Project 3 - EinsteinVision

Manoj Velmurugan*, Rishabh Singh†
Robotics Engineering
Worcester Polytechnic Institute
Email: *v.manoj1996@gmail.com, †rsingh8@wpi.edu

***Abstract*—In our project, we employed various deep learning models to detect cars, humans, their poses, lanes, and depth in images. These detections were then transformed into 3D coordinate space by using monocular depth. We developed a Blender python API based script to visualize these 3D detections, enabling a comprehensive and interactive representation of the data. The aim of the project to have a visually pleasing, yet informative display for a self driving car.**

I. Depth Estimation

In the broader scope of our project, while we successfully deployed various models to detect different elements within our scenes, these detections were constrained to 2D image coordinates, lacking the depth information needed for a realistic 3D visualization. We don't really need depth to scale since this is just for visualization. Our search led us to adopt Depth Anything, which stood out for its exceptional accuracy in generating depth data. To capture depth information with the highest possible fidelity, we wrote a script capable of producing high-resolution, 16-bit grayscale depth frames for each scene. This approach ensured we could extract detailed depth information, enriching our 2D detections with a semblance of three-dimensionality. Utilizing the camera's intrinsic parameters, we then converted this depth data into actual 3D coordinates for each detection. This process allowed us to accurately position and render each detected element within our Blender visualizations, significantly enhancing the realism and utility of our project's outputs.

Given a pixel (u, v) in the image plane and its corresponding depth Z, the 3D coordinates (X, Y, Z) in the camera coordinate system can be calculated as follows:

$$\begin{aligned} X &= \frac{(u - c_x) \cdot Z}{f_x}, \\ Y &= \frac{(v - c_y) \cdot Z}{f_y}, \\ Z &= Z \end{aligned} \tag{1}$$

where:

- u, v are the pixel coordinates in the image plane.
- c_x, c_y are the coordinates of the principal point (usually the optical center of the image).
- f_x, f_y are the focal lengths expressed in pixel units.
- Z is the depth of the object from the camera.

Fig. 1: Monocular Unscaled Depth

II. Detecting Lanes

In order to accurately detect lanes within diverse driving environments, we initially deployed CLRnet. However, this model demonstrated significant limitations, struggling with varying lighting conditions and being overly sensitive to color nuances. Despite several attempts to mitigate these issues through white balance correction, the use of monochrome video, and other image preprocessing techniques, CLRnet's performance remained suboptimal, failing to consistently recognize lane markings. Consequently, we pivoted to the Mask R-CNN lane detection model, which markedly improved our detection capabilities. This model not only identifies lanes but also provides detailed outputs including bounding boxes, segmentation masks, and classification IDs that distinguish between different types of lanes such as dotted or straight. To further refine our data, we fitted a second-order polynomial within the segmentation mask of each detected lane. This allowed us to generate a series of equidistant points along these polynomials, which we could then accurately plot in Blender. This approach not only improved the precision of our lane detection but also enhanced the visual representation of lanes in our 3D environment, complete with their specific classifications.

III. Object Detection

Our project required the identification of a wide array of objects, including traffic cones, traffic lights, cars, and humans, among others. Initially, we deployed YOLOv8, trained on the COCO dataset known for its speed and accuracy. However, we encountered a limitation in its pre-defined class categories,

Algorithm 1 Lane Detection and Polynomial Fitting

```
Initialize Mask R-CNN model with pre-trained weights
for each frame in video do
    Apply Mask R-CNN model to detect lanes
    Extract bounding boxes, segmentation masks, and class IDs
    for each detected lane do
        Extract segmentation mask for current lane
        Fit a 2nd order polynomial $f(y) = ay^2 + by + c$ to the mask
        Calculate lane length and divide into 10 equal segments
        Initialize list $P$ for points
        for $i = 1$ to 10 do
            Calculate $y_i$ as the $i$-th equidistant point on lane
            Calculate $x_i = f(y_i)$
            Append $(x_i, y_i)$ to $P$
        end for
    end for
end for
```

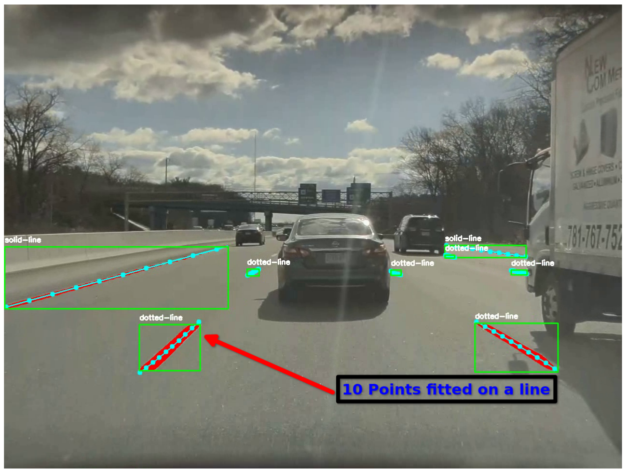


Fig. 2: Lane Detection with custom overlay

which did not encompass the full range of objects relevant to our application.

In search of a different solution, we transitioned to using Facebook's Detic, a model known for its extensive range of detectable classes. Detic significantly broadened our detection capabilities, covering every category of interest to our project. Despite our efforts, we faced challenges in further subclassifying cars into more specific types, such as SUVs or sedans, due to the absence of readily available pre-trained models tailored for such detailed classification.

Nevertheless, Detic empowered us to accurately identify and classify a comprehensive list of objects including cars, humans, traffic cones, traffic lights, and road signs, along with bounding boxes and confidence scores. Armed with this detailed object detection data, coupled with the depth information we had previously obtained, we were able to plot these detections in 3D with good accuracy.

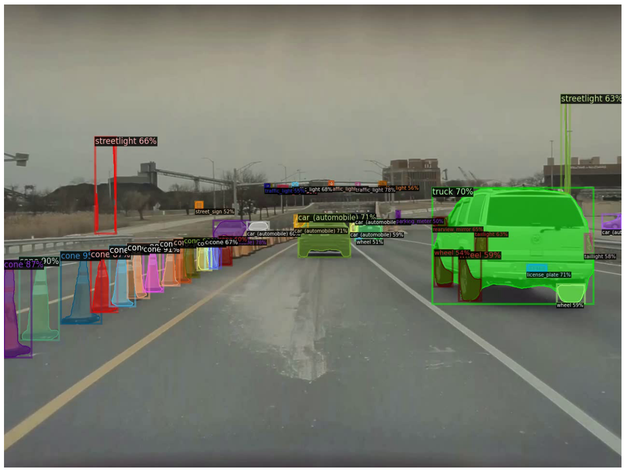


Fig. 3: Detic mask output, showing cars, trucks, traffic cones

IV. Car Pose

While our initial implementation of YOLO and Detic models nicely detected cars by providing 2D bounding boxes, we faced the challenge of lacking 3D orientation data for these vehicles. This dimension of data is crucial for accurately representing moving cars and their dynamics within our 3D visualizations. To overcome this limitation, we use the OpenPifPaf car keypoint model, which is trained on the ApolloCar3D Dataset. This model delivers precise keypoint detection for vehicles, marking critical points that define their structure and orientation.

Leveraging the depth information we had gathered, we were able to transform these 2D keypoints into 3D pose estimates with good accuracy. These pose estimates provide detailed spatial positioning for various parts of each car, enabling us to understand not just the location but also the orientation and movement direction of vehicles in the scene.

To visually represent these findings with fidelity in our 3D environment, we mapped the 3D pose estimates to Blender models of vehicles.

Later we found that the bounding box were not very accurate for this model. So we matched the centroid of the detected car bounding boxes, with th centroid of the bounding boxes given by detic bu using simple euclidean norm. This gave us a more precise bounding box along with good keypoints inside.

For estimating the pose, we posed the problem as a least squares problem. A standard car model was taken in blender and the worldpoints for every keypoint was measured. The car in the real scenario is assumed to be a rotated in yaw, translated and scaled version of our standard car model. Rotation can be expressed as $e^{i\phi}$ for linearizing the problem. This algorithm works really well even if only 3 keypoints are detected on the car as shown in Fig. 5. It works when the complete bounding box is absent or only part of the is visible.

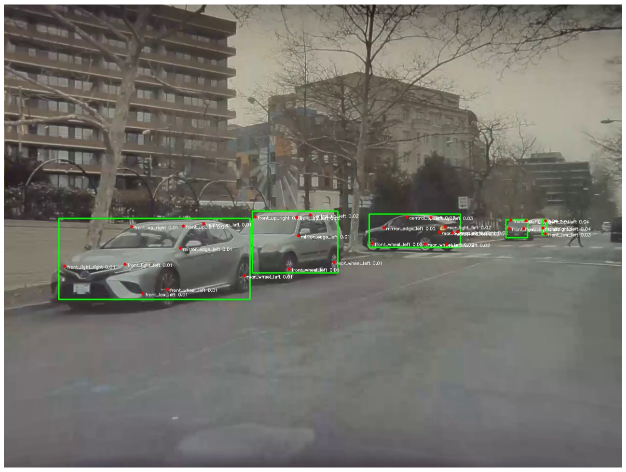


Fig. 4: Car Pose showing detected various key points

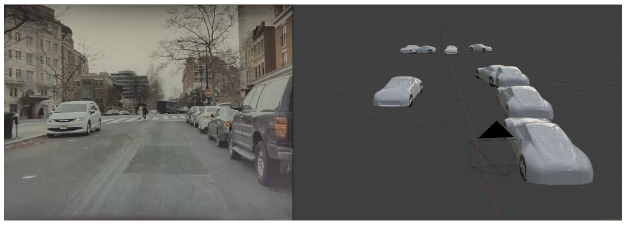


Fig. 5: Cars detected in scene with correct orientation

V. Human Pose

For detecting human poses within our scenes, we utilized the same OpenPifPaf network, trained on the COCO dataset. This allowed us to acquire detailed keypoints and bounding boxes for each detected human figure. By leveraging these keypoints, we could map each detected human pose onto corresponding models in Blender with high fidelity.

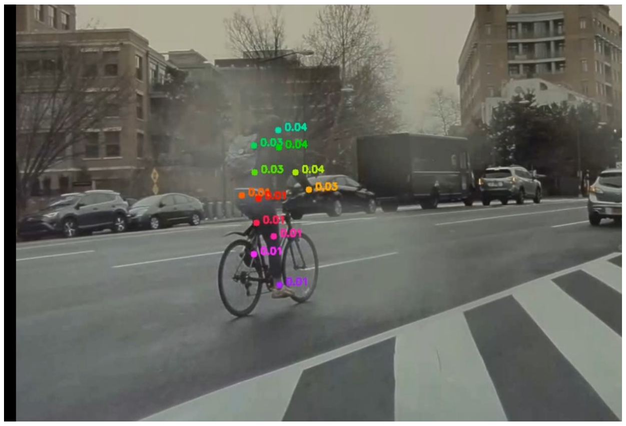


Fig. 6: Human Pose detected on Cycle

VI. Traffic Light and Tail Light Detection

Leveraging the car pose data obtained from OpenPifPaf, we were able to identify the tail lights of each vehicle. Additionally, Detic's provided us with accurate locations of traffic lights within our scenes. However, a key piece of information was missing from our dataset: the specific color within these detected bounding boxes, which is vital for understanding signaling and traffic flow.

To address this challenge, we developed a method to segment out the bounding box of each tail light and traffic light, calculating an average HSV (Hue, Saturation, Value) value for the area. We then applied a thresholded HSV mask to figure out the color changes within these segments. This approach enabled us to distinguish between different states of traffic lights and identify when a car's tail lights were activated, indicating braking or turning.

To facilitate the ”tuning” process of optimal HSV thresholds for each unique scene, we created a specialized tool that allowed for the manual adjustment of HSV values through sliders.

For traffic light we followed the exact same approach.

VII. Speed Bump Detection

We detected speed bump by using Detic to detect the speed bump sign next to it and then just put a custom cylindrical object next to it.

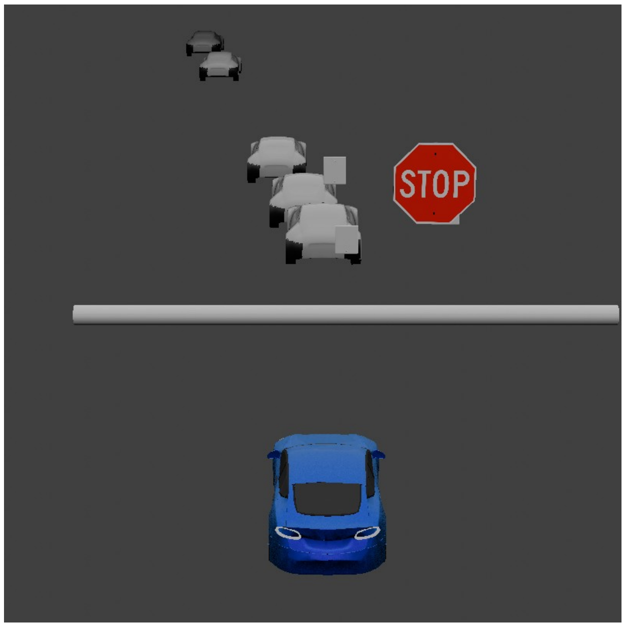
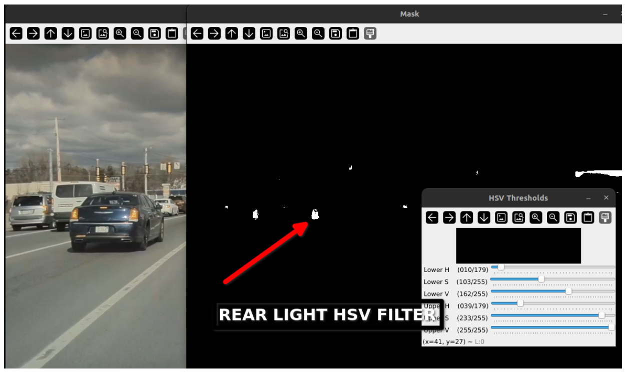


Fig. 7: Speed bump

VIII. Blender Output



Lower H (010/179)
Lower S (103/255)
Lower V (162/255)
Upper H (039/179)
Upper S (233/255)
Upper V (255/255)
(x=41, y=27) ~ L:0

Fig. 8: Custom made HSV tuning tool

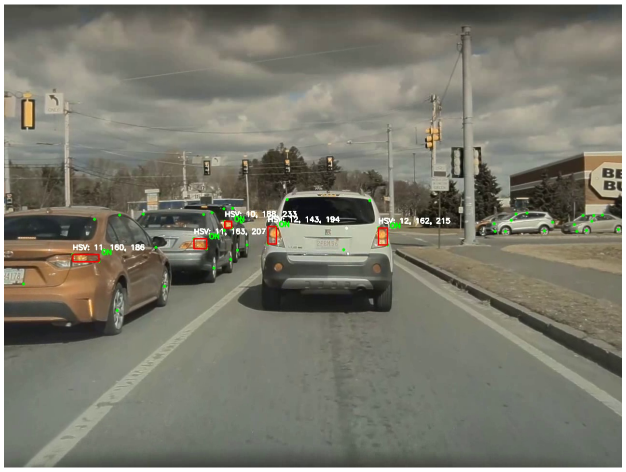


Fig. 9: ”ON or OFF” visualization created to test our thresholds of detected tail light bounding box

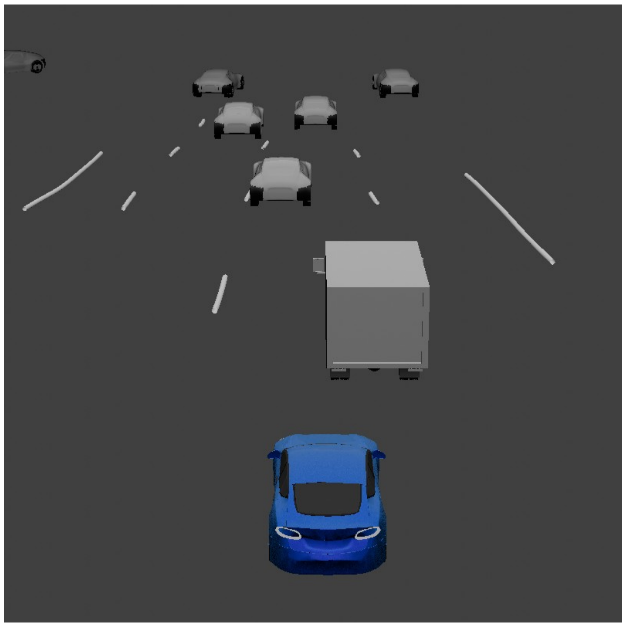


Fig. 10: Blender Highway Scene

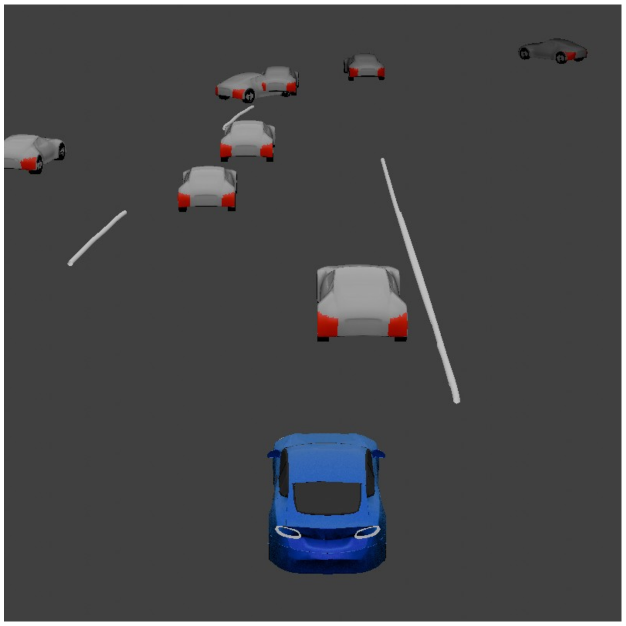


Fig. 11: Brake Lights

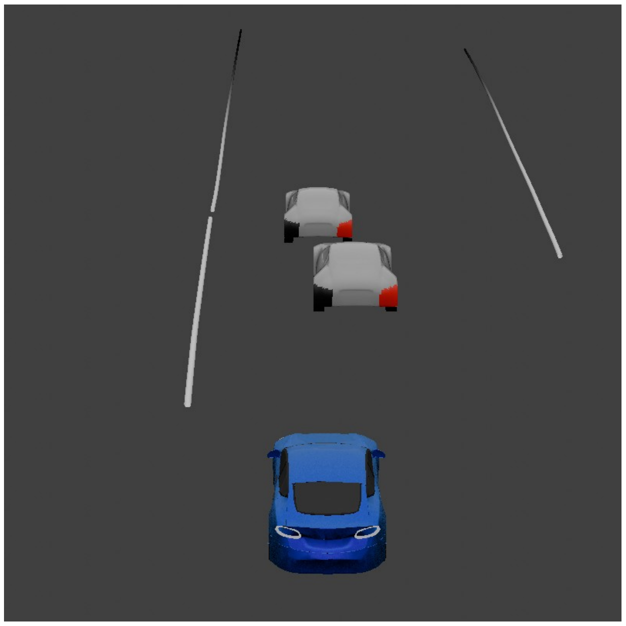


Fig. 12: Indicator

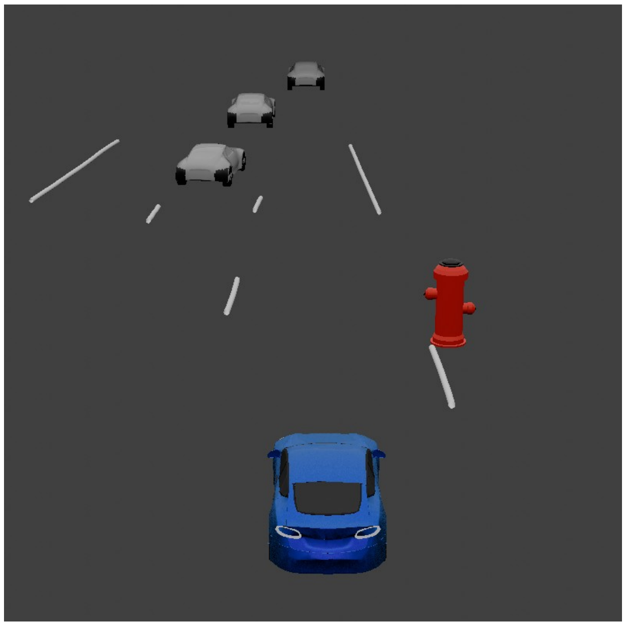


Fig. 13: Object Detected

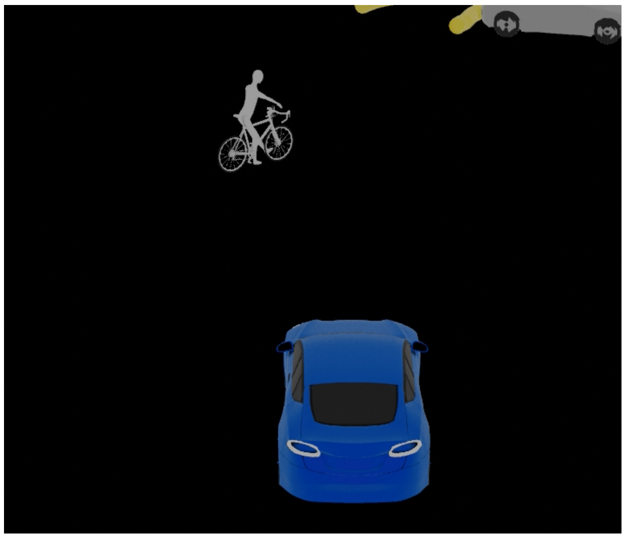


Fig. 15: Cycle with human as seen in Figure 6

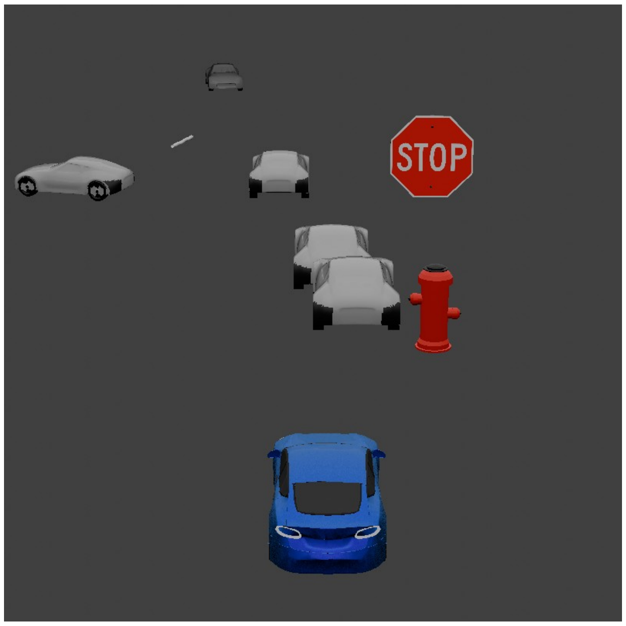


Fig. 14: Stop Sign

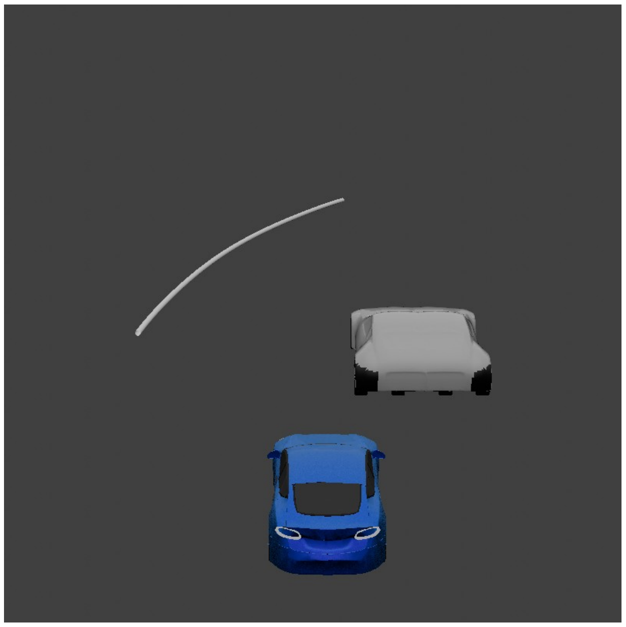


Fig. 16: Curved Lane

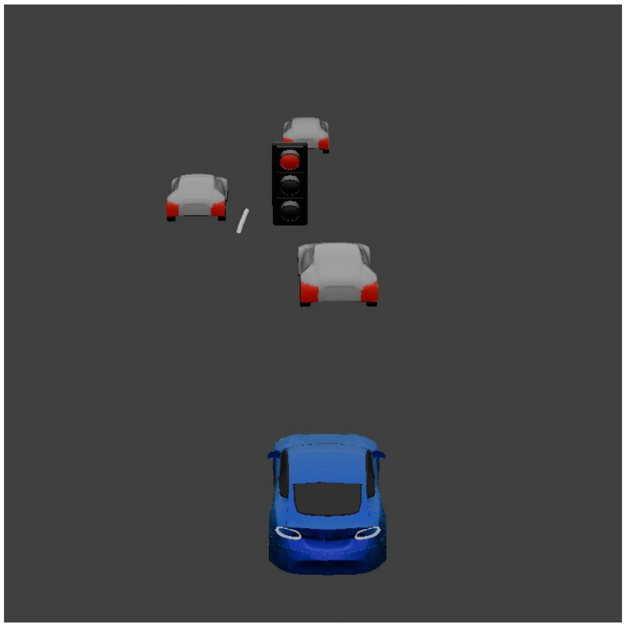


Fig. 17: Traffic Light