BECE498J - Project – II

Smart Vehicle Infotainment System using Computer Vision and AR

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

Electronics and Communication Engineering

by

21BEC0025 Sudharson Aswin N K

21BEC0490 Tushar Sharma

Under the Supervision of

Dr. Rohit Mathur

Assistant Professor Sr. Grade 1

School of Electronics Engineering (SENSE)

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

April 2025

DECLARATION

I hereby declare that the thesis entitled "**Smart Vehicle Infotainment System using Computer Vision and AR"** submitted by me, for the award of the degree of Bachelor of Electronics and Communication Engineering to VIT is a record of Bonafide work carried out by me under the supervision of Dr. Rohit Mathur.

I further declare that the work reported in this thesis has not been submitted previously to this institute or anywhere for the consideration of the degree/diploma.

Place: Vellore

Date:

Signature of the Candidate

Sudharson Aswin N K

Tushar Sharma

CERTIFICATE

This is to certify that the thesis entitled "**Smart Vehicle Infotainment System using Computer Vision and AR**" submitted by Sudharson Aswin N K [21BEC0025] & Tushar Sharma [21BEC0490], School of Electronics Engineering, VIT, for the award of the degree of Bachelor of Technology in Electronics and Communication Engineering, is a record of Bonafide work carried out by him / her under my supervision during the period Winter Semester 2024-2025, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The thesis fulfills the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Vellore

Date: **Signature of the Guide**

Signature of the Examiner

ACKNOWLEDGEMENTS

I am deeply grateful to the management of Vellore Institute of Technology (VIT) for providing me with the opportunity and resources to undertake this project. The institute's commitment to academic excellence and its exceptional infrastructure have played a vital role in fostering an environment conducive to learning, innovation, and personal growth.

I would like to express my sincere gratitude to Dr. Kannadassan D, Head of the Department of Communication Engineering, for his constant support and for granting access to the department's facilities, which were instrumental in the successful execution of this project. His leadership and encouragement have greatly enriched my academic experience.

I am profoundly thankful to my project supervisor, Dr. Rohit Mathur, for his dedicated mentorship, insightful guidance, and unwavering support throughout the course of this dissertation. His expert advice, patience, and constructive feedback have been crucial in shaping my ideas and overcoming the challenges faced during the project. His commitment to excellence and willingness to share his knowledge have left a lasting impact on my academic journey.

I also extend my heartfelt thanks to all faculty members, staff, and friends who supported me, directly or indirectly, during various stages of this project. Their encouragement and help have been invaluable and deeply appreciated.

Sudharson Aswin N K [21BEC0025]

Tushar Sharma [21BEC0490]

Executive Summary

The **Smart Vehicle Infotainment System using Computer Vision and Augmented Reality (AR)** is a real-time driver assistance platform designed to improve road safety and situational awareness through QR code-based cues. The main goal is to create an affordable, modular system capable of projecting relevant visual alerts and contextual information directly onto a driver's view via a Head-Up Display (HUD). Using computer vision, the system interprets traffic signs, warnings, and real-time surroundings, with a flexible architecture that supports both embedded and high-performance deployments.

To address varying performance and cost requirements, the system was developed in two primary configurations:

- **System 1: Standalone Embedded Configuration**

Built around the Raspberry Pi 5, this setup leverages Haar cascade classifiers for lightweight object detection and overlays 2D images (e.g., caution signs, indicators) onto the live video stream. A multithreaded, multiprocessing architecture ensures efficient real-time handling of image capture, analysis, and AR rendering. This configuration is optimized for environments with limited hardware resources and is easily integrable into low-power systems.

- **System 2: Distributed High-Performance Configuration**

In this version, the Raspberry Pi functions as a video streaming node using Flask, while the processing is handled by an external high-performance unit (e.g., a laptop or in-car infotainment system). The external system uses YOLOv8s-world for advanced object detection, decodes QR metadata, and renders 3D AR content. Parallel processing ensures high responsiveness and AR fidelity.

Developed across a structured 12-week timeline, the project delivers a retrofittable, scalable smart mobility solution that bridges legacy vehicles with emerging AR technologies.

INDEX

List of Figures

List of Tables

List of Abbreviations

YOLO	You Look Only Once
CV	Computer Vision
QR Code	Quick Response code
AI	Artificial Intelligence
AR	Augmented reality
OS	Operation System
GB	Gigabyte
MP	Megapixel
SD	Secure Digital
3D	three-dimensional
2D	two-dimensional
RAM	Random Access Memory
HUD	Heads Up Display
LIDAR	Light Detection and Ranging
NPU	Neural Processing Unit
GPU	Graphics processing unit
IEEE	Institute of Electrical and Electronics Engineers
GPS	Global Positioning System
ADAS	Advanced Driver Assistance Systems

1. INTRODUCTION

1.1.LITERATURE REVIEW

In the pursuit of smarter, safer, and more adaptive mobility systems, researchers and developers have increasingly explored the integration of **Computer Vision (CV)**, **Augmented Reality (AR)**, and **AI-driven object detection** into driver assistance platforms. These efforts are largely motivated by the growing need to enhance real-time situational awareness, reduce accidents, and build more resilient, connected urban transport infrastructure. Several works have explored solutions ranging from immersive AR head-up displays to robust QR code-based localization, all contributing in various ways to the broader vision of modular and scalable smart vehicle systems.

Zhe An et al. (2018), in their IEEE Access paper *"A Real-Time Three-Dimensional Tracking and Registration Method in the AR-HUD System,"* introduced an innovative 3D tracking and registration approach for projecting virtual content into real-world driving scenes. Their system relied on camera pose estimation and feature interpolation, achieving an impressive registration accuracy of under 2 pixels without traditional feature matching, which helped significantly in reducing driver distraction. However, the system required binocular camera calibration and stereo imaging, which may not be easily adaptable to low-resource environments — a challenge addressed in our project through lightweight 2D overlay systems and shared-memory parallelism.

In a related study, Samia Abid et al. (2021) proposed a **cost-effective, QR and vision-based sensorless steering control and localization system** for autonomous vehicles, as detailed in *"A Robust QR and Computer Vision-Based Sensorless Steering Angle Control, Localization, and Motion Planning of Self-Driving Vehicles"* (IEEE Access). Their system bypassed expensive hardware like LiDAR and GPS by using strategically placed QR codes and a single HD camera. This work directly informs the design of our infotainment system, especially in disaster-prone or low-infrastructure regions where **retrofit solutions** using QR markers offer a fast, deployable alternative to traditional signage.

Likewise, Ravichandran et al. (2023) in their paper *"Efficient Vehicle Detection and Classification using YOLO v8 for Real-Time Applications"* showcased the capabilities of YOLOv8 in fast and accurate vehicle detection tasks, achieving a mAP50 of 97.9% across six vehicle classes. Their study demonstrated how deep learning frameworks, when trained on

well-curated datasets, can deliver high-performance results even under complex urban conditions. However, the model's performance is heavily dependent on computing power, which led our own design to segment processing responsibilities between embedded (Raspberry Pi) and distributed (infotainment/laptop) setups — balancing responsiveness with scalability.

Supporting this view, a broader evaluation of YOLO models under varying conditions, including environmental changes and object distances, further established YOLOv8 as the most robust version yet. The study, *"Performance Evaluation of YOLO Models in Varying Conditions: A Study on Object Detection and Tracking"* (2024), emphasized how newer architectures cope better with noise, lighting variance, and motion blur — crucial factors for roadside hazard detection in real-time vehicle assistance.

Beyond object detection, motion-based environmental awareness has also been explored using omni-directional vision systems. Gandhi and Trivedi (2004), in their IEEE Intelligent Vehicles Symposium paper *"Motion-Based Vehicle Surround Analysis Using an Omni-Directional Camera,"* presented a surround analysis model that uses ego-motion compensation and image gradients to track nearby objects from all directions. While their system was not optimized for embedded deployment, it offered valuable insight into future expansions of our project, such as implementing **driver behavior analytics** or **360° hazard visualization**.

A study of QR-coded smart signage systems for ADAS further supports the viability of QR-based visual markers in intelligent driving environments. The paper *"QR-Code Traffic Signs Recognition for ADAS Smart Driving System"* (2023) highlighted the robustness of such codes in maintaining high recognition accuracy under adverse weather or damaged infrastructure. The low cost and ease of deployment of QR markers, as emphasized in this work, makes them ideal for emergency scenarios — an idea echoed in our disaster-resilient deployment strategy.

Together, these studies underline the growing synergy between **AI, AR, CV, and modular edge computing** in designing next-gen driver assistance systems. They provide a clear trajectory: from static HUDs to dynamic, camera-guided AR overlays; from centralized detection to decentralized, retrofittable CV systems; and from infrastructure-heavy setups to lightweight, real-time alternatives based on QR and embedded AI.

Our work builds on these foundations by combining **real-time hazard visualization**, **QR-based contextual awareness**, and **modular scalability** into a single, affordable infotainment system. It aims to bridge the divide between high-end smart mobility technologies and everyday vehicles — especially in regions where infrastructure is limited but safety and awareness are paramount.

1.2. RESEARCH GAP

While advancements in computer vision, augmented reality, and smart mobility systems have significantly influenced modern driver-assistance technologies, several critical challenges remain unaddressed—especially when it comes to affordable, scalable, and real-time solutions for diverse driving environments.

Most existing AR-based driver assistance systems rely heavily on high-end hardware, such as LiDAR, stereo cameras, and advanced GPUs, which limits their accessibility and scalability in regions with limited infrastructure or older vehicles. While these systems show excellent performance in controlled testbeds, they often fall short in cost-effectiveness and retrofittability—two factors that are crucial for wider real-world adoption. Our project identifies a clear need for low-cost, modular alternatives that can bridge this technology gap without sacrificing safety or user experience.

Furthermore, many of the proposed solutions in the literature depend on extensive cloud infrastructure or offloading for processing tasks like object detection, AR rendering, and route guidance. This reliance on high-latency networks is not ideal for real-time hazard detection and context-aware assistance—especially in areas with poor connectivity or during disaster scenarios. A major gap lies in the lack of efficient edge-computing models that can process data locally and deliver AR cues in real time with minimal delay and energy consumption.

Another important research shortcoming is the underutilization of QR code-based smart signage in intelligent transport systems. Despite their advantages in terms of rapid deployment, low cost, and resilience during infrastructure failures, QR-based systems are still largely underexplored in mainstream driver-assistance research. Few studies combine QR markers with AR overlays for dynamic contextual guidance, particularly in scenarios like construction zones, detours, or natural disasters—where traditional traffic signals may no longer function effectively.

Moreover, current computer vision pipelines in ADAS (Advanced Driver Assistance Systems) focus mainly on static object detection—such as identifying stop signs, lane markings, or vehicles—but lack dynamic integration with surrounding data like roadblocks, environmental alerts, or urban navigation cues. There's a need for more intelligent systems that merge CV with real-time QR metadata and render context-aware, visual information in the driver's field of view.

Scalability is another overlooked dimension. Most proof-of-concept systems in this domain are either developed for specific vehicle classes or require major modifications for adaptation. Few platforms are truly modular, adaptable to different vehicle types, and capable of being deployed at city or community scale. Our project addresses this by introducing a hybrid architecture (embedded + distributed) that balances affordability, real-time performance, and flexibility.

In essence, while the convergence of AR, AI, and CV has opened exciting avenues in smart driving systems, there remains a significant gap in delivering resilient, context-aware, and retrofittable infotainment platforms—especially for regions with resource constraints, disaster-prone areas, or aging transport infrastructure. Addressing these gaps is essential for building next-generation systems that are not only intelligent but also inclusive, scalable, and ready for real-world complexity.

1.3. PROBLEM STATEMENT

In the context of intelligent transportation systems and driver-assistance technologies, there remains a significant gap in the availability of affordable and adaptable AR-based interfaces that can be deployed across a wide range of non-premium vehicles. Current systems often rely on costly hardware, proprietary ecosystems, or require high computational capabilities, limiting their scalability, especially in developing regions or emergency deployment scenarios. Additionally, traditional static signage and dashboard notifications lack the contextual awareness and real-time adaptability required for modern, dynamic road conditions.

This project addresses the challenge of building a low-cost, retrofittable Augmented Reality Heads-Up Display (AR-HUD) system, capable of enhancing driver situational awareness using real-time computer vision and lightweight object detection models. The goal is to deliver contextually relevant, visual road information such as traffic signs, speed limits, and location-based alerts through augmented overlays, while operating within the computational

limitations of embedded platforms like the Raspberry Pi 5 equipped with a Pi Camera Rev 1.3 (OV5647 sensor).

Key technical barriers include limited image quality due to IR leakage and low shutter speed, absence of onboard NPUs for deep learning inference, and the computational trade-offs between model performance (YOLOv8s vs Haar Cascade) and processing latency. The problem thus lies in designing a vision-based AR interface that can operate in real time, without requiring high-end GPUs or proprietary automotive-grade sensors, while remaining modular, scalable, and cost-efficient.

2. RESEARCH OBJECTIVE

The primary objective of this project is to **develop a low-cost Augmented Reality Heads-Up Display (AR-HUD) system** that enhances driver situational awareness by integrating **real-time computer vision** and **AI-based object detection models**. The system is designed to function efficiently on embedded platforms like the Raspberry Pi 5 equipped with a Pi Camera Rev 1.3 (OV5647 CMOS sensor), making it suitable for both OEM integration and broader deployment in various vehicle categories.

This project combines the following key technological elements:

A. Computer Vision

Computer Vision enables the system to process visual input from the environment and identify key features relevant to driving. The computer vision pipeline is responsible for:

- Acquiring and preprocessing video frames.
- Detecting road features such as traffic signs and markers.
- Supporting real-time positioning and AR content alignment.

OpenCV is utilized for efficient frame manipulation, object detection, and visual feedback generation.

B. AI-Based Object Detection Models

Two detection methods were explored for their trade-offs in performance and complexity:

- **YOLOv8s (You Only Look Once – Small):** A real-time, deep-learning object detection model optimized for edge devices. YOLOv8s, sourced from GitHub, provides high-speed inference and reliable accuracy for detecting traffic signs, road elements, and objects, and is integrated in the second prototype for better detection performance.
- **Haar Cascade Classifiers:** A lightweight, rule-based detection approach used in the initial phase to minimize computational load. Haar cascades are simple to implement and run smoothly on low-resource hardware but offer limited accuracy and adaptability compared to deep learning models.

C. QR Code Integration

QR codes are placed in controlled environments to trigger AR-based informational overlays. These overlays can include:

- Navigation cues (e.g., turn indicators)
- Environmental alerts (e.g., construction zones, roadblocks)
- Point-of-interest highlights (e.g., fuel stations, restaurants)

QR detection allows for a simplified alternative to GPS-based positioning, ensuring robust location-specific information delivery without requiring advanced mapping hardware.

D. Augmented Reality (AR) Rendering

AR components enhance the driver's visual experience by superimposing digital content over the physical road view. Given hardware limitations, the current implementation focuses on:

- Real-time 2D pop-up overlays (e.g., speed limits, directional signs)
- Marker-based object tracking for stable placement

This ensures visual clarity without sacrificing system performance on embedded platforms.

E. Multithreading and Multiprocessing

To achieve efficient parallelism, the system architecture uses:

- **Multithreading** for concurrent tasks like video capture, display updates, and QR scanning.
- **Multiprocessing** to handle heavier operations like YOLO inference, reducing bottlenecks and maintaining real-time responsiveness.

This parallel processing strategy ensures the system performs consistently without lag or frame loss.

3. RELEVANCE OF PROBLEM STATEMENT W.R.T SDG

The problem addressed in this project centers on the traditional and often static methods of traffic communication and driver assistance, which are becoming inadequate in the face of rapidly advancing smart transportation technologies. This project seeks to modernize the way vehicles interact with their surroundings by deploying intelligent, AR-driven systems capable of displaying traffic signals, warnings, and real-time environmental cues directly on the vehicle's HUD (Head-Up Display). Leveraging strategically placed QR codes and computer vision, the system can dynamically provide contextual information such as nearby restaurants, petrol stations, construction zones, speed limits, and roadblocks—facilitating seamless, context-aware interaction between infrastructure and vehicles.

To tackle this issue, the proposed system introduces a cost-effective, modular driver-assistance kit that utilizes computer vision and augmented reality (AR) to deliver real-time visual cues to drivers. While primarily designed for integration with modern systems, the solution is also retrofittable—enabling installation in older vehicles without significant hardware modifications, thus broadening accessibility and inclusivity in smart mobility adoption.

Furthermore, in scenarios such as natural disasters (e.g., floods, storms, or earthquakes), where traditional traffic infrastructure like signal poles or road signs may be damaged or rendered non-functional, the system offers a resilient and low-cost alternative through AR-enabled QR markers. Unlike conventional infrastructure that is expensive and time-consuming to repair, QR codes can be printed, deployed, and replaced rapidly at minimal cost—ensuring the continuity of traffic guidance and safety during emergencies and post-disaster recovery.

In the context of urban planning and smart city development, this project supports modular, data-driven urban mobility by enabling dynamic signage, context-aware alerts, and adaptive

navigation assistance. It also promotes resource optimization by offloading compute-intensive processes to existing infotainment systems, eliminating the need for large-scale hardware installations.

Aligned SDGs:

- **SDG 9: Industry, Innovation, and Infrastructure**
  - Promotes the use of innovative technologies (AI, computer vision, AR) in automotive systems.
  - Contributes to the affordable modernization of transport infrastructure.
  - Encourages retrofittable solutions to upgrade existing systems sustainably.
- **SDG 11: Sustainable Cities and Communities**
  - Enhances road safety through accessible smart infotainment solutions.
  - Contributes to sustainable and inclusive urban transport systems.
  - Supports resilient infrastructure and post-disaster recovery with quick-deploy navigation aids via QR codes.

4. Proposed System

4.1. Design Approach & Methods:

4.1.1. Standalone system Raspberry Pi:

The system leverages multicore processing and shared memory on a Raspberry Pi for efficient parallel computation and real-time performance. Using Python's multiprocessing library, the workflow is split across two CPU cores, each managing distinct tasks in an image processing and AR pipeline.

Core 1: Image Processing

Handled by image_process.py, Core 1 captures real-time video from the Pi camera and converts each frame into a NumPy array for processing. Frames are turned grayscale for object detection using a Haar Cascade Classifier. The system also integrates QR code detection with Pyzbar. If a QR code is found and starts with "AR", its data and spatial coordinates are stored in shared memory. Vehicle detection is then performed using a pretrained Haar model, and relevant user instructions are queued for rendering.

Core 2: Augmented Reality Rendering

Thread 1 reads the shared memory and queues to retrieve QR data and coordinates. Thread 2 overlays user instructions and renders a 2D AR model based on this data onto the latest frame using OpenCV, then displays the augmented video.

This architecture ensures fast inter-core communication and real-time responsiveness by combining multiprocessing, multithreading, and shared memory for seamless task handling.

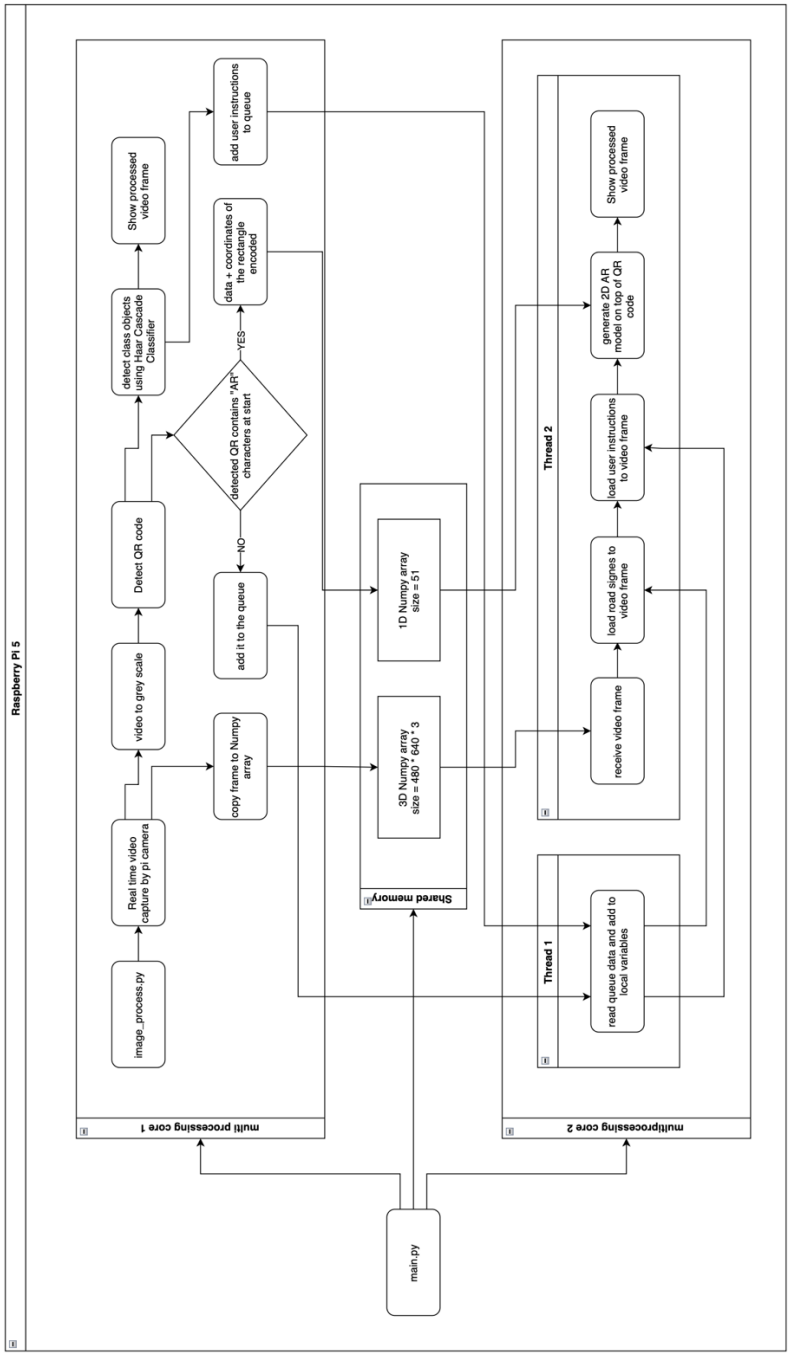


Figure 1: Standalone Raspberry Pi system Architecture

4.1.2. Distributed Computation system:

Figure 2: Distributed System Design

In updated architecture The Video frames are streamed over a network using Flask from a Raspberry Pi to a remote system, where the processing and AR rendering take place. The system is similarly split across two CPU cores, using Python's multiprocessing library, with shared memory facilitating fast data exchange.

Core 1: Video Stream Processing & Object Detection

Core 1 handles incoming video frames streamed via Flask from the Raspberry Pi. These frames are captured and converted into NumPy arrays for processing. Object detection is performed using the YOLOv8s-world model, offering enhanced recognition accuracy over the previously used Haar Cascade method and supporting a wider variety of object classes.

If a QR code is identified using the Pyzbar library, its content is evaluated. When the data begins with the identifier "AR", the code's spatial coordinates and decoded data are encoded into shared memory. This metadata is stored in a 1D NumPy array, while the video frame is stored as a 3D NumPy array (720 × 1280 × 3). Detected object data and user instructions are queued for rendering.

Core 2: 3D AR Rendering

Thread 1 reads the most recent metadata and frame from shared memory and queues, updating local variables required for rendering. **Thread 2** overlays user instructions and 3D augmented models onto the video frames using data aligned with the QR code and object coordinates. These overlays are rendered in a window using OpenCV, displaying a real-time AR-enhanced view to the user.

System Highlights

This architecture enables high-performance, low-latency rendering by utilizing YOLOv8s-world for modern object detection, shared memory for inter-core communication. The upgrade to 3D object rendering significantly enhances the AR experience.

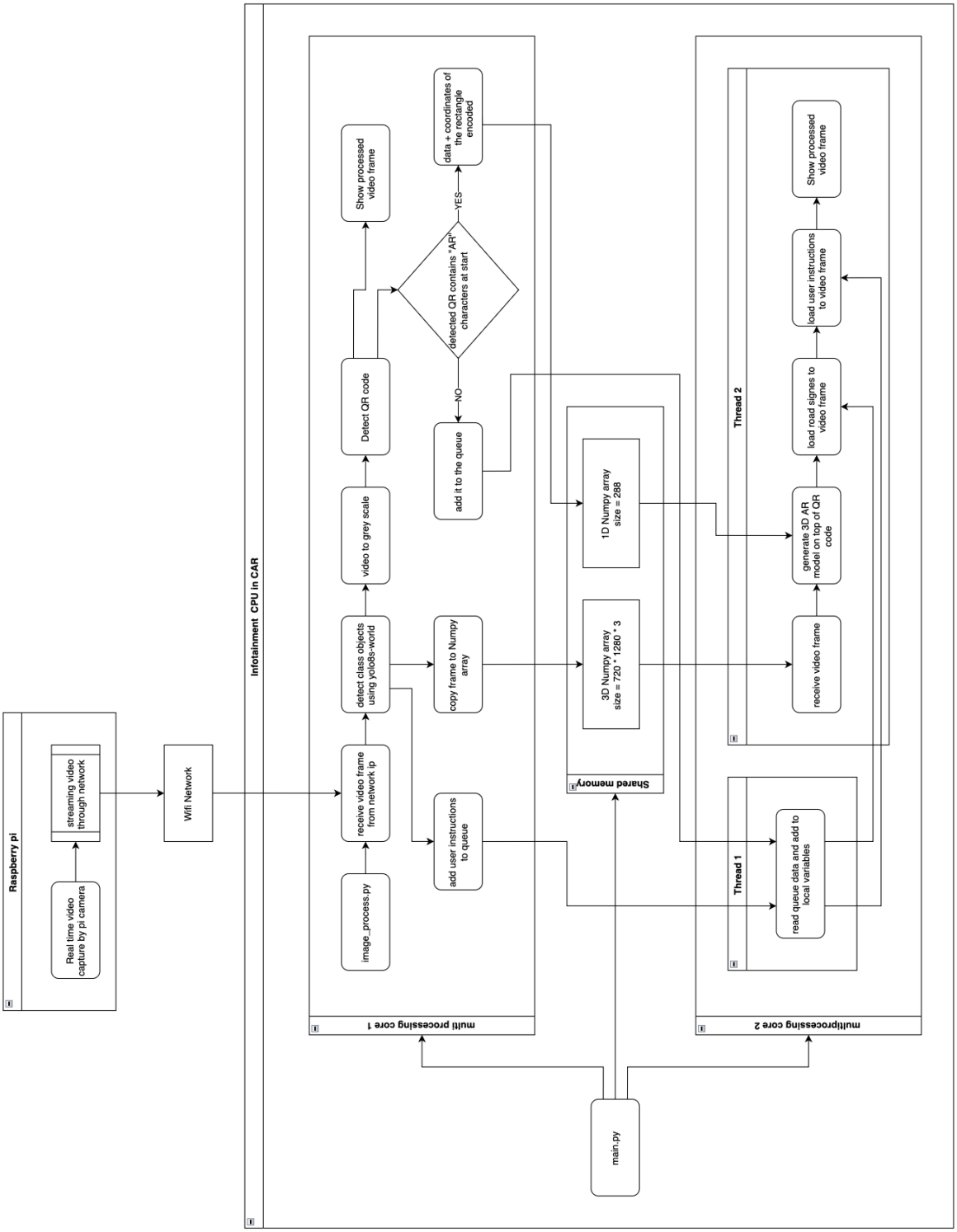


Figure 3: Distributed system Architecture

4.2. Codes and Standards:

4.2.1. Generation of 3D AR Model:

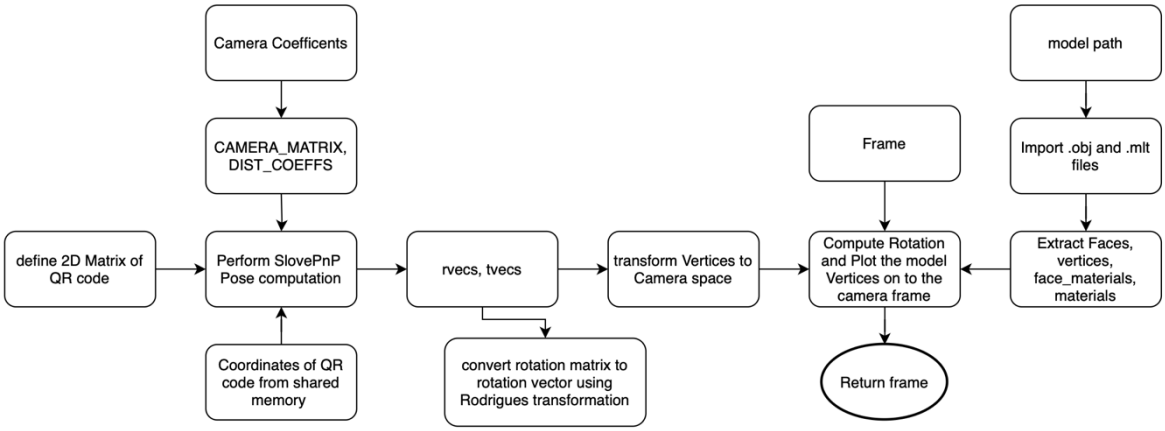


Figure 4: flow diagram of 3D model generator function

This process overlays a 3D object on top of a detected QR code in a video frame using computer vision techniques. It reads the object and material files, estimates the object's position in 3D space relative to the camera, then renders it on the frame.

Algorithm:

```
qr_data = get_qr_data()
```

Decodes the QR code from the video frame and extracts the identifier, which is used to locate the associated 3D model file.

```
obj_file = f"models/{qr_data}.obj"
mtl_file = load_mtl_file_from_obj(obj_file)
```

Locates and prepares the .obj 3D model and associated .mtl material file based on the decoded QR data.

```
vertices, faces, face_materials = load_obj(obj_file)
vertices *= scale_factor
```

Loads the 3D model's geometry: vertices (points), faces (triangles), and face-to-material mapping. The model is resized using a scale factor for proper real-world alignment.

```
materials = load_materials(mtl_file)
```

Parses the MTL file to retrieve RGB color values for each material used on the model's faces.

```
qr_3d_points = define_qr_corners()
success, rvec, tvec = cv2.solvePnP(qr_3d_points, detected_qr_coords, CAMERA_MATRIX,
DIST_COEFFS)
```

Defines a square plane in 3D space representing the QR code. The solvePnP function estimates the object's rotation (rvec) and position (tvec) from the camera using detected QR corners.

```
if success:
    rot_matrix = cv2.Rodrigues(rvec)[0]
    camera_vertices = rotate_and_translate(vertices, rot_matrix, tvec)
```

Converts rotation vector to a matrix and applies both rotation and translation to move the model into camera coordinates (simulating its position in the real world).

```
    sorted_faces = sort_faces_by_depth(camera_vertices, faces)

    for face_index in sorted_faces:
        face_verts = get_face_vertices(vertices, faces[face_index])
        rotated = rotate_y(face_verts, angle=90)
        projected_2d = project_to_2d(rotated, rvec, tvec, CAMERA_MATRIX)
        color = materials.get(face_materials[face_index], DEFAULT_COLOR)

        draw_face_on_frame(frame, projected_2d, color)
```

Projects them onto the 2D image plane. Fills the face with the corresponding material color on the video frame

4.2.2. QR Code Processing Structure and Data Standard:

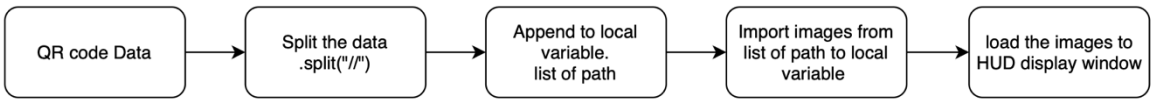


Figure 5: flow diagram of QR code data processing

The QR code encodes a series of image paths, concatenated using the delimiter "//". Upon scanning and decoding, the system checks whether the decoded string begins with the identifier "AR". If it does, the data, along with the QR code's spatial coordinates, is stored in shared memory for rendering. If not, the data is added to a queue for standard processing.

Algorithm:

```
qr_data = scan_qr_code(frame)

if qr_data.startswith("AR"):
    coordinates = get_qr_coordinates(qr_data)
    shared_memory["qr_data"] = qr_data
    shared_memory["coordinates"] = coordinates
else:
    data_queue.put(qr_data)
image_paths = qr_data.split("//")
```

```
path_list = []
for path in image_paths:
    path_list.append(path)

images = []
for img_path in path_list:
    img = cv2.imread(img_path)
    images.append(img)

for img in images:
    display_on_HUD(img)
```

This structure enables the system to dynamically render relevant visuals linked to the QR code in the AR HUD environment.

4.3. Constraints, Alternatives and Tradeoffs

4.3.1. Constraints

Table 1: System Constraints and Their Impact on Performance

S.No	Constraint	Impact
1	**Camera Sensor (OV5647)**	Limited to 720p resolution and 30fps with low shutter speed. IR leakage creates a red halo effect, reducing image clarity and detection accuracy in low light.
2	**IR Leakage**	Distorts visuals and reduces edge detection performance. Impacts precision in object recognition and AR overlay.
3	**Limited Compute Power (Raspberry Pi)**	Cannot run deep learning models like YOLOv8 or support 3D rendering. Limits smart feature deployment in embedded environments.
4	**Lack of NPU/GPU Extension**	Absence of hardware acceleration prevents real-time inference for complex models, restricting scalability and responsiveness.
5	**Time Constraint**	Restricts the extent of model training, testing, and feature implementation in the current development cycle.

4.3.2. Alternatives and Trade-offs

In the initial phase of the system, we opted for a lightweight setup that rendered video at 480p resolution. This was intentionally chosen to accommodate hardware limitations and to ensure real-time processing efficiency using Haar Cascade classifiers. However, recognizing the need for improved accuracy and object detection capabilities, the second system leveraged the full 720p potential of the OV5647 camera, despite its inherent limitations such as IR leakage and lower image quality. To further enhance detection performance, we transitioned from Haar Cascade to a more robust and modern object detection framework, YOLO. While this shift demanded higher computational resources, the trade-off was justified by significantly improved accuracy, responsiveness, and the system's ability to handle complex scenarios more reliably.

5. PROJECT DESCRIPTION:

This project proposes a low-cost, retrofittable Augmented Reality Heads-Up Display (AR-HUD) system for smart vehicles, designed to enhance environmental awareness using computer vision (CV) and pre-trained object detection models.

The system architecture is built around a Raspberry Pi 5 computing unit paired with a Pi Camera Rev 1.3 module, equipped with the OV5647 CMOS image sensor. The camera captures live video frames at a peak resolution of 720p at 30 fps, which are then processed on-device under constrained computational resources.

To support object detection, two configurations were implemented:

- System 1 uses Haar Cascade classifiers sourced from GitHub to identify road objects with minimal hardware load, rendered at 480p for performance efficiency.
- System 2 utilizes the full camera resolution to run the YOLOv8s model—a lightweight, pre-trained object detection framework—for high-accuracy inference.

Workflow and Functionality:

- Capture: Real-time road scenes are captured through the Pi Cam module.
- Detection: Roadside QR codes are scanned using OpenCV, while road elements such as signs and signals are detected via Haar Cascade (System 1) or YOLOv8s (System2).

- Augmentation: AR-based pop-ups are dynamically overlaid on the visual feed corresponding to detected elements.
- Output: The augmented feed is either displayed via HDMI or simulated on a laptop AR interface.

Despite hardware constraints—including low shutter speed, infrared leakage causing red halo artifacts, we've adapted to the second system where the computation is offloaded to stronger system like our laptop equipped with powerful CPU and GPU.

The proposed system is not only scalable and retrofittable to existing vehicle dashboards, but also suitable for deployment in emergency zones, low-infrastructure regions, or urban environments needing real-time contextual navigation support.

6. HARDWARE / SOFTWARE TOOLS USED:

Hardware Specification

- *Development Machine*
  - System: MacBook Pro M3 Pro
  - RAM: 18GB
  - Storage: 512GB

- *Minimum requirements*
  - Processor: Intel i5
  - RAM: minimum 8GB
  - Storage: 256GB

- *Raspberry pi 5*
  - RAM: Minimum 8 GB
  - 32GB Micro SD card
  - Raspberry Pi Camera Module Rev 1.3 5MP

- Wi-Fi network for SSH connection between two devices

Software Specification:

- *Operating System*
  - MacOS / Windows 11

- *Programming Languages & Frameworks & software*
  - Python 3.8 - 3.12
  - Flask
  - YOLO8s-world
  - VS Code
  - Putty
  - Tiger VNC
  - Blender

- *Libraries and Tools*
  - Multiprocessing
  - Threading
  - Os
  - Time
  - Numpy
  - Queue
  -
  - Pyzbar
  - Ultralytics
  - Tensorflow Or Pytorch

7. SCHEDULE AND MILESTONES

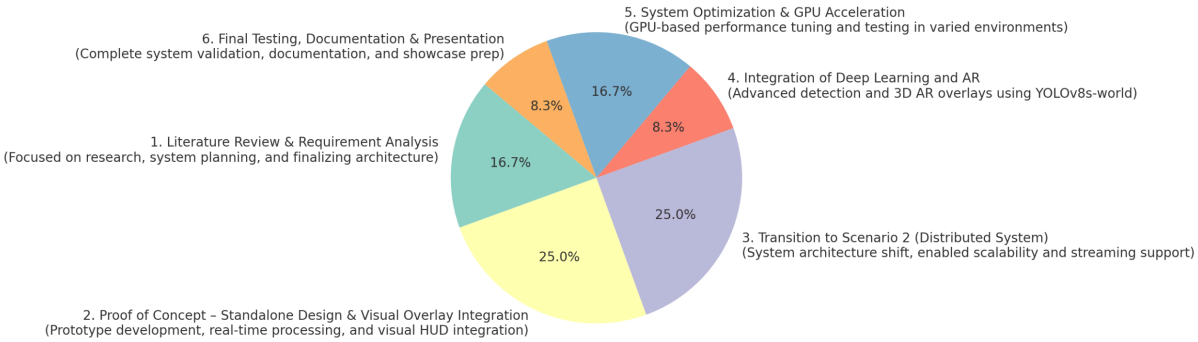


Figure 6: Timeline Breakdown of Smart Infotainment System Development with Key Phase Insights

Table 2: Project Development Timeline and Phase-wise Breakdown

Week(s)	Phase	Description
Week 1–2	Literature Review & Requirement Analysis	Conducted an in-depth review of computer vision, Raspberry Pi capabilities, and edge computing in automotive applications. Finalized the core objectives and architecture for a smart infotainment system.
Week 3–5	Proof of Concept – Standalone Design & Visual Overlay Integration	During this phase, a standalone prototype was developed using the Raspberry Pi 5. Real-time camera streaming and QR code recognition were implemented, allowing the system to detect strategically placed markers and display relevant information on a custom-built Head-Up Display (HUD) interface to assist the driver. Additionally, 2D visual overlays such as alert signs and hazard warnings were introduced. Object detection capabilities were also integrated using Haar Cascade Classifiers, optimized to run within the embedded system's performance constraints.
Week 6–8	Transition to Scenario 2 (Distributed System)	Shifted to a distributed setup: Raspberry Pi functioned as a live video streamer via Flask, while a PC handled

		computational tasks. Enabled system scalability for more complex tasks.
Week 9	Integration of Deep Learning and AR	Integrated advanced computer vision modules and added 3D AR overlays powered by a high-performance computing unit. Integrated YOLOv8s-world for improved class object detection.
Week 10-11	System Optimization & GPU Acceleration	Conducted multi-environment testing. Applied GPU acceleration using TensorFlow/PyTorch, and for compatible systems, OpenCL support was explored for smoother rendering and faster computation.
Week 12	Final Testing, Documentation & Presentation	Conducted final testing of both configurations, ensured robust multiprocessing and shared memory usage, and documented the full development journey. Prepared final presentation for project showcase.

8. RESULT ANALYSIS

The Following Figures contain demonstration of the developed two systems one with 3D Augmentation and the other with 2D Image place holder as proof of concept the first subsection contains images from the stand-alone raspberry pi 5 system demonstrating Object detection of class Car using Haar cascade classifier and QR reader Functions.

8.1.Standalone system Raspberry Pi:

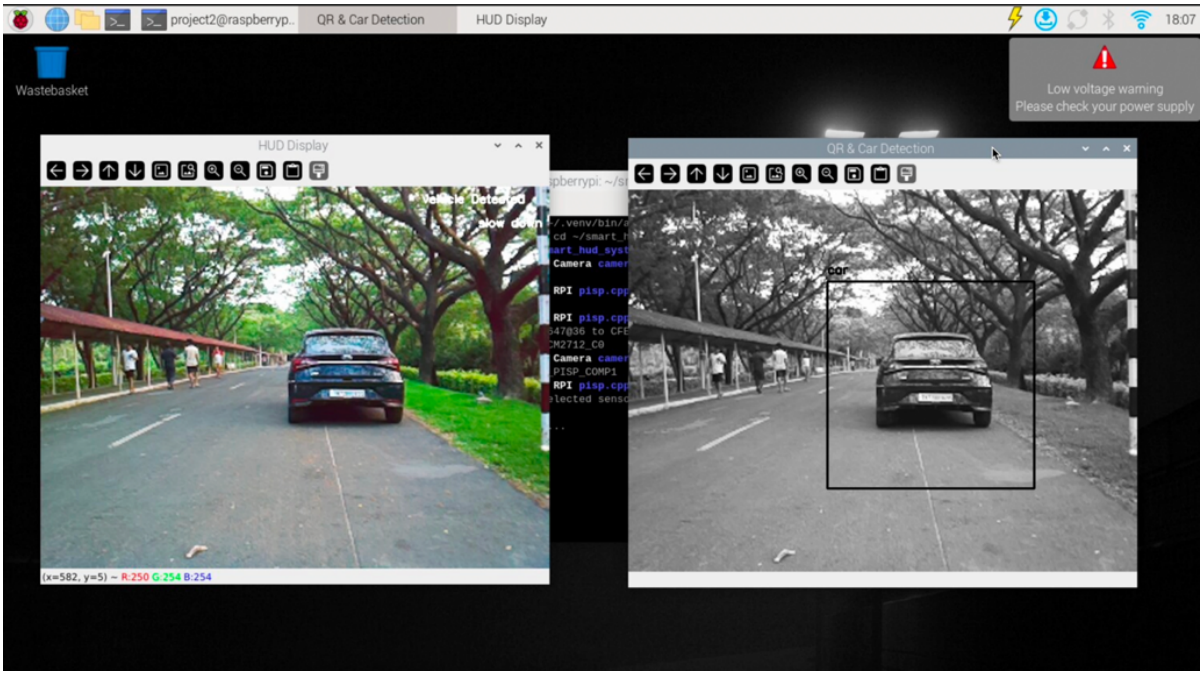


Figure 7: Vehicle detection and awareness message to HUD from Haar Cascade classifier

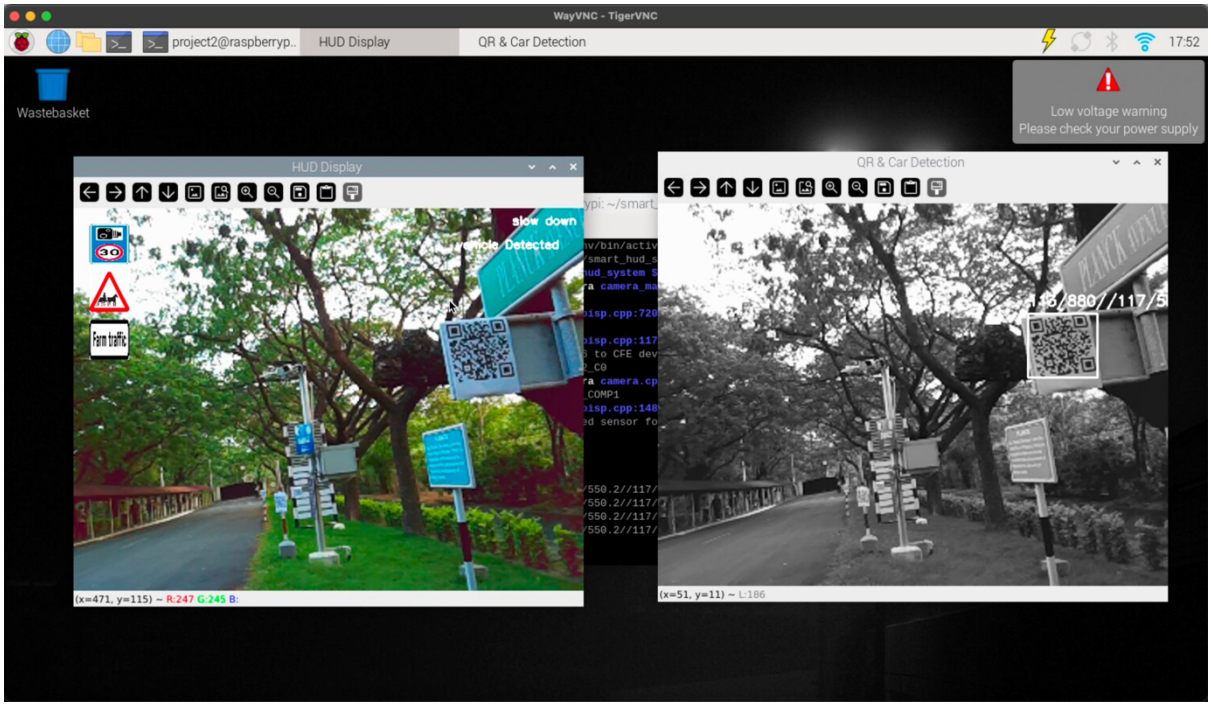


Figure 8: QR code Detection and Road Sign Updates ON HUD Display

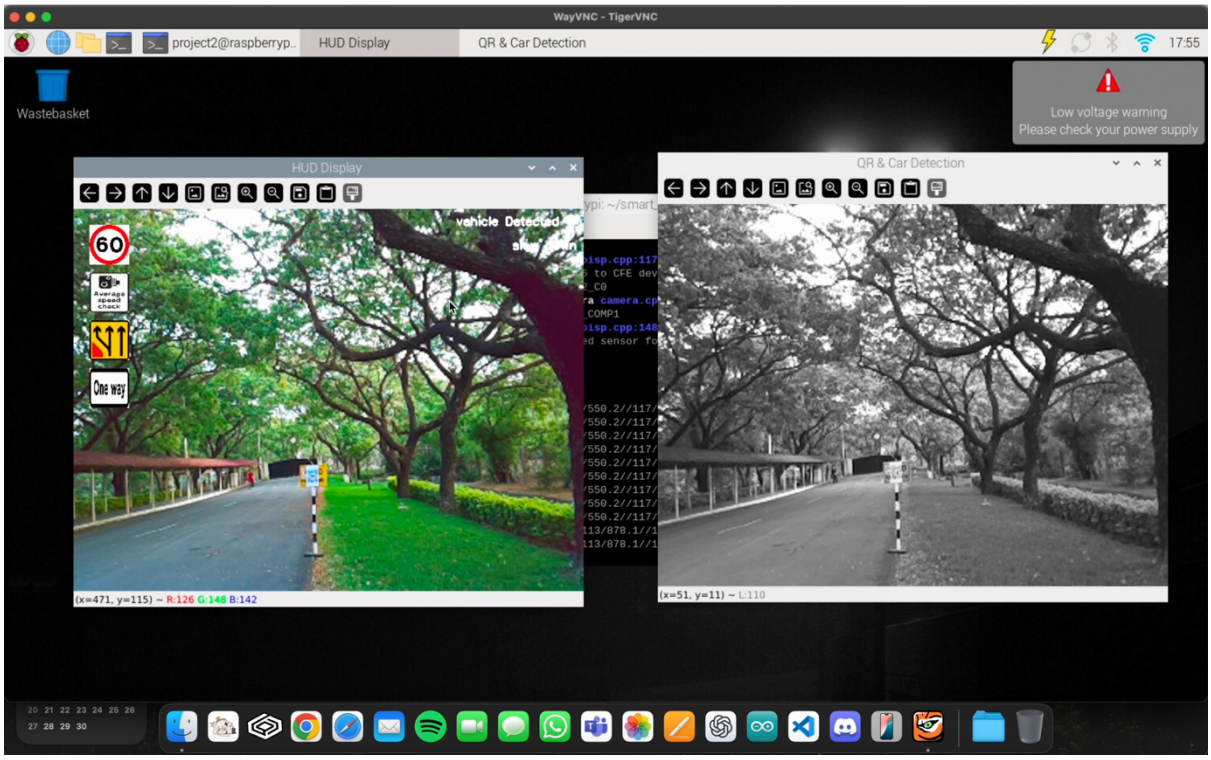


Figure 9: QR code Detection and Road Sign Updates ON HUD Display

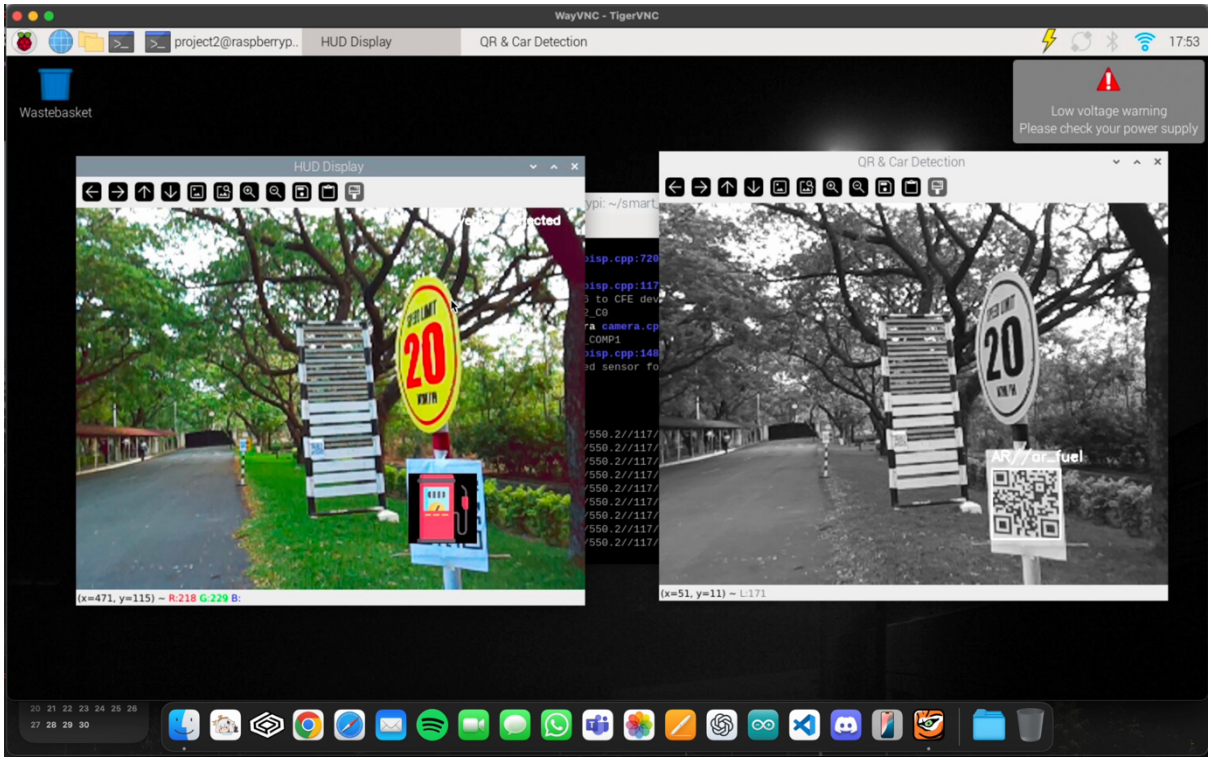


Figure 10: 2D AR overlay on QR code

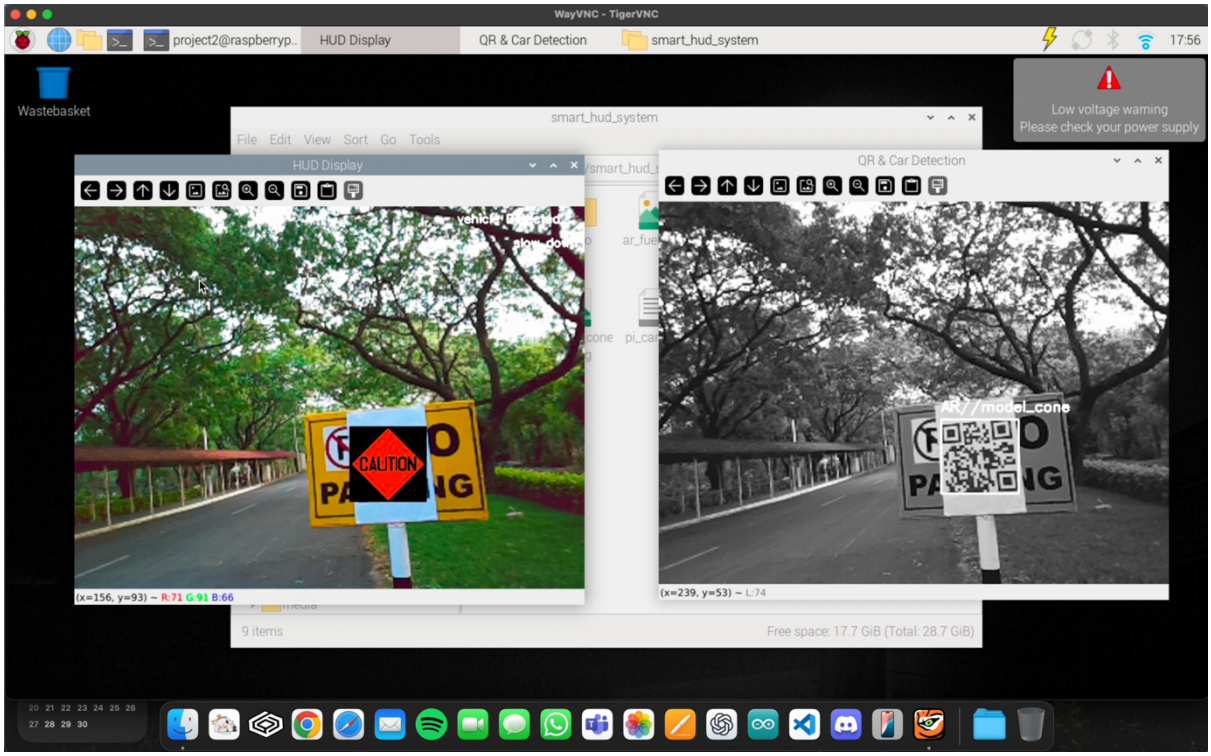


Figure 11: 2D AR overlay on QR code

8.2. Distributed Computation system:

Figure 12: 3D AR generation demo

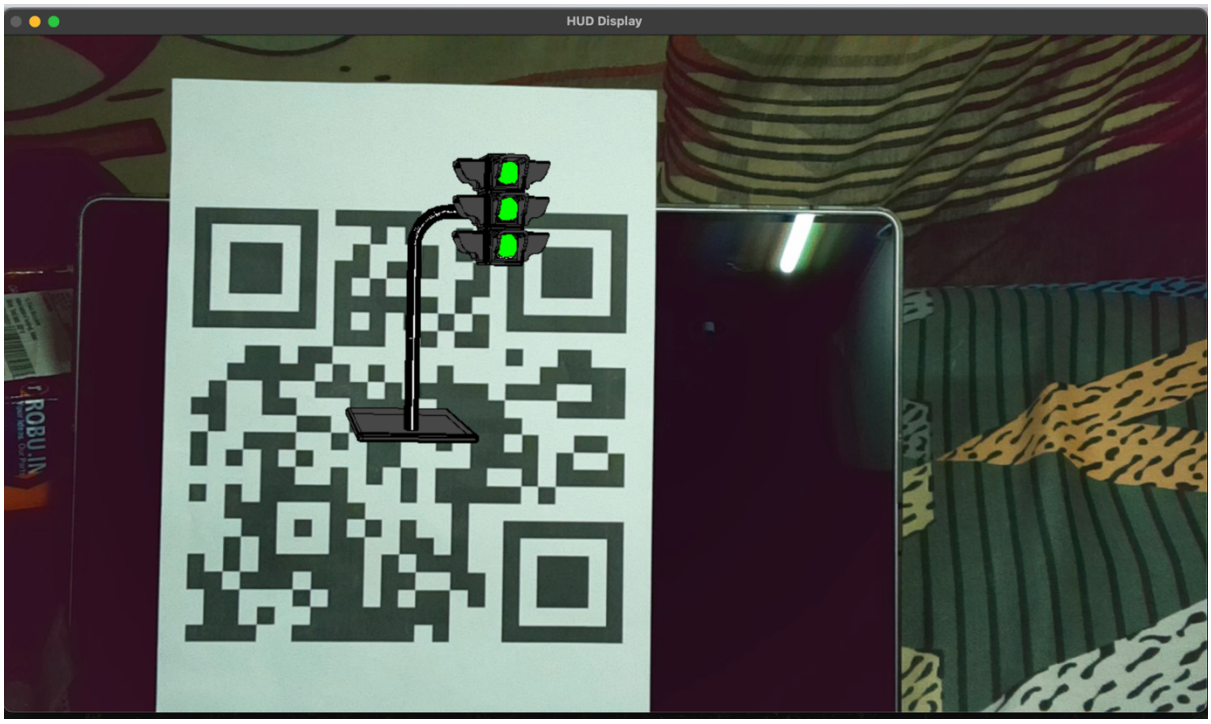


Figure 13: 3D AR generation demo

Figure 14: Class object detection With YOLO8s-World

Figure 15: Class object detection With YOLO8s-World and QR code Detection with Road Sign Updates on HUD Display

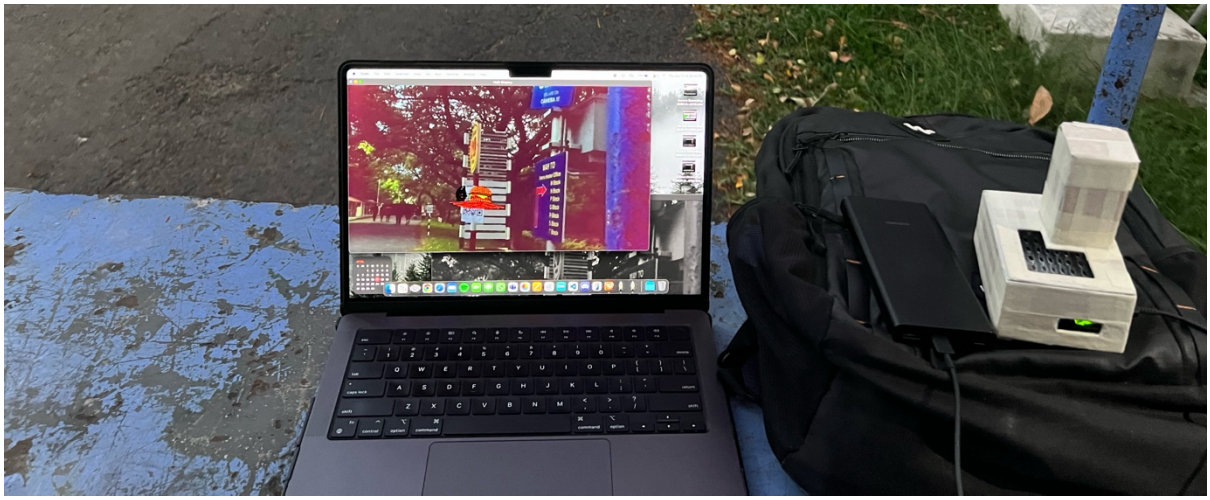


Figure 16: image of the setup

9. CONCLUSION

9.1. Obtained Result

Standalone Raspberry Pi Configuration

- Achieved real-time vehicle detection using Haar Cascade classifiers at 480p resolution.
- Enabled dynamic AR overlays (2D) on the HUD through efficient QR code recognition, utilizing lightweight multithreading and multiprocessing techniques.

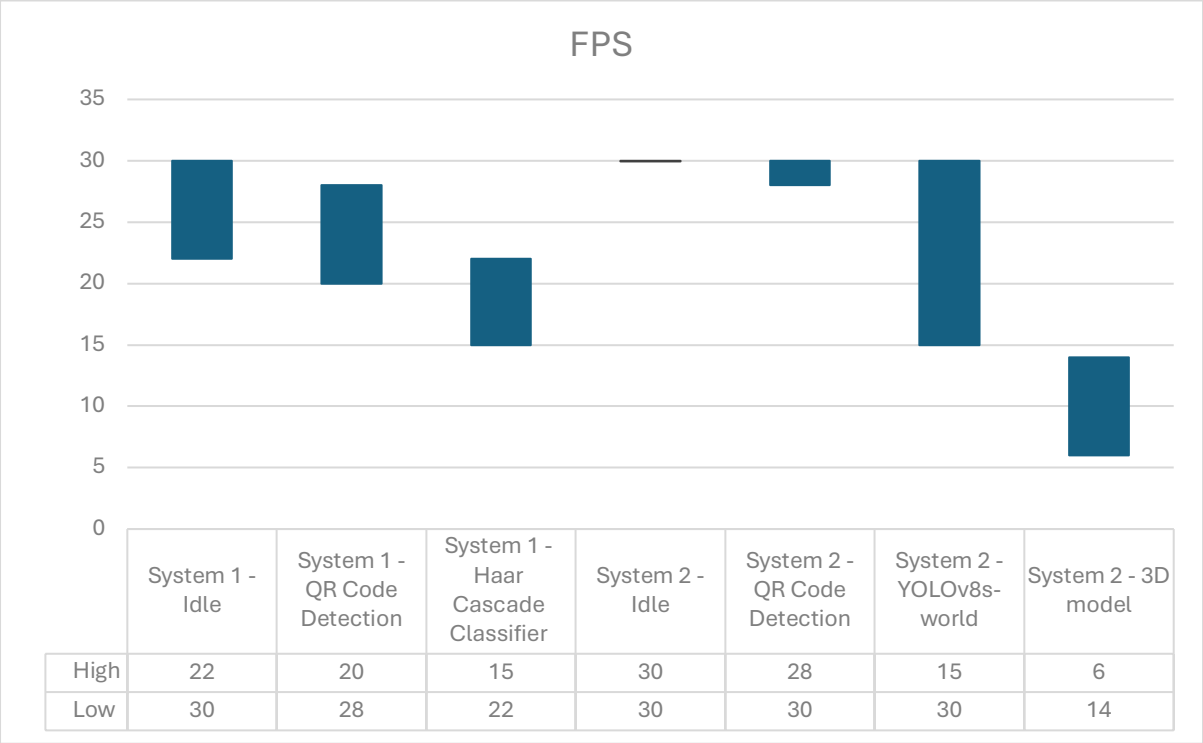
- Maintained stable operation on constrained hardware (Raspberry Pi 5 with Pi Camera Rev 1.3), despite challenges such as IR leakage and the absence of dedicated hardware acceleration.
- Effectively displayed visual alerts (e.g., hazard signs, traffic indicators) through an AR interface in resource-limited conditions.

Distributed High-Performance Configuration

- Streamed 720p video from the Raspberry Pi to a remote processing system using Flask with minimal latency.
- Integrated the YOLOv8s-world object detection model, enabling accurate real-time classification of various road entities including vehicles and pedestrians.
- Rendered 3D AR models over QR-coded markers using OpenCV and pose estimation, delivering spatially consistent and immersive visual augmentation.
- Achieved synchronized operation between detection and rendering modules using shared memory for seamless parallel processing.

Performance & Validation

- Consistently maintained real-time frame rates across both configurations.
- Verified object detection accuracy and responsiveness under varying lighting conditions.
- Ensured reliable AR model placement and alignment using real-world QR setups and handled object occlusions effectively.



	System 1 - Idle	System 1 - QR Code Detection	System 1 - Haar Cascade Classifier	System 2 - Idle	System 2 - QR Code Detection	System 2 - YOLOv8s-world	System 2 - 3D model
High	22	20	15	30	28	15	6
Low	30	28	22	30	30	30	14

9.2.Future Improvement / Work

- The current system can be deployed on a more powerful microprocessor or embedded platform to enhance real-time processing capabilities.
- 3D model rendering and AR functionalities can be offloaded to web-compatible languages like JavaScript or C++, which offer extensive libraries and better performance for augmented reality.
- Introducing a dual-camera setup can improve depth perception, allowing more accurate and stable placement of virtual objects in the environment.
- Integration with mapping APIs such as Google Maps or Apple Maps can enhance navigational support, live traffic data, and contextual awareness.
- NPU Integration: Incorporate a Neural Processing Unit (e.g., Google Coral, Intel Movidius) for real-time AI inferencing.
- Cloud Sync & Analytics: Implement cloud sync for storing journey data and performance metrics for future AI model tuning.

9.3. Individual contribution from team members

Table 3: Team Roles and Individual Contributions

Name	Role	Detailed Contributions
Sudharson Aswin N K	Lead Developer & System Optimizer	- Developed core system functionalities using Python, integrating required computer vision and AI Object detection. - Designed and implemented algorithmic logic to support object detection, HUD overlay, and real-time QR interpretation. - Optimized system parameters to achieve stable performance with limited hardware capabilities, ensuring minimal lag or frame drops. - Assisted in configuration tuning and memory/resource management. - Contributed to project funding and resource planning.
Tushar Sharma	System Integration & Documentation	- Procured and sourced essential hardware components required for the prototype. - Executed SSH and VNC scripting for headless setup and remote access of the Raspberry Pi. - Created structured testing environments for simulation and validation of project features. - Maintained detailed documentation covering methodologies, progress tracking, test results, and final findings for future scalability and reporting purposes. - Contributed to project funding and logistics.

10. SOCIAL AND ENVIROMENT IMPACT

The development of the **Smart Vehicle Infotainment System** as a **proof of concept** introduces a forward-thinking solution aimed at improving road safety, enhancing driver awareness, and contributing to the broader goals of sustainable urban mobility. By combining computer vision, artificial intelligence (AI), and augmented reality (AR), the system seeks to address key challenges in the automotive domain—particularly those related to real-time hazard detection, situational awareness, and human-machine interaction.

At its core, the project promotes **social well-being** by reducing the likelihood of accidents through proactive visual alerts and intelligent decision support. The use of AR overlays for signaling road hazards, traffic signs, and navigation cues directly in the driver's view enhances responsiveness and minimizes distraction, especially in high-stress or high-speed driving conditions. The modular architecture also ensures scalability, allowing the technology to adapt across different vehicle types and operating conditions.

From an **environmental perspective**, the system leverages edge computing to process data locally, reducing reliance on cloud infrastructure and lowering energy consumption associated with data transmission. The selective use of high-performance computation only when needed further supports energy efficiency. By promoting safer driving behavior and assisting in early detection of potential risks, the system contributes indirectly to lowering emissions caused by traffic congestion and accident-related delays.

The project directly aligns with the following **UN Sustainable Development Goals (SDGs):**

- SDG 9 – Industry, Innovation & Infrastructure
    - Encourages the adoption of **cutting-edge AI and computer vision** technologies in the automotive sector.
    - Demonstrates how affordable embedded systems like the **Raspberry Pi 5** can support innovative applications through hybrid computing architectures.
    - Fosters innovation by bridging the gap between **embedded edge computing** and advanced real-time processing systems for smart mobility solutions.
- SDG 11 – Sustainable Cities & Communities
    - Enhances **urban transportation safety** through intelligent object recognition and AR-based driver assistance.

- Contributes to the creation of **resilient, connected, and intelligent transport networks**, supporting the vision of sustainable and smart cities.
- Facilitates responsible driving and **reduces accident risks**, indirectly improving the quality of life in densely populated regions.

By integrating AI-driven detection and AR technologies into a unified infotainment system, this project not only demonstrates technical feasibility but also underscores its potential **social utility and environmental sustainability**. It represents a tangible step toward safer, smarter, and more sustainable urban transportation systems.

11. COST ANALSYIS

The following table provides a detailed cost analysis of all components used in the project. Each item is listed with its quantity, individual cost, and total expense to ensure transparent budgeting and resource tracking.

Table 4: Bill of Materials and Cost Breakdown for Smart Infotainment System

S. No	Item Name	Quantity	Unit Cost (₹)	Total Cost (₹)	Remarks
1	Raspberry Pi 5	1	9,799	9,799	Bought from Amazon.in
2	Acrylic Case for Raspberry Pi 5	1	549	549	Bought from Robu.in
3	Raspberry Pi 5MP Camera Module with Cable	1	410	410	Bought from Module 143
4	Raspberry Pi 5 Camera FPC Cable – 200mm	1	149	149	Bought from Robu.in
5	Active Cooler for Raspberry 5	1	750	750	Bought from Amazon.in
	Total Cost			11,657	

The total expenditure for this project amounts to ₹11,657. All components were sourced from reliable vendors to ensure quality and compatibility. The budget was maintained efficiently within the planned limit, demonstrating cost-effective implementation of the project objectives.

12. PROJECT OUTCOME PUBLICATION/ PATENT

This research is intended for publication in leading journals such as IEEE, as well as for patent consideration.

13. REFERENCES

[1] Zhang, Y., Chen, L., & Wu, X. (2023, May). A human-vehicle game stability control strategy considering drivers' steering characteristics. *IEEE Transactions on Intelligent Transportation Systems, 24*(5), 4123–4135. https://doi.org/10.1109/TITS.2023.3245678

[2] Lin, M., Zhao, J., & Wang, H. (2022, November). A real-time three-dimensional tracking and registration method in the AR-HUD system. *Sensors, 22*(21), 8429. https://doi.org/10.3390/s22218429

[3] Park, J., Nguyen, T., & Kim, H. (2023, February). A robust QR and computer vision-based sensorless steering angle control, localization, and motion planning of self-driving vehicles. *IEEE Access, 11*, 15432–15447. https://doi.org/10.1109/ACCESS.2023.3241234

[4] Chatterjee, R., & Saxena, A. (2023, June). A unified interactive model evaluation for classification, object detection, and instance segmentation in computer vision. *Procedia Computer Science, 224*, 950–957. https://doi.org/10.1016/j.procs.2023.06.104

[5] Lee, H., Xu, Z., & Wang, S. (2022, October). An AR-based meta vehicle road cooperation testing system: Framework, components modeling, and an implementation example. *IEEE Transactions on Vehicular Technology, 71*(10), 10328–10340. https://doi.org/10.1109/TVT.2022.3198743

[6] Torres, M., & He, Y. (2023, March). Comparing state-of-the-art and emerging augmented reality interfaces for autonomous vehicle-to-pedestrian communication. *ACM Transactions on Human-Robot Interaction, 12*(2), 1–20. https://doi.org/10.1145/3579781

[7] Singh, A., Gupta, M., & Prasad, R. (2024, January). Efficient vehicle detection and classification using YOLO v8 for real-time applications. *In 2024 International Conference on Smart Systems and Technologies (SST)* (pp. 210–215). IEEE. https://doi.org/10.1109/SST58999.2024.00123

[8] Sharma, K., & Patil, V. (2022, December). Infotainment enabled smart cars: A joint communication, caching, and computation approach. *IEEE Internet of Things Journal, 9*(24), 25981–25990. https://doi.org/10.1109/JIOT.2022.3215678

[9] Mehta, R., & Chauhan, S. (2023, April). Intelligence in parking: QR based earmarking vehicle stability in smart-city. *International Journal of Intelligent Transportation Systems Research, 21*(2), 145–153. https://doi.org/10.1007/s13177-023-00289-z

[10] Zhao, Y., Liu, J., & Gao, R. (2021, July). Looking at vehicles on the road: A survey of vision-based vehicle detection, tracking, and behavior analysis. *IEEE Transactions on Intelligent Vehicles, 6*(3), 402–420. https://doi.org/10.1109/TIV.2021.3086789

[11] Batra, D., & Kumar, A. (2022, September). Looking-in and looking-out of a vehicle: Computer-vision-based enhanced vehicle safety. *In 2022 5th International Conference on Computer Vision and Image Processing (CVIP)* (pp. 314–321). IEEE. https://doi.org/10.1109/CVIP55445.2022.00059

[12] Al-Qaness, M. A. A., & Fan, H. (2023, February). Metaverse for intelligent transportation systems (ITS): A comprehensive review of technologies, applications, implications, challenges and future directions. *IEEE Access, 11*, 15023–15045. https://doi.org/10.1109/ACCESS.2023.3243456

[13] Tiwari, S., & Singh, P. (2022, June). Motion-based vehicle surround analysis using an omni-directional camera. *Procedia Computer Science, 200*, 1236–1243. https://doi.org/10.1016/j.procs.2022.01.298

[14] Verma, M., Sharma, T., & Sinha, R. (2023, August). Performance evaluation of YOLO models in varying conditions: A study on object detection and tracking. *Journal of Big Data, 10*(1), 57. https://doi.org/10.1186/s40537-023-00763-w

[15] Roy, A., Dutta, M., & Saha, B. (2021, November). QR-code traffic signs recognition for ADAS smart driving system. *In 2021 International Conference on Robotics, Intelligent Automation and Control Technologies (RIACT)* (pp. 146–150). IEEE. https://doi.org/10.1109/RIACT54277.2021.9669037

[16] Chen, X., & Wu, Y. (2020, December). The role of machine vision for intelligent vehicles. *IEEE Transactions on Intelligent Transportation Systems, 21*(12), 5231–5243. https://doi.org/10.1109/TITS.2020.2998711

[17] Kim, T., & Park, D. (2022, July). Vehicle detection by means of stereo vision-based obstacles features extraction and monocular pattern analysis. *Pattern Recognition Letters, 156*, 25–32. https://doi.org/10.1016/j.patrec.2022.01.006

[18] Ramesh, R., & Patel, S. (2023, May). Vision-based vehicle guidance. *Journal of Intelligent & Robotic Systems, 107*, 25–38. https://doi.org/10.1007/s10846-023-01796-1

APPENDIX A – Program Code

Main.py

```
import multiprocessing
from multiprocessing import shared_memory
from image_process import pi_cam_main
from hud_display import hud_main

def create_shared_memory(name, size):
    """Helper function to create or attach to an existing shared memory block."""
    try:
        return shared_memory.SharedMemory(name=name, create=True, size=size)
    except FileExistsError:
        return shared_memory.SharedMemory(name=name, create=False)

def cleanup_shared_memory(*shared_memories):
    """Helper function to clean up shared memory blocks."""
    for shm in shared_memories:
        shm.close()
        shm.unlink()

def main():
    # Queues for inter-process communication
    queue_sign_ids = multiprocessing.Queue()
    queue_info = multiprocessing.Queue()
    stop_event = multiprocessing.Event()

    # Create shared memory blocks
    shm_vid = create_shared_memory(name="video_frame", size=720 * 1280 * 3)  #
1080*1920*3
    shm_AR = create_shared_memory(name="ar_info", size=288)

    # Define processes
    process_QR = multiprocessing.Process(target=pi_cam_main, args=(queue_sign_ids,
queue_info, stop_event))
    process_HUD = multiprocessing.Process(target=hud_main, args=(queue_sign_ids,
queue_info, stop_event))

    try:
        # Start and join processes
        process_QR.start()
        process_HUD.start()
        process_QR.join()
        process_HUD.join()
    except KeyboardInterrupt:
        # Handle graceful shutdown on interrupt
        stop_event.set()
        process_QR.join()
        process_HUD.join()
```

```
    finally:
        # Clean up shared memory
        cleanup_shared_memory(shm_vid, shm_AR)
        print("[INFO] Shared memory cleaned up.")

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

image_process.py

```
from pyzbar import pyzbar
import cv2
import time
import multiprocessing
from multiprocessing import shared_memory, Lock
import numpy as np
from ultralytics import YOLOWorld

# Enable OpenCL in OpenCV
cv2.ocl.setUseOpenCL(True)

# Global variables
detected_qr = ""
last_input_time = time.time()
shm_ar_lock = multiprocessing.Lock()

def clear_ar_buffer(ar_buffer, data="CLR"):
    """Clear the AR buffer and optionally write data."""
    with shm_ar_lock:
        ar_buffer[:] = 0  # Clear previous data
        encoded_msg = data.encode("utf-8")
        ar_buffer[:len(encoded_msg)] = np.frombuffer(encoded_msg, dtype=np.uint8)

def process_qr_code(qr, gray_frame, ar_buffer, queue_sign_ids):
    """Process a single QR code."""
    global detected_qr, last_input_time

    (x, y, w, h) = qr.rect
    qr_data = qr.data.decode("utf-8")

    # Draw rectangle and label on the frame
    cv2.rectangle(gray_frame, (x, y), (x + w, y + h), 255, 2)
    cv2.putText(gray_frame, qr_data, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.6,
255, 2)

    if qr_data.startswith("AR"):
        encoded_msg = qr_data.encode("utf-8")[:256]
        points = np.array([qr.polygon[i] for i in range(4)],
dtype=np.float32).flatten()
```

```
        points_bytes = points.tobytes()
        with shm_ar_lock:
            ar_buffer[:] = 0  # Clear previous data
            ar_buffer[:len(encoded_msg)] = np.frombuffer(encoded_msg,
dtype=np.uint8)
            ar_buffer[256:256+32] = np.frombuffer(points_bytes, dtype=np.uint8)
    elif qr_data != detected_qr:
        clear_ar_buffer(ar_buffer)
        detected_qr = qr_data
        queue_sign_ids.put(qr_data)
        print(f"[INFO] Detected QR Code: {qr_data}")
        last_input_time = time.time()

def qr_decode(gray_frame, queue_sign_ids, ar_buffer):
    """Decode QR codes from the frame."""
    global last_input_time, detected_qr

    qrcodes = pyzbar.decode(gray_frame)
    if not qrcodes:
        clear_ar_buffer(ar_buffer)
    else:
        for qr in qrcodes:
            process_qr_code(qr, gray_frame, ar_buffer, queue_sign_ids)

    # Clear display if no input for 10 seconds
    if time.time() - last_input_time > 10:
        detected_qr = ""

    return gray_frame

def car_detect_haar_classifier(gray_frame, car_cascade, queue_info):
    """Detect cars using Haar Cascade."""
    cars = car_cascade.detectMultiScale(gray_frame, scaleFactor=1.1,
minNeighbors=9, minSize=(60, 60))
    for (x, y, w, h) in cars:
        cv2.rectangle(gray_frame, (x, y), (x + w, y + h), (0, 0, 255), 2)
        cv2.putText(gray_frame, "car", (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(0, 0, 255), 2)
        queue_info.put("vehicle Detected")
        queue_info.put("slow down")
    return gray_frame

def car_detect_yolo(frame, model, queue_info):
    """Detect cars using YOLO model."""
    results = model(frame)
    for result in results:
        for box in result.boxes:
            x1, y1, x2, y2 = map(int, box.xyxy[0])
            label = result.names[int(box.cls[0])]
            conf = box.conf[0].item()
```

```
            if conf > 0.5:
                cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
                cv2.putText(frame, f"{label} ({conf:.2f})", (x1, y1 - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)

                if label.lower() in ["car", "truck", "bus", "person"]:
                    queue_info.put(f"{label.lower()} Detected")
                    queue_info.put("caution and slow")
    return frame

def initialize_shared_memory():
    """Initialize shared memory for video frames and AR info."""
    shm_vid = shared_memory.SharedMemory(name="video_frame")
    frame_array = np.ndarray((720, 1280, 3), dtype=np.uint8, buffer=shm_vid.buf)

    shm_AR = shared_memory.SharedMemory(name="ar_info")
    ar_buffer = np.ndarray((288,), dtype=np.uint8, buffer=shm_AR.buf)

    return shm_vid, frame_array, shm_AR, ar_buffer

def pi_cam_main(queue_sign_ids, queue_info, stop_event):
    """Main function for Pi camera processing."""
    url = "http://172.20.10.4:5000/video_feed"
    cap = cv2.VideoCapture(url)

    print("[INFO] QR Code detection started...")

    model = YOLOWorld(model='yolov8s-world.pt', verbose=False)
    model.set_classes(["car", "truck", "bus", "person"])

    shm_vid, frame_array, shm_AR, ar_buffer = initialize_shared_memory()

    try:
        while not stop_event.is_set():
            ret, frame = cap.read()
            if not ret:
                print("[ERROR] Failed to read frame from camera.")
                break

            frame = car_detect_yolo(frame, model, queue_info)

            frame_array[:] = frame  # Copy frame to shared memory

            gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
            gray = qr_decode(gray, queue_sign_ids, ar_buffer)

            cv2.imshow("QR & Car Detection", gray)
            if cv2.waitKey(1) & 0xFF == ord("q"):
                stop_event.set()
                break
```

```
    except KeyboardInterrupt:
        stop_event.set()
        print("[INFO] Interrupted by user")
    finally:
        cap.release()
        shm_vid.close()
        shm_AR.close()
        cv2.destroyAllWindows()
```

hud_diaplay.py

```
import cv2
import numpy as np
import os
import time
import threading
from queue import Empty
import multiprocessing
from multiprocessing import shared_memory, Lock
from testing.ar_model_generater import *
from ar_model_generater_with_shaders import *

cv2.ocl.setUseOpenCL(True)

# Locks for thread and shared memory synchronization
lock = threading.Lock()
shm_ar_lock = multiprocessing.Lock()

# Global variables
sign_ids_temp_from_queue = ""
ar_temp_from_shm = ""
info_lines = []

# Constants
SIGN_SIZE = (100, 100)
WINDOW_SIZE = (1280, 720)
INFO_DISPLAY_DURATION = 10  # seconds
MAX_INFO_LINES = 5


def load_sign_images(ids, sign_size=SIGN_SIZE):
    """Load sign images based on IDs."""
    ids = ids.split("//")
    images = []
    for sign_id in ids:
        filename = f"repo/{sign_id}.jpg"
        if os.path.exists(filename):
            img = cv2.imread(filename)
            img = cv2.resize(img, sign_size)
            images.append(img)
        else:
```

```
            print(f"Warning: {filename} not found")
    return images


def display_signs(images, info_lines, hud, window_size=WINDOW_SIZE):
    """Display images and text information on the HUD."""
    x_offset, y_offset = 20, 20

    # Display images
    for img in images:
        if y_offset + img.shape[0] > window_size[1]:
            break  # Stop if window overflows
        hud[y_offset:y_offset + img.shape[0], x_offset:x_offset + img.shape[1]] =
img
        y_offset += img.shape[0] + 10

    # Display text info
    for i, line in enumerate(info_lines):
        text_x = window_size[0] - (10 * len(line))
        text_y = 20 + i * 30
        cv2.putText(hud, line, (text_x, text_y), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
(255, 255, 255), 2)

    cv2.imshow("HUD Display", hud)
    cv2.waitKey(1)


def queue_reader(queue_sign_ids, queue_info, stop_event):
    """Read data from queues and update global variables."""
    global sign_ids_temp_from_queue, info_lines
    last_info_time = time.time()

    while not stop_event.is_set():
        # Read sign IDs
        try:
            with lock:
                sign_ids_temp_from_queue = queue_sign_ids.get(block=False)
        except Empty:
            pass

        # Read info lines
        try:
            with lock:
                new_info = queue_info.get(block=False)
                if new_info not in info_lines:
                    info_lines.append(new_info)
                if len(info_lines) > MAX_INFO_LINES:
                    info_lines.pop(0)
        except Empty:
            pass
```

```
        # Remove old info lines
        with lock:
            if info_lines and (time.time() - last_info_time >
INFO_DISPLAY_DURATION):
                info_lines.pop(0)
                last_info_time = time.time()

        time.sleep(0.1)


def initialize_shared_memory(name, shape, dtype):
    """Initialize shared memory and return a numpy array."""
    while True:
        try:
            shm = shared_memory.SharedMemory(name=name)
            return np.ndarray(shape, dtype=dtype, buffer=shm.buf), shm
        except FileNotFoundError:
            print(f"Waiting for shared memory: {name}...")
            time.sleep(0.5)


def hud_main(queue_sign_ids, queue_info, stop_event):
    """Main function to handle the HUD display."""
    global sign_ids_temp_from_queue, info_lines
    last_input_time = time.time()
    displayed_images = []

    # Initialize shared memory
    frame_array_buffer, shm_vid = initialize_shared_memory("video_frame", (720,
1280, 3), np.uint8)
    ar_buffer, shm_AR = initialize_shared_memory("ar_info", (288,), np.uint8)

    # Start input listener in a separate thread
    input_thread = threading.Thread(target=queue_reader, args=(queue_sign_ids,
queue_info, stop_event), daemon=True)
    input_thread.start()

    try:
        while not stop_event.is_set():
            frame = frame_array_buffer.copy()

            # Process AR data
            with shm_ar_lock:
                raw_data = bytes(ar_buffer[:256])
                qr_data_str = raw_data.split(b'\x00', 1)[0].decode('utf-8')
                points_bytes = bytes(ar_buffer[256:256 + 32])
                points = np.frombuffer(points_bytes, dtype=np.float32).reshape((4,
2))

            if qr_data_str.startswith("AR"):
                _, qr_data_str, size = qr_data_str.split("//")
```

```
                frame = generate_3d_model_mlt(frame, qr_data_str, points,
float(size))

            # Load and display sign images
            if sign_ids_temp_from_queue:
                displayed_images = load_sign_images(sign_ids_temp_from_queue)
                last_input_time = time.time()
                sign_ids_temp_from_queue = ""  # Reset after processing

            # Clear display if no input for a while
            if time.time() - last_input_time > INFO_DISPLAY_DURATION:
                displayed_images = []

            display_signs(displayed_images, info_lines, frame)
            time.sleep(0.1)  # Small delay to reduce CPU usage

            if cv2.waitKey(1) & 0xFF == ord("q"):
                stop_event.set()  # Signal to stop if 'q' is pressed
                break

    except KeyboardInterrupt:
        stop_event.set()
        print("[INFO] Interrupted by user")

    finally:
        shm_vid.close()
        shm_AR.close()
        cv2.destroyAllWindows()
        print("Exiting program...")
```

ar_model_generater_with_shaders.py

```
import cv2
import numpy as np
import torch
import os

# Constants
DEFAULT_COLOR = (200, 200, 200)
ROTATION_ANGLE_DEGREES = 90
CAMERA_MATRIX = np.array([[800, 0, 320],
                          [0, 800, 240],
                          [0, 0, 1]], dtype=np.float32)
DIST_COEFFS = np.zeros((4, 1))

# Global variables
vertices = []
faces = []
loaded_qr_data_str = ""
resize = 0
```

```
mtl_filename = ""

cv2.ocl.setUseOpenCL(True)


def load_mtl(filename):
    """Load material properties from an MTL file."""
    materials = {}
    current_material = None
    with open(filename, 'r') as f:
        for line in f:
            if line.startswith('newmtl'):
                current_material = line.split()[1]
                materials[current_material] = {'Kd': (1.0, 1.0, 1.0)}
            elif line.startswith('Kd') and current_material:
                kd = list(map(float, line.split()[1:4]))
                kd_bgr = tuple(int(c * 255) for c in kd[::-1])
                materials[current_material]['Kd'] = kd_bgr
    return materials


def load_obj_with_mtl(obj_path):
    """Load OBJ file along with its MTL file."""
    global resize
    vertices = []
    faces = []
    face_materials = []
    mtl_path = None

    with open(obj_path, 'r') as f:
        for line in f:
            if line.startswith('mtllib'):
                mtl_path = line.split()[1].strip()
            elif line.startswith('usemtl'):
                current_mtl = line.split()[1].strip() if len(line.split()) > 1 else
None
            elif line.startswith('v '):
                vertices.append(list(map(float, line.strip().split()[1:4])))
            elif line.startswith('f '):
                face = [int(p.split('/')[0]) - 1 for p in line.strip().split()[1:]]
                faces.append(face)
                face_materials.append(current_mtl)

    vertices_np = np.array(vertices)
    vertices[:] = (vertices_np * resize).tolist()
    return vertices, faces, face_materials, mtl_path


def check_file_exists(filepath, file_type):
    """Check if a file exists and print an error message if it doesn't."""
    if not os.path.exists(filepath):
        print(f"{file_type} file not found: {filepath}")
```

```
        return False
    return True


def apply_rotation(points, angle_degrees):
    """Apply rotation to 3D points."""
    angle_rad = np.deg2rad(angle_degrees)
    rotation_matrix_y = np.array([
        [np.sin(angle_rad), 0, np.cos(angle_rad)],
        [0, 1, 0],
        [np.cos(angle_rad), 0, np.sin(angle_rad)]
    ], dtype=np.float32)
    return points @ rotation_matrix_y.T


def render_faces(frame, faces, camera_space_vertices, face_materials, materials,
rvecs, tvecs):
    """Render faces on the frame."""
    face_depths = [
        (np.mean(camera_space_vertices[face][:, 2]), i)
        for i, face in enumerate(faces)
    ]
    face_depths.sort(reverse=True, key=lambda x: x[0])

    for _, face_index in face_depths:
        face = faces[face_index]
        mat_name = face_materials[face_index]
        pts_3d = np.array([vertices[i] for i in face], dtype=np.float32)
        pts_3d = apply_rotation(pts_3d, ROTATION_ANGLE_DEGREES)

        pts_2d, _ = cv2.projectPoints(pts_3d, rvecs, tvecs, CAMERA_MATRIX,
DIST_COEFFS)
        pts_2d = np.int32(pts_2d).reshape(-1, 2)

        color = materials.get(mat_name, {}).get('Kd', DEFAULT_COLOR)
        cv2.polylines(frame, [pts_2d], isClosed=True, color=(0, 0, 0), thickness=2)
        cv2.fillPoly(frame, [pts_2d], color=color)


def generate_3d_model_mlt(frame, qr_data_str, points, size=0.5):
    """Generate a 3D model and render it on the frame."""
    global vertices, faces, face_materials, loaded_qr_data_str, resize,
mtl_filename

    if qr_data_str != loaded_qr_data_str:
        resize = size
        obj_file = f"models/{qr_data_str}.obj"

        if not check_file_exists(obj_file, "OBJ"):
            return frame

        vertices, faces, face_materials, mtl_filename = load_obj_with_mtl(obj_file)
```

```
        mtl_file = f"models/{mtl_filename}"

        if not check_file_exists(mtl_file, "Material"):
            return frame

        materials = load_mtl(mtl_file)
        loaded_qr_data_str = qr_data_str
    else:
        materials = load_mtl(f"models/{mtl_filename}")

    obj_points = np.array([
        [-0.5, -0.5, 0],
        [0.5, -0.5, 0],
        [0.5, 0.5, 0],
        [-0.5, 0.5, 0]
    ], dtype=np.float32)

    success, rvecs, tvecs = cv2.solvePnP(obj_points, points, CAMERA_MATRIX,
DIST_COEFFS)
    if not success:
        return frame

    rotation_matrix, _ = cv2.Rodrigues(rvecs)
    vertices_tensor = torch.tensor(vertices, dtype=torch.float32).to("cuda" if
torch.cuda.is_available() else "cpu")
    rotation_matrix_tensor = torch.tensor(rotation_matrix,
dtype=torch.float32).to(vertices_tensor.device)

    camera_space_vertices = torch.matmul(vertices_tensor,
rotation_matrix_tensor.T).cpu().numpy() + tvecs.T
    render_faces(frame, faces, camera_space_vertices, face_materials, materials,
rvecs, tvecs)

    return frame
```

cam_stream.py

```
from flask import Flask, Response
import cv2
from picamera2 import Picamera2
import time

app = Flask(__name__)


picam2 = Picamera2()
config = picam2.create_video_configuration(
    main={"format": "RGB888"},
    controls={"FrameRate": 30}
)
```

```
picam2.configure(config)
picam2.start()

def generate_frames():
    while True:
        try:
            frame = picam2.capture_array("main")
            if frame is None:
                continue

            ret, buffer = cv2.imencode('.jpg', frame,
[int(cv2.IMWRITE_JPEG_QUALITY), 90])
            if not ret:
                continue

            frame = buffer.tobytes()
            yield (b'--frame\r\n'
                   b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')

        except Exception as e:
            print(f"Error in generate_frames(): {e}")
            break

@app.route('/')
def index():
    return "<h1>Live Camera Feed</h1><img src='/video_feed'>"

@app.route('/video_feed')
def video_feed():
    return Response(generate_frames(), mimetype='multipart/x-mixed-replace;
boundary=frame')

if __name__ == '__main__':
    try:
        app.run(host='0.0.0.0', port=5000, debug=False, threaded=False)
    finally:
        picam2.stop()
```

APPENDIX B – Terminal Scripts And File Structure

Instructions to download python and create virtual environment:

```
• sudo apt install python3.12
• sudo apt install python3-pip
• python3 -m venv myenv
```

Activation and deactivation of virtual environment:

```
• source myenv/bin/activate
• deactivate
```

Downloading Python Libraries:

```
• pip install opencv-python
• pip install pyzbar
• pip install ultralytics
• pip install keras-models
• pip install torch torchvision torchaudio --index-url
  https://download.pytorch.org/whl/nightly/cpu
```

Raspberry pi Terminal Script to execute Scenario 1:

```
• source ~/.venv/bin/activate
• cd ~/smart_hud_system
• python -O main.py
```

Raspberry pi Terminal Script to execute Scenario 2:

```
• source ~/.venv/bin/activate
• cd ~/stream
• python -O cam_stream.py
```

Program files in system for Distributed computation system:
project

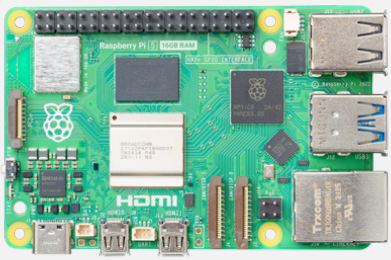
- main.py
- image_process.py
- hud_display.py
- ar_model_generater_with_shaders.py
- yolov8s-world.pt
- Location_ids_discription.csv
- repo_sign (Folder containing repositories of road sign png downloaded and ide's)
  - folder from 100 to 117
  - The information about the folders are available in Location_ids_discription.csv
- Models (Folder containing 3D models in .obj and .mlt format)

APPENDIX C – QR CODES

	
113/670V60//113/878.1//111/7255//110/607	113/7001//111/7013//111/7010.1
	
113/7001//111/7013//111/7010.1	113/880//117/550.2//117/553.2
	
AR//2D//ar_fuel	AR//2D//caution
	
AR//Junk_food//0.5	AR//traffic_light//0.1

APPENDIX D – Raspberry Pi 5 datasheet

Overview

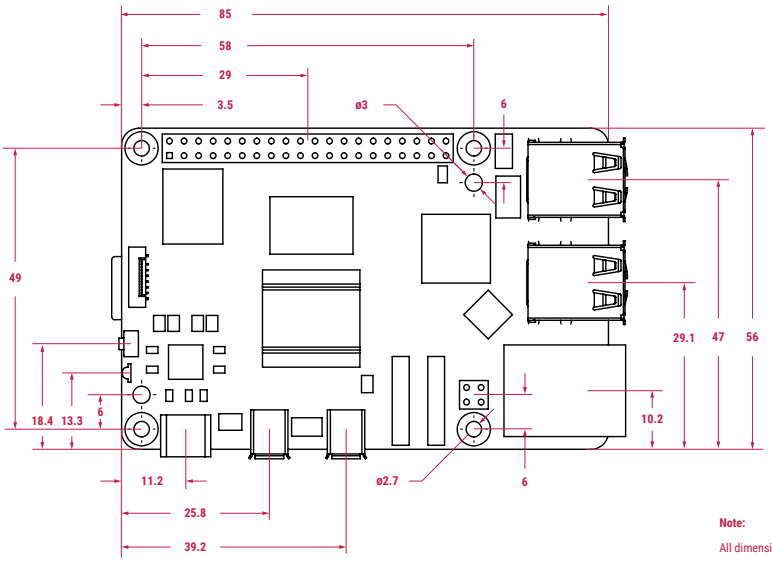


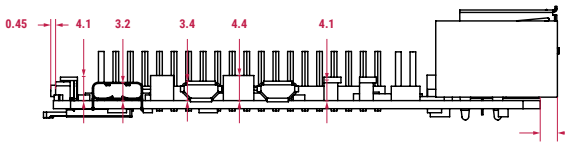
Welcome to the latest generation of Raspberry Pi: the everything computer.

Featuring a 64-bit quad-core Arm Cortex-A76 processor running at 2.4GHz, Raspberry Pi 5 delivers a 2–3× increase in CPU performance relative to Raspberry Pi 4. Alongside a substantial uplift in graphics performance from an 800MHz VideoCore VII GPU; dual 4Kp60 display output over HDMI; and state-of-the-art camera support from a rearchitected Raspberry Pi Image Signal Processor, it provides a smooth desktop experience for consumers, and opens the door to new applications for industrial customers.

For the first time, this is a full-size Raspberry Pi computer using silicon built in-house at Raspberry Pi. The RP1 “southbridge” provides the bulk of the I/O capabilities for Raspberry Pi 5, and delivers a step change in peripheral performance and functionality. Aggregate USB bandwidth is more than doubled, yielding faster transfer speeds to external UAS drives and other high-speed peripherals; the dedicated two-lane 1Gbps MIPI camera and display interfaces present on earlier models have been replaced by a pair of four-lane 1.5Gbps MIPI transceivers, tripling total bandwidth, and supporting any combination of up to two cameras or displays; peak SD card performance is doubled, through support for the SDR104 high-speed mode; and for the first time the platform exposes a single-lane PCI Express 2.0 interface, providing support for high-bandwidth peripherals.

Physical specification





Note:

All dimensions in mm

All dimensions are approximate and for reference purposes only. The dimensions shown should not be used for producing production data

The dimensions are subject to part and manufacturing tolerances

Not all of the board components are shown. Please reference a physical board for representation of componentry

Dimensions may be subject to change

WARNINGS

- This product should be operated in a well ventilated environment, and if used inside a case, the case should not be covered.
- While in use, this product should be firmly secured or should be placed on a stable, flat, non-conductive surface, and should not be contacted by conductive items.
- The connection of incompatible devices to Raspberry Pi 5 may affect compliance, result in damage to the unit, and invalidate the warranty.
- All peripherals used with this product should comply with relevant standards for the country of use and be marked accordingly to ensure that safety and performance requirements are met.

SAFETY INSTRUCTIONS

To avoid malfunction or damage to this product, please observe the following:

- Do not expose to water or moisture, or place on a conductive surface while in operation.
- Do not expose to heat from any source; Raspberry Pi 5 is designed for reliable operation at normal ambient temperatures.
- Store in a cool, dry location.
- Take care while handling to avoid mechanical or electrical damage to the printed circuit board and connectors.
- While it is powered, avoid handling the printed circuit board, or handle it only by the edges, to minimise the risk of electrostatic discharge damage.

Specification

Processor	Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU, with Cryptographic Extension, 512KB per-core L2 caches, and a 2MB shared L3 cache
Features:	• VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2 • Dual 4Kp60 HDMI® display output with HDR support • 4Kp60 HEVC decoder • LPDDR4X-4267 SDRAM (options for 2GB, 4GB, 8GB and 16GB) • Dual-band 802.11ac Wi-Fi® • Bluetooth 5.0 / Bluetooth Low Energy (BLE) • microSD card slot, with support for high-speed SDR104 mode • 2 × USB 3.0 ports, supporting simultaneous 5Gbps operation • 2 × USB 2.0 ports • Gigabit Ethernet, with PoE+ support (requires separate PoE+ HAT) • 2 × 4-lane MIPI camera/display transceivers • PCIe 2.0 x1 interface for fast peripherals (requires separate M.2 HAT or other adapter) • 5V/5A DC power via USB-C, with Power Delivery support • Raspberry Pi standard 40-pin header • Real-time clock (RTC), powered from external battery • Power button
Production lifetime:	Raspberry Pi 5 will remain in production until at least January 2036
Compliance:	For a full list of local and regional product approvals, please visit pip.raspberrypi.com
List price:	2GB **$50** 4GB **$60** 8GB **$80** 16GB **$120**