RBE 549 Project 3: Einstein Vision

UdayGirish Maradana
Robotics Engineering (MS)
Worcester Polytechnic Institute
Worcester, MA 01609
Email: umaradana@wpi.edu

Pradnya Sushil Shinde
Robotics Engineering (MS)
Worcester Polytechnic Institute
Worcester, MA 01609
Email: pshinde1@wpi.edu

***Abstract*—The following report consists of a detailed analysis of engineering a perception stack for autonomous vehicles that provides visualizations. A good visualization system takes in sensory information and provides intuitive insights into how the vehicle perceives the environment around it. Visualizations are important for debugging the issues in the software and retrieving feedback on the surroundings.**
Keywords: Object Detection, Lane Detection, Pedestrian Pose Estimation, Traffic Light Detection, Road Signs Detection,

I. Pipeline Overview

As this project involved, multiple concepts and the integration of them , initially we have tried to understand the overall pipeline and research around each of the individual concept separately. The initial plan is to build a system which is shown in Fig.1. So we started first with finding the calibration matrices of the front camera as that was the most important one. From the knowledge of previous projects, we were able to estimate the camera calibration matrix(K) and then Rotation and Translation and fixed according to the pose of camera in blender.

II. Methodology

In each of the following phases, we will deploy Deep Learning models to detect assets such as cars, traffic signals, traffic cones, road signs, and pedestrians. We are given the Blender models of each of the assets. We will render a scene through each phase where these assets will be detected by the autonomous vehicle.

A. Phase I: Basic Features

We will implement the following basic features:

- **Lanes**: Detect feature-specific lanes such as dashed, colored, and/or solid.
- **Vehicles**: Identify cars and represent each one as a car shape.
- **Pedestrians**: Identify and locate pedestrians where each pedestrian will look the same.
- **Traffic Lights**: Identify traffic lights and their colors.
- **Road Signs**: Identify Stop Signs and apply the textures.

B. Phase II: Advanced Features

We will implement this phase by building upon the previous one. We need to implement the following features to add more granularity to the vision system which can aid in algorithmic decisions in navigation modules.

- **Vehicles**: Classify (identify different vehicles) and sub-classify them (identify different kinds of a type of vehicle). We also need to identify the orientation of the vehicle and display them as a 3D model.
- **Traffic Lights**: Classify arrows on the traffic lights.
- **Road Signs**: Identify arrow and speed limit signs.
- **Objects**: Identify objects like dustbins, traffic poles, traffic cones, and traffic cylinders.
- **Pedstrain Pose**: Identify pedestrian pose in each frame.

C. Phase III: Bells and Whistles

- **Break Lights and Indicators of Vehicles**: Identify and display the vehicle brake lights and indicator signals
- **Parked and Moving Vehicles**: Distinguish between parked and moving vehicles and display. For the moving cars, we also need to identify their moving direction and display them.

D. Extra Credit: Cherry on Top

- SpeedBumps: We somehow got an approximate speed bump position through the scene and road sign-based information.
- Collison Prediction of Pedestrians or other vehicles as a red highlight: We have explored both averaging of optical flow segmented mask approach and using Multi object tracker out of which Multi Object tracker felt like a better approach as it does not need any threshold based approaches for understanding which part of the scene is moving. We have used Yolo Multi tracking approach and got some track ids but were not able to completely integrate this into the current Rendering pipeline.

III. Implementation

As mentioned in the Methodology, the primary aim is to extract info regarding several concepts of the scene. As it is hard to explain the different phases of each of the component separately we combined the information here regarding each exploration we did and the final model we have chosen with the result.

A. Lane Detection & Classification

Lane Detection is one of the important steps for a self-driving car pipeline. We initially started exploring Traditional methods such as color thresholding and then

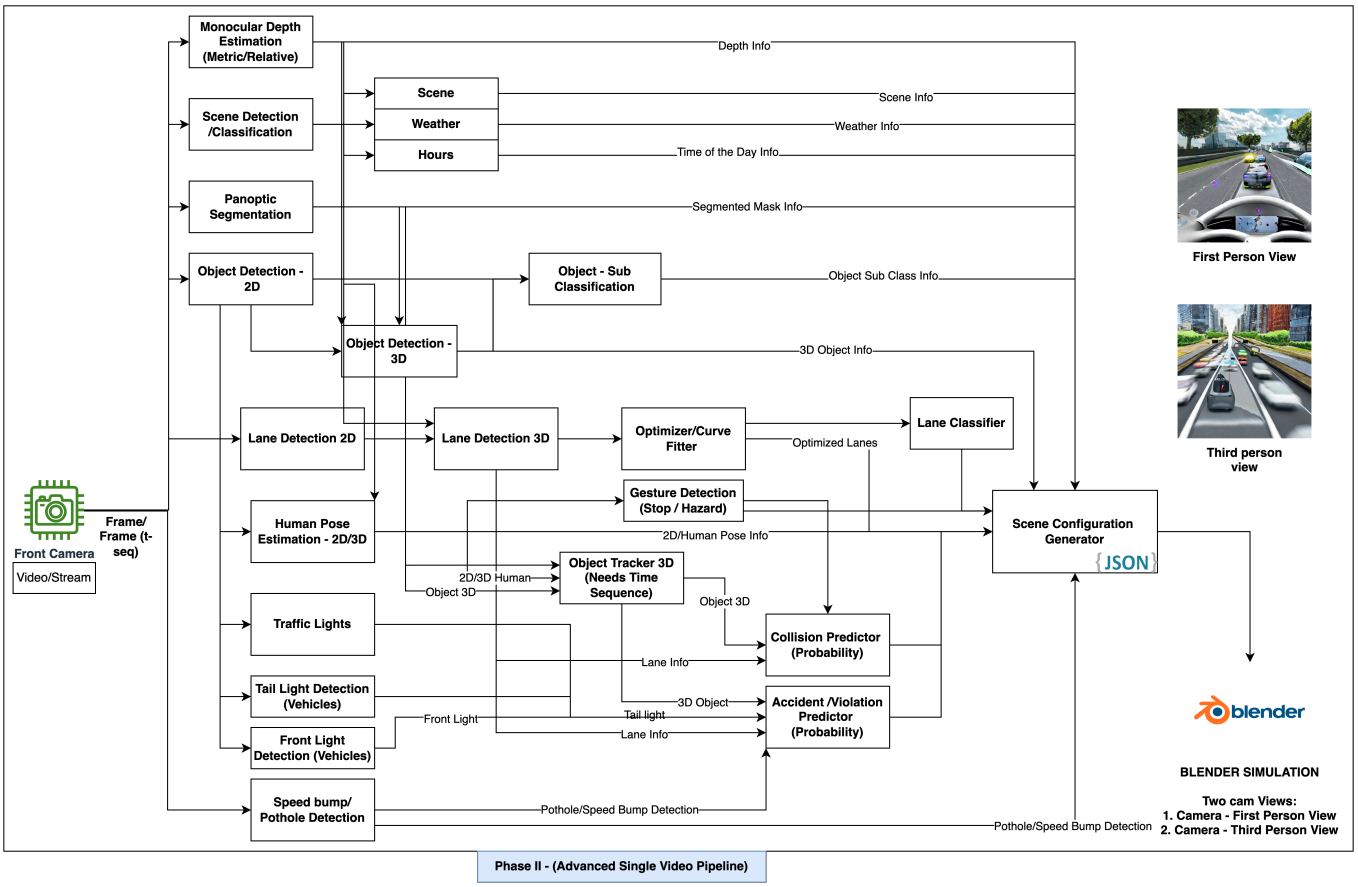

Fig. 1: Initial Planned Architecture

further understanding through Window search similar to the implementation here. Udacity Implementation. The traditional method comes with the downside of not being able to generalize much. Then we decided to try out Deep learning methods especially 3D Lane detection methods such as PerFormer but after trying out to write custom inference code for the same, we were not able to run the inference as the weights provided by the author are old and the architecture is modified in the recent branch. As all these are time-consuming we shifted back to 2D Detection methods. The primary reason for going towards models like Performer is that they are trained on Self Driving car datasets such as NuScenes which can give both the 3D lane information and also the type or class of the lane belongs to such as whether it is a white lane or dotted lane.

To achieve the same using 2D we went with a sequential approach involving finding the 2D lanes and corresponding the 3D points using the camera's intrinsic and extrinsic information. Later we are able to find a Repository that implemented a Mask RCNN-based segmentation and classifier of the lanes and the Free road. Even though Segmentation-based models work fine for lane detection and free lane detection, they are not as accurate as special lane detectors. So we used the Bounding box information from the Segmentation mask boundaries and ran a point inside rectangle check on all the 2d points we got before from the DL network (CLRer Net). Thus we were able to get the points and also get the labels through this matching process. This has a caveat that there are times when the matching fails, even though this can be solved by approximation methods, for now we went with the simple conditional where if the matching fails we resort to simply making it as default solid line as that is having the highest importance for safety. A simple explanation regarding CLRer net is written below.

CLRer Net [4]: CLRer Net builds upon **Cross Layer Refinement Network (CLRNet)** which aims at fully utilizing both high-level and low-level features in lane detection, by improving the confidence of Lane Detection with LaneIoU. In particular, it first detects lanes with high-level semantic features and then performs refinement based on low-level features. In this way, we can exploit more contextual information to detect lanes while leveraging locally detailed lane features to improve localization accuracy.

The results of CLRer Net + Mask RCNN-based classifier on 2D image and the 3D blender representation are shown in Fig. 5 and Fig. 6

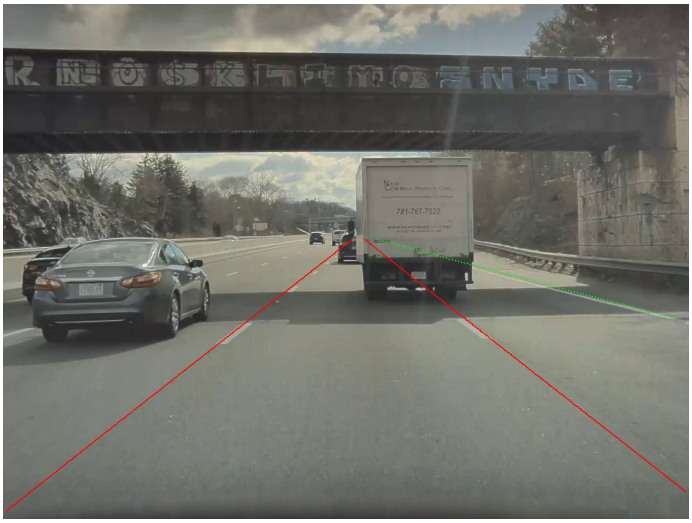

Fig. 2: Lane Points CLRerNet (Red - - Dotted Lane, Green - Solid White Lane

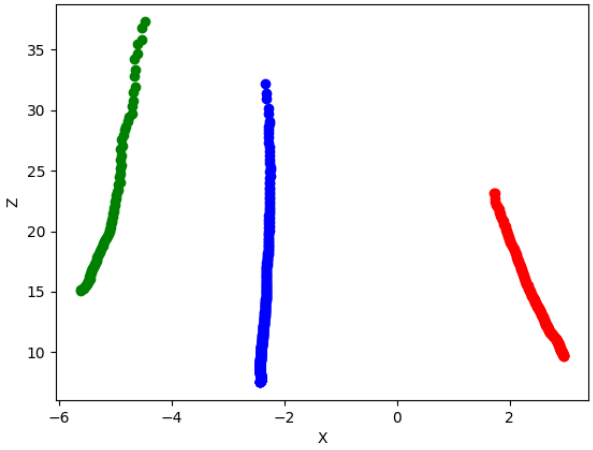

Fig. 4: Lane 3d points - Corrected and Noise Removed

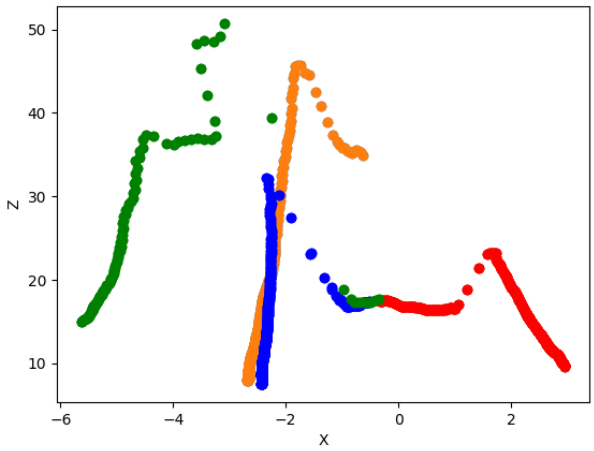

Fig. 3: Lane 3d points - RAW

Fig. 5: Original Image of Lanes

B. Scene Object Level Information - Vehicles, Pedestrians, Traffic Assets

Object detection is the core task of recognition, detection, and classification of objects in an image. This can achieved by many classifers, one such famous one who works out of the box is Yolo from Ultralytics. Even though the latest stable version from them (Yolov8) works out of the box and almost detected people, cars, trucks, traffic lights, and stop signs from the scene it failed at most of the Road sign detection and traffic asset detection which are two important things which we need for the project.

One more issue is the traffic light classification, we tried doing the traditional way where we get the traffic light bbox and extract the crop which gets passed to an HSV space thresholding algorithm to get the information regarding the dominating color in the crop. Even though this method was

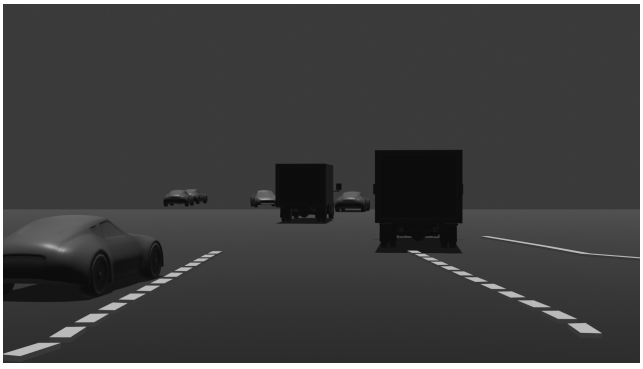

Fig. 6: Blender Output

Fig. 7: Original Image for Object Detection

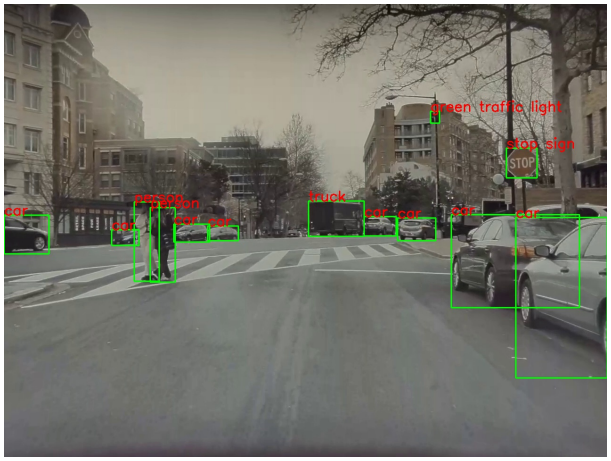

Fig. 8: 2D Object Detection - Yolo World

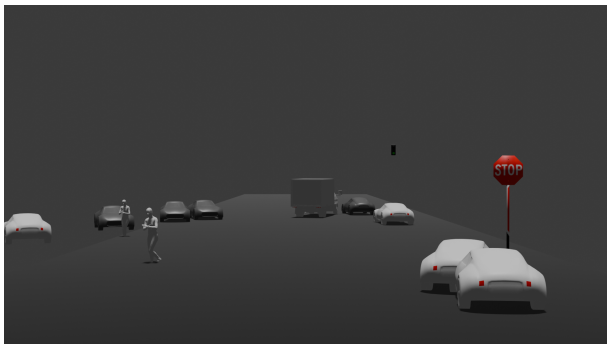

Fig. 9: Blender Ouptut with Brake lights implemented

able to perform to some extent, it was very hard to make it generalizable (Common case of Failure to generalization in 2D Computer vision). So, we shifted towards thinking deep learning-based methods, and we kind of trained a ResNet-based Traffic Sign Classification on Bosch Dataset, and the results were good.

But in the end, we kind of searched for a simple method as it again adds to the entire computation, even though this project does not have constraints on computation, we took it as a challenge to optimize as much as possible. So, we were searching for more methods, and then we kind of got the idea of why not to use Language-based modeling to understand scene-level information to do this.

We found about Yolo V2 which kind of trained with open vocabulary detection capabilities. With proper vocabulary prompting, we were surprised to see that it almost gave all the information we required and even performed decently to subclassify and detect traffic signals based on the addition of color to the text prompt. This resulted in us getting all the information we required for the Objects. Some of the detections and the information are shown in Fig.7 - 9.

Once we get the outputs, we need to process, the information regarding each specific object separately which is explained in the below sections.

C. Pedstrain Pose Estimation

After extracting the Object-level information from the scene, the next thing to do is to extract pose-level information to model the 3D human in the Blender. We tried to get the 3D pose estimation method of humans and also tried to find 2D pose estimation but both were not able to directly translate to the modification of the 3D human pose in Blender. We initially took the approach of getting the COCO 17 Keypoint model (Mostly from Yolo Pose) (2D/3D) to model the mesh in Blender. We tried the armature-based mesh modification approach, and even though we were able to move the armature and mesh it was pretty hard to generate good 3D configurations. As we found some singularities in the position even after going with the Quaternion-based angle modeling. This involves solving for a perfect Inverse Kinematics solution to represent a human pose back to 3D as it is not a matching problem. We found one of those methods which is the HybrIK but this also took more time to implement.

Finally, we resorted to 3D Mesh estimation models such as AlphaPose or PyMAF and found PyMAF was too easy to implement and easy to import into Blender as it directly gives a 3D mesh obj model which can be imported to Blender. PyMAF also used a human model from SMPL which is the most common format for human modeling-based approaches in Deep learning. We Kind of used this to implement the overall pipeline, these results are extracted and saved to a folder. A brief explanation of both YoloPose and PyMAF is here.

1) **YOLO POSE** [3]: It uses a novel heatmap-free approach for joint detection and 2D multi-person pose estimation in an image based on the popular YOLO object detection framework
2) **PyMAF** [8]: To overcome misalignments between estimated meshes and image evidence, **Pyramidal Mesh Alignment Feedback (PyMAF)** loop leverages a feature pyramid and rectifies the predicted parameters explicitly based on the mesh-image alignment status.

Even though PyMAF works open out of the box, there were

Fig. 10: PyMAF3D

times it was wrong and mostly tried to fit the entire human mesh even in the crop of humans. We read that this was solved to some extent in PyMAF SMPLX but we haven't explored that completely. Some of the results of Yolo Pose and PyMAF are shown below in Fig. 10.

D. Tail Light /Brake Detection

Detection of Tail lights or indicator lights is crucial as that helps the car behind to make better decisions as to whether to stop or not. We have again tried to approach with the help of Traditional CV algorithms, but the output of the method is sometimes random and not generalizable, and took time to tune the values. Further, we tried approaching the problem with DL, and when we searched for it, we found a repo that does Auto brake classification using Yolo v5. We adopted that here and were able to understand whether the brake lights were on or not.

E. Traffic Assets Detection

1) Road Signs & Stop Signs: We were able to classify most of the Traffic Assets from the Yolo World but still, we were left with classifying two important road signs. One is a Stop Signal and the other one is a Parking meter or speed.

- From YOLO world we were getting Stop Sign detection and further we also got generic road sign detection.
- For Generic road sign detection, we take the crop of the Road sign detection image and then pass the image to Optical Character Recognition (OCR), here we use EasyOCR.
- The output from Easy OCR we use a matching logic to see if numbers are there in OCR output if numbers are there we extract them and get the speed on the road sign.
- If there is text we kind of extract the text and put it on the Road sign.
- Overall this is quite an experimental approach and might need more tuning. But it works!

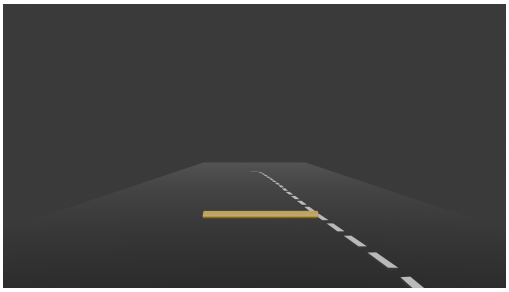

Fig. 11: Speed Hump

*2) Road Signs & Speed Humps - **Extra**:* From the Yolov2 model, we were also able to identify the speed bumps, this was done with two types, one by prompting the Yolo World and the other by detecting of road signs and further checking for "road hump" and "hump" text matching. Assuming in North American datasets, we have a road sign near speed humps, we could either do a detection from the scene or from the text on the road sign and this was the intention behind approaching in this way. Some of the outputs can be seen in Fig. 11.

3) Road Markings: One more information, we need to extract is the Road sign information painted on the road. This is very important for Lane changing and taking turns on a highway or parking. To do this, what we did is to use Yolo P to get the drivable segmentation area and further get a mask of this segmented area. Once we got this masked area we did Edge and Contour Operations on it to understand the contours. From the contours, we were able to perfectly get a Polygonal bounding box around the Road signs, especially arrows. Further, we used a Maximum directional change estimate to understand the arrow direction. From this method, we were perfectly able to know what the sign especially especially whether the arrows are showing towards left/right/forward. This we haven't got time to render in a blender with some additional flags. Some of the 2D outputs from this can be seen in Fig. 13.

F. 3D Object Detection

Even though we have 2D object detection, we need the orientation and scale to place the 3D assets in the scene. To do this we need some sort of 3D estimation and Yaw orientation of the object. This can be achieved with the help of 3D regressors. One such method is Yolo 3D estimation which is open source implementation of a regressor combined with Yolov5 2D model outputs. We used this model to get the Orientation and scale of the 3D object. Some of the Yolo 3D outputs can be seen here in Fig. 14 .

G. Depth Estimation

As most of the information we get from the above methods, is 2-dimensional. We need a way to convert them to 3D information. One way to do this is to use the stereo method or use Monocular depth estimation. Here as we concentrate

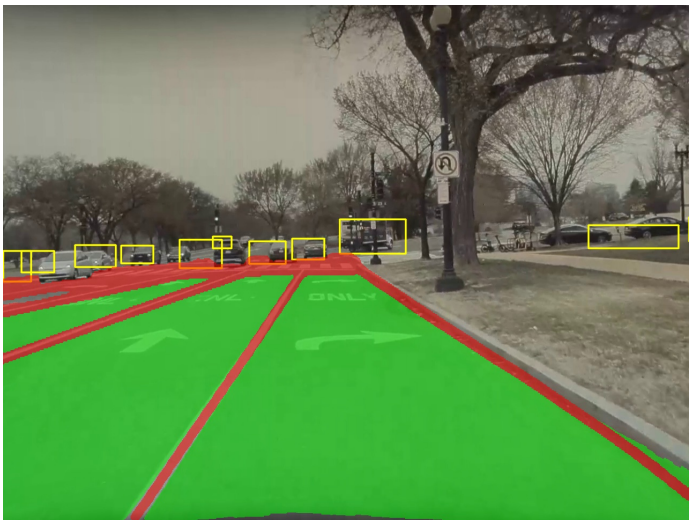

Fig. 12: Frame Output from YoloPV2

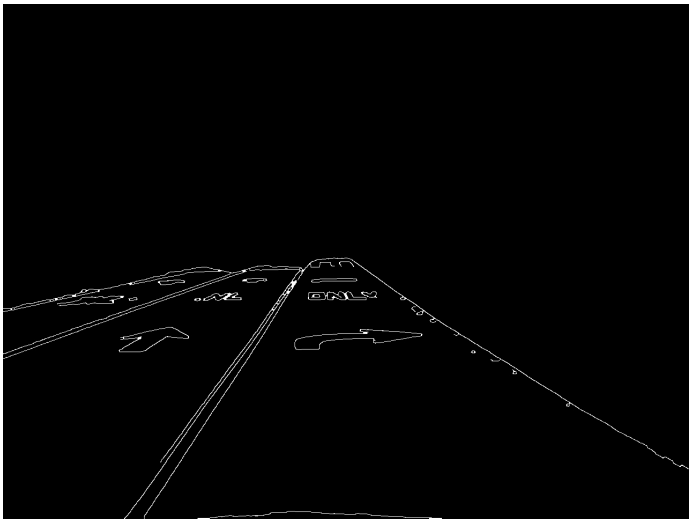

Fig. 13: Frame Output from YoloPV2

Fig. 14: Frame Output from YoloPV2

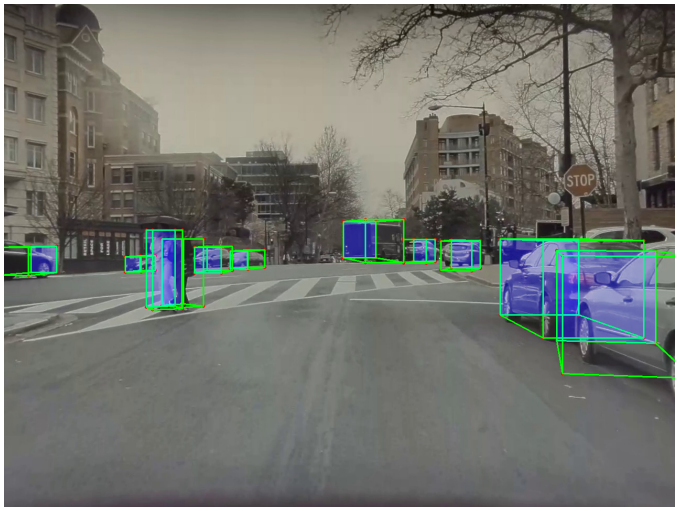

Fig. 15: Yolo 3D Ouptut

on the front camera view, we use Monocular Depth from the Front camera image. So, for every 2D estimation, we take the center of the bounding box get the respective depth, and transform the coordinates to 3D world representation using camera intrinsic and extrinsic information. Two such popular implementations are ZoeDepth and Marigold. We tried both, implementations and found both good and work out of the box. Marigold is along the concepts of Diffusion models therefore takes heavy memory we even tried the faster version but still it was consuming almost 8GB VRAM, which might not fit in the current pipeline we designed as we are running locally on 16 GB VRAM. So, we stick with ZoeDepth, even though sometimes it gives a bit bad result, because of that we might see some random flickering in the scenes, these can be further reduced by using good Depth maps adding more constraints, and better fine-tuning the params.

1) **ZoeDepth** [5]: This model has been deployed to estimate depth from a single image. It focuses on generalizing performance across multiple datasets while maintaining the metric scale (metric depth estimation).
2) **Marigold** [6]: Another model that focuses on depth estimation from a single RGB-D image. Marigold exhibits excellent zero-shot generalization and without ever having seen real depth maps, it attains state-of-the-art performance on several real datasets.

H. Classification of Static and Moving Objects

Understanding the movement of objects in a frame is highly important as it helps us to avoid potential dynamic objects. So in a Driving scenario understanding moving of people and the vehicles is important. This can achieved by multiple methods such as Multi-Object trackers tracked across t frames, Feature based Point level tracking, Traditional KCF Tracker with Hungarian methods, and one of the other cool concept to understand motion is literally understanding pixels. This understanding of pixels is Optical flow, when optical flow is combined with Sampson Distance or any other geometrical

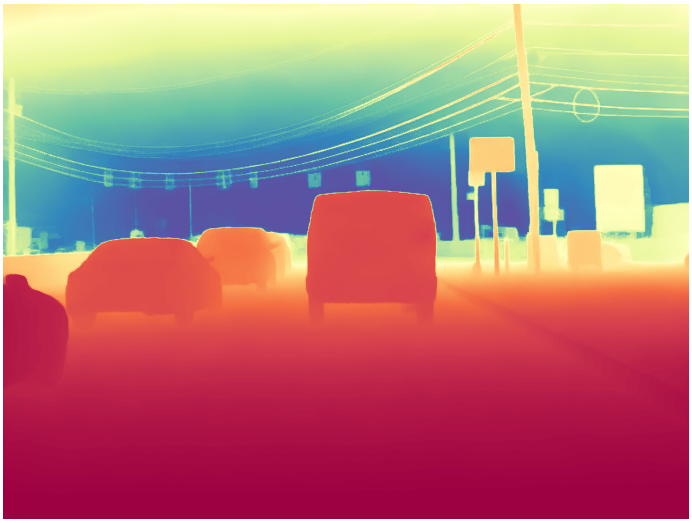

Fig. 16: Depth Output from Marigold - Colored

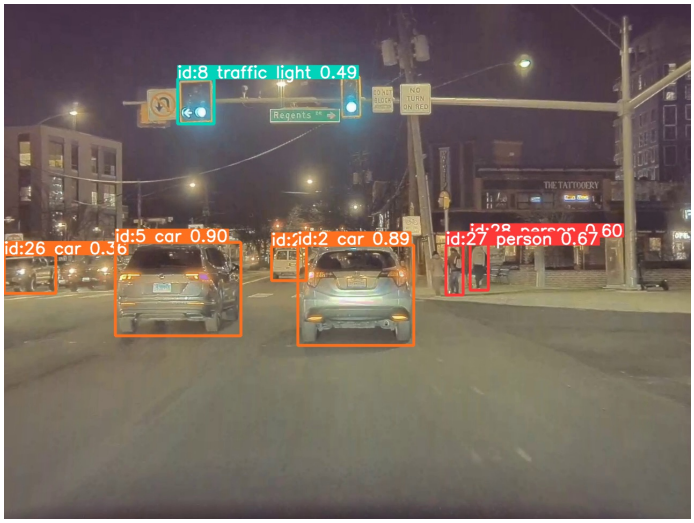

Fig. 17: Multiple Object Tracking

distance calculation and threshold we should be able to relatively understand which objects are dynamic and which are static in the scene assuming we have some prior information about sizes of the objects and perfect threshold.

Optical flow Concept: Let two image frames captured at t and $t+\delta t$ be denoted as $\mathcal{I}_t$ and $\mathcal{I}_{t+\delta t}$ respectively. Further, let the 3D linear and angular velocities of the camera in this time be $V = \begin{bmatrix} V_x & V_y & V_z \end{bmatrix}^T$, $\Omega = \begin{bmatrix} \Omega_x & \Omega_y & \Omega_z \end{bmatrix}^T$. Optical flow at a pixel $\mathbf{x} = \begin{bmatrix} x & y \end{bmatrix}^T$ is defined as the apparent velocity of the corresponding 3D world point and is given by

$$\dot{\mathbf{p}}_{\mathbf{x}} = \frac{1}{Z_{\mathbf{x}}} \begin{bmatrix} xV_z - V_x \\ yV_z - V_y \end{bmatrix} + \begin{bmatrix} xy & -(1+x^2) & y \\ (1+y^2) & -xy & -x \end{bmatrix} \Omega$$

Here, $Z_{\mathbf{x}}$ denotes the depth at a pixel $\mathbf{x}$ which is commonly called the *structure*.

Further we use Sampson distance,

$$d_S(x_i, x_i') = \frac{(x_i' F x_i)^2}{(\text{row}(F_1 x_i)^2 + \text{row}(F_2 x_i)^2 + \text{row}(F_1^T x_i')^2 + \text{row}($$

Here we use the Fundamental matrix which we obtained between the images at t and $t+1$.

Here, we use RAFT as a Deep learning method, and further use the flow from the network and apply sampson distance. Further based on a threshold we classify whether a object is moving or not.

*I. Multiple Object Tracking -**Extra***

We have also explored the Multi Object Tracking to understand the Track id's and our current pipeline is constructed keeping in mind to adapt to track the ID's but because of time constraints we didn't complete the logic to

J. Scene Classification

We further went ahead to understand Scenes, which we felt as one of the good information to give an overall probability on how well models can perform. This is just an ideation step and usually, this is a parameter for some of the DL frameworks where they are trained on different scene types data to ensure predictability and generalizability. We have trained a custom ResNext 34 to get this, for which the results can be seen in Fig. 17 - 18. We did not get time to add this information to the current Blender implementation.

1) ResNeXt 34 on Multiclass Weather Dataset [7]: We build a custom Scene Classification model based on ResNeXt [9].

Fig. 18: Scene Classification

IV. Final Pipeline

The Final pipeline architecture is in Figure . This shows the complete flow of getting the complete flow from it.

Fig. 19: Scene Classification

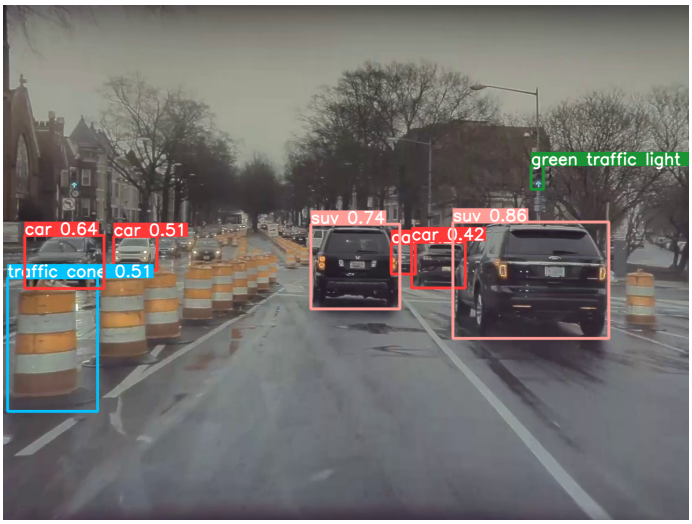

Fig. 20: YOLO World Output

V. Discussion

- We are successfully able to run and integrate most of the models all running on the same system with less than 20 GB RAM and 12 GB VRAM.
- We could make the results much better with integrating more deeper and diffusion based or attention based models but that will maxout the current configuration and it may happen that we get lot of files to play with it.
- Our pipeline generates everything online and saves ¡ 20 MB file for the entire video which contains essential information to render the video from Blender.
- This pipeline can essentially run on an Edge GPU.
- Some of the issues we faced are the timeline is a bit constraint and also may be it would be better to restructure this project into phases of each algorithms rather than upgrading everything one after the other. Means, it is better we fully make one concept working and make the whole pipeline, changing concepts in the middle is a bit challenge and sometimes we are finding some interesection which might need the pipeline structure to change a bit. But this needs proper planning.
- Most of the models which are latest in CVPR and papers with code are heavy so might not be feasible to use them.

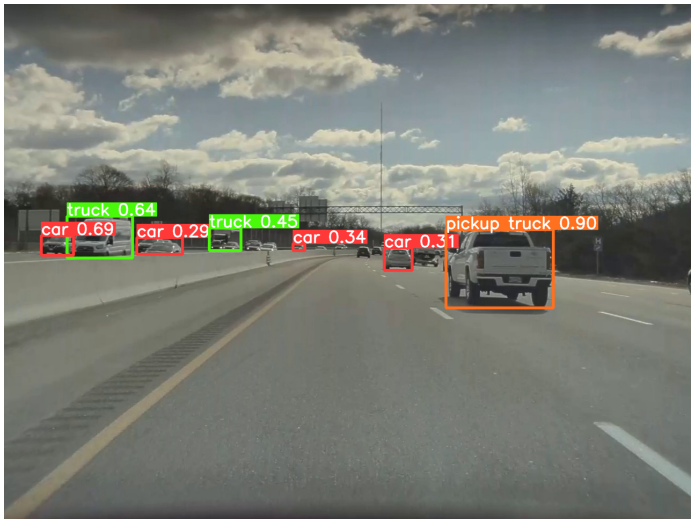

Fig. 21: YOLO World Output

References

[1] T. Cheng, L. Song, Y. Ge, W. Liu, X. Wang, and Y. Shan, 'YOLO-World: Real-Time Open-Vocabulary Object Detection', arXiv [cs.CV]. 2024.
[2] W. Ali, S. Abdelkarim, M. Zahran, M. Zidan, and A. E. Sallab, 'YOLO3D: End-to-end real-time 3D Oriented Object Bounding Box Detection from LiDAR Point Cloud', arXiv [cs.CV]. 2018.
[3] D. Maji, S. Nagori, M. Mathew, and D. Poddar, 'YOLO-Pose: Enhancing YOLO for Multi Person Pose Estimation Using Object Keypoint Similarity Loss', arXiv [cs.CV]. 2022.
[4] H. Honda and Y. Uchida, 'CLRerNet: Improving Confidence of Lane Detection with LaneIoU', arXiv [cs.CV]. 2023.
[5] S. F. Bhat, R. Birkl, D. Wofk, P. Wonka, and M. Müller, 'ZoeDepth: Zero-shot Transfer by Combining Relative and Metric Depth', arXiv [cs.CV]. 2023.
[6] B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt, and K. Schindler, 'Repurposing Diffusion-Based Image Generators for Monocular Depth Estimation', arXiv [cs.CV]. 2024.
[7] Multi Class Weather Dataset
[8] H. Zhang et al., 'PyMAF: 3D Human Pose and Shape Regression with Pyramidal Mesh Alignment Feedback Loop', arXiv [cs.CV]. 2021.
[9] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, 'Aggregated Residual Transformations for Deep Neural Networks', arXiv [cs.CV]. 2017.

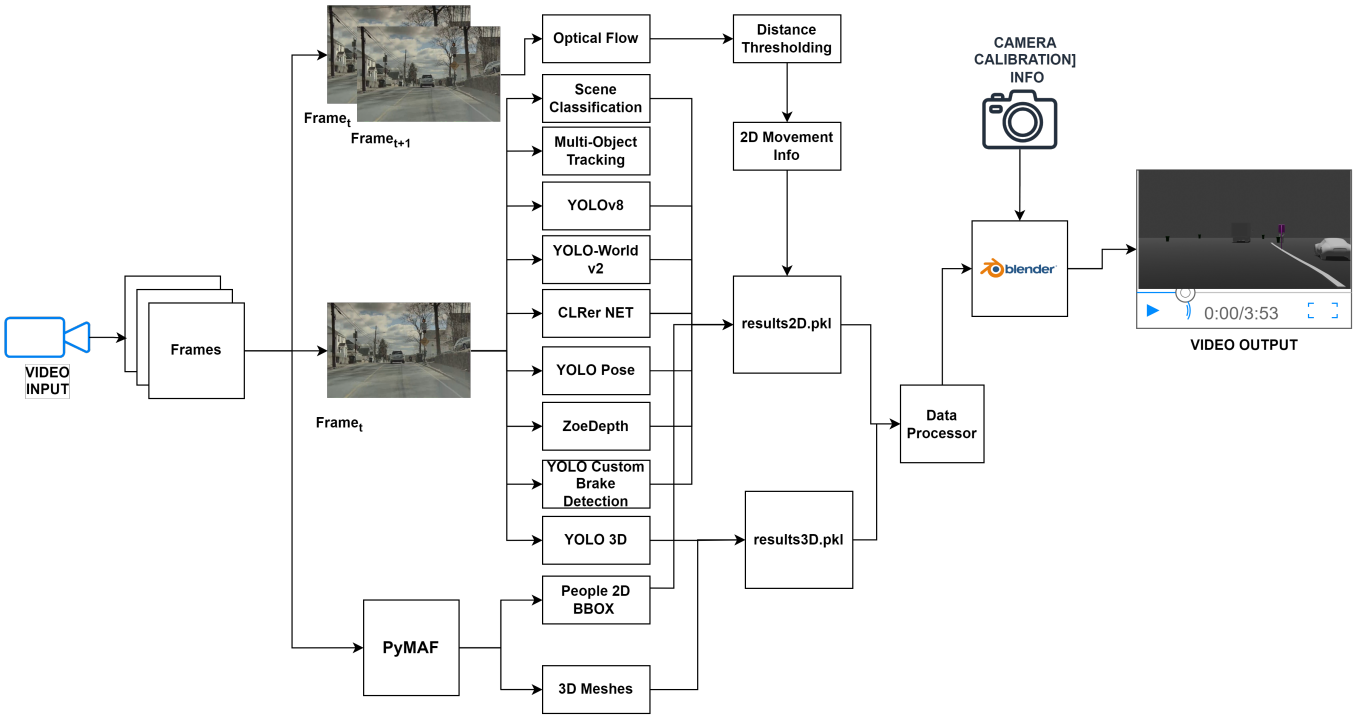

Fig. 22: Final Pipeline Architecture