RBE/CS 549 Project 3
Einstein Vision

Blake Bruell
Worcester Polytechnic Institute
babruell@wpi.edu

Cole Parks
Worcester Polytechnic Institute
cparks@wpi.edu

***Abstract*—This report outlines the process of creating a 3D visualization of a driving scene from a monocular forward facing camera view. The process is broken down into three phases, each of which encompass a certain set of target goals. Techniques were explored and implemented to detect lane lines, detect objects in the scene, localize objects in 3d space, and create a 3D visualization of the scene using Blender. The report also discusses the challenges faced and the results obtained from the implementation of the techniques.**

I. Phase 1

In the first phase of this project our goals were to get basic features of a scene localized in 3D space, including vehicles, lanes, pedestrians, traffic lights, and road signs. To accomplish this, we looked at different possible pipelines for the object detection and localization.

A. Object Detection

Some techniques for object detection, such as described in *3D Bounding Box Estimation Using Deep Learning and Geometry* [1], will directly output a 3D bounding box of teh detected objects using a single image. These techniques have the distinct advantage of providing all needed information to render the object from a single technique, but in general the performance of these technique can be limited by the fact that are doing so much. In addition, pre-trained weights on classes other than vehicles were not easily available, and as such our team looked for other solutions.

The obvious choice, then was to use a information from multiple different networks, each providing some needed information. At the bare minimum, two networks would be needed: one to detect objects, and one to predict depth. Our team chose solutions which were easy to get running, as we wanted to focus on getting outputs from the networks as quickly as possible. To this end, we elected to use the YoloV9 framework [2], due to its outstanding performance, and also ease of use due to the fantastic Ultralytics framework for YOLO models. The output of the YOLO model is shown in Figure 1.

Fig. 1: Output of YOLO model

Fig. 2: Output of ZoeDepth model

B. Depth Estimation

For depth estimation, there are many different techniques available. One technique in particular stuck out in the early research that our team did, namely ZoeDepth [3]. ZoeDepth is monocular metric depth estimation technique which builds upon the MiDaS architecture. The ZoeDepth model is trained on both indoor and outdoor scenes, and is able to generalize very well. ZoeDepth also had the distinct advantage of being setup to be run via TorchHub, a online repository of networks. As such, getting the network running, and integrating it into the existing YOLO pipeline was much easier. The output of the ZoeDepth model is shown in Figure 2.

C. 3D Localization

Once the outputs from YOLOv9 and ZoeDepth were obtained, the next step was to localize the objects in 3D space. This was done by first taking the center of each bounding box, and then using the depth map to get the depth of the object at the center of the bounding box. The pinhole projection model was then used to get the 3D coordinates of the object. We began with the pinhole projection model, which is shown in Equation 1. In this equation, λ is a scalar, $\mathbf{K}$ is the camera intrinsics matrix, $\mathbf{R}$ is the rotation matrix, $\mathbf{t}$ is the translation vector, X, Y, and Z are the 3D coordinates of the object, and x and y are the 2D coordinates of the object in the image.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

The equation was then simplified by assuming $\mathbf{R}$ and $\mathbf{t}$ are the identity matrix and zero vector, respectively. This simplification is shown in Equation 2.

$$\lambda \mathbf{K}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2}$$

From this equation, we could get the ray direction of object

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \lambda \mathbf{K}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{3}$$

where λ was a normalization factor, so that the magnitude of the ray direction vector was 1. The 3D coordinates of the object were then calculated by multiplying the ray direction vector by the depth of the object at the center of the bounding box, which gave the object's 3D location.

This pipeline was very effective, but it was limited by the fact that the ZoeDepth estimates the depth of the object's surface, and not the center of the object. This meant that the 3D localization was not perfect, but it was still very good. The output of the 3D localization is shown in Figure 3.

The benefit of this pipeline is that is simple, fast, and applies for all classes of objects that the YOLO model can detect. The downside is that the 3D localization is not perfect, and no information about orientation is provided.

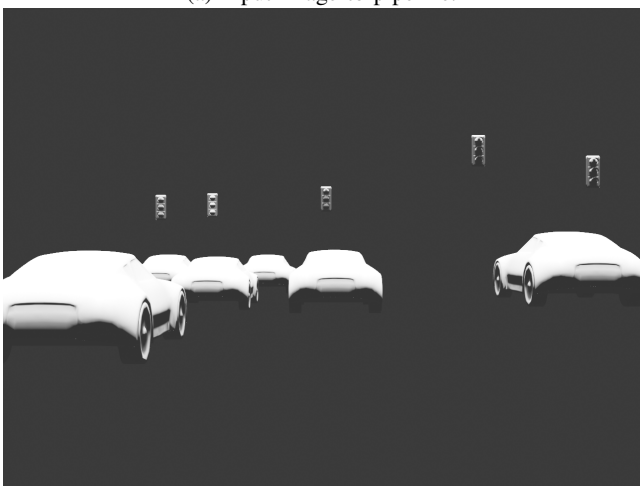
D. 3D Rendering

Finally, to 3D render the scene, a json data-structure was created which contained the 3D coordinates and class of each detected object. This json file was then read in by a Blender python script which placed each object in the scene at the correct location. The output of the 3D rendering is shown in Figure 3.

E. Lane Detection

There were many choices for lane detection, but we sought to use a technique which gave 3D lane outputs directly. We found many such powerful techniques, such as BEV-LaneDet [4] and PersFormer [5], but we ran into some serious challenges with getting these networks running. In both cases, the code was publicly available on GitHub, but dependencies were not well documented, and when documented were often many years out of date, making it difficult to setup the environment to run the network. In some cases pre-trained weights were missing, making it difficult to run the test the network, or simply prohibitively expensive to train the network. In many cases, the networks were only setup to be trained and validated on datasets, and not to perform inference on new data. As a result, we were unable to get these networks running in time for the first phase of the project.

(a) Input image to pipeline.



(b) Output of 3D localization.

Fig. 3: 3D localization of objects.

II. Phase 2

In the second phase of this project, our goals were to successfully detect lanes, traffic light colors and type, and estimate car poses. We began this section by performing a comprehensive literature review to understand the state-of-the-art techniques for each of these tasks. We then moved forward with the techniques which showed the most promise.

Fig. 4: Output of lane detection from YOLOPv2 [7].

A. Lane Detection

Given that lane detection was missing feature from our Phase 1, we wanted to get this working as soon as possible. The techniques which provided 3D lane output discussed in the previous section were still not working, so our team looked into more techniques.

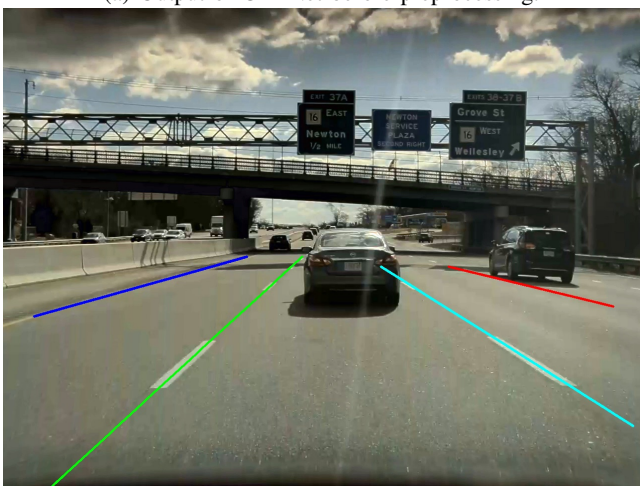
1) Other 3D Lane Detection Techniques: Other lane detection techniques were considered, such as Anchor3DLane [6]. Again, the same issues with functionality were encountered, and proved too time consuming to resolve. As such our team decided to look into techniques which provided lane output in 2D image space.

2) Segmentation of Lane Lines: The first class of technique that we found in the literature were techniques which segmented the lane lines in an image. The first technique we came across was an extension of YOLO for the express purpose of driving perception, called YOLOPv2 [7]. This network not only segmented the lane lines, but also provided segmentation of other regions, such as drivable area, and also detected vehicles in the scene. The output of this paper on our test data is shown in Figure 4. As is apparent in the image, the lane segmentation was not very precise, which would make it difficult to fit 2D lane lines to the output of this network. Even more problematic, the different lane lines were all classified with the same mask. This was a problem, as our rendering pipeline used Bézier curves to interpolate between a set of sampled points of each lane line separately. This meant that the output of this network was not suitable for our purposes.

Another segmentation technique using a Mask RCNN backbone was considered, but again, the output was not suitable for our pipeline, and the documentation and code for the technique were not as easily available as the YOLOPv2 technique.

3) 2D Lane Line Fitting: The final class of techniques considered would return lane line coordinates fitted to each individual lane line. These techniques were much more useful than the segmentation style 2D detection networks, as we could simply make the assumption that the lanes lay on the ground plane, and project the 2D lane sample points onto the ground plane to recover lane lines. The first technique our team considered was CLRNet [8]. This network provided decent output, and had documentation on how to run inference with the network on new images. The output of the network is shown in Figure 5a.

(a) Output of CLRNet before preprocessing.



(b) Output of CLRNet after preprocessing.

Fig. 5: Output of CLRNet [8].

Initially, the output of the network was underwhelming, but it was noted that the input images lacked contrast and brightness, and so did not match the training images well. We therefor applied a simple preprocessing step to the images before detecting lanes, which proved effective at improving the output of the network. The output of the network after preprocessing is shown in Figure 5b.

An extension of CLRNet, discussed in *Recursive Video Lane Detection*, added temporal information to the network greatly improving the consistency of the output on video inputs [9]. Again, the codebase could not be made to function on our test data, and so our team settled on using the CLRNet network for lane detection.

4) Lane Line Reprojection: Once the lane lines were detected, the next step was to re-project the 2D lane lines

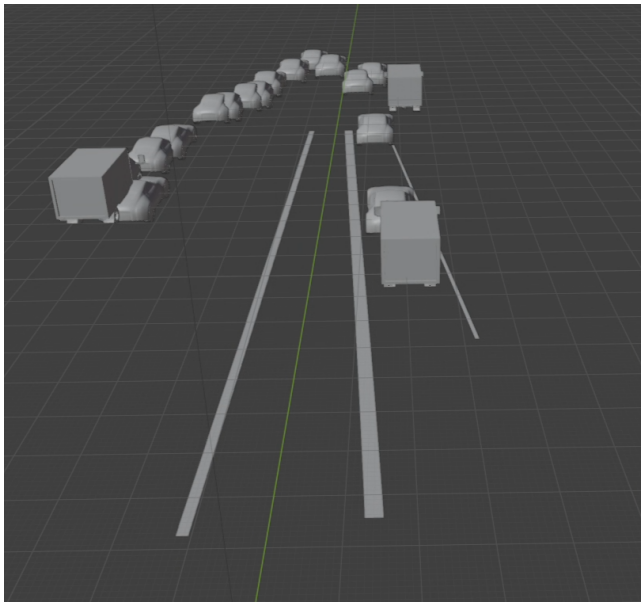


Fig. 6: Re-projected lane lines.

onto the ground plane. This again began with the pinhole projection model, as described in Equation 1. We then made the assumption that the camera was looking straight ahead, and that the ground plane was perpendicular to the XZ plane with $Y = -1.5$. This allowed us to simplify the pinhole projection model to the form shown in Equation 4.

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} X \\ -1.5 \\ Z \\ 1 \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} X \\ -1.5 \\ Z \end{bmatrix} = \lambda \mathbf{K}^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

From this equation, we could get the 3D coordinates of the lane lines. The output of the re-projected lane lines is shown in Figure 6.

B. Traffic Light Color Detection

While the YOLOv9 pipeline from Phase 1 provided 3D localization of traffic lights, it did not provide the color of the traffic light. Our team looked into the literature, and found that some people had fine tuned a YOLO network to detect traffic lights and their colors. We decided to use this technique, as it would fit nicely into our existing pipeline. We used data from cinTA_v2 dataset to fine tune a YOLOv8 network [10], but the output proved very poor, as detection of traffic lights in the first place was much more inconsistent, and even when detected the color was not correct. Some data manipulation was attempted, such as increasing the contrast, changing the color space, and changing the size of the images, but the output of the fine-tuned networks still remained unusable. Due to time constraints, our team decided to ignore the traffic light color detection for the time being, and focus on the other tasks.

(a) Output on training data.

(b) Output on test data.

Fig. 7: Output of 3D Bounding Box Estimation Using Deep Learning and Geometry [1].

Given more time, classical methods would have been pursued using brightness andn some color thresholding to perform the traffic light classification.

C. Car Pose Estimation

Crucially missing from the first phase of this project was the ability to estimate the pose of the cars in the scene. This was a difficult task, as the only information we had about the cars was their bounding boxes. Many techniques have been proposed over the years, but our team turned back to *3D Bounding Box Estimation Using Deep Learning and Geometry* [1], as it not only provided pose of the object, but also the 3D location. The output of the network on data from the dataset the network was trained on is shown in Figure 7a, and the output of the network on our test data is shown in Figure 7b. As is apparent from the comparison, the network did not generalize well to our test data, or some other issue was present. This failure, along with the fact that networks which predicted both bounding box and pose would preclude the use of the YOLO pipeline, prompted out team to look into other techniques for car pose estimation.

The next technique which our team looked into was called EgoNet [11], which had the advantage that it could take in the output of the YOLOv9 network, and then predict the pose of the cars in the scene. The network also came with pre-trained weights, necessary due to our time constraints, but again the code base failed to have good support or documentation on how to run inference on new images. Due to these issues, our

team did not include any pose estimation in our pipeline for Phase 2.

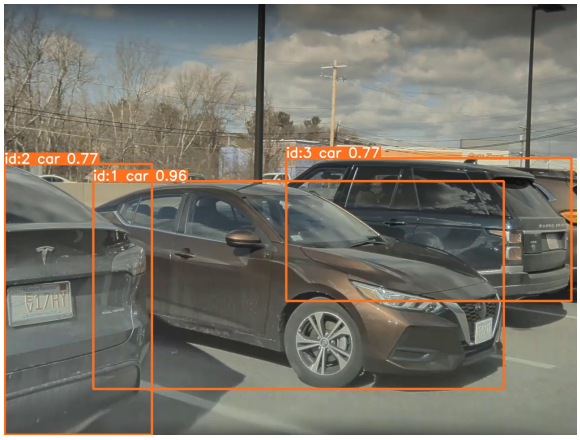
III. Phase 3

In the third phase of this project, our goals were to flesh out pose detection for cars, and add in the ability to detect traffic light colors, and improve the rendering pipeline so that entire scenes could be rendered in a reasonable amount of time.

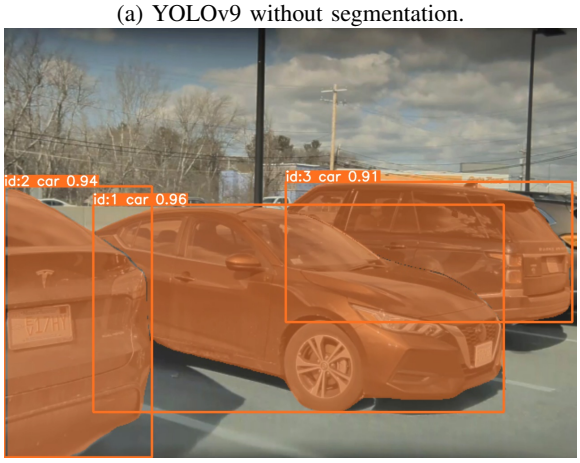
A. Improved Depth Detection

Another issue which presented itself in previous phases was the issue with using the centroid of the bounding box for depth estimation. Objects could be occluded in a such a way that the centroid lay on pixels which did not actually belong to the object, and as such the depth would be predicted incorrectly. To solve this, the segmentation version of YOLOv9 was used, which provided a mask for each bounding box of a detected object. A simple scheme wherein the median of the depth of each pixel in the mask was used as the depth of the object was implemented. The difference between the outputs of the two network versions, as well as the resulting depth mask, is shown in Figure 8.

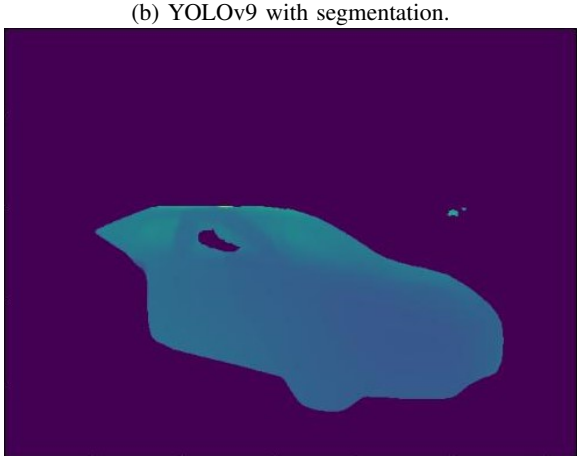
Another consideration with the depth estimation, as mentioned in Phase 1, is the fact that the ZoeDepth estimates the depth of the object's surface. This was accounted for partially by offsetting the depth of detected vehicles by one meter.



(a) YOLOv9 without segmentation.



(b) YOLOv9 with segmentation.



(c) Masked depth of one car. Notice the outlier points, which are ignored using the median scheme.

Fig. 8: Improved depth detection using segmentation.

B. Improved Lane Detection

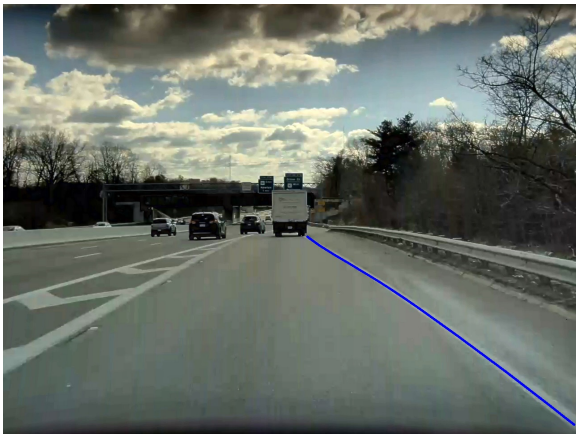
When running the CLRNet pipeline on some test videos, the output proved very poor, and so causes of the poor performance were assessed. First, hyperparameters of the network were tuned, specifically the cut distance, image size, and sampling range parameters, but these did not lead to consistently better results. Our team then noticed that the pre-trained weights and configuration of the network were set up for the CULane dataset, which did not match our input data well. The network also had weights and configuration for the LLAMAS dataset, which matched our input data much better, and so the hyperparameters were transferred to the existing LLAMAS configuration, and the network was tested again. This time, the output was significantly better, as shown in Figure 9.

C. Pedestrian Pose Estimation

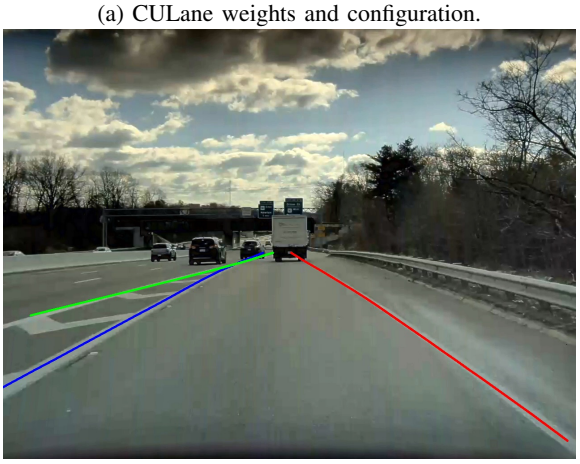
Pedestrian pose estimation was performed using derivative of YOLOv8 trained on poses, provided by Ultralytics [12]. This network detected pedestrians and accurately predicted poses, but we were unable to integrate it into our pipeline. The output of the network is shown in Figure 10.

D. Improved Rendering Pipeline

The pipeline for rendering an entire scene was also improved in this stage. The details of the rendering pipeline are shown in Figure 11 in Appendix A.



(a) CULane weights and configuration.



(b) LLAMAS weights and configuration.

Fig. 9: Comparison of CLRNet lane detection on a difficult frame using two different weights and configurations.

Fig. 10: Pedestrian pose estimation using YOLOv8.

IV. Shortcomings

After completing our entire pipeline, there are clearly some shortcomings, and things our team could simply not get working. Most notably car pose estimation and traffic light color detection were not fully implemented. There were some other subtle things missing, such as traffic cones, rigging pedestrian model in blender, lane type, speed limit signs, break light detection, and moving vs stationary vehicles. Most of these features were not implemented as networks found for the task could not be made to function on our data, but others were simply not implemented due to time constraints.

Of the pipelines our team did get working, the biggest limitations were the use of 2D lane line estimation, and the lack of robust 3D orientation information for detected objects.

V. Reflection

Our team put in many hours of work into this project, and most of that time was spent attempting to get techniques described in literature to run on our data, or even run at all. We found many powerful techniques, all with source code available, but despite this could not manage to get things actually working in time for the deadlines. We noted that the task of reproducing results from pervious papers, and generalizing the code to new data was tedious, and also very exhausting, frustrating, and time consuming. This made it difficult to make progress, as we wanted to avoid committing to a technique which would not work. The unbounded nature of the target goals for this project also made it harder to keep work focused and directed at the most important tasks. To address these challenges, our team frequently created priority lists and split up work into tasks which were entirely parallelizable to ensure that the most worthwhile tasks were completed first, and that work was never held up by the progress of the other individual.

The second aspect of this project which made it difficult to motivate (but something which cannot be avoided with this kind of work) was the fact that the project was not about *understanding* or developing intuition or rigorous knowledge about a topic, but rather was concerned purely with results. This project showed the difficulties with the latter goal, as well as allowed our team to gain a first hand idea of what we enjoyed and did not enjoy about this aspect of the field of computer vision.

On the positive side, while we may not have found the actual work enjoyable, and did not learned much about the actual techniques being used, as it would have simply been too much work, we did learn about the *process* of doing reviews of papers and replicating results. We learned both what makes it challenging and frustrating, and also the value in having experience in doing so. This project gave our team a initial taste of this experience, and also provided a good opportunity to learn about the challenges of working with other people's code, and the importance of good documentation and code quality. The challenges we faced in this project provided motivation to continue to ensure that the code we write is reproducible, understandable, and applicable by others.

REFERENCES

[1] A. Mousavian, D. Anguelov, J. Flynn, and J. Kosecka, "3d bounding box estimation using deep learning and geometry," *CoRR*, vol. abs/1612.00496, 2016. arXiv: 1612.00496. [Online]. Available: http://arxiv.org/abs/1612.00496.

[2] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao, "YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information," *arXiv e-prints*, arXiv:2402.13616, arXiv:2402.13616, Feb. 2024. DOI: 10.48550/arXiv.2402.13616. arXiv: 2402.13616 [cs.CV].

[3] S. F. Bhat, R. Birkl, D. Wofk, P. Wonka, and M. Müller, *Zoedepth: Zero-shot transfer by combining relative and metric depth*, 2023. arXiv: 2302.12288 [cs.CV].

[4] R. Wang, J. Qin, K. Li, Y. Li, D. Cao, and J. Xu, *Bev-lanedet: A simple and effective 3d lane detection baseline*, 2023. arXiv: 2210.06006 [cs.CV].

[5] L. Chen, C. Sima, Y. Li, *et al.*, *Persformer: 3d lane detection via perspective transformer and the openlane benchmark*, 2022. arXiv: 2203.11089 [cs.CV].

[6] S. Huang, Z. Shen, Z. Huang, *et al.*, *Anchor3dlane: Learning to regress 3d anchors for monocular 3d lane detection*, 2023. arXiv: 2301.02371 [cs.CV].

[7] C. Han, Q. Zhao, S. Zhang, Y. Chen, Z. Zhang, and J. Yuan, *Yolopv2: Better, faster, stronger for panoptic driving perception*, 2022. arXiv: 2208.11434 [cs.CV].

[8] T. Zheng, Y. Huang, Y. Liu, *et al.*, *Clrnet: Cross layer refinement network for lane detection*, 2022. arXiv: 2203.10350 [cs.CV].

[9] D. Jin, D. Kim, and C.-S. Kim, *Recursive video lane detection*, 2023. arXiv: 2308.11106 [cs.CV].

[10] W. Pradana, *Cinta_v2 dataset*, Open Source Dataset, visited on 2024-04-05, Jun. 2022. [Online]. Available: https://universe.roboflow.com/wawan-pradana/cinta_v2.

[11] S. Li, Z. Yan, H. Li, and K.-T. Cheng, *Exploring intermediate representation for monocular vehicle pose estimation*, 2021. arXiv: 2011.08464 [cs.CV].

[12] Ultralytics, *Pose*, Mar. 2024. [Online]. Available: https://docs.ultralytics.com/tasks/pose/.

Appendix A
Pipeline Details

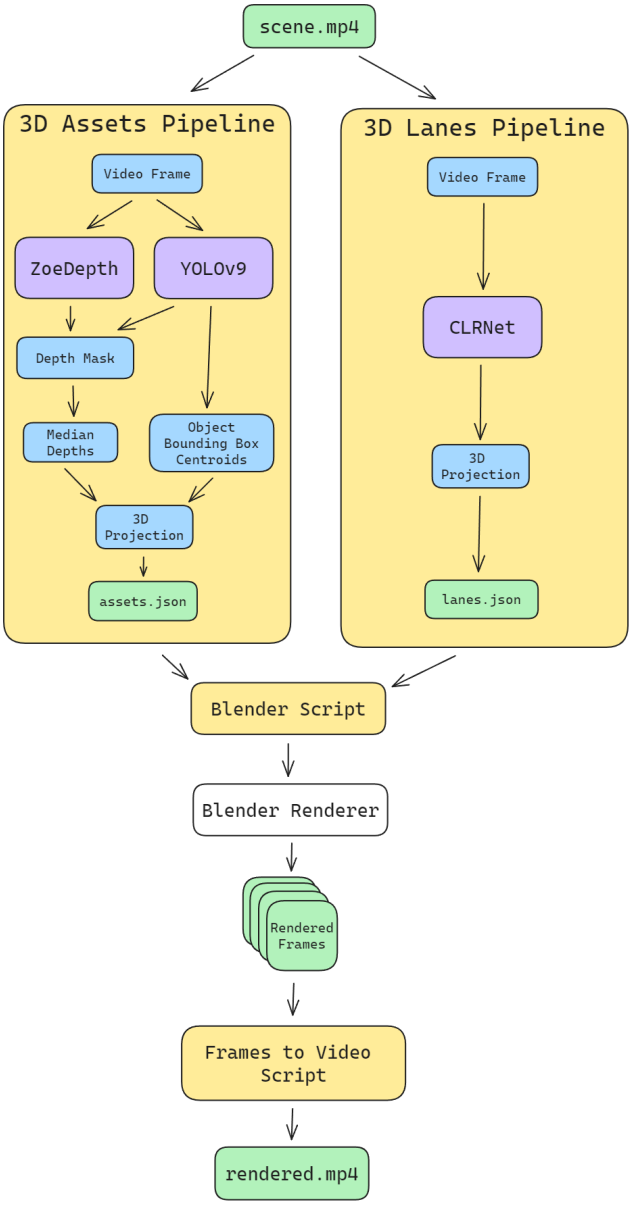


Fig. 11: Final rendering pipeline. Green blocks are files, yellow blocks are Python scripts. Within the Python blocks, purple blocks are networks, and blue blocks are hand written steps.

Appendix B
Results

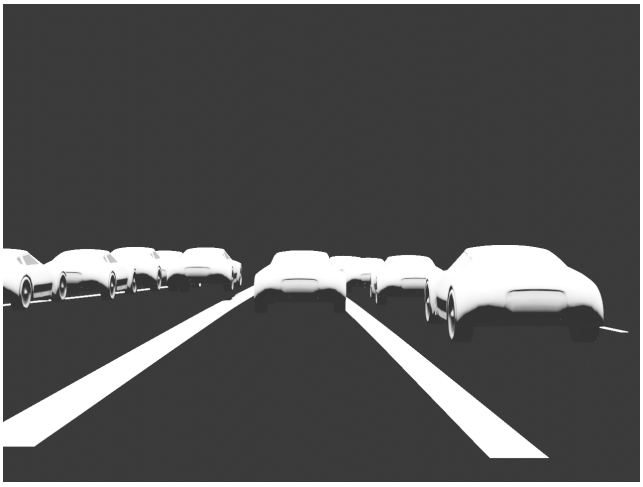


Fig. 12: Scene 6 output at frame 943.

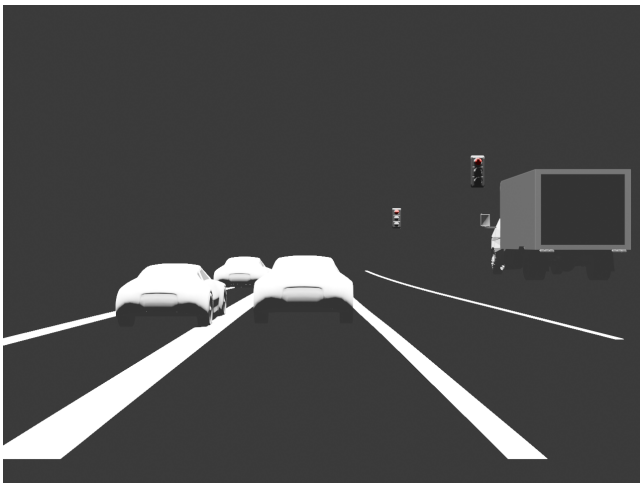


Fig. 13: Scene 7 output at frame 583.

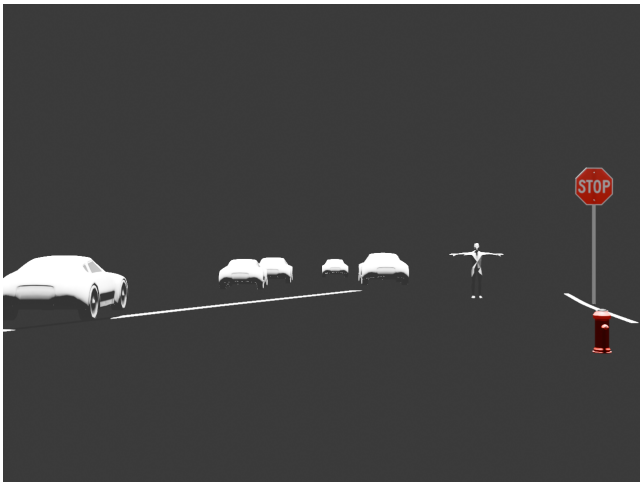


Fig. 14: Scene 8 output at frame 547.

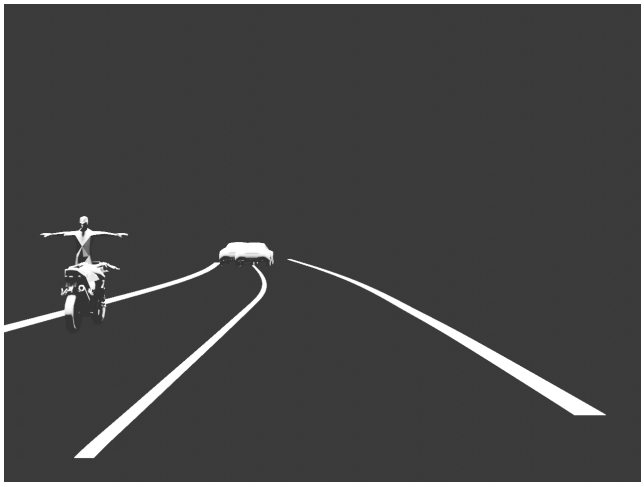


Fig. 15: Scene 13 output at frame 1165.