Indoor Navigation for the Blind: Real-Time Object Detection and Mapping

Abhishek Bansal
IIITD, Delhi
abhishek22021@iiitd.ac.in

Dhruv Sharma
IIITD, Delhi
dhruv22170@iiitd.ac.in

Pratyush Gupta
IIITD, Delhi
pratyush22375@iiitd.ac.in

Vinayak Agrawal
IIITD, Delhi
vinayak22574@iiitd.ac.in

Abstract

Navigational assistance for visually impaired individuals often relies on specialized hardware or GPS, which face limitations in cost, infrastructure dependency, and indoor effectiveness. We introduce a real-time assistive system using a client-server architecture accessible via smartphone. The perception engine integrates YOLOv5 object detection, MiDaS depth estimation, and ORB visual odometry with depth-aided PnP fusion for robust scene understanding. This information populates a dynamic cost map for collision-free path planning, with waypoints translated into natural language instructions. The Android frontend captures camera frames, communicates with the backend API, and delivers guidance through Text-to-Speech and haptic feedback. Our approach provides an accessible, infrastructure-free navigation solution at interactive frame rates.

1. Introduction

Indoor navigation poses unique challenges for visually impaired individuals due to GPS signal limitations and dynamic environments. Traditional aids like white canes provide insufficient spatial awareness without semantic understanding of surroundings.

This project presents a mobile application that employs computer vision for assistive indoor navigation. The system integrates optimized deep learning models—YOLOv5s for object detection, MiDaS for depth estimation, and visual odometry—to deliver real-time environmental interpretation. Path planning is handled through the A* algorithm, ensuring safe navigation routes. The application features an Android frontend built with Kotlin and a scalable FastAPI backend containerized using Docker.

The system processes visual data and delivers real-time audio guidance, enabling visually impaired users to navigate indoor spaces independently and safely.

1.1. Motivation

The motivation for this work stems from:

- **Accessibility:** Providing an affordable, camera-only solution that leverages existing smartphone or wearable hardware.
- **Scalability:** Eliminating the need for infrastructure (e.g., beacons, LiDAR) to enable deployment in diverse indoor settings.
- **Real-time Guidance:** Ensuring low-latency perception and instruction so users receive timely alerts.
- **User-Centric Design:** Prioritizing the needs of visually impaired users by delivering intuitive, non-visual cues—such as voice prompts —that translate complex spatial information into simple, actionable guidance, thereby enhancing autonomy and reducing cognitive load.

1.2. Project Outcomes

- Developed a user-friendly Android application using Kotlin, enabling real-time interaction and feedback.
- Implemented a lightweight deep learning model optimized for mobile devices, combining YOLO for object detection, MiDaS for depth estimation, and visual odometry techniques to interpret indoor environments effectively.
- Integrated the A* algorithm for efficient path planning, allowing the system to compute optimal routes while avoiding obstacles.
- Designed a robust backend using FastAPI, containerized with Docker, ensuring scalability and ease of deployment.
- Conducted comprehensive testing in various indoor environments to validate the system's effectiveness and reliability.

2. Problem Statement

Navigational assistance for visually impaired individuals in indoor environments remains challenging due to the complex, variable nature of indoor spaces and the need for reliable, real-time perception and guidance. To develop a truly scalable, vision-only indoor navigation aid with natural language output, we must address the following core challenges:

- **Spatial and appearance variability:** Indoor spaces exhibit diverse layouts, textures, and lighting conditions that degrade detection and mapping algorithms [11, 12].
- **GPS-denied localization:** In the absence of satellite signals, vision-based SLAM must provide drift-free pose estimates over long trajectories [5].
- **Depth estimation reliability:** Monocular and stereo depth cues are often noisy under low-texture or poor lighting; fusing multi-scale networks and stereo disparity improves robustness [2, 7].
- **Visual odometry drift:** Without loop closures or global corrections, frame-to-frame VO accumulates error, degrading localization accuracy [5].
- **Real-time computational constraints:** Running YOLOv5 object detection and MiDaS depth estimation at interactive frame rates (¡100 ms/frame) on mobile GPUs requires highly optimized inference [3, 8].
- **Instruction clarity vs. cognitive load:** Natural language directions must balance informativeness with brevity to avoid overwhelming users in cluttered environments [4].
- **Seamless multi-module integration:** Vision, planning, and text-to-speech components must interoperate efficiently on resource-limited hardware without perceptible latency [4, 10].

Addressing these challenges is critical to realizing a vision-only indoor navigation system that delivers accurate, low-latency guidance to visually impaired users in real-world settings.

3. Literature Review

Majorily done in interim report an extension of it

Davison [1] introduced one of the earliest real-time frameworks for monocular Simultaneous Localization and Mapping (SLAM), demonstrating that a single moving camera can estimate its own pose and map surrounding features in real time without relying on external sensors. Although our system does not perform full SLAM, we build upon these foundational ideas by adopting lightweight monocular visual odometry techniques. Using OpenCV-based feature tracking, we enable continuous localization and navigation through visual inputs alone, which is critical for developing scalable, infrastructure-free assistive technologies for the visually impaired.

4. Main Pipeline

Flowchart 10,illustrates the overall architecture.

4.1. Computer Vision Module

The computer vision (CV) module executes three parallel pipelines—object detection, depth estimation, and visual odometry—using separate CUDA streams.

Object Detection (YOLOv5s) We employ the `yolov5s` model, pretrained on the COCO dataset, processing 640×480 RGB frames. Detections with confidence scores above 0.4 are retained, and non-maximum suppression (NMS) with an intersection-over-union (IoU) threshold of 0.45 is applied to eliminate redundant boxes. The outputs include bounding boxes, class labels, confidence scores, and box centers. Inference latency is approximately 12 ms per frame.

Depth Estimation (MiDaS DPT_Hybrid) RGB frames are resized to 384×384 and processed by the MiDaS `DPT_Hybrid` model. The resulting depth map is filtered using a 5×5 median filter followed by a bilateral filter with diameter $d = 7$, color sigma $\sigma_c = 80$, and spatial sigma $\sigma_s = 80$. The depth map is normalized to a closeness map $C(u, v) \in [0, 1]$, where higher values indicate closer objects, and upsampled to the original frame resolution. The latency is approximately 18 ms per frame.

Visual Odometry (ORB + PnP) We detect up to 1500 ORB features on undistorted grayscale frames and match them across frames using Hamming distance. If a prior depth map is available, keypoints are unprojected to 3D using a depth scale factor (`depth_vo_scale`). We solve the Perspective-n-Point (PnP) problem with a reprojection error threshold of 4 pixels to estimate the relative pose $(\mathbf{R}_{\text{rel}}, \mathbf{t}_{\text{rel}})$. If no prior depth map exists, pose updates are skipped. Poses are accumulated as $(\mathbf{R}_{\text{cum}}, \mathbf{t}_{\text{cum}})$ to construct a 2D trajectory in pixel coordinates.

Fusion and Obstacle Map The closeness map $C(u, v)$ is sampled at the center of each detected object. Pixels with $C(u, v) > 0.5$ are marked as obstacles. The obstacle map is dilated by 15 pixels, and a distance transform is applied to compute the Euclidean distance transform $D_{\text{edt}}(u, v)$. The navigation cost map is defined as:

$$C_{\text{nav}}(u, v) = w_{\text{prox}}\big(1 - \text{norm}(D_{\text{edt}}(u, v))\big) + w_{\text{close}} C(u, v),$$

where $w_{\text{prox}} = 300$, $w_{\text{close}} = 50$, and $\text{norm}(\cdot)$ normalizes values to $[0, 1]$.

4.2. Trajectory Planning

Trajectory planning is performed using the A* algorithm on an 8-connected grid with 15-pixel steps. The cost function for a point p is:

$$f(p) = C_{\text{nav}}(p) + w_h \|\mathbf{p} - \mathbf{p}_{\text{goal}}\|_2,$$

where $w_h = 1.5$ is the heuristic weight, and $\mathbf{p}_{\text{goal}}$ is the goal position. Planning terminates when the current point is within 1.5 steps (22.5 pixels) of the goal, producing a sequence of waypoints $\{\mathbf{p}_i\}$

4.3. Instruction Generation and TTS

Waypoint deltas are converted to heading changes $\Delta\theta_i$ and classified as *forward*, *slight left or right* (if $|\Delta\theta_i| < 30°$), or *turn left or right* (otherwise). Distances are accumulated in 15-pixel units. A dedicated `TTSManager` thread, implemented using `pyttsx3`, speaks the first instruction per frame, ensuring non-blocking audio feedback.

4.4. Implementation & Performance

Implemented in Python with OpenCV, PyTorch, NumPy, and pyttsx3. Parallelism uses `concurrent.futures`. On an RTX 3080 + i7 CPU, we achieve $\approx$ 25 FPS end-to-end. Module latencies are roughly: YOLOv5s 12 ms, MiDaS 18 ms, VO 10 ms, planning & TTS 5 ms.

Please refer to the demo video for our CV pipeline Google Drive Link.

4.5. Experimental Metrics

We evaluate the key components of our system—object detection, depth estimation, inference speed—and overall navigation performance in terms of success and collision rates. Tables 1–3 summarize these results.

Object Detection Metrics

Refer Table 1,

Depth Estimation Metrics

Refer Table 2.

Navigation Performance

Refer Table 3. We logged end-to-end navigation trials with our `MetricsLogger`. Table 3 summarizes overall success and collision rates across all tasks.

4.6. Compute Requirements and Resource Analysis

Our client-server architecture distributes computational load to optimize performance and battery life.

- **Backend Server Resources:**
  - **GPU:** NVIDIA RTX series or cloud equivalents (T4, V100) for accelerating YOLOv5s and MiDaS DPT_Hybrid inference. Requires 6GB+ VRAM.
  - **CPU:** Multi-core for FastAPI server, ORB feature processing, PnP calculations, image processing, and network I/O.
  - **RAM:** 16GB+ for OS, Python environment, and concurrent request handling.
  - **Storage:** 400MB for models (YOLOv5s: 14MB, MiDaS: 350MB) plus codebase.
  - **Network:** Low-latency connection for frame transmission (100-500KB per frame) and JSON responses (¡1KB).
- **Frontend Mobile Client:**
  - **CPU:** Handles Android OS, Kotlin app logic, camera operations, network communication, UI, and TTS engine.
  - **RAM:** 4-8GB for system, application, camera and network buffers.
  - **Battery:** Critical constraint. Offloading ML inference reduces power consumption, but camera, network, and TTS remain significant power draws.
  - **Storage:** Minimal requirement (50-100MB) for APK and cached data.
  - **Network:** Stable connection (Wi-Fi preferred) for backend communication.
- **Model Selection:** YOLOv5s and MiDaS DPT_Hybrid balance real-time performance with accuracy. ORB feature processing adds moderate CPU load.

Average Inference Time Plots

Performance comparisons of object detection and depth estimation models shown in Figures 5 and 6.

4.7. Backend Processing Engine

The core visual processing engine operates on the backend, developed using Python and the high-performance FastAPI framework. We designed to efficiently receive image frames via HTTP POST requests from the client application.

Upon receiving an image frame, the backend performs the following sequence of operations:

1. **Input Validation and Preprocessing:** The incoming image data is validated and preprocessed to meet the requirements of downstream models.
2. **Machine Learning Inference:** The frame is processed through pre-trained machine learning models (to perform tasks such as object detection and spatial attribute estimation (e.g., determining bounding boxes).
3. **Instruction Generation:** Based on the inference results—including detected objects, their locations, and potentially depth or contextual information—the backend generates contextually relevant navigational cues or alert messages tailored for non-visual feedback.
4. **Response Formulation:** The processed information, including object details (labels, locations) and generated

instructions or alerts, is packaged into a structured JSON response.

This processed information is then transmitted back to the originating client application.

4.8. Frontend Mobile Application (Android Client)

The frontend component is a native Android application developed using Kotlin and the modern, declarative Jetpack Compose UI toolkit for building an accessible user interface. The primary responsibilities of the frontend application include:

1. **Camera Management:** Accessing and managing the device's camera to continuously capture the live video feed from the user's perspective.
2. **Network Communication:** Efficiently encoding and transmitting selected camera frames to the backend's `/predict` API endpoint over the network, and handling the reception and parsing of JSON responses containing the processed visual information and instructions.
3. **Non-Visual Feedback Delivery:** Translating the received instructions or alerts into user-perceptible feedback through integrated non-visual modalities, primarily including:
    - **Text-to-Speech (TTS):** Utilizing the native Android TTS engine to vocalize instructions and alerts clearly.
4. **User Interface and Control:** Providing an accessible interface for users to initiate and terminate the visual assistance process, manage application settings (e.g., TTS rate, haptic intensity), and optionally view the live camera feed for sighted assistance or debugging purposes.

The frontend serves as the crucial link between the user and the backend's computational power, focusing on efficient data transfer and effective non-visual communication.

4.9. Implementation Details

The system leverages standard libraries for core functionalities: `FastAPI` for backend routing and asynchronous handling, `OpenCV`and potentially `PyTorch` or `TensorFlow` (depending on model choice) for backend image processing and inference, `Kotlin` with `Jetpack Compose` for the Android frontend, and native Android APIs for Camera, TTS, and Haptics. Network communication relies on standard HTTP libraries (e.g., Retrofit/OkHttp on Android, HTTPX/Requests in Python).

4.10. System Summary

In summary, our system presents a practical client-server solution for real-time visual assistance. By offloading heavy computation to a cloud backend accessed via a simple API, the native Android client can focus on efficient camera handling, network communication, and delivering crucial information through accessible TTS and haptic feedback. This architecture provides a foundation for a scalable and adaptable visual aid for diverse user needs.

5. Future Work and Limitations

Current Limitations

- **Monocular Depth Noise:** Ambiguity in texture-less regions, low lighting, and cluttered scenes.
- **Lack of Absolute Scale:** Up-to-scale depth estimates without direct metric distance understanding.
- **Inference Latency:** Performance fluctuations depending on scene complexity.

Future Work

- **Improving Depth Estimation:** Self-supervised learning, stereo-based refinement, and ensemble approaches for greater robustness.
- **Scale Disambiguation:** VO-based dynamic scaling and smartphone dual camera stereo fusion for absolute measurements.
- **Optimization:** Model compression (quantization, pruning) and deployment optimization via TensorRT or ONNX Runtime.
- **Advanced Scene Understanding:** Integration of *SpatialLM* [9] for structured 3D understanding and semantic layout recognition from monocular inputs.
- **Enhanced SLAM:** Adoption of *Mast3r-SLAM* [6] for improved 3D reconstruction and localization through multi-view depth fusion.

These advancements would transform our system from basic obstacle avoidance to comprehensive 3D semantic navigation with context-aware guidance for visually impaired users.

6. Contributions

Our team equally contributed to all aspects of this project, collaboratively developing the entire pipeline from vision module integration to real-time navigation and feedback systems.

7. Conclusion

We present a real-time vision-based navigation system for visually impaired users that combines YOLOv5s, MiDaS, and visual odometry in a client-server architecture. Our system achieves interactive frame rates with reliable environmental understanding across varied conditions. Despite current limitations in monocular depth estimation, our work with emerging technologies like *SpatialLM* and *Mast3r-SLAM* establishes a foundation for advancing affordable assistive technologies that enhance autonomy in real-world indoor settings.

References

[1] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[2] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in Neural Information Processing Systems*, 2014.

[3] Glenn Jocher. Ultralytics yolov5. `https://github.com/ultralytics/yolov5`, 2020.

[4] Payal Mahida, Seyed Shahrestani, and Hon Cheung. Deep learning-based positioning of visually impaired people in indoor environments. In *Proceedings of CVPR*, 2025.

[5] Raúl Mur-Artal and Juan D. Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. In *IEEE Transactions on Robotics*, 2015.

[6] Riku Murai, Eric Dexheimer, and Andrew J. Davison. Mast3r-slam: Real-time dense slam with 3d reconstruction priors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

[7] Rupert Ranftl, Alexey Bochkovskiy, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[8] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, 2015.

[9] Manycore Research. Spatiallm: Large language model for spatial understanding. `https://manycore-research.github.io/SpatialLM/`, 2024.

[10] Adarsh Jagan Sathyamoorthy, Jing Liang, Utsav Patel, Tianrui Guan, Rohan Chandra, and Dinesh Manocha. Densecavoid: Real-time navigation in dense crowds using anticipatory behaviors. In *Proceedings of CVPR*, 2025.

[11] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *European Conference on Computer Vision*, 2012.

[12] Scott Song, Jean Ponce, et al. Sun rgb-d: A rgb-d scene understanding benchmark suite. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 567–576, 2015.

Table 1. Object detection performance using YOLOv5s

Metric	Value
mAP@0.50	*0.80*
Precision	*0.56*
Recall	*0.50*
F1 Score	*0.60*

Table 2. Depth estimation performance for MiDaS variants

Metric	MiDaS_Small	DPT_Hybrid
AbsRel	1.2269	1.2840
RMSE_log	1.3655	1.1121
$\delta < 1.25$	0.1862	0.1847
$\delta < 1.25^2$	0.3441	0.3450
$\delta < 1.25^3$	0.4794	0.4788

Table 3. Overall navigation success and collision rates

Metric	Value
Success Rate	71%
Collision Rate	22%

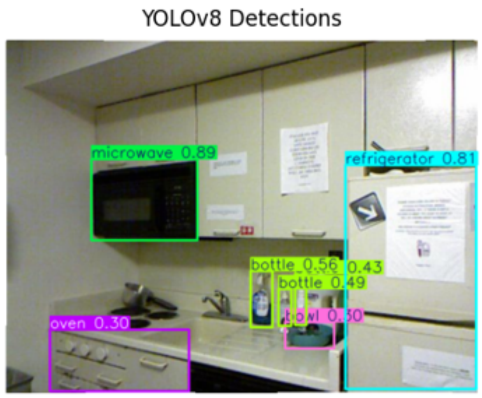


Figure 1. Sample object detection output using YOLOv8-nano.

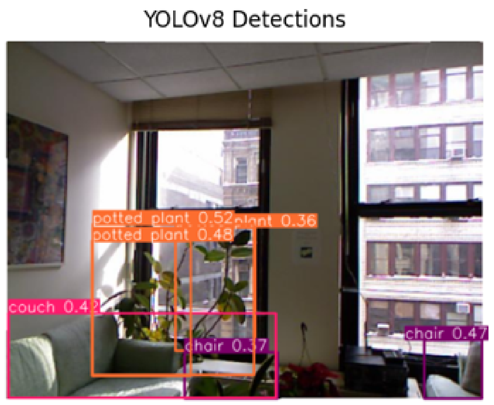


Figure 2. Additional sample detection output from YOLOv8-nano.

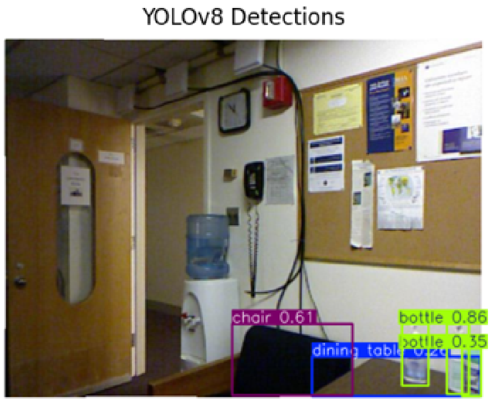


Figure 3. Another detection output using YOLOv8-nano.

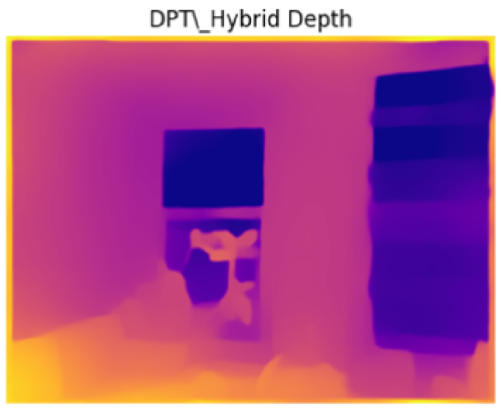


Figure 4. Sample depth map predicted by DPT_Hybrid.

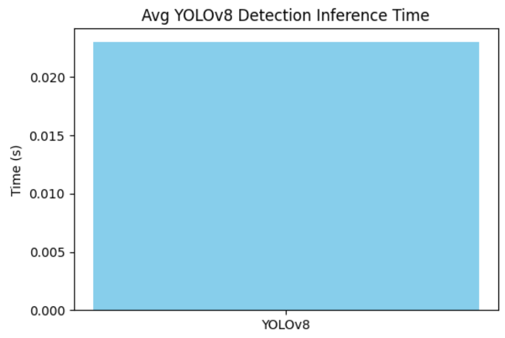


Figure 5. Avg YOLOV8 Detection time

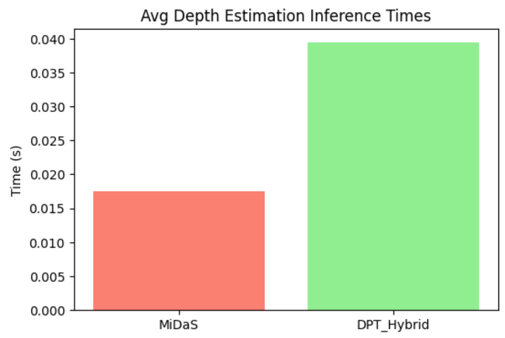


Figure 6. Avg depth estimation inference time

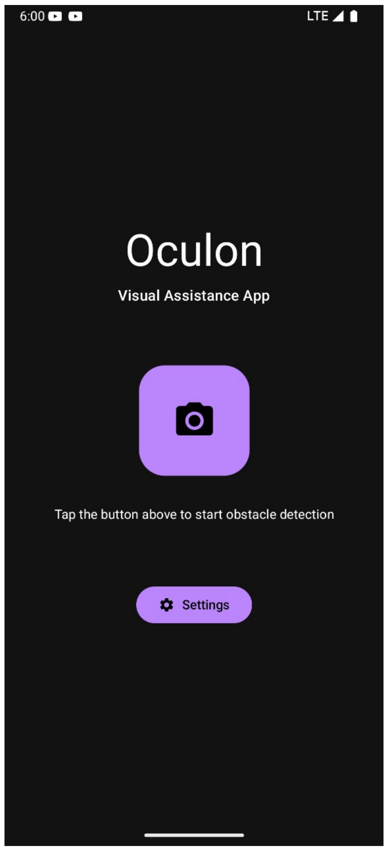
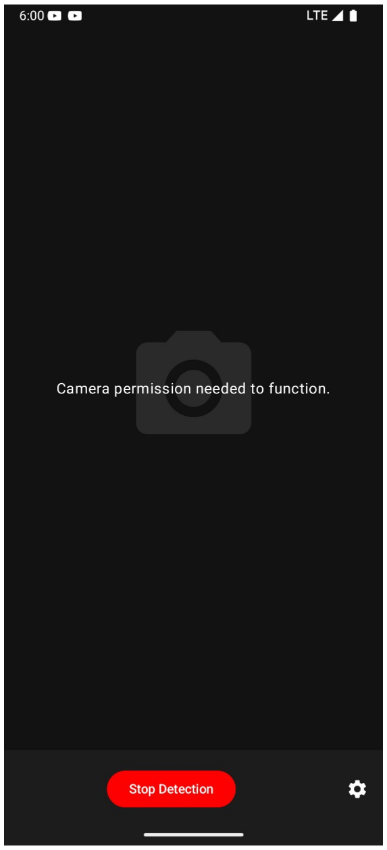
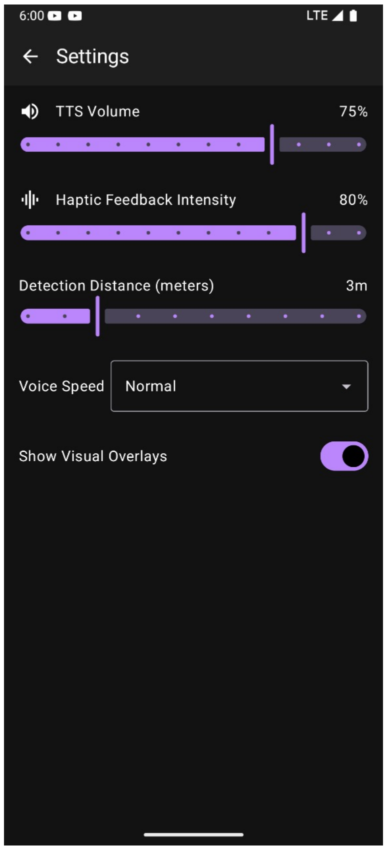


Figure 7. Landing page of the app



Camera permission needed to function.

Figure 8. Main page that takes the video feed



Settings

TTS Volume 75%

Haptic Feedback Intensity 80%

Detection Distance (meters) 3m

Voice Speed Normal

Show Visual Overlays

Figure 9. Settings

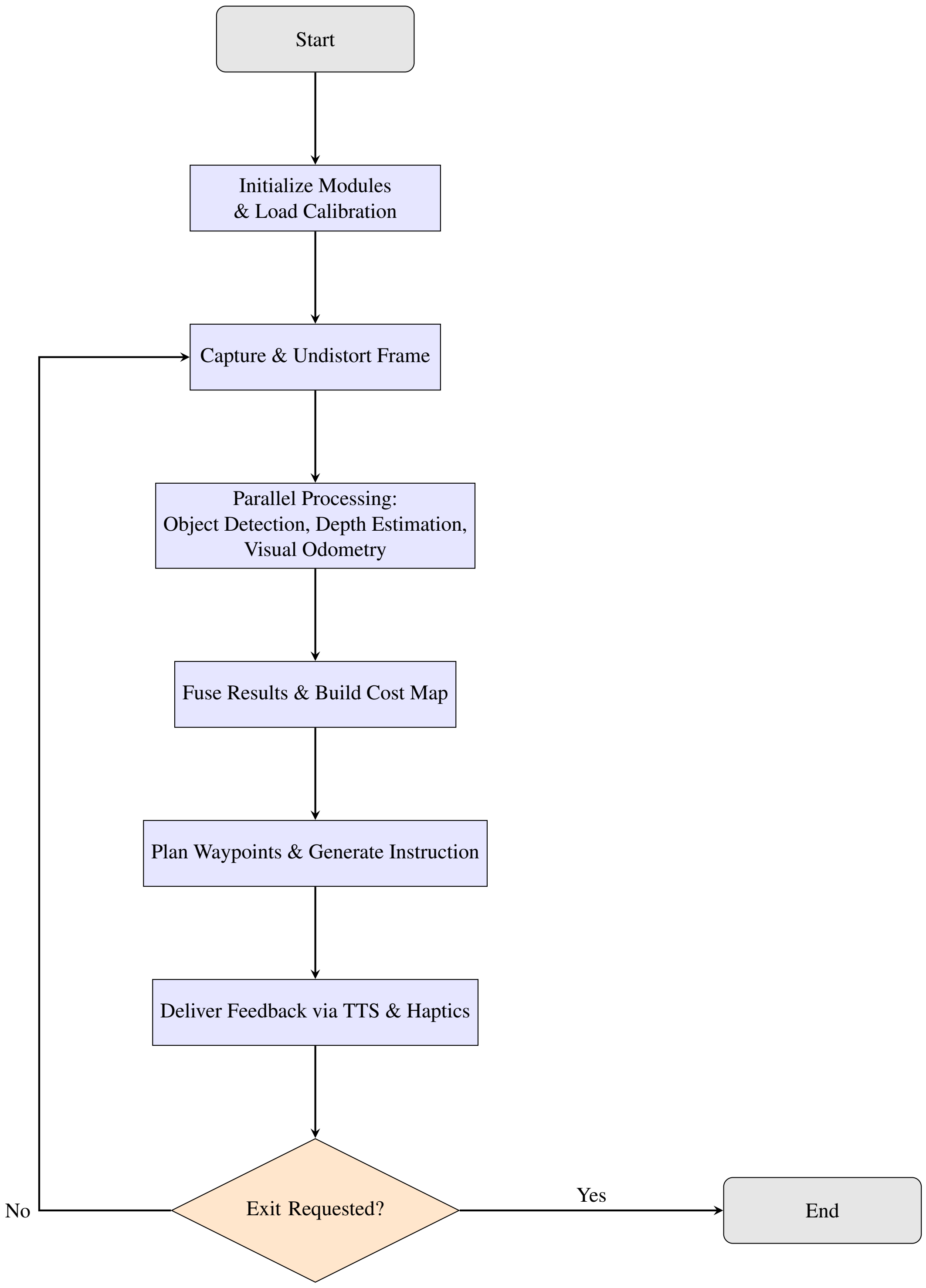

Start
Initialize Modules & Load Calibration
Capture & Undistort Frame
Parallel Processing: Object Detection, Depth Estimation, Visual Odometry
Fuse Results & Build Cost Map
Plan Waypoints & Generate Instruction
Deliver Feedback via TTS & Haptics
Exit Requested?
Yes
No
End

Figure 10. Flowchart of the Real-Time Visual Assistance Pipeline