Computer Vision - Project 3: Einstein Vision

USING 2 LATE DAYS in Phase 2

Jesdin Raphael
Worcester Polytechnic Institute
Worcester, MA, USA
Computer Science
Email: jraphael@wpi.edu

Harsh Verma
Worcester Polytechnic Institute
Worcester, MA, USA
Robotics Engineering
Email: hverma@wpi.edu

Muhammad Sultan
Worcester Polytechnic Institute
Worcester, MA, USA
Robotics Engineering
Email: msultan@wpi.edu

I. Phase 1

The goal in phase 1 was to detect all the Basic Features which included:

1) Lanes
2) Vehicles
3) Pedestrians
4) Traffic Lights
5) Road Signs (Primarily Stop Signs)

A. *Overview of pipeline*

For generating the data, we obtained every 10th frame from the undistorted videos of the front camera. The next step was to detect objects (including lanes) using various deep learning models on all these 'key' frames, and extract bounding box centers and labels. Bounding box centers were then computed in pixels, and these pixel coordinates were then converted to real-world coordinates, using depths (Z) obtained from a monocular depth estimation model. The equations used to project the points were:

$$X = \frac{Z(x - c_x)}{f_x} \qquad (1)$$

$$Y = \frac{Z(y - c_y)}{f_y} \qquad (2)$$

Where, X, Y and Z are real-world coordinates, while x and y are pixel coordinates. f_x and f_y are focal lengths in pixels, while c_x and c_y are principal point coordinates. These camera intrinsic parameters were obtained by calibrating the camera ourselves. One important thing to note is that in our coordinate frame setup in Blender, the Y coordinate was only relevant for the traffic lights since we assumed that everything is flat, so nothing on the road has any vertical height, so for traffic lights all three coordinates were relevant, but for the rest we assumed vertical height was always 0. All objects were placed similarly in the blender scene, even the lanes.

B. *Object Detection*

In phase 1, for all object detection which included cars, pedestrians, traffic lights and road signs, we used a model called YOLOv8 [1]. Figure 1 shows a sample output of the YOLOv8 object detection model.

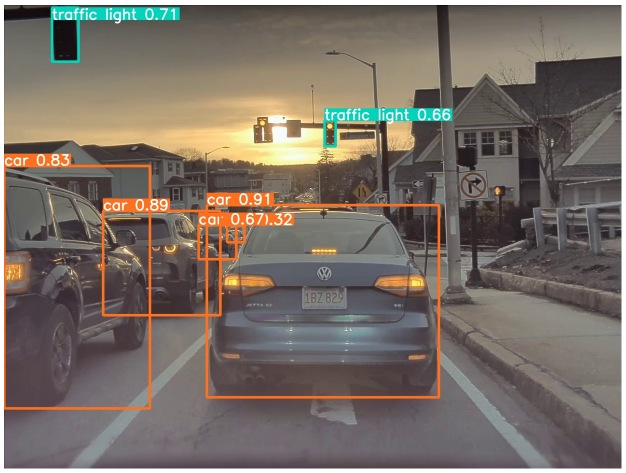

Fig. 1: Yolov8 Sample Output

This model gave us the object labels, but it did not give us their orientation. We did the orientation part in phase 2.

C. *Lane Detection*

The lane detection part was done using a Mask RCNN [2]. This model classified the lane markings into six different classes:

1) divider-line
2) dotted-line
3) double-line
4) random-line
5) road-sign-line (Arrows)
6) solid-line

A sample output of the model is shown in figure 2. We treated the solid line, divider line and double all as a solid line for ease of plotting in blender. Using the bounding boxes, we obtained multiple points on the marking area and then projected those points into the real-world in blender (exactly like objects) and then fit bezier curves to them. For the direction markers, we obtained all the points in the entire mask and similarly project them in blender.

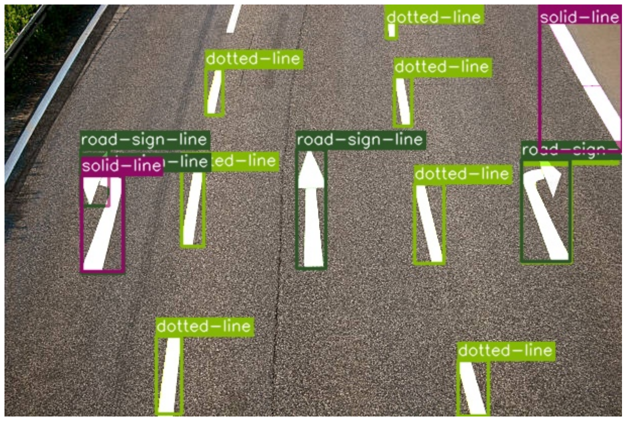

Fig. 2: Mask RCNN Sample Output

Figure 3 shows how we plotted the arrows in blender by projecting each point in the mask of the arrow.

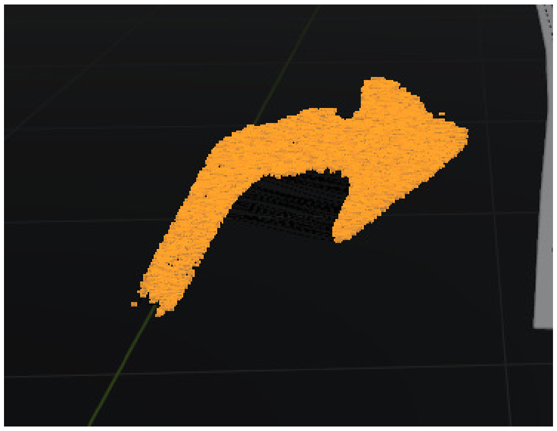

Fig. 3: Arrow plot in blender

D. Depth Estimation

For depth estimation, in phase 1, we used ZoeDepth [3] to get metric depth maps and then use those to get depth at the detected object bounding box center pixel coordinates. The depth was then used in equations 1 and 2 to project the objects. This model was good, but the scale sometimes changed from one frame to other quite drastically. This resulted in objects suddenly coming very close to the camera or going very far away. This model was later replaced in phase 3.

E. Blender

The blender part was executed as explained in the previous sections. One extra thing to note is that all distances had to be scaled, because the object models provided were quite large compared to the metric coordinates we were calculating. However, the relative positioning of the objects stayed the same. Figure 4 and Figure 5 show a sample output of a scene render in Blender and its corresponding image.

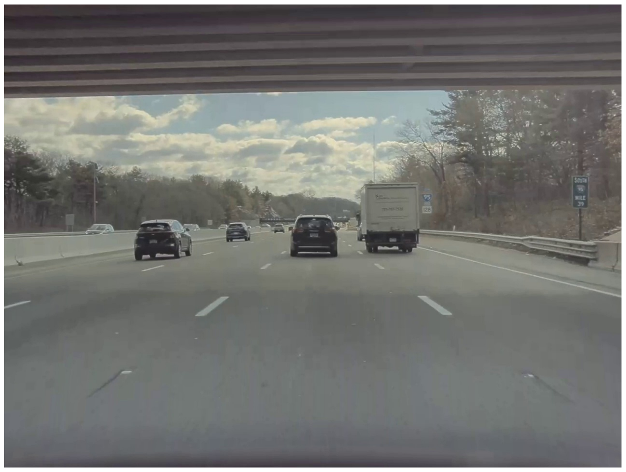

Fig. 4: Actual Image from Scene 1

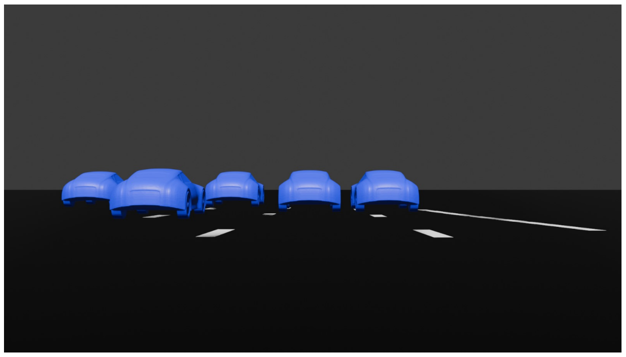

Fig. 5: Rendered Image from Scene 1

II. Phase 2

A. Vehicle Classification

In this phase, we performed vehicle classification, as previously we treated all vehicles as cars. The vehicles were classified into car, truck, bus, motorcycle and bicycle. This classification was also don't using YOLOv8. Further sub-classifications were done in phase 3, where YOLOv8 was replaced with another model which had a lot more labels.

B. Vehicle Orientation Estimation

For vehicle orientation estimation, we used a framework with two models called YOLO3D [4]. The first model was a YOLOv5 used to detect 2D bounding boxes, and the other one was used to detect 3D bounding boxes using the results of YOLOv5 model. We already had the position using YOLOv8, so we only extracted the yaw angle from this model's outputs. In this approach there was a problem that YOLO3D did not detect the same number of cars as YOLOv8 so there were sometimes vehicles detected by YOLOv8 that did not have a corresponding orientation from YOLO3D. Furthermore, we did not know which bounding box in YOLO3D corresponded to which bounding box in YOLOv8. To fix, this we assigned assigned the YOLOv8 detections the orientation of the closest 3D bounding box (using euclidean distance). That way each vehicle would have an orientation, even in cases where there is a mismatch in the detections from the two models.

The results of this model were not too promising. Especially when the cars were occluded, or not completely visible in the image. This was becuase the model assumed the 2D boudning box to be tight, and that would not be the case when objects are occluded or cut in the scene. The model also only detected certain object like cars, trucks, bicycles, etc so it did not give orientations of other objects, because it wasn't trained to do so. These problems could have been solved had we trained the model on a better dataset, or if we had found a better model, but due to time constraints, we stuck to this one. All the 3D bounding boxes predicted by this model had a pitch value that made them point upwards or downwards. We ignored this, since we were only interested in the yaw value. A sample output of this model's detection is shown in Figure 6.

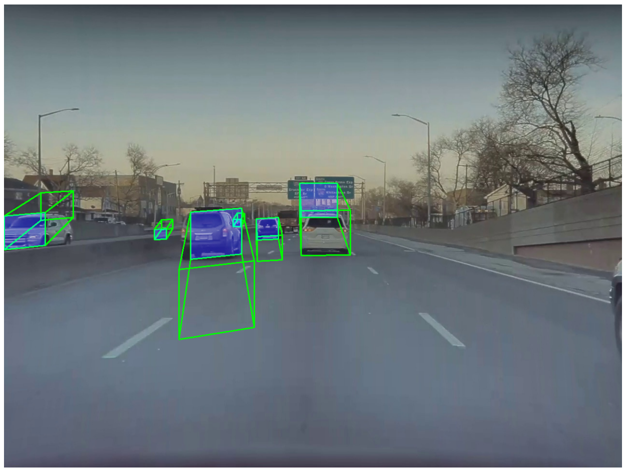
Fig. 6: Sample output of YOLO3D

C. Human Pose Detection (Pedestrians)

For human pose detection, we decided to use a model called Hybrik [5], because its output could be loaded into blender. However, there was a problem with the model, that it only detected humans properly if they were in the center of the image. This is shown in Figure 7 where the model detected the car as a human and tried to fit a model to it. To counter this we cropped out the human bounding boxes, placed them in the center and padded the image, and then input those images to the model, however the model would only give an animated output, and we were not able to convert those animations to individual frames (manually it was possible but could not be automated) so we decided to move onto another model, which we implemented in Phase 3.

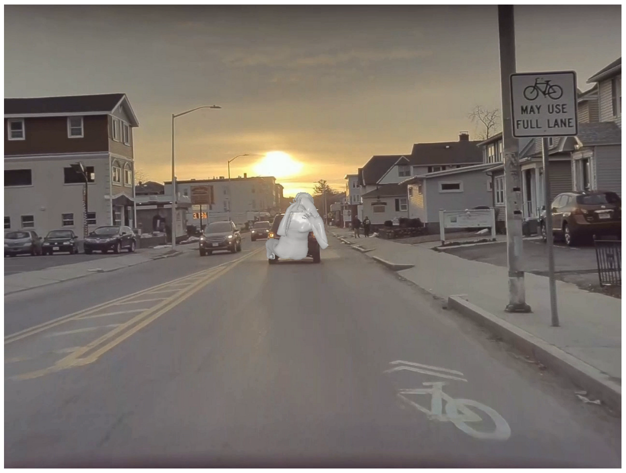

Fig. 7: Sample output of Human Pose using Hybrik

D. Traffic Light Color Classification and Arrow detection

In this phase, we tried using a pre-trained YOLOv3 model [6] which was trained specifically to detect and classify traffic lights. It also detected arrows on the traffic lights. However, upon using this model we found out that results were not good, so we decided to move this part to phase 3 since we were running out of time for phase 2. However, even in phase 3 we could not find a better way to predict the arrows, so we decided to use the YOLOv3 model anyways, but we would have had to modify our code and due to time constraints we did not end up doing this part, and only did traffic light color detection.

E. Detecting Other Objects

Other objects like trash cans, traffic cones, road signs were detected using another object detection model called Detic [7] by Facebook Research, as these objects were not detected by YOLOv8. A sample output of a frame with trash cans is shown in Figure 9.

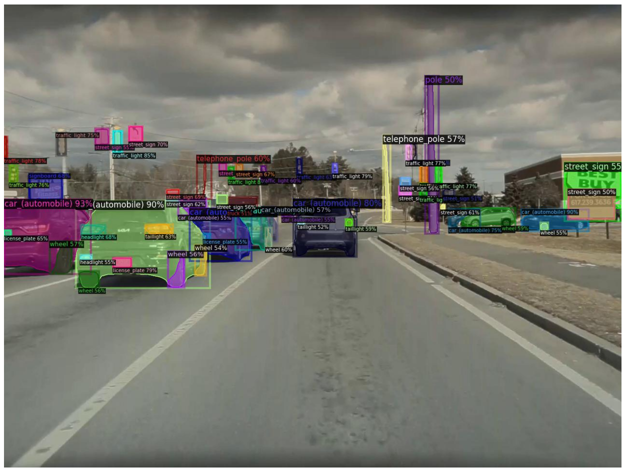

Fig. 8: Object detection done by DETIC

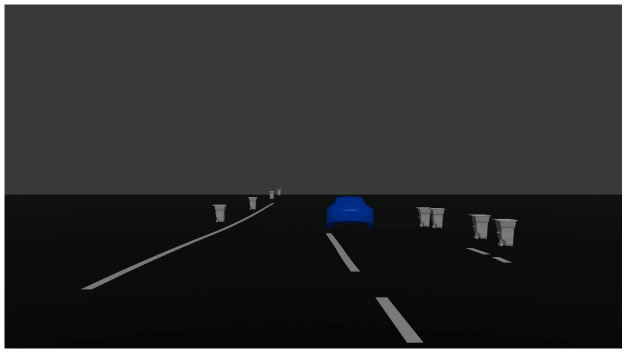

Fig. 9: Render of scene with trash cans detected using Detic

This model had a significantly larger set of labels that it could predict, so we completely shifted from YOLOv8 to Detic in Phase 3, for the vehicles, signs, and everything else. The one thing we did not implement as expected was the speed limit sign, in which we were expected to display the detected speeds as well. We eventually did find a model [8] that did this, but that model was not trained on a very large dataset, and it would have taken too much time to incorporate it into our existing code so we ended up only detecting the speed limit signs using Detic and not the speeds.

III. Phase 3

A. Classification shift to Detic and Depth shift to Marigold

We decided to shift all object detections to Detic. It is able to not only able to classify the same objects as YOLOv8, but also classify cars into sedans and SUVs as well.

However, the pretrained model we got was trained on the objects365 dataset, and was not as good as the one trained on the KITTI dataset, which took a lot of time to generate results. In the light of subpar human detection results by Detic, we decided to use Yolov8 for the same.

Furthermore, we shifted our depth estimation model from ZoeDepth to Marigold [9] as Marigold's scale estimation was not changing drastically from frame to frame. However, using Marigold did cause our lanes to get curved for some reason that we are not sure of.

B. Human Pose Detection

For human pose detection we decided to try another model called MeshNet [10] which provided us with individual *.obj* files instead of a whole animation. This worked, and sample output is shown in Figure 10.

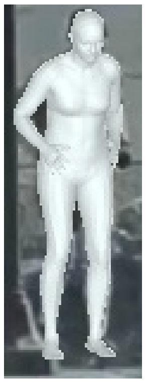

Fig. 10: Sample output of Human Pose using MeshNet

C. Traffic Light Color Classification

For this, we got the bounding boxes from Detic, cropped them out, divided them into three equal parts (this assumes that the bounding box is relatively tightly fit and includes the entire traffic signal), converted the crop into HSV format and thresholding based on average brightness of each part. If the top part had the highest average brightness, the light was assigned the color red, if the middle part had the highest brightness the color would be yellow, and if the bottom part had the highest average brightness the color would be green. Furthermore, we applied another threshold for the light being off. A sample blender output of traffic light color detection is shown in Figure 11.

This method assumes that the bounding box for the traffic light is tightly fit, and encapsulates the entire traffic signal, otherwise this approach fails. As shown in Figure 12, the bounding box does not completely cover the traffic signal, and hence this particular case gave the prediction that the light was yellow since the middle part of the bounding box is the brightest. We wanted to make the approach agnostic to color so that even if car headlights or the sun caused color thresholding methods to fail, this would still work using overall brightness.

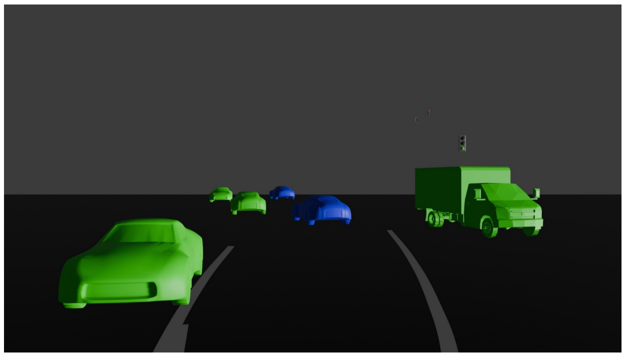

Fig. 11: Traffic light detection blender output

However, this method has its own drawbacks. A better approach would be to train some deep learning model specifically for this task with lots of data.

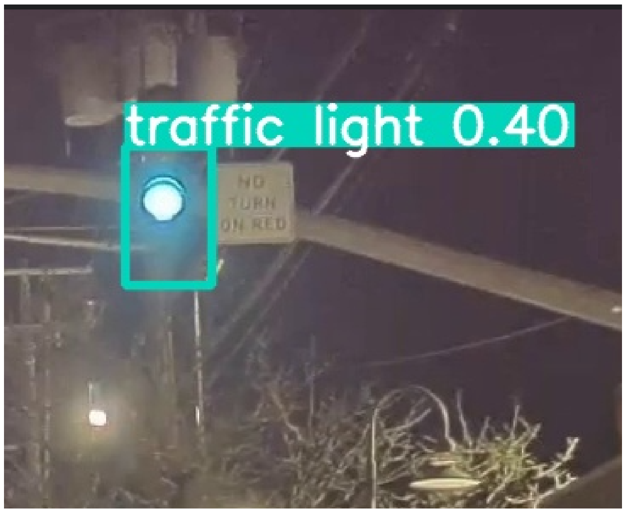

Fig. 12: Traffic light detection corner case

D. Taillight and Brake light Detection

The taillights and brake lights were detected using detic as well, but there was a lot of overlap between their bounding boxes so we treated them as the same thing, a brake light. The idea was to use brightness thresholding, where if the overall brightness of a bounding box was less than a certain value the light would be considered off. We would have further added a color threshold to see if it had enough red color as well because brightness could be caused by other sources as well. Now if both lights were on, it would be safe to say that the car was braking, and if one of them was on it most likely meant that the car indicator was on. However this method did not take into account the situations where both the indicator and brakes were on, so the method was not very good. Another improvement we thought would be to add color thresholding separately to separate out yellow and red colors. If there was not a lot of overlap between the yellow and red parts, it would mean that the yellow was most likely coming from the indicator and not some external source. However, due to time constraints we could not implement all of this.

Instead, we came up with an easier but a very naive approach where we would crop out a car's bounding box, color threshold and if it had a certain level of red value, the brakes would be considered on. We did not implement indicator in this phase.

E. Stationary and moving Objects

For identifying Stationary and moving Objects, we used the optical flow concept. Moving cars were colored blue, and stationary cars were colored green, while for every last frame the car is pink to denote unknown state. A sample output is shown in Figure 13. The code we referred to has been referenced [11] in the Bibliography. The code uses OpenCV's $calcOpticalFlowFarneback$ function. It takes two consecutive frames and a point and determines whether the point is moving or stationary. However, this method is very naive and only works well if all the cars are moving in the same direction and at relatively same speeds as the camera. The reason for this is that the code calculates the magnitude by which the particular pixel moves, and if that magnitude is less than a certain threshold it means that the object is moving along in the same direction as the camera at similar speeds for it to have low relative velocity and hence a lower change in position. However, a key flaw in this approach is to assume motion of cars in roughly the same direction. If the cars are moving at angles to the camera or are moving in the opposite direction to the camera, this approach will definitely fail.

The results using this approach were not good, as it classified a lot of the moving cars as stationary. Some threshold tuning on the pixel movement could be done to improve the results. A better way would be to use RAFT model to get the optical flow, and calculate the Sampson Distance. Following that, we could compare the results with a threshold and classify moving and stationary cars.

Fig. 13: Optical flow: img at t_1

Fig. 14: Optical flow: img at t_2

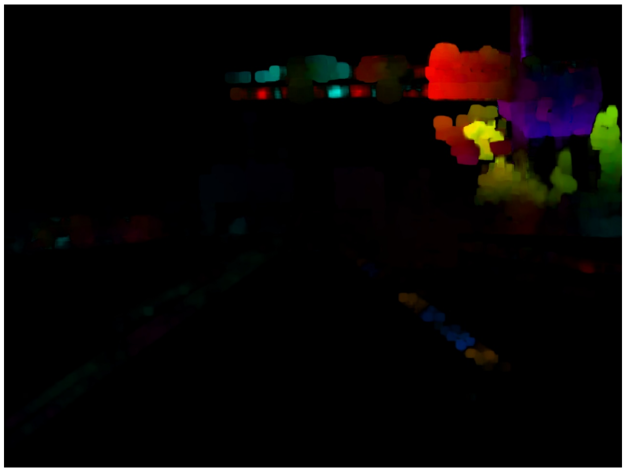
Fig. 15: Optical flow map using OpenCV's inbuilt function

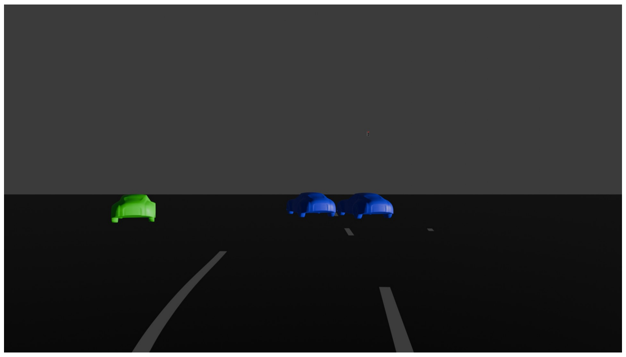
Fig. 16: Render showing stationary and moving objects (Green is stationary)

IV. Implementation Notes

Implementation of the models with their dependencies was a challenging task. Due to inaccuracies in detection of humans by YOLOv8, meshing by the Meshgrid Model was not up to the mark, resulting in inaccurate human pose representation in the renders.

Blender had a few major errors. The object files of some vehicles, like the truck was fixated on a particular orientation (facing backwards). After repeated attempts to change the orientation of the truck using the Blender UI manually, we were unable to do so. Similarly, human object models spawned half underground, and we were unable to change the translation manually. The same object files when run on our peers' machine worked well.

References

[1] D. Reis, J. Kupec, J. Hong, and A. Daoudi, "Real-time flying object detection with yolov8," 2023.

[2] S. R. Rath, "Lane detection using mask rcnn - an instance segmentation approach," https://debuggercafe.com/lane-detection-using-mask-rcnn/, Aug 2023.

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

[4] ruhyadi, "Yolo 3d object detection for autonomous driving vehicle," 2024. [Online]. Available: https://github.com/ruhyadi/YOLO3D

[5] J. Li, S. Bian, C. Xu, Z. Chen, L. Yang, and C. Lu, "Hybrik-x: Hybrid analytical-neural inverse kinematics for whole-body mesh recovery," 2023.

[6] S. R. Rath, "Traffic light detection using yolov3," https://github.com/sovit-123/Traffic-Light-Detection-Using-YOLOv3, 2023, accessed: [Insert current date here].

[7] X. Zhou, R. Girdhar, A. Joulin, P. Krähenbühl, and I. Misra, "Detecting twenty-thousand classes using image-level supervision," 2022.

[8] M. Gallacher, "speed-limit-sign-detection: Automated annotator for images with stop signs," https://github.com/michaelgallacher/speed-limit-sign-detection, 2023.

[9] B. Ke, A. Obukhov, S. Huang, N. Metzger, R. C. Daudt, and K. Schindler, "Repurposing diffusion-based image generators for monocular depth estimation," 2024.

[10] G. Moon and K. M. Lee, "I2l-meshnet: Image-to-lixel prediction network for accurate 3d human pose and mesh estimation from a single rgb image," 2020.

[11] S. O. contributors, "What is output from opencv's dense optical flow farneback function & how can this be visualised?" 2016, available online at: https://stackoverflow.com/questions/38131822/what-is-output-from-opencvs-dense-optical-flow-farneback-function-how-can-th.