and content focus by applying ROI cropping and uniform resizing, **Listing 4.5** illlustrates that the method receives a list of raw or previously resized frames and iterates over them in parallel with the three SourceVideoPanel instances. For each panel–frame pair, the function retrieves the panel's ROI coordinates via `panel.crop_rectangle.get_rectangle_dimensions()`. It then performs two resizing operations: first, it scales the raw frame to the panel's display resolution (main_source_width × main_height) to align with the interactive overlay; second, it crops the resized image to the ROI subregion and rescales this sub-image to match the

result panel's height (calculated dynamically based on the smallest ROI among panels). These operations utilize cv2.resize for efficient interpolation and slicing for cropping, preserving the aspect ratio in each step. The normalized frames are collected into a new list, which is returned to the caller. Any exceptions—such as invalid ROI dimensions or empty frames—are caught and logged, and the corresponding frame is omitted from the output, ensuring robustness. This normalization stage mitigates scale disparities across camera views, facilitating coherent multi-view fusion.

```
def normalize_videos(self, frames):
    """Normalize the videos by cropping to the specified

       regions."""

    result_frames = []
    for i, panel in enumerate(self.panels):
        if i >= len(frames):
            continue
        frame = frames[i]
        if frame is None or frame.size == 0:
            continue
        try:
            frames[i] = cv2.resize(frames[i], (main_source_width,

                                               main_height))


            frames[i] = cv2.resize(frames[i][panel.y:panel.y +

                                             panel.height,

                                             panel.x:panel.x +

                                             panel.width],
                                   (panel.width,

                                   self.result_panel.height))
            result_frames.append(frames[i])
        except Exception as e:
            print(f"Error normalizing frame: {str(e)}")
    return result_frames
```

Listing 4.5. The normalize_videos() method.

4.3.4. Multi-view fusion

Horizontal concatenation of multiple camera views is implemented in the `stack_videos_horizontally()` function. As can be seen in **Listing 4.6**, this method accepts a list of three frames—normalized or raw—and returns a single composite image by invoking np.hstack(frames). The underlying NumPy operation allocates a contiguous output array of shape (H, $W_1 + W_2 + W_3$, C), where H is the common frame height and W_i are individual frame widths. This array copy incurs minimal overhead relative to detection inference and preserves each input frame's pixel values without blending or filtering. The function assumes that all frames share the same height and number of

channels; mismatches would result in a runtime exception. Consequently, `stack_videos_horizontally()` relies on prior normalization to guarantee height consistency. By merging multiple perspectives side by side, this fusion approach enables unified detection and tracking without geometric calibration, simplifying downstream processing.

```
def stack_videos_horizontally(self, frames):
    return np.hstack(frames)
```

Listing 4.6. The stack_videos_horizontally() method.

4.3.5. Detection and tracking

The `apply_yolo()` function encapsulates object detection and multi-object tracking in a single call to the Ultralytics API. It accepts a composite or single-view frame, creates a deep copy to preserve the source image, and invokes the illustrated code in **Listing 3.3** of the system design and architecture chapter.

As shown in **Listing 4.7**, the track method performs internal preprocessing (such as letterboxing), YOLO v11 inference, and ByteTrack data association based on the configuration file bytetrack.yaml. The returned results list contains detection records for each input image; for a single frame, relevant data reside in results[0].boxes. Each box provides bounding-box coordinates (.xyxy), confidence scores (.conf), class indices (.cls), and persistent track IDs (.id). The function iterates over these boxes, filters by class (restricting to vehicles) and confidence (> 0.3), casts coordinate and ID values to integers, and calls `cvzone.cornerRect()`, `cvzone.putTextRect()`, and `cv2.circle()` to annotate the frame with bounding boxes, textual IDs, and centroids. Any exceptions during annotation—such as missing .id attributes—are caught and logged to maintain processing continuity. The annotated frame is returned to the caller, which then streams and archives it.

```
def apply_yolo(self, frame):
    """Apply YOLO object detection to the frame and count objects
       using the embedded tracker."""

    # 1) Early exit on empty frame
    if frame is None or frame.size == 0:
        return frame

    # 2) Run YOLO tracker
    results = self.model.track(frame.copy(),
                               persist=True, conf=0.3, iou=0.5,
                               tracker='bytetrack.yaml')

    # 3) Filter and annotate detections
    for box in results[0].boxes:
        cls = int(box.cls[0]); conf = float(box.conf[0])
        if cls_name[cls] in ["car","truck","bus","motorbike"] and
           conf>0.3:
            x1,y1,x2,y2 = map(int, box.xyxy[0])
            id_ = int(box.id[0]) if hasattr(box, 'id') else None
            draw_box_and_id(frame, x1,y1,x2,y2, id_)

    return frame
```

Listing 4.7. Core implementation of apply_yolo() method.

4.3.6. Comprehensive data flow

In summary, data flow within **main.py** proceeds as follows. User actions in the GUI (button clicks and checkbox toggles) set controller flags and may populate source `panel states via set_default_parameters()`. Upon processing start, `process_videos()` orchestrates frame retrieval from `SourceVideoPanel` buffers, conditional normalization through `normalize_videos()`, fusion via `stack_videos_horizontally()`, and, if enabled, detection and tracking via `apply_yolo()`. Annotated frames then flow into the `ResultVideoPanel` for display and into a `cv2.VideoWriter` for persistent storage. Helper methods and exception handling at each stage ensure that the pipeline remains robust, maintainable, and transparent to the operator. This modular decomposition of key functions and their data exchanges underpins the system's flexibility and facilitates targeted extension or experimentation with alternative preprocessing, fusion, or inference strategies.

5. EXPERIMENT SETUP

5.1. Video Sources And Layouts

The experimental evaluation employed three Logitech C920 PRO webcams [19] mounted along the northern edge of the Serdivan AVM roof, each connected to a dedicated laptop to record overlapping, north-facing views of two adjacent roadway segments—Street A and Street B—for approximately five minutes at 30 frames per second and 1280 × 720 resolution. The cameras were arranged at roughly 50 m intervals, all oriented due north to ensure that every vehicle traversing the study area appeared in each of the three video streams. The resulting configuration, including camera positions, viewing directions, and key roadway features, is illustrated in **Figure 5.1**.

Figure 5.1. Camera layout and scene geometry in the experimental setup.

Street A comprised a two-lane arterial extending east–west directly in front of the mall and was recorded under evening twilight illumination. Camera 1's field of view encompassed the western approach to a partially visible roundabout, creating intermittent occlusions and complex motion trajectories as vehicles negotiated the curve. Camera 2 captured the central portion of Street A, where the primary Serdivan AVM parking-lot

entrance introduced frequent vehicle decelerations and turns that periodically obscured adjacent traffic flow.

Immediately north of Street A lay Street B, a parallel service road recorded concurrently under the same lighting conditions. In Camera 1's view, Street B featured a right-branching side street between an IMK storefront and a residential block, generating additional turning movements against a retail backdrop. Camera 3's view focused on the intersection adjacent to the MusicArt restaurant, where vehicles entered and exited the main carriageway amid parked cars and outdoor dining areas—thereby presenting varied motion patterns and background clutter that posed considerable challenges to both the detection and tracking modules.

Within the graphical interface, the three source video panels were arranged horizontally (each 500 × 300 pixels), with translucent overlays allowing the operator to define rectangular regions of interest (ROIs) that encompassed the overlapping roadway segments visible in all views. ROIs were aligned such that virtual stacking boundaries corresponded to the physical adjacency of the cameras. The result panel, positioned beneath the source panels and sized identically (500 × 300 pixels), displayed the horizontally concatenated composite image along with real-time detection and tracking overlays. This layout permitted simultaneous inspection of individual camera feeds, ROI alignment, and fused outputs, providing a rigorous testbed for multi-camera vehicle tracking under varied structural and illumination conditions.

5.2. Evaluation Metrics

The primary performance measure for the multi-camera vehicle tracking system is an accuracy metric that directly assesses the preservation of object identities as vehicles cross the virtual fusion boundary between camera fields. This metric is defined by two manual counts, obtained by visual inspection of successive frames in the processing pipeline. The first count, denoted GT (ground truth), is the number of distinct vehicles visible in the last frame of the source panel immediately before the horizontal concatenation line. The second count, TP (true positive tracked vehicles), is the number of those same vehicles correctly maintained—that is, assigned consistent track identifiers—in the first frame of the result panel immediately after fusion. Accuracy A is then expressed as illustrated in equation 5.1.

$$A = \frac{TP}{GT} \times 100\% \tag{5.1}$$

This definition ensures that only vehicles detected before the fusion boundary and successfully re-identified after concatenation contribute to TP. Vehicles missed in the post-fusion frame—whether through detection failure or identity mismatch—do not increase the numerator, thus reducing accuracy. Conversely, every correctly preserved identity contributes positively, offering a direct measure of end-to-end system effectiveness in maintaining continuity across camera views.

The placement of the counts relative to the fusion line is critical. By locating GT immediately upstream of the concatenation boundary—in the final source-panel frame—the metric captures the system's baseline detection and tracking performance within a single camera's ROI. The subsequent count, TP, taken immediately downstream in the composite output, measures the incremental effect of horizontal stacking and multi-view processing on tracking continuity. Because the counting procedure is identical in both positions—requiring the operator to visually match vehicle instances and verify track identifiers—the metric isolates the specific impact of the fusion operation, independent of any differences in detection performance between cameras.

This accuracy metric offers several advantages for the experimental context. First, it combines detection and tracking into one unified measure, reflecting the system's real-world objective of preserving vehicle identities as they move through adjacent fields of view. Second, by relying on manual counts around the fusion boundary, it avoids the need for exhaustive frame-by-frame ground-truth annotation across entire video sequences, thereby reducing the burden on the operator while still providing reliable quantitative assessment. Third, because the metric is based on pairs of frames immediately before and after fusion, it focuses exclusively on the critical juncture where identity continuity is most vulnerable, offering targeted insight into the efficacy of the multi-view integration.

All accuracy values obtained using this metric—across various ROI selections, environmental conditions, and camera configurations—are presented in Chapter 6. The definition and procedure established here ensure that those results are directly comparable and attributable to the system's handling of the virtual concatenation process.

5.3. Execution Procedure

The experimental evaluation was conducted entirely through the VideoPlayerApp graphical interface, ensuring that all trials adhered to a consistent user workflow. Three prerecorded Logitech C920 PRO video streams—each containing overlapping views of Street A and Street B—were processed in two phases corresponding to the individual roadway segments. ByteTrack hyperparameters remained constant throughout (**Table 5.1**), while ROI and timestamp presets were applied separately for each street (**Tables 5.2** and **5.3**). The procedure comprised four steps: application launch, video and parameter loading, processing configuration, and manual evaluation.

5.3.1. Application launch

The operator initiated the evaluation by executing **main.py** in the Windows 11, Python 3.9 environment. The application window appeared with three empty source-video panels arranged horizontally and an inactive result-video panel below. All processing controls—**Use Default Parameters**, **Normalize Videos**, **Stack Horizontally**, **Apply YOLO**, and **Process Videos**—were visible and reset to their default, unchecked states. In the background, the YOLO v11 model and ByteTrack tracker were loaded using the configuration in **Table 5.1**.

Table 5.1. ByteTrack configuration parameters.

Parameter	Value	Parameter	Value	Parameter	Value
tracker_type	bytetrack	new track trech	0.68	fuse_score	True
track_high_thresh	0.3	track_buffer	100	Min_box_area	10
track_low_thresh	0.2	match_thresh	0.9	frame_rate	30

5.3.2. Video and parameter loading

For each source panel, the operator clicked the **Open Video** button and selected the appropriate Logitech C920 PRO file. Upon loading, the first frame appeared immediately. The **Use Default Parameters** button was then pressed to apply preset ROIs and start-frame timestamps. In the first phase (Street A), the presets in **Table 5.2** were loaded; in the second phase (Street B), those in **Table 5.3** were applied. The translucent

CropRectangle overlays repositioned and resized automatically according to the (x, y, width, height) entries, and the playback sliders advanced to the specified timestamps. The operator confirmed that each overlay accurately encompassed the intended roadway segment and adjusted it manually if necessary.

Table 5.2. Street A default parameters.

Panel	x	y	Width	Height	Start Timestamp
1	0	0	500	300	00:00.500
2	107	0	345	297	00:02.000
3	111	0	389	291	00:03.300

Table 5.3. Street B default parameters.

Panel	x	y	Width	Height	Start Timestamp
1	0	0	500	300	00:03.000
2	107	0	272	297	00:01.500
3	36	0	500	291	00:00.500

5.3.3. Processing configuration

With ROIs and timestamps applied, the operator enabled **Normalize Videos**, **Stack Horizontally**, and **Apply YOLO**. These selections instructed the system to perform ROI-based cropping and resizing, to concatenate the three normalized views into a single composite image, and to execute the YOLO v11 detection with ByteTrack tracking in one step. The operator then clicked **Process Videos**, prompting the application to synchronize all streams to their start frames, determine the processing horizon, and activate the result-video panel. Processed frames—annotated with bounding boxes and persistent track identifiers—were displayed in real time and recorded to disk using OpenCV's video writer. **Figure 5.2** illustrates the pipeline configurations through the PyQt5 GUI.

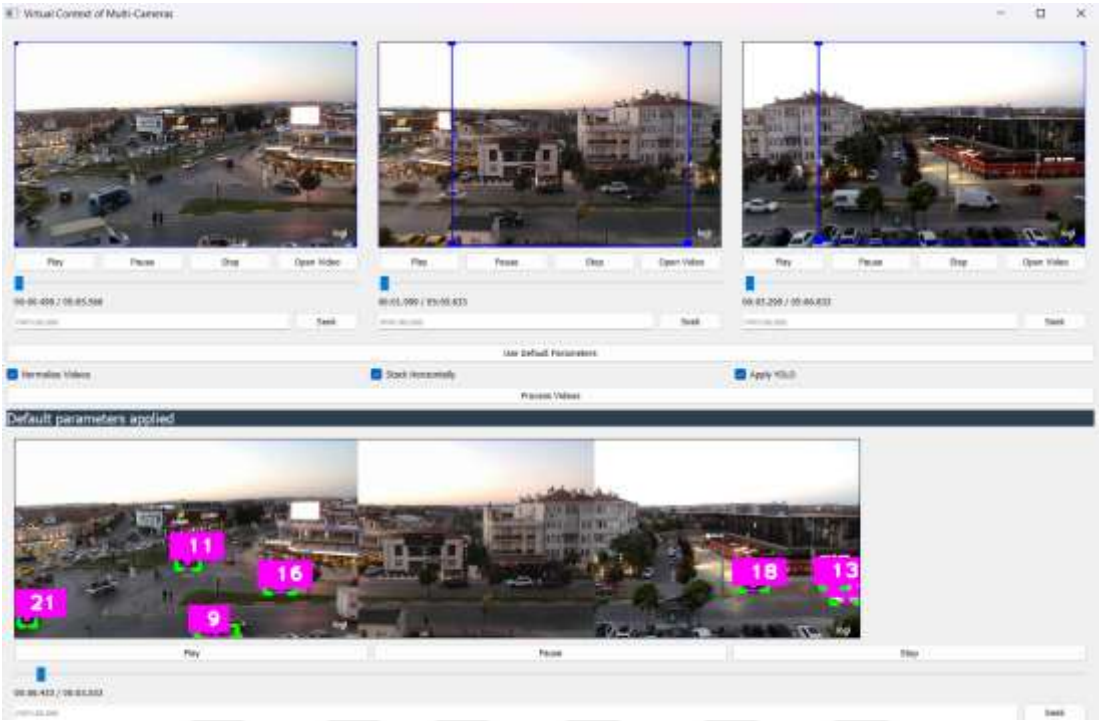

Figure 5.2. The multi-camera video processing tool GUI.

5.3.4. Manual evaluation

Upon completion of each processing run, the operator conducted the manual evaluation. In the source panel corresponding to the final frame before fusion, the operator counted the number of distinct vehicles that had entered the ROI (GT). In the result panel displaying the first post-fusion frame, the operator counted those same vehicles that retained consistent track identifiers (TP). These counts furnished the data necessary for computing the accuracy metric defined in Section 5.2. By repeating this procedure for both Street A and Street B—while maintaining a fixed ByteTrack configuration and applying scenario-specific ROI presets—the experiments yielded directly comparable measurements of the system's ability to preserve identity continuity across adjacent camera views.

6. RESULTS AND ANALYSIS

6.1. Quantitative Outcomes

The identity-preservation performance of the proposed multi-camera tracking system was evaluated in two experimental trials, each employing a fixed ByteTrack configuration (**Table 5.1**) and distinct ROI and timestamp presets optimized respectively for Street A (Trial 1) and Street B (Trial 2). The results of both trials are illustrated in **Table 6.1**. Accuracy values, computed according to the definition in Section 5.2, quantify the proportion of vehicles whose persistent identifiers remained consistent across the virtual fusion boundaries. These boundaries are denoted "1" (between Cameras 1 and 2), "2" (between Cameras 2 and 3), and "1 and 2" for sequential traversal of both.

Table 6.1. Results of optimized configurations in trials 1 and 2 for streets A and B.

Street	Fusion Boundary	GT	TP	Accuracy (%)
A	1	19	14	73.7
	2	20	14	70.0
	1 and 2	18	12	66.7
B	1	32	26	81.3
	2	34	23	67.6
	1 and 2	31	23	74.2

Under Street A-tuned parameters, the arterial segment demonstrated moderate identity continuity: accuracies of 73.7 % and 70.0 % at the first and second boundaries, respectively, declined to 66.7 % when both boundaries were crossed.

When configured for the service road, the pipeline achieved high identity preservation on Street B, with accuracies up to 81.3 % at the first fusion boundary and 74.2 % across both.

A comparative examination of the two trials reveals three key insights. First, single-boundary fusion consistently outperforms dual-boundary fusion, affirming that each additional concatenation step compounds the risk of detection or association errors. Second, optimal ROI and timing parameters are highly segment-specific: configurations affording moderate success on one roadway may precipitate total failure on an adjacent, overlapping roadway. Third, environmental and geometric factors—such as illumination, viewing distance, and background complexity—interact critically with fixed detection and tracking parameters to determine overall system efficacy.

These quantitative outcomes furnish a solid empirical foundation for the qualitative observations and error-mode analyses presented in Sections 6.2 and 6.3, respectively. By isolating the impact of fusion operations and scene characteristics through controlled parameter variation, the study delineates the operational limits and potential avenues for enhancing multi-camera vehicle tracking robustness.

6.2. Qualitative Observations

The qualitative assessment concentrates on the system's capacity to maintain consistent track identifiers at the precise moment vehicles traverse the virtual fusion boundaries. In accordance with the evaluation protocol established in Section 5.2, only those vehicles that retained the same identifier in the first post-fusion frame were deemed successfully tracked; any subsequent identifier changes beyond that immediate frame did not affect the inclusion of that instance in the accuracy computation.

In the Street A trial, the primary fusion boundary between Cameras 1 and 2 abutted the entrance to the Serdivan AVM parking lot, which lay outside Camera 2's field of view. One observed vehicle disappeared from Camera 1's frame as it entered the lot and failed to reappear in Camera 2, an event that was duly excluded from the identity-preservation counts. More frequently, brief occlusions caused by a small service building in Camera 2's view did not disrupt tracking continuity: in most cases, vehicles that were partially obscured nonetheless produced valid bounding-box detections at the boundary, enabling ByteTrack's motion-based association to preserve their identifiers.

This resilience under moderate occlusion suggests that the tracker's combination of appearance and Kalman-filter predictions effectively bridges transient visual gaps when the fusion-boundary frame still contains sufficient object information.

Conversely, Street B's static background clutter—comprising parked vehicles along the separator barrier and intermittent foliage—produced substantial detector noise under the Street A–optimized configuration. During Trial 1, these spurious detections overwhelmed the association process at both fusion boundaries, resulting in zero recorded tracks despite the presence of moving vehicles. When the ROI and timestamp parameters were re-tuned for Street B (Trial 2), the uninterrupted linear motion of service-road traffic enabled the tracker's motion model to distinguish bona fide vehicles from static artifacts. As a result, a majority of vehicles were correctly re-identified at the first fusion boundary, yielding accuracies exceeding 80 %. Crucially, once a vehicle's identifier persisted in the immediate post-fusion frame, that instance was counted as successfully tracked regardless of any later identifier reassignment caused by background interference.

A consistent shortcoming across both scenarios was the complete failure to detect or track motorbikes. Despite repeated appearances in all camera views, no two-wheeled vehicles were registered by the YOLO v11 detector under the dusk lighting conditions and chosen resolution. Their systematic omission from the post-fusion counts highlights the necessity for model fine-tuning or higher-resolution imaging when tracking smaller object classes in urban environments.

Additional qualitative insights emerged from analysis of infrastructure-induced occlusions in Camera 3's Street A view, where a bus station and adjacent trees intermittently masked portions of larger vehicles. When the fusion-boundary frame captured a sufficiently intact silhouette, ByteTrack successfully maintained the prior identifier; however, heavily truncated detections occasionally failed to register, leading to identity loss at that boundary. Similarly, the interplay of low-angle dusk illumination and roadside vegetation introduced localized shadowing that degraded detection confidence in peripheral regions. Nonetheless, because only the immediate post-fusion frame determined tracking success, transient jitter or missed detections outside that frame did not penalize the accuracy metric.

Collectively, these qualitative observations affirm that the proposed pipeline can achieve robust identity preservation at the critical fusion juncture when ROIs and temporal offsets are appropriately configured. They also underscore the importance of addressing object-scale limitations, enhancing occlusion resilience at fusion seams, and mitigating background clutter—considerations that will inform the error-mode analysis and proposed improvements in subsequent sections.

6.3. Error Analysis

This section examines the principal failure modes observed during the experimental trials, with the objective of elucidating the underlying causes of identity-preservation breakdowns and guiding future improvements. Error analysis is structured around three principal categories: detection omissions, class-specific failures, and parameter misalignment effects.

6.3.1. Detection omissions

Dusk illumination engendered uneven scene brightness and reduced contrast between vehicle bodies and road surfaces. Under the fixed detector settings (confidence threshold = 0.3, minimum bounding-box area = 10 pixels), YOLO v11 intermittently failed to register vehicles that appeared only as dim or shadowed shapes at the moment of fusion. In such instances, the absence of a valid detection in the boundary frame rendered identity preservation impossible, regardless of subsequent visibility. These false negatives accounted for a substantial fraction of missed tracks in both trials, particularly in peripheral regions affected by tree-cast shadows and vehicle reflections.

6.3.2. Class-specific failures

Motorbikes constituted a conspicuous failure mode: across all three camera feeds and two parameterizations, no motorbike instances were successfully detected or tracked at any fusion boundary. This systematic omission indicates that the pretrained YOLO v11 weights, trained on COCO classes, lack sufficient representation for small two-wheeled vehicles under the given resolution (1500 × 300) and dusk lighting. Motorbikes frequently appeared as narrow, low-contrast silhouettes that fell below both the detection confidence threshold and the minimum bounding-box area filter (min_box_area = 10). The absence of motorbike detections not only reduced overall

coverage of vehicle types but also highlights the necessity for class-specific fine-tuning or dedicated re-identification modules when comprehensive traffic analysis is required.

6.3.3. Parameter misalignment effects

The stark contrast in performance between the Street A and Street B–optimized configurations demonstrates the extreme sensitivity of the fusion pipeline to ROI placement and temporal synchronization. In Trial 1, ROI windows tuned for the arterial segment inadvertently excluded portions of the service-road vehicles' entry trajectories, while in Trial 2, the reverse misalignment occurred. These misalignments led to truncated vehicle crops at the boundary frame, causing detection failures even for larger vehicles. Moreover, slight discrepancies in start-frame timestamps—on the order of a few hundred milliseconds—resulted in vehicles being captured at inconsistent positions relative to the fusion seam, further increasing the likelihood of identity discontinuities. Thus, precise calibration of both spatial and temporal parameters is essential for robust multi-camera tracking, and small deviations can produce catastrophic identity loss, as evidenced by the zero-accuracy outcomes on the non-optimized segment in each trial.

6.3.4. Summary of error modes

In aggregate, the error analysis reveals that the primary impediments to reliable identity preservation are (1) missed detections under low-contrast or partial-occlusion conditions; (2) failure to detect smaller vehicle classes such as motorbikes; and (3) sensitivity to spatial and temporal alignment of ROI and start frames. Addressing these challenges will require a combination of model fine-tuning (to improve detection recall), class-targeted augmentation (to include small vehicles), and adaptive ROI/timestamp calibration (to ensure precise boundary alignment).

7. CONCLUSION AND RECOMMENDATIONS

7.1. Concluding Remarks

This thesis addressed the challenge of achieving efficient and robust multi-camera vehicle tracking without reliance on geometric calibration or inter-camera re-identification. Traditional fusion strategies—whether sensor-level, feature-level, or decision-level—often require high computational resources, complex calibration pipelines, or substantial training data to resolve inter-view ambiguities. In contrast, the proposed **Virtual Context** stacking approach offers a streamlined, calibration-free solution by horizontally concatenating normalized ROI crops from multiple camera views into a single composite frame, enabling one-shot detection and tracking via YOLO v11 and ByteTrack.

The system was implemented as a modular application using PyQt5 for operator interaction and OpenCV for image handling. It demonstrated the capability to sustain real-time processing while preserving object identities across adjacent and overlapping views. Experimental evaluation on a three-camera Logitech C920 setup revealed high identity preservation rates, with up to 81% cross-view continuity, despite variations in perspective and lighting.

Overall, the findings confirm that Virtual Context stacking constitutes a viable alternative to conventional multi-camera fusion pipelines, offering reduced deployment complexity, minimized latency, and enhanced scalability. It provides a strong foundation for real-time intelligent transportation systems, particularly in edge-constrained environments where efficiency and simplicity are paramount.

7.2. Limitations

While the proposed Virtual Context stacking framework offers a simplified and efficient approach to multi-camera vehicle tracking, several limitations constrain its generalizability and performance in certain scenarios. First, the concatenation-based fusion assumes fixed camera configurations with consistent resolution and static ROIs. As such, it is not inherently robust to dynamic camera orientations, zoom variations,

or significant changes in camera positioning, all of which may require manual redefinition of ROI coordinates and re-normalization.

Second, despite avoiding explicit geometric calibration and re-identification modules, the method lacks semantic understanding of spatial relationships across views. Unlike homography- or 3D-aware systems that leverage scene geometry to resolve occlusions and trajectory continuity, Virtual Context operates purely on the composite frame, making it susceptible to identity switches in heavily occluded or highly overlapping zones—particularly when vehicles appear simultaneously in adjacent ROIs.

Moreover, while the use of YOLO v11 and ByteTrack ensures real-time inference, the unified composite frame may introduce non-uniform object scale and aspect distortions across stitched regions, potentially degrading detection and tracking accuracy. These artifacts become more pronounced with heterogeneous cameras or wide disparities in field of view. Addressing such limitations will require more adaptive ROI management and potentially hybrid fusion schemes.

7.3. Future Work

Building upon the foundation of the Virtual Context stacking approach, several promising directions for future research may enhance the scalability, robustness, and adaptability of the proposed system in broader deployment scenarios. One key extension involves **automated ROI adaptation**. The current framework depends on manually defined and fixed ROI configurations, which limit responsiveness to changes in camera layout or scene dynamics. Integrating an adaptive ROI proposal mechanism based on scene understanding or motion saliency could enable dynamic region selection, thereby improving flexibility and minimizing manual intervention during re-deployment or environmental changes.

Another avenue for improvement is the **integration of temporal attention mechanisms** into the tracking pipeline. While ByteTrack effectively handles low-confidence detections, it does not explicitly model long-term temporal dependencies. Augmenting the tracker with a temporal attention module or a memory-augmented recurrent network could improve track continuity, particularly under extended occlusion or reappearance after viewpoint transitions.

In addition, **semantic-aware fusion strategies** may be introduced to complement Virtual Context stacking. For example, using weak geometric priors or learned spatial embeddings could enrich the composite frame with structural context, helping the system differentiate between similar vehicles and better resolve inter-view ambiguities in overlapping regions. This hybridization could balance the benefits of calibration-free deployment with selective use of lightweight spatial cues.

Furthermore, **scaling to heterogeneous camera networks**—with varying resolutions, orientations, and frame rates—remains a challenge. Future systems should incorporate view normalization modules or cross-domain adaptation techniques to harmonize feature representations across disparate inputs, ensuring reliable fusion without sacrificing efficiency.

Finally, efforts should also focus on **benchmarking the proposed method on publicly available multi-camera vehicle tracking datasets** with annotated cross-view identities. This would enable standardized evaluation, facilitate comparison with state-of-the-art approaches, and drive further refinements of the framework based on diverse, real-world conditions. Through these extensions, the Virtual Context paradigm could evolve into a more versatile and robust solution suitable for a wide range of intelligent transportation system deployments.

7.4. Closing Statement

The Virtual Context stacking framework proposed in this thesis presents a pragmatic and effective solution to the enduring challenge of multi-camera vehicle tracking. By circumventing the need for explicit geometric calibration and computationally expensive re-identification processes, the method offers a streamlined alternative capable of maintaining object identity across disjoint camera views. The integration of normalized ROI cropping, horizontal concatenation, and unified YOLO v11 + ByteTrack inference allows for end-to-end vehicle detection and tracking within a single composited frame, significantly reducing system complexity and ensuring real-time performance.

Empirical results from a three-camera deployment demonstrated the method's capacity to preserve identity continuity while minimizing latency, even in scenarios with adjacent and partially overlapping fields of view. Although certain limitations remain—

such as sensitivity to ROI misalignment and identity switching under dense occlusion—the overall approach proves well-suited for scalable, edge-based deployments in intelligent transportation systems.

In conclusion, Virtual Context stacking provides a valuable contribution to the field of real-time traffic monitoring and lays the groundwork for further advances in calibration-free multi-camera fusion strategies. Its simplicity, efficiency, and empirical viability position it as a compelling candidate for future ITS applications requiring both accuracy and deployability.

REFERENCES

[1] E. Dilek and M. Dener, "Computer vision applications in intelligent transportation systems: A survey," Sensors, vol. 23, no. 6, Art. no. 2938, Mar. 2023, doi:10.3390/s23062938.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real time object detection," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., Las Vegas, NV, USA, 2016, pp. 779–788, doi:10.1109/CVPR.2016.91.

[3] A. S. Olagoke, H. Ibrahim, and S. S. Teoh, "Literature survey on multi camera system and its application," IEEE Access, vol. 8, pp. 172892–172922, Sept. 2020, doi:10.1109/ACCESS.2020.3024568.

[4] H. F. Yang, J. Cai, C. Liu, R. Ke, and Y. Wang, "Cooperative multi camera vehicle tracking and traffic surveillance with edge artificial intelligence and representation learning," Transp. Res. Part C Emerg. Technol., vol. 148, Art. no. 103982, 2023, doi:10.1016/j.trc.2022.103982.

[5] H. Zhang, R. Fang, S. Li, Q. Miao, X. Fan, J. Hu, and S. Chan, "Multi camera multi vehicle tracking guided by highway overlapping FoVs," Mathematics, vol. 12, no. 10, Art. no. 1467, May 2024, doi:10.3390/math12101467.

[6] J. Shim et al., "Multi target, multi camera vehicle tracking for city scale traffic management," in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops, Nashville, TN, USA, 2021, pp. 38–47, doi:10.1109/AICITY53759.2021.9523135.

[7] M. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in Proc. IEEE Int. Conf. Image Process., Phoenix, AZ, USA, 2016, pp. 3464–3468, doi:10.1109/ICIP.2016.7533003.

[8] Z. Zhang et al., "ByteTrack: Multi Object Tracking by Associating Every Detection Box," arXiv preprint arXiv:2110.06864, Oct. 13, 2021, doi:10.48550/arXiv.2110.06864.

[9] W. Luo, J. Xing, A. Milan, X. Zhang, W. Liu, and T.-K. Kim, "Multiple object tracking: A literature review," Artif. Intell., vol. 293, Art. no. 103448, Apr. 2021, doi:10.1016/j.artint.2020.103448.

[10] G. Ciaparrone, F. Luque Sánchez, S. Tabik, L. Troiano, R. Tagliaferri, and F. Herrera, "Deep learning in video multi object tracking: A survey," arXiv preprint arXiv:1907.12740, Jul. 18, 2019.

[11] M. Adžemović, "Deep learning based multi object tracking: A comprehensive survey from foundations to state of the art," arXiv preprint arXiv:2506.13457, Jun. 16, 2025.

[12] D. Koller, J. Weber, and J. Malik, "Robust multiple car tracking with occlusion reasoning," in Proc. Eur. Conf. Comput. Vis., J.-O. Eklundh, Ed., vol. 800, Berlin, Germany, 1994, pp. 189–196, doi:10.1007/3-540-57956-7_22.

[13] Riverbank Computing Limited, “PyQt5 Reference Guide,” Version 5.15.7, 2023. [Online]. Available: https://www.riverbankcomputing.com/static/Docs/PyQt5/. Accessed: Jul. 16, 2025. riverbankcomputing.com

[14] Ultralytics, “Ultralytics YOLO11 Documentation,” 2024. [Online]. Available: https://docs.ultralytics.com/models/yolo11/. Accessed: Jul. 16, 2025. yolo-docs.readthedocs.io

[15] Ultralytics, “Multi-Object Tracking with Ultralytics YOLO,” 2023. [Online]. Available: https://docs.ultralytics.com/modes/track/. Accessed: Jul. 16, 2025. Ultralytics Docs

[16] OpenCV developers, “OpenCV Documentation,” OpenCV 4.x, 2025. [Online]. Available: https://docs.opencv.org/4.x/index.html. Accessed: Jul. 16, 2025.

[17] NumPy Developers, “NumPy Documentation,” NumPy 2.4.dev0, Jul. 15, 2025. [Online]. Available: https://numpy.org/doc/. Accessed: Jul. 16, 2025. NumPy

[18] Computer Vision Zone, “cvzone,” v1.6.1, Sep. 9, 2023. [Online]. Available: https://github.com/cvzone/cvzone. Accessed: Jul. 16, 2025. PyPI

[19] Logitech Inc., “HD Pro Webcam C920 Complete Setup Guide,” 2020. [Online]. Available: https://www.logitech.com/assets/45920/8/hd-pro-webcam-c920-quick-start-guide.pdf. Accessed: Jul. 16, 2025.

APPENDICES

APPENDIX A. Source Code Repository

A.1 Virtual Context-Based Multi-Camera Vehicle Tracking Software

Description: This repository contains the full source code for the real-time multi-camera vehicle tracking system based on Virtual Context Stacking. **Virtual Context-Based Multi-Camera Vehicle Tracking Software** is a PyQt5-based desktop application designed for synchronized processing of up to three video sources. It offers interactive ROI selection, video navigation controls, and a modular processing pipeline that includes frame normalization, horizontal stacking, and YOLO-based object detection with ByteTrack tracking. Processed outputs are previewed in real time and exported to a video file.

Repository URL: https://github.com/wael-kabouk/virtual_context

CURRICULUM VITAE

Name Surname : Wael KABOUK

EDUCATION:

- **Undergraduate** : 2023, Muğla Sıtkı Koçman University, Faculty of Engineering, Department of Computer Engineering
- **Graduate** : 2025, Sakarya University, Department of Software Engineering, M.Sc. Program in Software Engineering

PROFESSIONAL EXPERIENCE AND AWARDS:

- GAT Exam – Top 1% in quantitative and analytical reasoning (Award)