Einstein Vision
RBE549 Project 3

Sumukh Porwal, Piyush Thapar, Sarah Semy
MS Robotics Engineering
Worcester Polytechnic Institute
Email: sporwal@wpi.edu, pthapar@wpi.edu, srsemy@wpi.edu
Using 7 Late Days

***Abstract*—This project presents a vision-based dashboard designed to enhance driver awareness and support autonomous systems by visualizing how the vehicle perceives its environment. Inspired by Tesla's interface, it uses deep learning techniques such as object detection, depth mapping, and pose estimation to create a 3D scene of surrounding vehicles, pedestrians, lanes, and signs. The resulting visualizations help in both real-time understanding and debugging of the perception stack.**

I. Pipeline Overview

We worked with undistorted videos from 13 scenes recorded by a Tesla Model S, focusing on the front view for object visualization. Every 5th frame was processed using deep learning models to extract object type, position, and rotation, which we stored in a structured .txt format. This data was later used in Blender, where scripts read the .txt and render 3D models of vehicles, pedestrians, and traffic elements.

To build this pipeline, we began by understanding and integrating key components such as camera calibration, perception, and rendering.

II. Checkpoint 1: Basic Features

In the first phase, we implement basic features that are absolutely essential for a self-driving car. Includes the following: different types of lanes, vehicles (without classification), pedestrians (without pose), Traffic lights, and stop signs.

A. Lanes:

Accurate recognition of lane geometry is fundamental for autonomous–vehicle localisation and control. We used a Mask R-CNN model [1] on our custom dataset to segment four lane categories—*solid*, *dotted*, *divider*, and *double* lines—and post-process each binary mask to obtain a succinct set of centre-line points that are subsequently rendered in Blender. The point-sampling procedure is:

- obtain the binary mask produced by the network;
- extract all non–zero pixel coordinates (y_s, x_s); return an empty array if none exist;
- generate 2-6 evenly spaced target rows y_{new} between $\min(y_s)$ and $\max(y_s)$;
- for every $y \in y_{\text{new}}$:
  1) locate indices where $|y_s - y| \leq 0.5\,\text{px}$; if none exist, linearly interpolate an x value from (y_s, x_s);
  2) otherwise, compute the mean of the corresponding x_s;
- pair each computed x with its y to form a new point and cast the list to integers.

Sampling six such points per mask yields a robust lane skeleton. Each pixel's monocular depth estimate is then used to lift the 2-D points into 3-D metric space under the ground-plane assumption $Z = 0$. A Bézier curve is fitted through these world points, coloured according to the lane class, and inserted into the Blender scene. The same network also flags ground-painted arrows and other road signs. Empirically, the system performs best on highway imagery; performance degrades in urban scenes where markings are faint or missing.

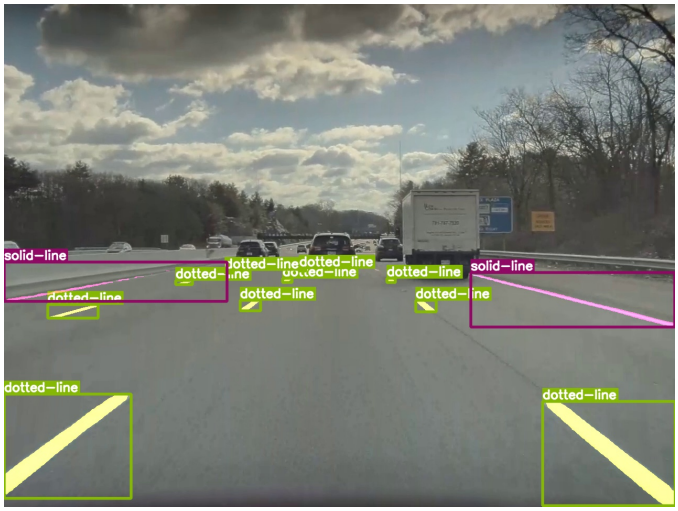

Fig. 1. Lanes and Road Sign Segmentation using Mask-RCNN

B. Vehicles, Pedestrians and Stop Signs:

For the first phase, we are required to detect the cars and not classify them. We have used `Detic` [2] for all the major detections, it's robust and easy to infer for an image. It outputs class name, bounding box and the confidence score for each of the detected object. The normalized depth map is obtained using `Marigold` [3]. Using the object's center coordinates of the bounding box for each of the class, we mapped it to the depth image. Using the camera intrinsics, pixel values, and normailized depth, we estimated the 3D point of the object and

visualized it in Blender. A rotation matrix was employed to rotate the 3D points with respect to the camera frame. The Z-axis (depth) is perpendicular to the image plane. The equation for the pixel to the 3D point is given below.

$$x = \frac{(u - c_x)z}{f_x}$$

$$y = \frac{(u - c_y)z}{f_y}$$

where f_x, f_y are the focal lengths and equal to 1594.7, 1607.7 mm. The principal points c_x, c_y are 655.2961, 414.3627 pixels respectively. z is the depth and u, v are the pixel points of the object in the image frame. The output x, y are the points in the 3D world frame.

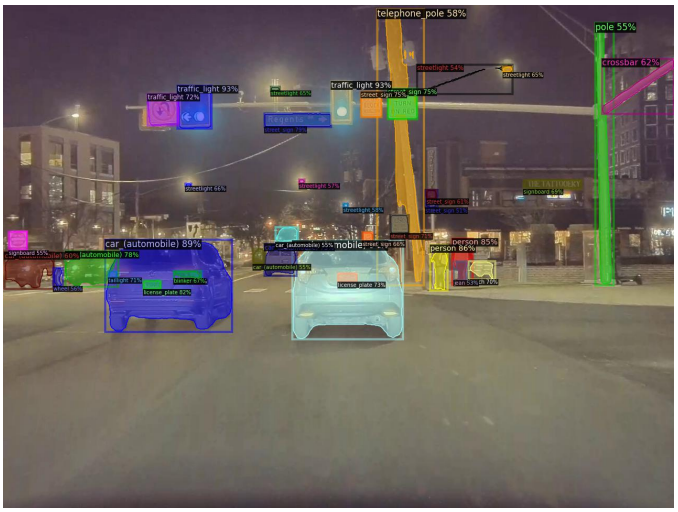

Fig. 2. Detic Output

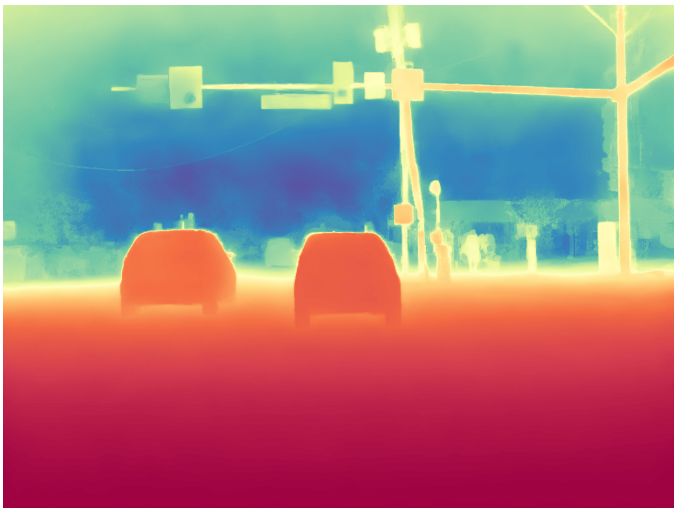

Fig. 3. Depth using Marigold

C. Traffic light detection:

Detic was able to detect the traffic signal but was not able to identify its color. To do so we deployed `Traffic-Light-Detection-Using-YOLOv3` [4] model which had pre-trained weights and built over the YOLOv3 framework. This model was able to classify the green light as "go", the red light as "stop", the yellow light as "warning", the left arrow as "goLeft" and so on. However, the detection isn't very robust as it seems to miss detections from far away.

Fig. 4. Traffic-Light-Detection-Using-YOLOv3

III. Checkpoint 2: Advanced Features

In this phase, we built over the previous work by enhancing and adding more features.

A. Vehicle classification:

In this phase, are still using Facebook's DETIC to do instance segmentation. DETIC gives the existing COCO dataset classes additionally also having the lvis classes. It also works with a custom dictionary which we use to identify the car sub-types such as sedan, SUV, pickup truck, hatchback, and truck using custom vocabulary. We do essentially a non-max suppression to get rid of multiple masks on the vehicles. The other classes were identified using the standard predictor including motorcycles and bicycles. As we are using a custom vocabulary, the confidence of the predictions is on the lower side as it uses CLIP embeddings as input. Hence we only opt to identify car sub-types using the custom vocabulary while the other objects like stop signs, cones, barrels, etc from the standard predictor. Using custom vocabulary also brings the confidence in predictions down.

B. Vehicle pose estimation:

We also had to estimate the orientation of the vehicles for this phase. We use the implementation of YOLO3D [5] trained on the KITTI dataset. We match the bounding boxes from DETIC to the regressor which predicts the bounding box

Fig. 5. Detic's out for car subclasses

form YOLO3D. We use the yaw from this to spawn the cars with the corresponding orientations. The model used the image corp from DETIC i.e. the 2D bounding box from DETIC predictions, to estimate the 3D location and back project onto the image. The results were pretty accurate for all vehicle subtypes except the trucks in the scene which had an offset in the orientation. The possible reason could be the smaller number of truck images in the dataset compared to other vehicles. However, it should be noted that the yaw from this is not very accurate when the car is very close or is occluded. A more advanced network might be utilized for better results.

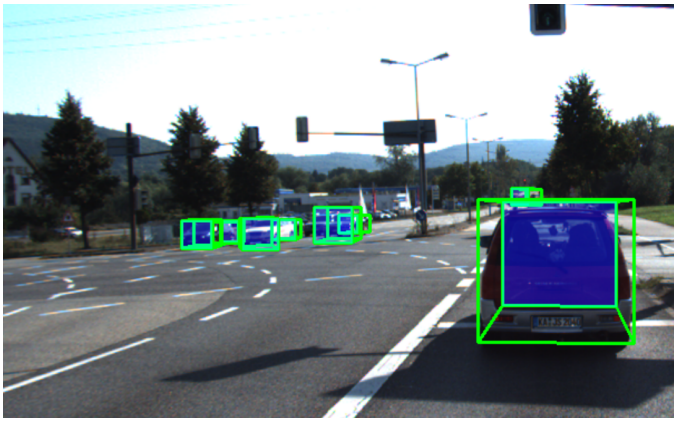

Fig. 6. 3D bounding box projected in 2D from YOLO3D

C. Road Signs:

The same Mask R-CNN backbone is reused to segment on–pavement road-sign symbols (e.g. arrows, turn indicators). For every mask that corresponds to a road sign we crop the RGB image with the predicted bounding box, convert the crop to grayscale, and apply a fixed threshold that preserves the high-contrast sign while replacing asphalt pixels with a uniform background colour. The resulting binary crop is flipped horizontally (so it faces the driver) and mapped onto a textured plane that is placed in Blender at the depth returned by the monocular map, reproducing the sign's true position on the road surface.

To recognise vertical speed-limit signs we combine Detic's object detections with text extracted by the EasyOCR engine [6]. Whenever Detic labels a bounding box as `road_sign`, we used output from OCR for that image and search the transcribed string for the keywords `speed`, `limit`, or the phrase `speed limit`. If a match is found, the first numerical token is parsed as the posted speed value, which is then stored with the sign's pose for downstream planning modules. In our evaluation this joint Detic–EasyOCR pipeline achieved high recall and near-perfect precision.

Fig. 7. Speed Limit Text Detection using OCR

D. Objects:

Apart from the trivial objects, there are plenty of objects such as dustbins, traffic cones, and cylinders present in the surroundings which we aim to detect. DETIC was able to handle the detection of these objects and was found to be pretty robust. The blender models for all such objects were already provided. Using the depth information, each object was seamlessly spawned in the environment.

E. Pedestrian pose:

We have utilised the OSX [7] framework. By inputting monocular images of human figures, OSX generates 3D mesh representations encapsulated in .obj files. This streamlined our workflow by providing readily usable mesh data for integration into Blender. To ensure precise positioning within our Blender environment, we integrated depth and pixel information from the input images to estimate the 3D world coordinates of the mesh vertices. This approach facilitated the seamless integration of human poses into our Blender scenes, enhancing the overall efficiency and effectiveness of our project workflow.

Fig. 8. Miscellaneous Objects (Dustbins) using Detic

Fig. 9. Miscellaneous Objects (traffic cones and barrels) using Detic

IV. Checkpoint 3: Bells and Whistles

A. Brake light and Indicators

Detic provides class-agnostic bounding boxes for vehicle tail-lamp clusters. For each ROI we run the classical routine to decide whether the lamp is active and to infer its colour (red / yellow). The crop is first heavily blurred, converted to HSV, and analysed on the *value* channel V. A dynamic threshold $T = \max\{m_V, \bar{V} + k\sigma_V\}$ is formed from the mean $\bar{V}$ and standard deviation σ_V (k=1.0, m_V=130). Pixels with $V > T$ are deemed "bright"; if their ratio exceeds the decision margin (≥ 0.10) and $\bar{V} \geq m_V$ the lamp is flagged `ON`, otherwise `OFF`. The hue of these bright pixels determines whether the emission is predominantly red (brake) or yellow (turn indicator). While computationally cheap, this heuristic suffers when ambient reflections raise the background brightness or when LED patterns are partially occluded, and therefore attains only modest accuracy compared with the learning-based pipelines described earlier.

Fig. 10. Breaklight detection

Fig. 11. OSX keypoints for detected person

B. Parked and moving vehicles

To distinguish between parked and moving vehicles, we utilized optical flow analysis with the RAFT [8] algorithm, which provides monocular optical flow images indicating relative flow between pixels. Higher flow rates are represented by intensified colors in the flow images. Initially, we attempted to use the Sampson distance to mask moving vehicles based on their higher flow rates. To classify as moving or parked, first we evaluated the net flow within each bounding box provided by DETIC and compared it with the flow from neighboring regions. If the difference in flow was relatively low, indicating minimal movement of the object relative to the scene, and if the net flow of the mask itself fell within a specified range (indicating that cars moving in the same direction were below a minimum threshold), we categorized the vehicle as parked.

This approach is not 100% accurate as if an object is right in front of camera it fails but works in most of the cases.

Fig. 12. RAFT's Output

Fig. 13. Moving or Parked Classification - 1

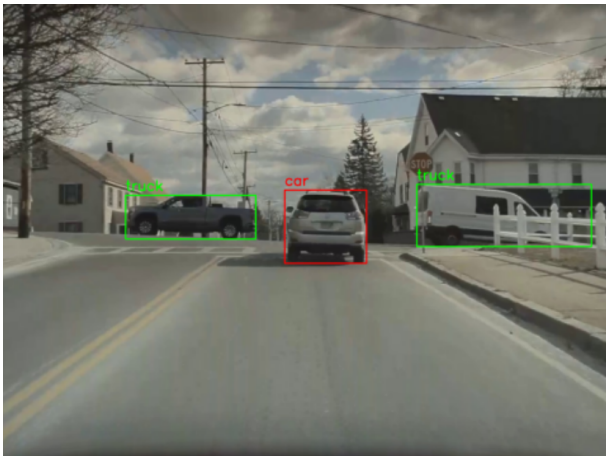

Fig. 14. Moving or Parked Classification - 2

V. Extra Credit

A. *Speed Humps:*

To recognise vertical speed-limit signs we combine Detic's object detections with text extracted by the EasyOCR engine. Whenever Detic labels a bounding box as road sign, we used output from OCR for that image and search the transcribed string for the keywords speed, hump, or the phrase speed hump. If a match is found, a speed hump is spawned near the sign on the road.

B. *Collision Prediction:*

We just did a simple logic here that we any car or pedestrian is very close to the car i.e. within specified bounds and is also moving towards the car then its considered as an potential threat and marked red as the collision probability with that object is high.

VI. Rendering Samples

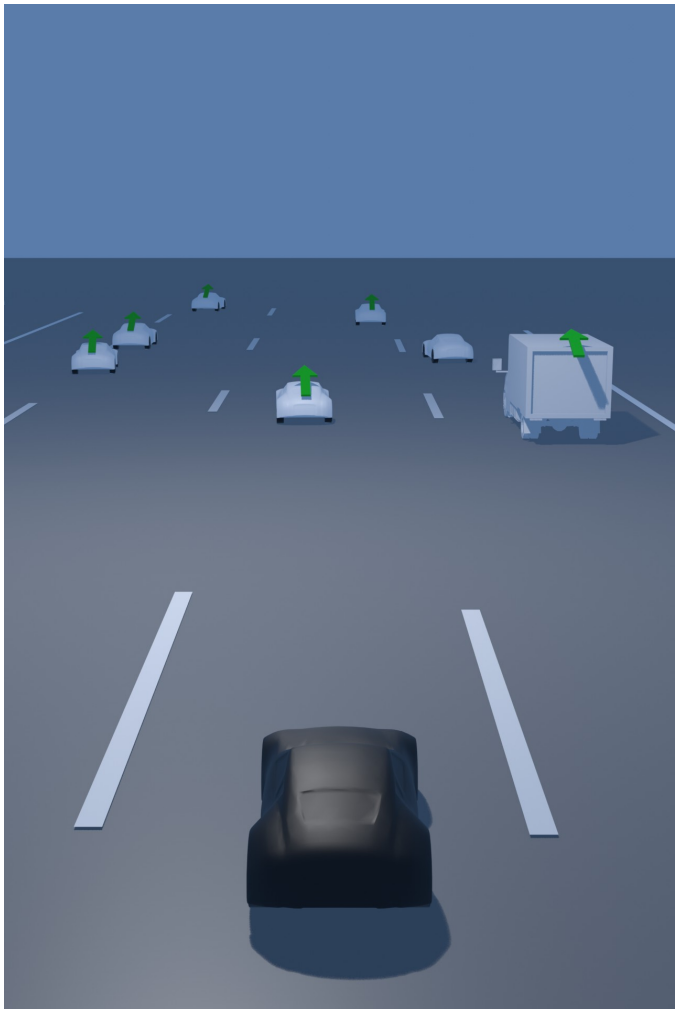

Fig. 15. Render with Lanes and Vehicles

Fig. 16. Render with Green Traffic Light with a direction arrow

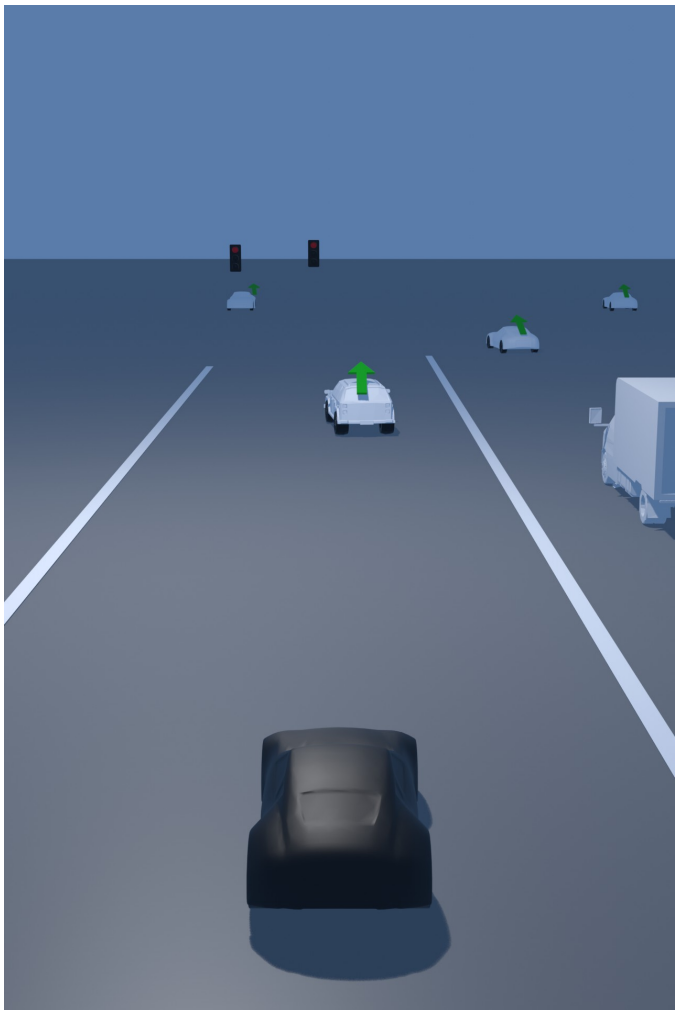

Fig. 17. Render with Red Traffic Light

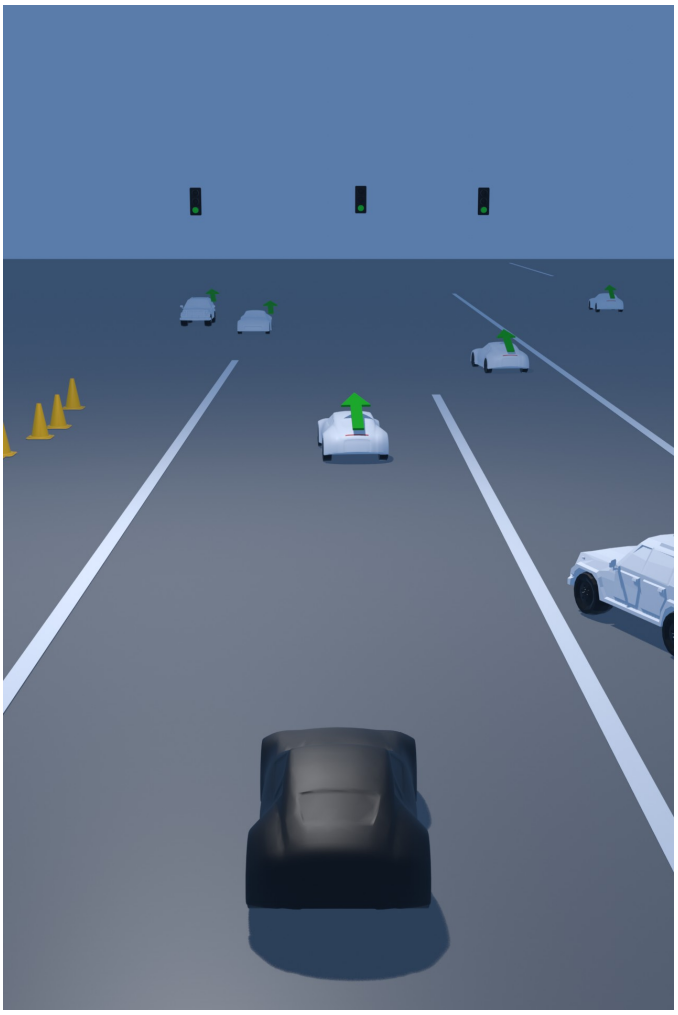

Fig. 18. Render with Traffic Cones

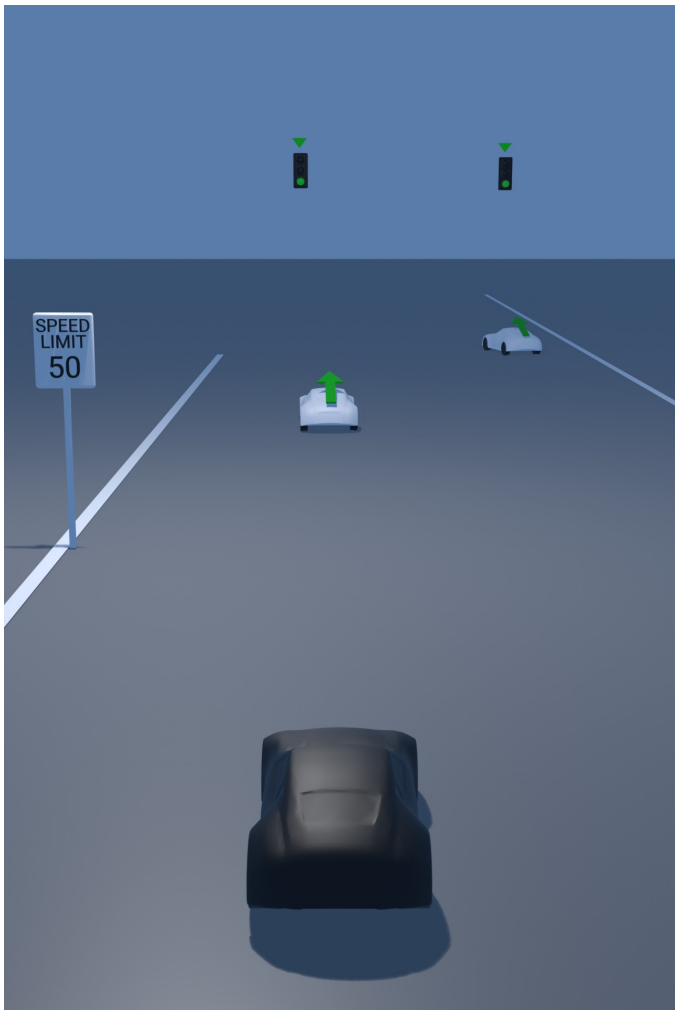

Fig. 19. Render with Speed Limit

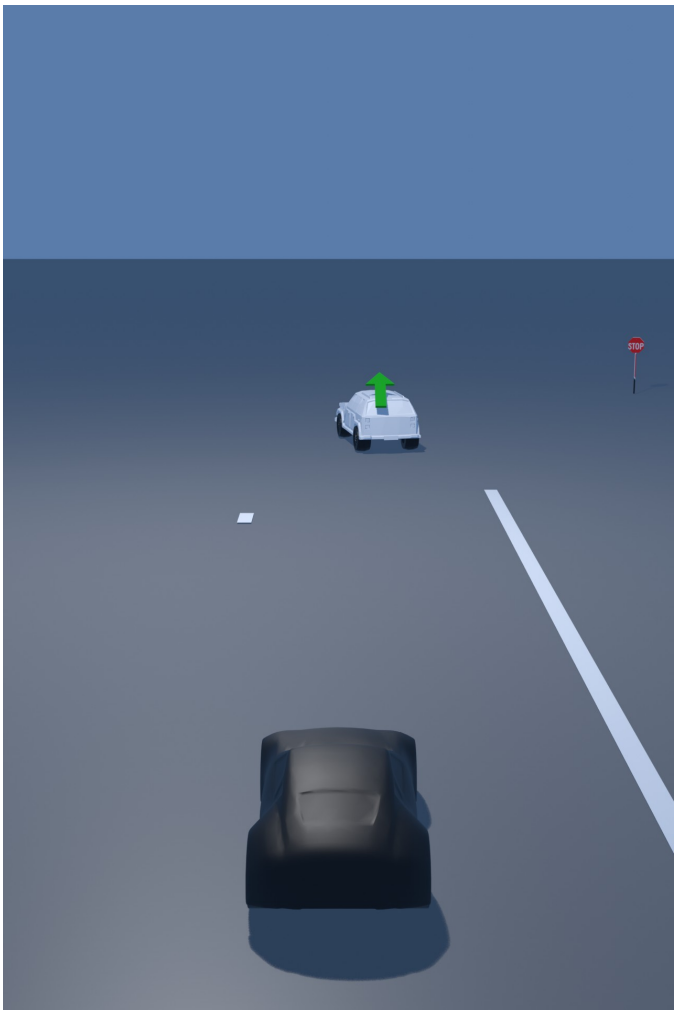

Fig. 20. Render with Stop Sign

Fig. 21. Render with Dustbins

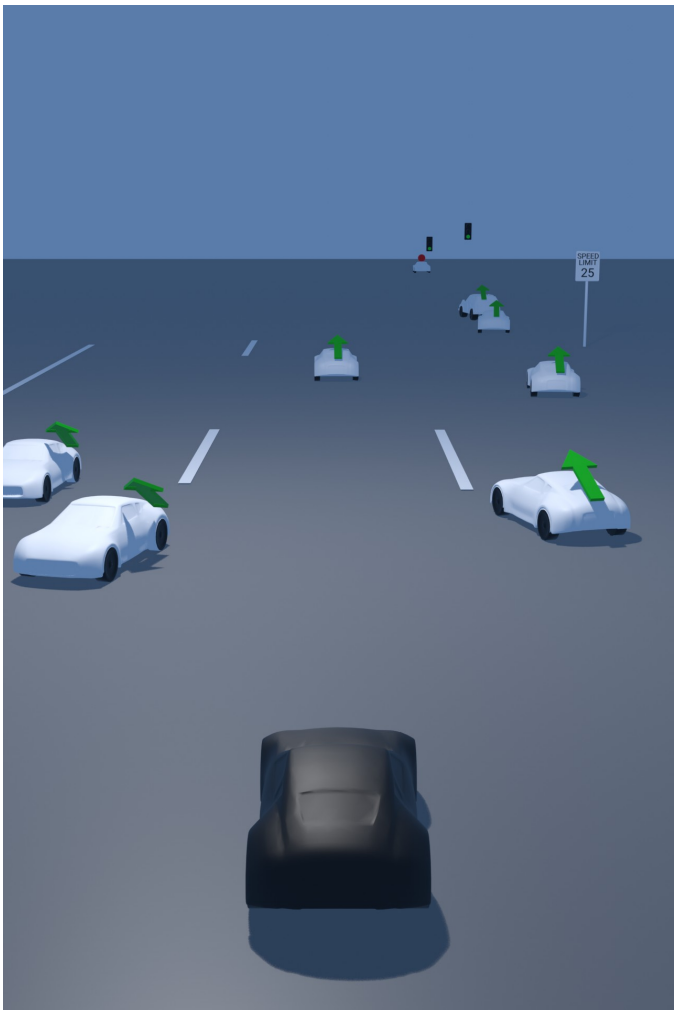

Fig. 22. Render with Moving or Parked vehicle marked

Fig. 23. Render with correct Human joint pose

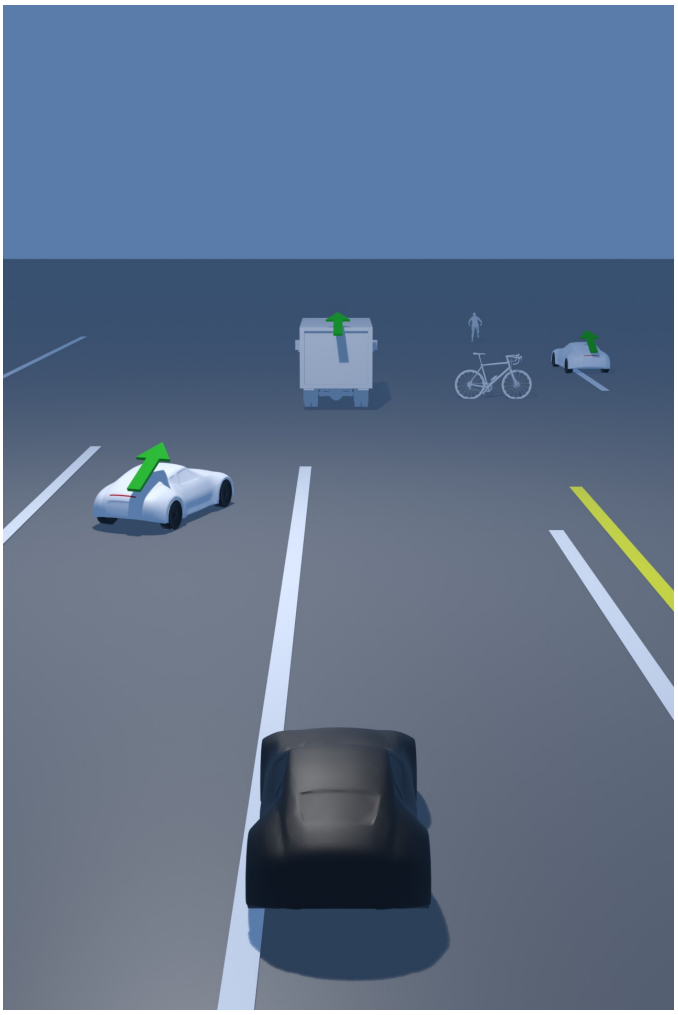

Fig. 24. Render with a Bicycle

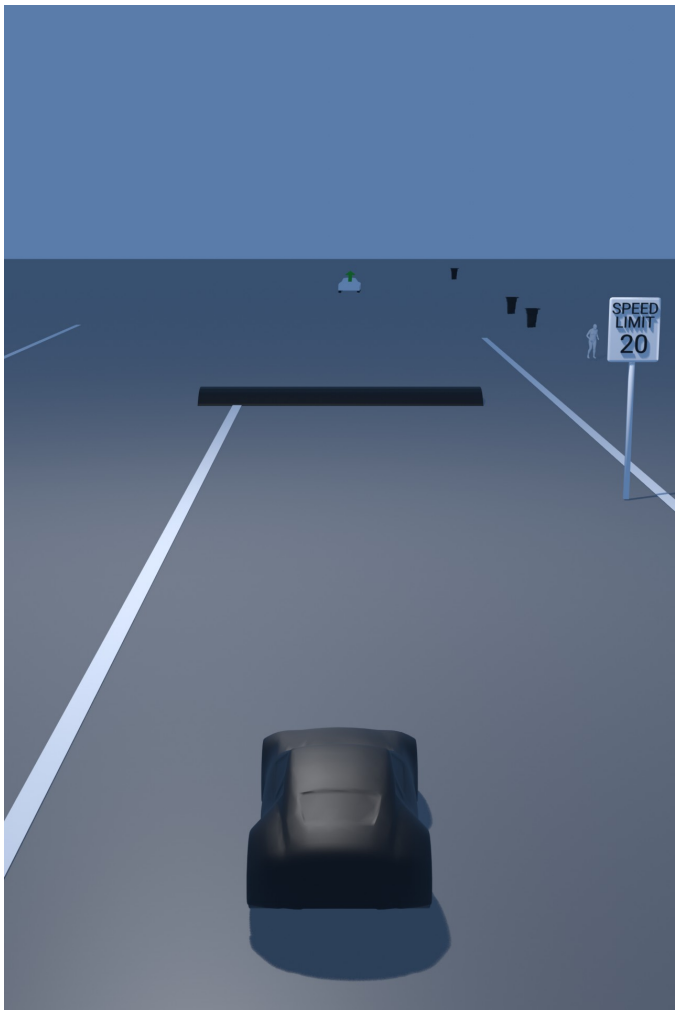

Fig. 25. Render with Speed Limit and Speed Hump

Fig. 26. Render with Motorcycle

Fig. 27. Render with Collision Detection

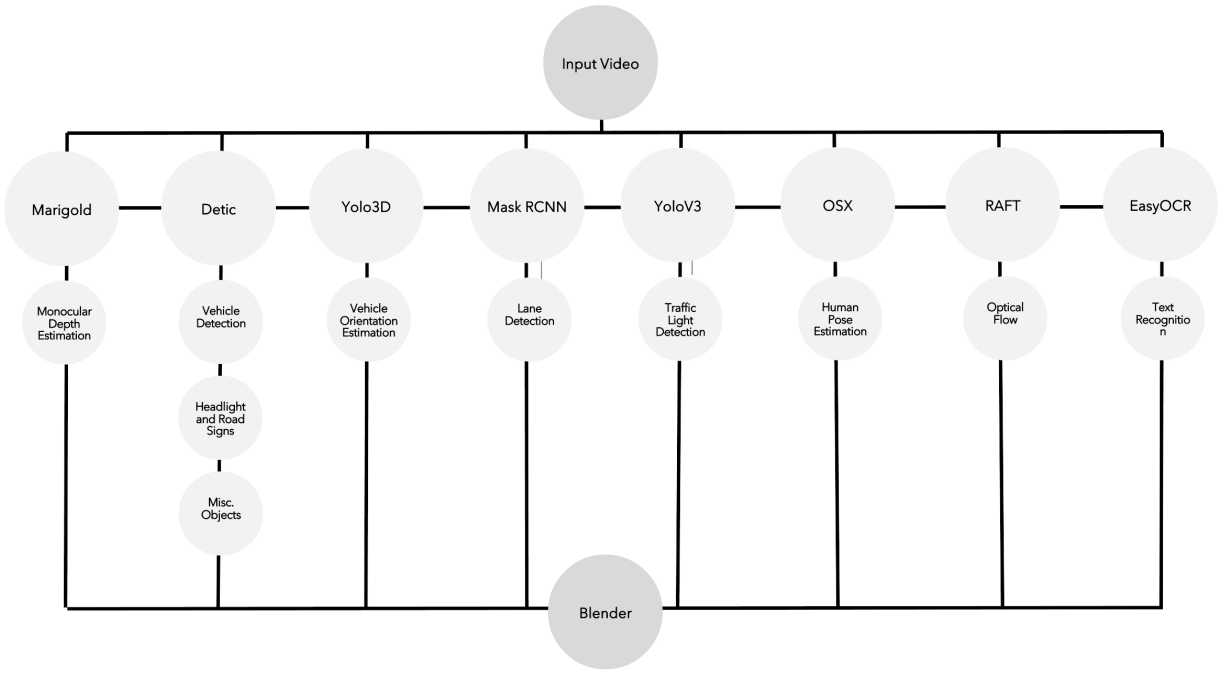

Fig. 28. Einstein Vision Pipeline

REFERENCES

[1] Sovit Ranjan Rath. Lane detection using mask r-cnn, 2020.

[2] Xingyi Zhou, Rohit Girdhar, Armand Joulin, Philipp Krähenbühl, and Ishan Misra. Detecting twenty-thousand classes using image-level supervision, 2022.

[3] Bingxin Ke, Anton Obukhov, Shengyu Huang, Nando Metzger, Rodrigo Caye Daudt, and Konrad Schindler. Repurposing diffusion-based image generators for monocular depth estimation, 2024.

[4] Sovit Ranjan Rath. Traffic light detection using yolov3. https://github.com/sovit-123/traffic-light-detection-using-yolov3, 2020.

[5] Arsalan Mousavian, Dragomir Anguelov, John Flynn, and Jana Kosecka. 3d bounding box estimation using deep learning and geometry, 2017.

[6] Jaided AI. Easyocr: Ready-to-use ocr with 80+ supported languages. https://github.com/JaidedAI/EasyOCR, 2020.

[7] Jing Lin, Ailing Zeng, Haoqian Wang, Lei Zhang, and Yu Li. One-stage 3d whole-body mesh recovery with component aware transformer, 2023.

[8] Zachary Teed and Jia Deng. Raft: Recurrent all-pairs field transforms for optical flow, 2020.