Technical Documentation

Edit New page

Jump to bottom

anuunchin edited this page yesterday · 1 revision

Technical Documentation

This document breaks down the technical implementation of OptiBot's real-time object detection pipeline. It explains how we stream video via WebRTC, process frames with YOLO and MiDaS, and wire everything together.

1. Architecture

The system follows a three-tier architecture with microservices connected via WebRTC.

- The **Webcam service** captures camera feed and streams raw video.
- The **Analyzer service** receives the stream, performs YOLO object detection and MiDaS depth estimation, and forwards the annotated video.
- The **Frontend** displays the processed stream in the browser.

For detailed architecture diagrams and explanations, see the Software Architecture Document.

2. AI models

At the core of our processing pipeline are two AI models.

We use **YOLOv8 Nano** for object detection. The model outputs bounding boxes with class IDs and confidence scores for 80 COCO object classes. We cache inference results to maintain smooth frame rates. See `src/backend/common/core/detector.py` .

For distance estimation, we use **MiDaS Small** to predict depth from single-camera frames. The depth estimator samples the center region of each detected object and converts relative depth values to metric distances. See `src/backend/common/utils/geometry.py` .

Both models are initialized once at startup.

3. Webcam service: Capturing and streaming

The pipeline begins with the webcam service capturing camera frames and streaming them over WebRTC. We use an async background loop to continuously read frames. Camera capture is managed with reference counting to support multiple consumers. When a client connects, the service creates a WebRTC connection and streams frames. Each frame is flipped horizontally for mirror effect, converted from BGR to RGB, and encoded to H.264 for efficient streaming.

See `src/backend/webcam/` for implementation.

4. Analyzer service: Receiving and processing

The analyzer acts as a middleware, connecting to the webcam as a client while serving the browser as a server. This role allows it to intercept the video stream for processing.

The analyzer establishes a WebRTC connection to the webcam service to receive raw frames. It then wraps this incoming stream with AI processing (YOLO detection and MiDaS depth estimation), renders the overlays, and streams the annotated frames to the browser through a separate WebRTC connection.

See `src/backend/analyzer/` for implementation.

5. Drawing detection overlays

Once the analyzer runs object detection and depth estimation, it renders the results directly onto video frames before streaming to the browser. The implementation uses OpenCV's drawing functions. For each detection, `draw_detections()` draws a green rectangle using `cv2.rectangle()` and adds a label with format `"{class_id}: {confidence:.2f} {distance:.1f}m"` using `cv2.putText()`.

Source code `src/backend/common/utils/geometry.py`.

6. Frame transformation pipeline

As frames flow through the three services, they undergo several transformations to match the requirements of different libraries and protocols.

The **webcam service** captures raw frames from the camera, flips them for a mirror effect, and converts them to the color format expected by WebRTC. The frames are then encoded and streamed to the analyzer service.

The **analyzer service** receives the stream, converts frames to the format expected by the AI models, runs object detection and depth estimation, draws overlays on the frames, and streams the annotated video to the browser.

The **frontend** receives the encoded stream, decodes it, and renders it in a standard video element.

The main transformations are color space conversions to match different library expectations, format conversions between video and array representations, and video encoding for efficient network transmission. See `src/backend/webcam/tracks.py` and `src/backend/analyzer/tracks.py` .

7. Frontend: Displaying the stream

The pipeline ends with the frontend displaying the annotated video stream. The browser establishes a WebRTC connection to receive the video stream from the analyzer. It negotiates the connection by exchanging session details with the server, then receives the video track and displays it in a standard HTML `<video>` element.

See `src/frontend/src/hooks/useWebRTCPlayer.ts` and `WebRTCStreamPlayer.tsx` .

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/amosproj/amos2025ws04-robot-visual-perception.wiki.git
```