dotLumen Software Challenge 2 Submission

Submission by Vlad-Ioan Durdeu

Introduction

When I read the challenge description, I knew this was right up my alley. Machine learning, image analysis, object detection. All things I've wanted to explore more seriously, and this gave me a solid reason to dive in. I chose Python from the start because I've used it professionally, and honestly, it's just the best tool when I need to test, fail fast, and iterate quickly.

This challenge also came at the right moment. I recently decided that my bachelor's project would be on a topic related to machine learning, and this lined up perfectly with that goal. It was a great chance to apply my skills, learn something new, and get hands-on experience building something that could actually help people, especially those with visual impairments.

Thanks for the opportunity to go deep into a field I care about, while also tackling a real-world problem. Let's get into it.

Research

I started with research, searching for applications of machine learning and image detection and problems I can fix using it and Python code. I picked Python because I already used it professionally and also it's the best language when prototyping and building fast.

For researched I used Google alongside OpenAi's Deep Search feature, which is really useful to get good research fast, and by reading the sources and googling I can be confident that I got a pretty good picture of the problem and potential solutions.

Here are the 5 problems I came up with, inspired by problem validation research and what technical solutions seemed the most viable. Some of them are also suggested in the document:

1. Urban object detection with distance estimation for warning the user of pending dangers and interest points (for example, while traversing a street to announce that cars are coming and if they are slowing down or not) [2] [3]
2. Sidewalk, road and crosswalk detection for helping the user navigate. This can be paired with an AI assistant that will allow the user to set destinations in google maps API and then the assistant will guide the user to the destination. [2] [3] [4]
3. Text detection and playing audio of the text or paired with idea 2 for confirming the destination was reached if that is the case [1]

4. Using image detection to detect known faces of people and their face visions. [2] [3]
5. Text detection may be also used to confirm the bus or tram line. This was inspired by actually seeing a blind person take the 25 route bus here in Cluj. I was worried about him more than for myself. Luckily the stations and the bus has useful audio queues but the station one seemed the most unreliable, especially when multiple busses are coming in. [1]

After that, I started searching for suitable models for implementing the first and maybe second image detection task, found the first model I want to try out: yolov8n. This model will be used exclusively for the next 3 versions of the code. Then I will add SegFormer-B0 (or higher) fine-tuned on Cityscapes for better detection while also being able to detect objects needed for task 2, so I hit two tasks with one code. But I will keep around the other model just for comparison.

There are 7 versions of this code, each was iterated over the last to fix mistakes and deliver something useful. I will tell the story of the many tries and versions the code went through, and then I will go first through the last version and explain every class and what it does. There is still a lot of work and fine-tuning, but I am confident this is a strong starting point. I am also running out of time and I have to submit this soon.

Writing the code

First version code

What I want to do:

- I want to make the detection of objects and their distance
- I will render all the objects boxes on every detected object and the estimated distance from the user

Result can be found in `v1_annotated.mp4` it works, but it is very janky and needs some smoothing. I also took forever to finish because I forgot to trim the video and it did on all of it.

- Also, from now on I will generate only the first 30 seconds
- The good thing is that cars and people are detected very well, but anything else is not detected at all or very poorly.
- Distance is also not always detected properly, but I will focus on detecting objects for now.

Second version

This new version, found in file `v2_30s_annotated.mp4` looks much better and clearer, here are the changes:

- Added a confidence filter to get rid of random false boxes

- Used EMA smoothing with `self.alpha = 0.4` to stop distance labels from jumping around
- Color coded boxes to quickly show what's safe (green) and what's not (yellow/red)
- Made text background slightly opaque so it's readable even when the video is messy
- Ignored tiny boxes since they’re usually noise or way too far to matter

But I want to detect even more objects, not just cars and people. Time to switch to a bigger model and work on some improvements.

Third version

This new version, found in file `v3_60s_annotated.mp4`, looks even more promising, but this came at a cost. Here are the changes:

- Switched from the small version yolo8n.pt to the medium version. It takes forever to run, but detects more objects
- Reduced the culling minim size from 30x30 to 10x10

Forth version

Forth version demo can be found at `v4_30s_annotated.mp4`, I wanted to try a new model because this was was lacking in features. Here is what I tried:

- Tried to switch to model detectron2 but had import error and after trying multiple options to import, nothing worked.
- Uses SegFormer-B0 (or higher) fine-tuned on Cityscapes for street-scene segmentation.
  I imported cv2 in my python code and.. I forgot to switch to RGB to BGR and now this detection is useless and detects the sky as a pavement. Great.

Fifth version

The fifth demo can be found at `v5_30s_annotated.mp4`, here are the patch notes:

- Switched from BGR to RGB before sending frames into SegFormer, we are working in OpenCV after all
- Highlighted road in red and pavement in blue, this doesn’t work properly, everything ends up being road
- Upsampled the 1024×1024 map back to video resolution and did a quick morphological close to clean holes
- Made the SegFormer init flexible so I can swap from B0 to B1, B2, B3, B4 or B5 if needed
- Did some small cleanup, better variable names, added docstrings, but didn’t touch the main pipeline. This can really add up.

Sixth version

Sixth version found at `v6_120s_annotated.mp4` was a debug run to see if I can separate the roads and the sidewalk. This are the patch notes.

- Put SegFormer in `eval` mode so BatchNorm and dropout act right during inference
- Added two debug prints in `segment()`:
  - `print("Unique seg classes:", unique_ids)`
  - `print("Sidewalk pixels:", np.sum(seg_map == side_id))`
- Commented out the 5×5 morphological closing to see raw segmentation without smoothing
- In `overlay_segmentation()`:
  - Road still gets a 50% red blend (`[0,0,255]`)
  - Sidewalk now gets an 80% yellow blend (`[0,255,255]`), replaced the old 50% blue
  - Drew a bright yellow contour around sidewalk areas using `findContours` + `drawContours` so edges pop more. As per the theme of this document, this didn't work as expected.

It didn't work, but It was interesting to see that in the park, the sidewalk was marked, even though they are labeled road. Well, at least I can see just like drivers in Bucharest see the road when trying to park or when pulling a 'romanian' to skip some traffic.

Seventh version

For the last version (demo at `v7_60s_annotated`), I knew I had to go big or go home. So I created a overline for all classes that are found in id2label and showed them on the screen. It's a mess of tags and colors, but at least I can confirm that this model it's just better than the other one and if used correctly, can truly help the visually impaired to navigate the streets of our city. Patch notes:

- Looped over all classes in `id2label` , not just road and sidewalk
- Created a binary mask for each class (`mask = (seg_map == class_id)`) and skipped the ones with zero pixels
- Used a simple modulus trick to give each class a unique, repeatable color instead of hardcoding red/blue/yellow
- Used `cv2.findContours` + `cv2.drawContours` to draw outlines instead of just filling stuff in
- Calculated centroid using image moments (fallback to bounding box if needed) to place the label
- Drew a background box and put the class name in white text on top of the centroid

Code Documentation

How to Run the Program

Running the program is simple once you've set everything up. This is how I did it every time during testing.

You just call the script from the terminal using:
`python script.py --rgb path/to/rgb_video.mp4 --depth path/to/depth_video.mp4 --out output.mp4 --duration 30 --conf 0.3 --seg segformer`

Let me break it down a bit:

Required parameters:

- `--rgb`
  The path to the RGB video (color stream). Mandatory.
- `--depth`
  The path to the depth video (grayscale from the RealSense). Also mandatory.
- `--out`
  Where to save the final annotated video. Should be an `.mp4` file.

Optional parameters:

- `--duration`
  How many seconds of video you want to process. Default is 30 seconds.
  I usually kept it short while debugging to avoid wasting time on rendering.
- `--conf`
  Confidence threshold for object detection. Anything lower gets ignored. Default is 0.3.
  If you're getting too many random boxes, raise this to 0.5+.
- `--seg`
  What segmentation model to use:
  - `none` = no segmentation, just object detection
  - `segformer` = use SegFormer to label roads, sidewalks, etc.

Code Structure

Config

Handles arguments and settings. This is the starting point. When I run the script, this is where everything like input paths, how long to process, confidence threshold, and what segmentation model to use gets pulled in. It's used everywhere downstream.

```
class Config:
    """
    Stores everything we need: paths, duration, confidence, and segmentation type
    """
    def __init__(self, rgb_path, depth_path, output_path, duration, confidence,
segmentation):
        """
        Initialize config values from args
        """
        self.rgb_path = rgb_path
        self.depth_path = depth_path
        self.output_path = output_path
        self.duration = duration
        self.confidence = confidence
        self.segmentation = segmentation  # 'none' or 'segformer'

    @staticmethod
    def from_args():
        """
        Parse CLI args and return a Config object
        """
        parser = argparse.ArgumentParser(description='Assistive Vision Pipeline')
        parser.add_argument('--rgb', required=True, help='Path to RGB video')
        parser.add_argument('--depth', required=True, help='Path to depth video')
        parser.add_argument('--out', required=True, help='Path to output video')
        parser.add_argument('--duration', type=float, default=30.0, help='Max
processing duration in seconds')
        parser.add_argument('--conf', type=float, default=0.3, help='Confidence
threshold for detection')
        parser.add_argument('--seg', choices=['none','segformer'], default='none',
help='Which segmentation model to use')
        args = parser.parse_args()
        return Config(args.rgb, args.depth, args.out, args.duration, args.conf,
args.seg)
```

RGBCamera & DepthCamera

Both of these are just video readers. RGB gives the main visual feed, Depth gives the depth data. They also expose FPS and resolution so I don't hardcode anything later. If the videos aren't there or broken, these two will throw errors immediately.

```
class RGBCamera:
    """
    Handles reading frames from the RGB video
    """
    def __init__(self, video_path):
        """
        Open the video and grab metadata (fps, size)
        """
        self.cap = cv2.VideoCapture(video_path)
        if not self.cap.isOpened():
            raise IOError(f"Cannot open RGB video: {video_path}")
        self.fps = self.cap.get(cv2.CAP_PROP_FPS)
        self.width = int(self.cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        self.height = int(self.cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

    def read(self):
        """
        Return the next frame
        """
        return self.cap.read()

    def release(self):
        """
        Release the video file
        """
        self.cap.release()



class DepthCamera:
    """
    Handles reading and normalizing frames from the depth video    """    def
__init__(self, video_path):
        """
        Open the depth video file
        """
        self.cap = cv2.VideoCapture(video_path)
```

```
        if not self.cap.isOpened():
            raise IOError(f"Cannot open depth video: {video_path}")

    def read(self):
        """
        Read next frame and scale it to 0.0–10.0 meters
        """
        ret, frame = self.cap.read()
        if not ret:
            return ret, None
        depth_frame = frame.astype(np.float32) / 255.0 * 10.0
        return True, depth_frame

    def release(self):
        """
        Release the video file
        """
        self.cap.release()
```

FrameAligner

Right now, it doesn’t do anything. It just returns the frames as they are. But I left it in place in case later I need to sync the RGB and depth frames properly. If I need calibration or warping between the two, this is where it will go. I didn't need it for this project, but when starting I didn't know that.

```
class FrameAligner:
    """
    Stub class to align RGB and depth frames (currently a passthrough)
    """
    @staticmethod
    def align(rgb_frame, depth_frame):
        """
        Return frames as-is
        """
        return rgb_frame, depth_frame
```

ObjectDetector

This is the YOLO wrapper. It’s what finds the objects in the RGB video. It gives me bounding boxes, class IDs, and confidence scores. It’s the base of all detection logic. If this fails, everything downstream becomes garbage.

```
class ObjectDetector:
    """
    YOLOv8 wrapper for detecting objects in a frame
    """
    def __init__(self, model_path='yolov8n.pt'):
        """
        Load the YOLO model
        """
        self.model = YOLO(model_path)


    def detect(self, frame):
        """
        Run inference and return boxes, classes, confidences
        """
        results = self.model(frame)[0]
        boxes = results.boxes.xyxy.cpu().numpy()
        classes = results.boxes.cls.cpu().numpy().astype(int)
        confidences = results.boxes.conf.cpu().numpy()
        return boxes, classes, confidences
```

DistanceEstimator

This one is simple but useful. It takes the bounding boxes from YOLO and then uses the depth map to figure out how far away each object is. It doesn't do anything fancy, just takes a small patch at the center of the box and averages it, but it's enough to give decent distance estimates.

```
class DistanceEstimator:
    """
    Estimate distance to the center of each bounding box using depth info
    """
    def estimate(self, boxes, depth_frame):
```

```
        """
        For each box, average a small patch in the depth map at the box center
        """
        distances = []
        for box in boxes:
            x1, y1, x2, y2 = box.astype(int)
            cx, cy = (x1 + x2) // 2, (y1 + y2) // 2
            patch = depth_frame[max(cy-2, 0):min(cy+2, depth_frame.shape[0]),
                                max(cx-2, 0):min(cx+2, depth_frame.shape[1])]
            dist = float(np.nanmean(patch)) if patch.size else float('nan')
            distances.append(dist)
        return distances
```

Reporter

Draws everything on the video. This includes boxes, labels, and smoothed distance readings. It also handles writing the final video to disk. If something looks bad in the output video, it's usually a problem here. I also added EMA smoothing here to avoid flickering distances between frames.

```
class Reporter:
    """
    Handles drawing boxes and distances on frames, and writing output video    """
def __init__(self, output_path, frame_size, fps):
        """
        Set up video writer and EMA smoothing state        """        fourcc =
cv2.VideoWriter_fourcc(*'mp4v')
        self.writer = cv2.VideoWriter(output_path, fourcc, fps, frame_size)
        self.ema = {}
        self.alpha = 0.4

    def _color_for_distance(self, d):
        """
        Pick a box color based on distance        """        if d < 1.0:
            return (0, 0, 255)
        if d < 2.5:
            return (0, 255, 255)
        return (0, 255, 0)
```

```
    def draw(self, frame, boxes, classes, distances, confidences, class_names,
min_conf):
        """
        Draw boxes, distance labels, and smooth distance output using EMA
"""        for i, (box, cls, dist, conf) in enumerate(zip(boxes, classes,
distances, confidences)):
            if conf < min_conf:
                continue
            x1, y1, x2, y2 = box.astype(int)
            if (x2 - x1) < 10 or (y2 - y1) < 10:
                continue
            prev = self.ema.get(i, dist)
            dist_s = prev * (1 - self.alpha) + dist * self.alpha
            self.ema[i] = dist_s
            color = self._color_for_distance(dist_s)
            label = f"{class_names[cls]} {dist_s:.1f}m"
            (tw, th), _ = cv2.getTextSize(label, cv2.FONT_HERSHEY_SIMPLEX, 0.6, 2)
            cv2.rectangle(frame, (x1, y1-th-8), (x1+tw+4, y1), (0, 0, 0),
cv2.FILLED)
            cv2.rectangle(frame, (x1, y1), (x2, y2), color, 2)
            cv2.putText(frame, label, (x1+2, y1-4),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 2,
cv2.LINE_AA)
        return frame

    def write(self, frame):
        """
        Write the annotated frame to the output video        """
self.writer.write(frame)

    def release(self):
        """
        Release the writer        """        self.writer.release()
```

SegFormerDetector

Used when I want scene segmentation, like figuring out what's a road, sidewalk, building, etc. It loads the SegFormer model from Hugging Face and runs it frame by frame. Right now I'm using it mainly to identify sidewalks and roads to help navigation. Could be extended later to detect stuff like crosswalks or traffic signs.

```
class SegFormerDetector:
    """
    Handles loading and running the SegFormer segmentation model    """    def
__init__(self, device=None, backbone='b2'):
        """
        Load the model and put it into eval mode        """        self.device =
device or ('cuda' if torch.cuda.is_available() else 'cpu')
        model_name = f'nvidia/segformer-{backbone}-finetuned-cityscapes-1024-1024'
        self.feature_extractor =
SegformerFeatureExtractor.from_pretrained(model_name)
        self.model =
SegformerForSemanticSegmentation.from_pretrained(model_name).to(self.device)
        self.model.eval()

    def segment(self, frame: np.ndarray):
        """
        Run segmentation on an RGB frame and return resized segmentation map
"""        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        inputs = self.feature_extractor(images=rgb,
return_tensors='pt').to(self.device)
        with torch.no_grad():
            outputs = self.model(**inputs)
        logits = outputs.logits
        seg_map = logits.argmax(dim=1).squeeze().cpu().numpy().astype(np.uint8)

        unique_ids = np.unique(seg_map)
        print("Unique seg classes:", unique_ids)
        side_id = next(k for k, v in self.model.config.id2label.items() if v ==
'sidewalk')
        print("Sidewalk pixels:", np.sum(seg_map == side_id))

        seg_map = cv2.resize(seg_map, (frame.shape[1], frame.shape[0]),
interpolation=cv2.INTER_NEAREST)
        kernel = np.ones((2,2), np.uint8)
        seg_map = cv2.morphologyEx(seg_map, cv2.MORPH_CLOSE, kernel)
```

```
        return seg_map
```

---

overlay_segmentation

This is where I take the segmentation map and draw the contours + labels for each class. Instead of just slapping a color overlay on the frame, I go class by class, draw borders, and label them with readable tags. It makes everything cleaner and easier to understand visually.

```
def overlay_segmentation(frame: np.ndarray, seg_map: np.ndarray, id2label: dict):
    """
    Draw contours for each segmented class and label them on the frame    \
    """
    overlay = frame.copy()
    for class_id, class_name in id2label.items():
        mask = (seg_map == class_id).astype(np.uint8)
        if cv2.countNonZero(mask) == 0:
            continue
        color = (
            int((class_id * 47) % 256),
            int((class_id * 97) % 256),
            int((class_id * 167) % 256),
        )
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        cv2.drawContours(overlay, contours, -1, color, 2)
        M = cv2.moments(mask)
        if M["m00"] != 0:
            cx = int(M["m10"] / M["m00"])
            cy = int(M["m01"] / M["m00"])
        else:
            x, y, w, h = cv2.boundingRect(contours[0])
            cx, cy = x + w // 2, y + h // 2
        (tw, th), _ = cv2.getTextSize(class_name, cv2.FONT_HERSHEY_SIMPLEX, 0.5, 2)
        cv2.rectangle(overlay, (cx, cy - th - 5), (cx + tw + 4, cy), (0, 0, 0),
cv2.FILLED)
        cv2.putText(overlay, class_name, (cx + 2, cy - 3),
```

```
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)
    return overlay
```

VideoPipeline

This is the glue that ties everything together. Reads frames, runs object detection, estimates distances, draws annotations, and writes the result. It also checks whether segmentation is enabled or not and handles both paths. If the pipeline breaks, nothing works.

```
class VideoPipeline:
    """
    The main pipeline that ties together detection, segmentation, and video output
"""    def __init__(self, config: Config):
        """
        Initialize all components and prepare for processing        """
self.rgb_cam = RGBCamera(config.rgb_path)
        self.depth_cam = DepthCamera(config.depth_path)
        self.detector = ObjectDetector()
        self.estimator = DistanceEstimator()
        self.reporter = Reporter(
            config.output_path,
            (self.rgb_cam.width, self.rgb_cam.height),
            self.rgb_cam.fps
        )
        self.max_frames = int(self.rgb_cam.fps * config.duration)
        self.conf = config.confidence
        self.segmentation = config.segmentation
        if self.segmentation == 'segformer':
            self.seg_detector = SegFormerDetector()

    def run(self):
        """
        Run the full pipeline frame by frame        """        frame_count = 0
        class_names = self.detector.model.names
        while frame_count < self.max_frames:
            ret_rgb, rgb_frame = self.rgb_cam.read()
            ret_dep, depth_frame = self.depth_cam.read()
            if not ret_rgb or not ret_dep:
```

```
                break
            rgb_aligned, depth_aligned = FrameAligner.align(rgb_frame, depth_frame)
            boxes, classes, confs = self.detector.detect(rgb_aligned)
            dists = self.estimator.estimate(boxes, depth_aligned)
            annotated = self.reporter.draw(
                rgb_aligned, boxes, classes, dists,
                confs, class_names, self.conf
            )
            if self.segmentation == 'segformer':
                seg_map = self.seg_detector.segment(rgb_aligned)
                annotated = overlay_segmentation(
                    annotated, seg_map,
                    self.seg_detector.model.config.id2label
                )
            self.reporter.write(annotated)
            frame_count += 1
        self.release()
        print(f"Processed {frame_count} frames
({frame_count/self.rgb_cam.fps:.1f}s)")

    def release(self):
        """
        Release all resources        """        self.rgb_cam.release()
        self.depth_cam.release()
        self.reporter.release()
```

Entry point (`if __name__ == '__main__'`)

Takes the CLI args, builds the config, creates the pipeline, and runs it. This is the starting point when running the script from the terminal. Doesn’t do anything fancy, just connects everything and kicks it off.

```
if __name__ == '__main__':
    """
    CLI entry point to run the pipeline from command line args
    """
    cfg = Config.from_args()
    os.makedirs(os.path.dirname(cfg.output_path) or '.', exist_ok=True)
```

```
    pipeline = VideoPipeline(cfg)
    pipeline.run()
```

References

1. **OrCam MyEye.** *Wearable device for face recognition and text reading.*
   Integrates face detection/recognition and robust OCR in a single wearable device, supporting social interaction and environmental information access.
   theguardian.com
2. **Empathy Glasses.** *AR display recognizing facial expressions and gaze direction.*
   Provides discreet cues (audio or haptic) to signal interlocutor gaze direction, aiding in one-on-one or group conversations.
   springer.com
3. **YOLO-OD: Obstacle Detection for Visually Impaired Navigation Assistance.**
   Proposes an obstacle detection method for visual navigation assistance, improving the ability to detect and differentiate between different sized obstacles in outdoor environments.
   researchgate.net
4. **Assistive Systems for Visually Impaired Persons: Challenges and Opportunities for Navigation Assistance.**
   Highlights state-of-the-art assistive technology, tools, and systems for improving the daily lives of visually impaired people, evaluating multi-modal mobility assistance solutions for both indoor and outdoor environments.
   researchgate.net
5. **Mathew, J. (2022).** *Estimating depth for YOLOv5 object detection using Intel RealSense.*
   Demonstrates combining YOLO with D435i depth to get object distances.
   medium.com
6. **Zhang et al. (2024).** *Improved YOLOv5 with depth camera for blind assistance.*
   Used YOLO + RealSense D435i to detect objects and range them for an indoor aid.
   nature.com
7. **Asante et al. (2023).** *Obstacle avoidance for visually impaired using stereo.*
   Proposed selecting nearest obstacle via adaptable grid and YOLOv5, 95% accuracy in prioritizing hazards.
   mdpi.com
8. **SegFormer B0 fine-tuned on Sidewalks – HuggingFace Model (2021).**
   Example of a segmentation model that specifically labels sidewalk pixels.
   huggingface.co
9. **Intel RealSense Forum.** *Reflective surface depth issues.*
   Noted that glossy surfaces like water yield no depth – appearing as black in depth map.

10. **PyImageSearch (2018).** *OpenCV OCR and text recognition with Tesseract.*
    Tutorial on using EAST detector and Tesseract OCR for scene text.
    pyimagesearch.com
11. **Nanda, C. (2020).** *OCR: pytesseract vs EasyOCR.*
    Describes EasyOCR using CRAFT for detection and CRNN for recognition.
    medium.com
12. **Teed & Deng (2020).** *RAFT: Recurrent All-Pairs Field Transforms for Optical Flow.*
    RAFT model yields dense optical flow; Torchvision example used for flow visualization.
    pytorch.org
13. **Avi Singh (2017).** *Monocular Visual Odometry – Blog.*
    Explains feature-based VO with code, using KLT tracking and recoverPose; highlights need for scale input.
    avisingh599.github.io
14. **Wang et al. (2023).** *RGB-D-Based Stair Detection for the Visually Impaired.*
    Shows use of a four-channel RGB-D network to identify stairs, helping blind navigation by fusing depth into input.
    mdpi.com

Conclusion

Seven versions later, here we are. I started with shaky object detection, clunky boxes, and noisy outputs. Ended up with a working pipeline that can detect, estimate distances, segment environments, and annotate the video with contextual labels. It's not perfect, but it's getting close.

The goal was never to build the final product. It was to build something that works, can be iterated on, and proves that the core ideas are solid. This version does that. I also learned a lot during the process. Model choices, common failure points, OpenCV quirks, and the reality of working with noisy real-world input.

There are still things I want to improve. Better model fusion, smarter logic around danger zones, maybe audio feedback. But I’m out of time for now. This was one of the most useful and enjoyable challenges I’ve worked on.

One last thing: I did track my time. The development took around 5 hours (excluding waiting around for video generation), research and documentation took around 3 hours. This totals to 8 hours. This was done in 2 consecutive days working afternoons.

Thanks again for letting me explore this domain.
I wish you all the best,
Vlad-Ioan Durdeu