Adaptive Cane for Visually Impaired People

Simon Dourte
Edwyn Eben
Nathan Lambrechts
Louca Mathieu
Florian Stormacq
simon.dourte@student.unamur.be
edwyn.eben@student.unamur.be
nathan.lambrechts@student.unamur.be
louca.mathieu@student.unamur.be
florian.stormacq@student.unamur.be
University of Namur, Namur Digital Institute (NaDI) and Research Center on Information Systems Engineering (PReCISE)
Namur, Belgium

Figure 1: Overview of the Adaptive Cane System

Abstract

This project presents a connected and adaptive cane for blind or visually impaired people, designed to improve their perception of the environment and enhance their safety when facing obstacles and potential hazards. It integrates a video sensor (Intel RealSense D435i depth camera) and an audio sensor (USB microphone) to detect obstacles and assess the surrounding sound environment in real time. The acquired data are processed by a Raspberry Pi 3 using a multi-threaded producer-consumer architecture, with queue management and temporal synchronization to fuse audio and video information. The fused information is converted into adjustable intensity levels and transmitted to an Arduino, which drives haptic feedback devices simulated by RGB LEDs. Each system module was developed and tested individually, and the complete prototype was validated in a controlled environment.

CCS Concepts

• **Hardware** → **Sensors and actuators**; **Haptic devices**; *Sensor devices and platforms*; *Signal processing systems*; • **Computer systems organization** → *Embedded hardware*; • **Human-centered computing** → **Accessibility technologies**.

ACM Reference Format:

Simon Dourte, Edwyn Eben, Nathan Lambrechts, Louca Mathieu, and Florian Stormacq. 2025. Adaptive Cane for Visually Impaired People. In *Proceedings of the UNamur Symposium on Advanced Interaction Methods (AIM 2025 Proceedings), December 25, 2025, Namur, Belgium.* ACM, New York, NY, USA, 7 pages. https://doi.org/00.0000/0000000.0000000

1 Introduction

People with visual impairments represent a significant proportion of the global population. It is estimated that at least 2.2 billion people have some form of vision impairment. It is therefore essential to develop solutions that meet their needs and improve their independence. Traditional canes provide basic assistance, but they are limited in their ability to detect complex obstacles or to provide real-time feedback about the environment.

In response to these limitations, the central objective of this project is to develop an adaptive smart cane designed to improve spatial awareness and safety for visually impaired users. The system relies on two main sensors: a depth camera and a microphone to detect obstacles or hazards in real time. The data are collected by a Raspberry Pi, which analyzes the visual and auditory information from the environment. This multimodal approach leverages auditory perception and haptic interaction to compensate for visual impairments and enhance user awareness of the surrounding environment.

The system focuses entirely on haptic feedback. A set of three vibrating motors is placed on the cane to provide tactile information to the user. Each motor corresponds to a direction, meaning that one direction is not covered. Indeed, the rear direction was deliberately excluded, since the camera is positioned like a pair of glasses and cannot see behind the user. The intensity and duration of the vibrations reflect the proximity of the danger, allowing the user to interpret the environment through haptic feedback. This cane aims to assist blind people during navigation in urban environments or in any activity involving potential danger for the user.

This report first presents some existing approaches that address the main problem. It then discusses the architecture of our project and its implementation. A brief overview of each member's contribution, as well as the reasons why we were unable to evaluate our project under real-world conditions, is also provided. Finally, we discuss the various limitations and challenges encountered, along with ideas for potential improvements.

2 Related work

Several assistive technologies have been developed to enhance mobility for visually impaired individuals. Manuel et al. [2] present AI-enhanced smart glasses that combine YOLOv8-based obstacle detection with OCR and image captioning capabilities for comprehensive environmental awareness.

In the domain of smart canes, Loor Guaycha et al. [1] developed an ultrasonic-based prototype capable of detecting obstacles up to 100 cm away, providing real-time auditory alerts to users. Their field tests demonstrated high precision in obstacle detection.

For multimodal approaches, Xu et al. [3] propose a wearable navigation system that combines GPS positioning, machine vision, and remote companion functionality, enabling independent travel and daily activities for blind users.

Unlike these approaches, the system developed in this project integrates a depth camera and an audio sensor to provide real-time haptic feedback for obstacle avoidance. This multimodal approach aims to enhance user safety and spatial awareness through tactile signals.

3 Methodology

The project is structured into several key components to ensure optimal functionality, performance, and maintainability.

3.1 Project Architecture

3.1.1 Audio Sensor (Microphone). To capture ambient sound, we use a USB microphone (*"USB PnP Sound Device"*). The goal is to continuously record the surrounding audio signal in order to determine the noise level around the user. This allows us to know whether the environment is noisy (*e.g.*, a crowd) or quiet.

3.1.2 Video Sensor (Intel RealSense Camera). To capture the image and depth information, we use an Intel RealSense Depth D435i camera connected to the Raspberry Pi via a USB-C cable. The aim is to continuously record the depth stream perceived by the camera in order to locate potential dangers, estimate their distance, and determine the alert mode of the smart cane as well as the direction to avoid obstacles.

The camera is configured with a resolution of 640×480 pixels and a frame rate of 15 frames per second. Only the depth stream is used in our application.

AIM 2025, Namur, Belgium

ACM ISBN 000-0-0000-0000-0/00/00
https://doi.org/00.0000/0000000.0000000

Each depth image is then divided into three vertical zones (left, center, and right). For each zone, the median distance is calculated to reduce the influence of noise and obtain a more stable estimate of the distance to obstacles.

Two distance thresholds are defined to assess the level of danger: an attention zone when the obstacle is located less than 2 meters away, and an alert zone when it is less than 1 meter away. Any obstacle located beyond 2 meters is considered non-hazardous.

3.1.3 Processing Unit. The processing unit used is a Raspberry Pi 3, which handles data acquisition, processing, and communication with the Arduino board. The Raspberry Pi runs multiple threads to manage audio and video data streams, synchronization, and command generation for the output layer.

The producer-consumer architecture is implemented with separate queues for the audio and video data streams. Each queue has a maximum size to prevent memory overflow. The queues use a LIFO (Last-In-First-Out) strategy to ensure real-time performance by prioritizing the most recent data. To avoid blocking the processing and communication threads, when a queue reaches capacity, the three oldest items are automatically dropped, and the dropped count is tracked for monitoring purposes.

This architecture allows for efficient and parallel processing of multimodal data, ensuring real-time feedback to the user. The system monitors queue statistics including total items processed, current queue sizes, and processing frequencies, which are logged every 10 seconds to track system performance.

3.1.4 Arduino Board. In this project, Arduino is used as an interface between the central processing unit (Raspberry Pi) and the physical actuators (vibration motors). It receives already-processed instructions that specify how the motors should behave.

First, the Raspberry Pi sends data divided into three components: where the danger comes from, the intensity, and the duration of the vibration. The Arduino receives the information, interprets it, and converts it into appropriate signals for the vibration motors.

In the overall architecture, the Arduino is responsible for executing the physical output (vibration motors) in real time. It does not process data; it applies the received instructions to the vibration motors.

3.2 Project Implementation

First, to ensure an easy setup of the project, UV was used to manage the Python dependencies. All required libraries are listed in the `pyproject.toml` file, allowing for a straightforward installation process.

3.2.1 Code Structure. The codebase is organized into several modules, each responsible for a specific functionality of the system. The entry point of the project is the `main.py` script, which orchestrates the different threads. This script initializes and starts all the necessary threads for data acquisition, processing, communication, and monitoring of the sent data. The main script supports several command-line arguments: `--no-audio` to disable audio capture, `--no-video` to disable video capture, `--debug` to enable verbose logging, and `--simulate` to simulate inputs for testing purposes. Each producer (audio and video) runs as a daemon thread to ensure proper shutdown when the main process terminates.

In addition to this main script, a secondary Python script (`monitor_serial.py`) is provided to monitor the serial communication between the Raspberry Pi and the Arduino board. This lightweight monitoring script connects to the Arduino via `/dev/ttyACM0` at 115200 baud and displays all incoming messages in real time. To facilitate the startup of the entire system along with the monitoring script, a shell script (`start.sh`) is provided that launches both the main system and the monitor simultaneously. The script captures the process ID of the main system and ensures proper cleanup when the monitor is stopped with Ctrl+C.

3.2.2 Queue Manager. To simplify the management of the queues used for inter-thread communication, a dedicated module was implemented: `queue_manager.py`. This module creates a singleton class, `QueueManager`, which encapsulates the functionality of all queues used in the project. The `QueueManager` class initializes five separate queues: `micro_queue` and `video_queue` (maxsize=10) for raw sensor data, `audio_processed_queue` and `video_processed_queue` (maxsize=5) for processed features, and `arduino_queue` (maxsize=5) which was initially designed for Arduino commands but is not used in the current implementation.

Each queue is configured with a maximum size to prevent memory overflow. The module provides dedicated methods for adding and retrieving data from each queue (`put_micro_data()`, `get_micro_data()`, `put_video_data()`, etc.). When adding data, the current timestamp is automatically attached as a tuple `(data, timestamp)` to facilitate temporal synchronization. The queue manager implements comprehensive monitoring, tracking both the total number of items processed and the number of dropped items for each queue. When a queue becomes full, the system drops the three oldest items before adding the new data, preventing blocking while maintaining real-time responsiveness. The `get_queue_stats()` method provides detailed statistics about queue usage, enabling performance monitoring and system diagnostics. Encapsulating the queue management logic within a dedicated module simplifies the code, making it easier to understand and maintain.

3.2.3 Sensors Implementation.

3.2.4 Video Sensor (RealSense D435i). The `camera.py` code uses the RealSense Depth D435i camera to detect obstacles in front of the cane and estimate the distances to them in real time.

It uses the pyrealsense2 library to communicate with the camera. The camera is configured with a resolution of 640×480 pixels at 15 FPS to capture depth images. At startup, a detection phase checks if the camera is available. If it is not detected, the system switches to simulation mode, generating artificial distances to test the code.

The `process_frame()` function divides each image into three vertical zones corresponding to the left, center, and right directions. For each zone, the distance is estimated from the depth data measured by the camera. After converting this perceived data to meters, a median is applied to all the values to obtain a representative value for each image using the `median_calculator()` function. This approach helps limit measurement errors and sensor noise. To make the measurements even more stable, the system takes into account distances calculated over several successive images for each zone, instead of relying on a single image; this is called smoothing.

The distances from this history are stored, and then a second median is calculated to obtain a more stable final distance. Based on the smoothed central distance, the `danger_zone()` function determines which danger zone the user is in (peaceful, caution, or alert). Beyond 2 meters, the user is in peaceful mode, indicating that there are no obstacles that could pose a real danger. Between 2 and 1 meter, the user enters a caution zone, where they know there is an obstacle in front of them and that they must avoid it. Within 1 meter, the user knows they must move if they don't want to collide with the obstacle.

Finally, the `start_video_capture()` function manages the main operation of the video module. It has a loop that runs continuously to capture images, but also to calculate distances and detect nearby obstacles in each area. Based on this information, a direction is chosen to avoid the obstacle, and this direction is determined by selecting the area where the distance is greatest (where there are no obstacles). The essential data is then sent to the video data queue via the `queue_manager`.

3.2.5 Audio Sensor (Microphone). A Python script in the `micro` folder is used to capture the data recorded by the microphone. Audio capture relies on the `sounddevice` library, which creates an audio stream that captures samples in real time. The signal is retrieved as chunks, then sent into a queue to be processed.

To achieve this, the script first searches for the desired audio device. Then, it defines various parameters (chunk size, chunk duration, etc.). After that, audio capture is performed through an audio stream (`InputStream`) associated with a callback function. Each time new data is received, the samples are added to an internal buffer. As soon as this buffer contains enough samples to form a complete chunk, the chunk is extracted and converted into a NumPy array before being sent into a queue for processing. Additionally, the module provides a simulation mode to facilitate testing and debugging of the system without requiring physical audio hardware.

3.2.6 Raspberry Pi Module. As the Raspberry Pi is the core processing unit of the system, it hosts several modules that handle different aspects of data processing and communication.

The `raspberry.py` module is the main module that integrates all functionalities, managing data flow between the audio and video modules, synchronization buffer, and Arduino communication.

As discussed earlier, the project architecture is based on a producer-consumer model, with multiple threads handling different data streams. Each sensor modality has its own dedicated thread for data acquisition and preprocessing, which then feeds the shared queues.

The `raspberry.py` script is responsible for all the heavy processing tasks. For audio data, the `heavy_audio_processing()` function computes the Root Mean Square (RMS) of the signal, converts it to dB level using the formula $20 \times \log_{10}(\text{rms}/\text{reference})$, and classifies the sound into four categories: "Chillax" (< -45 dB), "Some noise" (-45 to -30 dB), "Be Careful" (-30 to -15 dB), or "Danger" (> -15 dB). Additionally, it performs Fast Fourier Transform (FFT) analysis to detect the dominant frequency in the audio signal. For video data, the `heavy_video_processing()` function analyzes obstacle positions and distances, computing a danger level (0–3) based on the number and position of detected obstacles, with special priority given to center obstacles. The function also assigns risk classifications ("safe", "medium", "high", or "critical") corresponding to each danger level.

The script runs separate processing threads for audio and video data (`micro_processing_thread()` and `video_processing_thread()`), each continuously consuming data from their respective queues. These processed results are then added to intermediate queues before being synchronized using the `sync_buffer.py` module. Finally, commands are generated for the Arduino board based on the synchronized information, and these commands are sent via serial communication at 115200 baud.

Synchronization Buffer. In order to correctly merge the information from the audio and video sensors, a time synchronization mechanism has been implemented using the `sync_buffer.py` module. This module is responsible for associating the processed audio and video data that correspond to the same point in time, despite different capture frequencies and processing times.

The system is based on two separate buffers, implemented as limited-size `deque` structures, one for audio data and the other for video data. Each buffer can hold up to five items. The stored data is encapsulated in `SensorData` objects, which contain three essential pieces of information: a precise timestamp, the processed sensor data, and the origin of the data (audio or video).

When a new audio or video result is produced by the respective processing modules, it is added to the corresponding buffer with a timestamp obtained using the `time.time()` function.

In order to avoid the accumulation of obsolete data and maintain low latency, an automatic cleaning mechanism is applied. Any data older than a maximum threshold of 150 milliseconds is deleted from the buffer.

The synchronization itself is performed by the `get_synchronized_pair()` method. This method compares all possible combinations between the elements present in the audio and video buffers in order to determine the pair with the smallest time difference. If this difference is less than a synchronization threshold set at 50 milliseconds, the pair is considered valid. The elements used are then removed from the buffers to prevent any subsequent reuse. The value of 50 milliseconds was chosen arbitrarily based on what worked best for our tests.

If no sufficiently synchronized pair is found, the method returns a null value, indicating that no reliable multimodal fusion can be performed at that time. The module also provides methods for retrieving the most recent audio or video data, which are used as a fallback mechanism when one of the two modalities is temporarily unavailable.

Intensity Calculator. To adjust the output intensities based on the processed sensor data, the `intensity_calculator.py` script is used. This module implements the `IntensityCalculator` class with two static methods for converting sensor data into intensity values ranging from 0 to 100. These values represent the perceived level of risk and are used to modulate the intensity of the feedback provided to the user. The class is based on two distinct static methods, corresponding to the two sensory modalities of the system: audio and vision.

Audio conversion: The `audio_to_intensity()` method converts the ambient sound level, expressed in decibels (dB), into an intensity between 0 and 100. This conversion is performed using a piecewise linear function, defined over several sound level intervals in order to gradually reflect the increase in risk associated with ambient noise. Four sound level ranges are defined:

- For levels below −45 dB, corresponding to a very quiet environment, the intensity varies linearly from 0 to 20.
- Between −45 dB and −30 dB, the environment is considered moderately noisy, and the intensity increases gradually from 20 to 50.
- Between −30 dB and −15 dB, the noise becomes significant and the intensity is between 50 and 80.
- Above −15 dB, the sound level is considered potentially dangerous and the intensity is saturated between 80 and 100 in order to avoid excessive values.

Visual conversion by zone: The `vision_to_intensity_by_zone()` method converts the distances estimated by the depth camera into distinct intensities for the three spatial zones considered: left, center, and right. For each zone, a default distance of 5 meters is used in the absence of a valid measurement. The base intensity is calculated based on the minimum distance detected in the zone:

- For distances greater than 2 meters, the intensity remains low (0 to 15), indicating a non-hazardous situation.
- For distances between 1 and 2 meters, the intensity increases linearly from 15 to 70, indicating an area requiring vigilance.
- For distances less than 1 meter, the intensity is high (70 to 100), signaling immediate danger.

In addition to this distance-based estimate, a reinforcement mechanism is applied when an obstacle is explicitly detected in a given area. In this case, a fixed bonus of 20 points is added to the calculated intensity. This increase allows obstacles clearly identified by the vision algorithm to be prioritized, even if their distance remains slightly above the critical thresholds. Finally, all intensities are limited between 0 and 100 to ensure system stability and compatibility with the output device control protocol.

Message Generator. A final relevant module is the `lcr_message_generator.py`, which is responsible for formatting the computed intensity levels into the specific command protocol required by the Arduino board. This module implements the `LCRMessageGenerator` class that maintains state including the last sent message and a message counter for tracking.

The command format used is `LxxxCxxxRxxx`, where `L`, `C`, and `R` represent the left, center, and right output intensities, respectively, each followed by a three-digit zero-padded intensity value (000–100). The module provides two generation methods: `generate_synchronized_message()` for cases where both audio and video data are available, and `generate_fallback_message()` for single-modality scenarios.

The synchronization method implements a sophisticated weighted averaging scheme to merge audio and video intensities. For the center zone, the formula is: $(4 \times \text{vision\_intensity} + \text{audio\_intensity})/5$, giving 80% weight to visual data and 20% to audio. For lateral zones (left and right), audio influence is further reduced to 70% of its original value: $(4 \times \text{vision\_intensity} + 0.7 \times \text{audio\_intensity})/5$. This weighting strategy prioritizes visual obstacle detection while still incorporating ambient sound information, particularly emphasizing visual data for the critical center zone where the user is heading. The weighted average calculation ensures smooth transitions between different environmental conditions.

These weighting values can be modified in the `lcr_message_generator.py` module to adjust the system's sensitivity according to user preferences, and this opens the possibility for future improvements, such as implementing a user-tunable sensitivity setting.

All these modules work together seamlessly to ensure that the system operates in real time, providing appropriate feedback to the user based on the multimodal sensory input. The entire processing pipeline—from sensor data acquisition through queue management, processing, synchronization, intensity calculation, and message generation—is designed to maintain low latency and high responsiveness, crucial for assistive navigation applications.

3.2.7 Arduino Board Implementation. The Arduino is used as a hardware interface between the Raspberry Pi and the output devices. Its role is limited to receiving commands sent by the Raspberry Pi in real time and applying them to the vibration motors.

The Arduino communicates with the Raspberry Pi via a serial connection configured at 115200 baud. Commands are transmitted using a simple and robust format: `LxxxCyyyRzzz`, where each value represents the intensity of the danger expressed between 0 and 100. The letters stand for left, center, and right directions.

Upon reception, each command is directly delivered to the Arduino side. The received values in the range of 0–100 are converted to an 8-bit scale (0–255) required for proper interpretation by the Arduino. In this part, there is no buffer or queue; each new command immediately replaces the previous output state. This choice ensures that the feedback always reflects the most recent perception of the environment.

Originally, the system was designed to accommodate three vibration motors, one for each spatial direction. Due to hardware and voltage constraints, these motors were first replaced by simple LEDs. However, the brightness of the single color of the LEDs was not fully reliable because the intensity values were difficult to distinguish visually, making it hard to verify whether the Arduino was correctly interpreting the intensity of the danger. To overcome this limitation, RGB LEDs are used as a visual proxy for the haptic feedback. Each LED corresponds to one direction, and its color and brightness simulate the vibration intensity. The color can smoothly transition from green (safe) through yellow and orange (increasing risk) to red (critical danger), providing a clearer and more intuitive representation of the intensity level.

The Arduino code is structured into multiple C++ files:

- A main file (`main.cpp`) is used to handle initialization and the main execution loop.
- A get file (`get.cpp`) that manages the serial input.
- The logic file (`logic.cpp`) is used to parse and convert the intensity values. The RGB architecture is also present here.
- The send file (`send.cpp`) is used to abstract the hardware control of the output devices.

In addition, three corresponding header files (`get.h`, `logic.h`, and `send.h`) are used so that the functions of these files can be used across the different files of the project.

Several hardware considerations had to be addressed. Indeed, directly powering vibration motors from the Arduino was unsafe due to current limitations and voltage mismatches, which could damage the Arduino board. Using LEDs as a placeholder allowed functional testing of the communication from the Raspberry Pi to the outputs and control logic while avoiding electrical risks and hardware damage. Integrating proper motor drivers remains future work, as well as environmental tests.

4 Contribution

This project was developed by our student group; each team member was responsible for specific components of the system to ensure a clear separation of tasks and efficient development.

Edwyn Eben was responsible for the video detection module with the RealSense Depth D435i camera, obtaining depth data, obstacle detection, and distance-based risk assessment.

Louca Mathieu was responsible for capturing audio data (`micro.py`) and synchronizing audio and visual data.

Nathan Lambrechts developed the Arduino board code, handling serial communication and output device control. The hardware integration and safety considerations were also managed by him.

Florian Stormacq implemented the Raspberry Pi processing unit, including configuration and dependencies management, multi-threading architecture, the initial queue manager, and the main processing pipeline.

5 Evaluation

The system was not evaluated in real-world conditions due to time constraints and the unavailability of certain hardware components. However, each prototype module was tested individually to ensure correct functionality. Additionally, the prototype was tested in a controlled environment to validate the overall system integration, as well as the visual feedback provided by the LED strip.

6 Discussion

6.1 Limitations

To begin with, the choice of hardware components imposed certain limitations on the system's performance. To ensure real-time processing, the Raspberry Pi 3 was selected as the processing unit. To replicate this project, it is important to use a Raspberry Pi 3 or a more powerful model, as older versions struggle to handle the computational load of processing all sensor data in real time.

Another limitation of the prototype is the limited processing rate of the audio module. While the video module can process data at a rate of approximately 15 frames per second, the audio module is limited to processing only 6 items per second. This leads to a situation where audio data becomes the bottleneck for the overall system performance, as the synchronization buffer assembles data based on the produced timestamps. Consequently, the system can only provide updates at a maximum rate of 6 items per second, which may not be sufficient for fast-changing environments.

A final limitation to consider is the absence of real-world testing with visually impaired users. Due to time constraints and the unavailability of certain hardware components, the system could not be evaluated in practical scenarios. Real-world testing is crucial for assessing the effectiveness and usability of the system, as well as for gathering feedback from actual users to inform future improvements.

6.2 Problems Encountered

During the development of this project, several challenges were encountered that required problem-solving and adaptation. Some of the most notable issues included:

(1) Configuring the Raspberry Pi to work with the RealSense camera, which involved building the PyRealSense2 library from source. This process was time-consuming and error-prone, requiring careful attention to detail and extensive troubleshooting. For more information, please refer to the documentation provided in the GitHub repository.
(2) The initial project aimed to use vibration motors as output devices, one for each direction (left, center, right). These motors would have been placed on a handlebar to provide haptic feedback to the user. However, due to time limitations and difficulties in assembling the desired prototype, the vibration motors were replaced by an LED strip for visual feedback. This change allowed for a simpler and quicker implementation while still demonstrating the core functionality of the system.
(3) Limited documentation for the PyRealSense2 library made it difficult to install and utilize the camera effectively across different operating systems.
(4) Debugging multi-threaded applications proved to be, as expected, a complex task. Ensuring that all threads operated correctly and efficiently required careful design, testing, and the implementation of comprehensive logging and monitoring systems.
(5) Synchronizing audio and video data streams accurately to ensure that the output feedback corresponded correctly to the sensory input, particularly given the different capture rates and processing times of each modality.
(6) Saturating the Arduino communication buffer when sending commands too rapidly, which led to unresponsive output devices. To address this issue, a monitoring script (`monitor_serial.py`) was implemented to visualize the commands being sent to the Arduino in real time, allowing for better debugging and optimization of the communication process.

6.3 Ideas for Future Work

As this project was developed within a limited timeframe, there are several areas of improvement and expansion that could be explored in future work. Some potential directions for future development include:

(1) Enhancing user interaction by implementing a user interface that allows for real-time adjustment of system parameters, such as the weighted averaging coefficients between audio and video inputs, sensitivity thresholds, and output modes.
(2) Integrating the originally intended vibration motors to provide haptic feedback and evaluating their effectiveness compared to the visual feedback provided by the LED strip. This modification would not require significant changes to the existing codebase, as the communication protocol with the Arduino would remain the same.
(3) Optimizing the audio processing module to increase the data processing rate beyond the current 6 items per second, allowing for more frequent updates and smoother feedback. This

could involve implementing more efficient FFT algorithms or utilizing hardware acceleration.

(4) Conducting real-world testing with visually impaired users to evaluate the system's performance in various environments and scenarios, providing valuable insights into its practical usability, effectiveness, and user acceptance. Such studies would also help identify necessary adjustments to the sensitivity settings and feedback mechanisms.

(5) Implementing adaptive sensitivity that automatically adjusts based on environmental conditions and user preferences learned over time.

7 Conclusion

This project successfully developed a multimodal assistive system that integrates audio and video sensors and provides real-time feedback to the user through a set of output devices. The system architecture, based on a producer-consumer model with multi-threading and sophisticated queue management, effectively handles the complexities of data acquisition, time-sensitive processing, temporal synchronization, and communication. The implementation includes comprehensive modules for audio and video processing, intensity calculation using weighted averaging, and command generation following a well-defined protocol.

As with all prototype projects, there are several areas for improvement that could be explored in future work, including enhancing user interaction through adaptive interfaces, optimizing processing modules for higher throughput, integrating the intended haptic feedback hardware, and conducting real-world evaluations with target users. Overall, this project demonstrates the feasibility and potential of multimodal systems for enhancing mobility and safety for visually impaired individuals through intelligent sensory feedback.

Open Science

The complete code for this project is available on GitHub at the following repository: https://github.com/fstormacq/Master1-IIA-project. A video demonstration of the prototype is also available online[1].

Acknowledgments

This work was supported by the University of Namur, the Namur Digital Institute (NaDI), and the Research Center on Information Systems Engineering (PReCISE). The authors would like to thank all individuals who contributed to this project. A particular thanks to the providers of the hardware components that made this research possible.

References

[1] Andrés Manuel Loor Guaycha, José Andrés Matute Palma, and Patricia Nuñez Panta. 2026. Functional Prototype of an Ultrasonic Smart Cane for Visually Impaired People. In *Lect. Notes Networks Syst.*, Vol. 1429 LNNS. Springer Science and Business Media Deutschland GmbH, 18–30. doi:10.1007/978-3-031-99339-8_2 Journal Abbreviation: Lect. Notes Networks Syst..

[2] Austine S. Manuel, Alana Ance John, Hamna Rafeeq, Thalhah Anas, and Manoj V. Thomas. 2026. AI-Enhanced Glasses for the Visually Impaired. In *Lect. Notes Networks Syst.*, Vol. 1463 LNNS. Springer Science and Business Media Deutschland GmbH, 397–411. doi:10.1007/978-981-96-7514-2_32 Journal Abbreviation: Lect. Notes Networks Syst..

[3] Jingjing Xu, Caiyi Wang, Yancheng Li, Xuantuo Huang, Meina Zhao, Zhuoqun Shen, Yiding Liu, Yuxin Wan, Fengrong Sun, Jianhua Zhang, and Shengyong Xu. 2025. Multimodal Navigation and Virtual Companion System: A Wearable Device Assisting Blind People in Independent Travel. *Sensors* 25, 13 (2025). doi:10.3390/s25134223 Publisher: Multidisciplinary Digital Publishing Institute (MDPI).

[1] https://unamurbe-my.sharepoint.com/:v:/g/personal/florian_stormacq_student_unamur_be/IQCELicuX961Qoh6YVtFpLJMAbkyzlf1O0I0-dX3hws1TNk?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciO
e=0QfZOO (available until June 30, 2026)