A Pioneering Scalable Self-driving Car Simulation Platform

Haolun Wu
Department of Computer Science and Engineering
Northeastern University
Shenyang, China
wuhl0203@163.com

Yunfei Feng*
Smart Home Lab Tech Lead
Iowa State University
Ames, USA
yunfei@iastate.edu

Abstract—**This paper proposes a novel hybrid, cross-platform 3D self-driving simulator, which offers a platform for researchers in the field of automatic vehicles to validate algorithms with ease and criteria to evaluate models in the community. Primarily our system decomposes the achievement of self-driving into four distinct steps by using a Tensorflow framework in machine learning: data generating, data balancing, model training, and model testing. The core part of our training section, the Alexnet, underlies training a deep neuron network with high accuracy at a fast speed. Some techniques, like balancing data, are employed to avoid overfitting, thus improving the robustness and accuracy; other functions, such as speed control, are designed for emulating the reality more vividly. By many study cases, it shows a strong flexibility of our simulator substantially. Various lane detection and obstacle detection algorithms are embedded smoothly in our system. In addition, Canny algorithm and Hough Transfer algorithm detect and draw out the edge of lanes perfectly, while YOLO architecture manages to detect obstacles.**

Keywords—self-driving, automatic vehicles, simulation, lane detection, obstacle detection, Tensorflow, machine learning

I. INTRODUCTION

In response to the public demand for convenience in this high-tech era full of artificial intelligence, there has been a growing interest in the development of self-driving car during the past decade. In the meantime, increasing companies are jumping into this promising ocean with the intention of a fully automated system invention that does not involve any human interference to be in the loop.

In 2011, Google announced its self-driving car project, which was once a secret [6]. Stanford University Professor Sebastian Thrun and Google engineer Chris Urmson led this project, discussing the details in a keynote at the *IEEE International Conference on Intelligent Robots and Systems* in San Francisco. Google's self-driving car is equipped with cameras, lidars, radars, GPS, and wheel encoders, using lidar to create 3D image of its surroundings. Meanwhile, BOSCH, one of the most leading multinational engineering and electronics company in the world nowadays, is working on a *Parking Steering Control* project with automated steering and partly automated braking [17].

Prior to a real production of sophisticated self-driving car system, especially for proof-of-concept, a simulation phrase is always willingly preferred at a relatively low cost. Typically, several crucial steps are required in the construction of a self-driving car simulator, in which the first stage is to generate data; the application of good datasets that enables testing and validation of results is a prerequisite all the time. In order to record a car movement, camera placement and video capture are worth serious considerations.

Even though many kinds of research and studies have been executed on lane detection [2,3,8,11–13] and obstacle detection [2,10] in the automatic vehicles field, the works on a reliable simulator on the whole self-driving car controller is still uncommon up to now. Consequently, it is a novelty to create a self-driving car simulator to offer convenience to the academic community.

In this paper, we designed a self-driving car simulator architecture and implemented it. It is a hybrid, cross-platform 3D simulator with high flexibility. Our simulator is able to generate data, making it possible to train the model later, to get a good performance in automatic driving. The architecture of our training model is based on AlexNet [9] to train a much deeper neural network in a fast speed rolling out overfitting. Many technical methods are applied to make the system more robust, such as data balancing, due to the fact that there are always far more quantities of data representing "Go forward" than those representing "Turn left", "Turn right", or "Go backward" when driving vehicles on a road in reality. Speed control technique is also implemented in the simulator to emulate the "throttle" of the car in the real world. The flexibility of our simulator is also obvious. Various lane detection and obstacle detection algorithms were also supported by our simulator. Canny algorithm and HT (Hough Transfer) algorithm were applied to detect and draw out the edge lines of the lanes; YOLO [15] framework was successfully embedded into our simulator to fulfill a real-time obstacle detection. The ultimate goal of our achievement is to provide an efficient platform to test others' algorithms and their own models, in addition, to have fun playing self-driving car.

The rest of this paper is structured as follows. Section II reviews existing technologies on automated vehicles and simulation tools. Section III presents the architecture and implementation. Section IV presents the flexibility of our simulator, mainly from the application of two different algorithms: lane detection and obstacle detection. Section V lists the limitations of our simulator and future work for this project, at the end the paper concludes.

II. RELATED WORK

Recently, many automatic vehicles techniques were proposed and implemented in the community. As we see it, three of the main issues for a self-driving simulator are lane detection, obstacle detection and controller. In this part, lane detection and obstacle detection are mainly described.

A. Lane Detection

Typically, the road lane detector consists of four main parts: warping, filtering, detecting and de-warping [13].

In the process of warping, the traditional method first selects a region of interest (ROI), which contains the road information. This step reduces the time for image-processing and shows more details. Afterward, the system converts an RGB image into a gray style without compromising the original road lines information [3]. Here the weight of R, G, and B during this transformation can be set as 0.02900, 0.58700, and 0.11400 separately [11].

The filtering process to distinguish the road pixels from the background plays a quite substantial part. Reference [13] just selects the scope of yellow and white color from the images by using LUV [14] and LaB image formats, thus separating the pixels of roads from those of the background. Alternatively, reference [12] uses Gabor filters, which is a best-known quadrature filters. Gabor filter is characterized by Gaussian-formed band pass filters. It is a suitable choice to accomplish functions demanding simultaneous measurement in both space and frequency domains [1].

As for the road boundary detection, many detectors were previously proposed. Khalifa, Hashim and Assidiq implemented a "canny" edge detector [8], which adaptively adjusts thresholds. M. Bertozzi and A. Broggi developed a GOLD system [2] to detect both generic obstacles and the lane position in a structured environment based on a full-custom massively parallel hardware. Also, Hough Transform (HT) is an effective and typical approach on lane detection. The principle is to convert the image on a Cartesian coordinate system to a Polar system, therefore each point on the same line $P(x, y)$ must fit the equation: $r = x\cos\theta + y\sin\theta$ [11], where r is the distance from the origin point to the straight line, θ is the angle with the positive X axis, the range of θ is $\pm 90^\circ$.

De-warping is a reverse process of the warping, which is necessarily the first step. Several methods have been used in this process to convert a 2-D perspective image into a 3-D perspective one. Polynomial regression is one of the good choices to reshape the lane and curve by fitting lane points [16].

B. Obstacle Detection

It is widely accepted that obstacle detection is indispensable for safe driving. Some technologies, such as LIDAR (Light Detection And Ranging device), can yield good performances. A ray cast is used to detect the distance by emitting laser, so this approach is not affected by the weather condition. Besides, the scale of its measurements is uniform despite distance [16].

Reference [10] sets up a novel obstacle detection and recognition method based on convolutional neural network, containing 15 layers in addition to the input layer. The network is also combined with a four-layer RPN (Region Proposal Network) work for generating obstacle recommended region and is trained by using the *Back Propagation* method. Also, the dropout strategy is used in this work to suppress overfitting.

However, some tasks with huge training data and unsupervised learning do not work well with the CNN method; therefore, reference [4] proposes a DSA (deep stacked auto-encoders) model, which combines a greedy learning method and an unsupervised k-nearest neighbors (KNN) algorithm. The DSA model treats the obstacle detection as an anomaly detection problem based on the V-Disparity data distribution, using the V-Disparity dataset to initialize training at a cold stage. Besides, the KNN algorithm is flexible to deal with the non-linearity in the data [7].

III. Architecture and Implementation

Our work proposes a novel hybrid, cross-platform 3D automatic vehicles simulation, which is particularly designed for the field of self-driving car. It attempts to build a standard open-source simulation for researchers in this area. Moreover, it follows a hybrid approach to generate data and integrate various algorithms on automatic vehicles. Hopefully, other researchers can use our platforms to practice and test their algorithm; companies can utilize our platform to prove their works.

This section presents the architecture of our simulation from different steps during the whole work. At the end of this part, the implementation of our system is briefly discussed.

A. Data Generation

It is very convenient to generate training dataset on our simulator. In any self-driving system, the first and most important step is always driving the car by human beings to collect plenty of data. After running our simulator, the image of driving will be recorded on the fly. In the former research and experiments, it is very popular to record the whole image on the screen in each certain time, then deliver to the training process in the next step. Nevertheless, as is known that the top half of the image recorded by the driving camera is usually the sky and other scenery which is involved less with the training part; the former way, sometimes leads to a waste of calculation resource. Therefore, in our simulation, *grabscreen* module allows users freely grabbing the screen just by defining the 4-tuple coordinate *($start_x$, $start_y$, width, height)*. Many coordinate screenshoting applications have such a functionality.

Additionally, *getkeys* module can record humans' manipulation of the car in our simulator. Our platform record the operation of these 4 keys when the player drives the car in the simulator with the grabbed screen simultaneously. We define an array named *output* which has 4 dimensions, mapping to different operations. The detail information is shown in Table I.

TABLE I. The key and output mapping to different operation.

Operation	Key	Output
Turn Left	A	[1,0,0,0]
Go Forward	W	[0,1,0,0]
Turn Right	D	[0,0,1,0]
Go Backward	S	[0,0,0,1]

B. Balancing Data

Overfitting is a headache to be cured in machine learning. Imagining one drives a real car on the road, normally the human driver seldom turns the steering wheel to turn the vehicle left or right; instead, during the majority of the time, he or she just steps on the throttle to keep the speed of car, remaining the steering wheel unchanged. This life experience tells us that if the human driver wants to collect the driving data in the real world, there must be a large amount of data denoting "Go forward", while only a few about "Turn left" or "Turn right", especially on a road with a perfect traffic condition.

Likewise, driving the car in our simulator has the similar situation. This may be convenient for the driver: it appears very easy to drive; while it is not a good thing for the training. Most of the data are about "Go forward"; nevertheless, the most important part during driving is about how to turn the car at a certain corner of the road. Therefore, if we use the data collected directly without any processing, it is likely that overfitting will occur.

Several approaches are discussed in reference [5] against overfitting, such as

- *Cross validation*
- *Training with more data*
- *Removing features*
- *Stopping early*
- *Ensembling (bagging and boosting)*

In our project, the *balancing* trick is applied to solve this issue.

Originally, we have 4 two-dimensional arrays to record the images. Keystrokes (mentioned in *Data Generation*) correspond to the 4 different control of the vehicle separately. Actually, we can ignore the array about "Go backward", because people rarely drive backward on the road, instead of parking. After saving the data collected during the *Data Generation* stage, we find out the array of the shortest length among *Forward*, *Left* and *Right*. Then it cuts the other two arrays based on the shortest length. It is better to remove the elements of the arrays from the beginning, because the traffic is much easier from the start, so to speak less training will be needed. Eventually, all the 3 arrays have the same length; thus, the overfitting is largely avoided and the following step is to train our model on the processed data. The detail is shown in Fig. 1 and Fig. 2, for which the blue boxes refer to the elements in the array; the white boxes refer to the abandoned elements after balancing data.

C. Training Model

Our training section is based on a famous deep convolutional neural network, the AlexNet [9].

There are 5 convolutional layers and 3 fully-connected layers in the architecture of AlexNet. Among the first 5 convolutional layers, the 2nd ,4th ,and 5th layers are merely connected to part of the kernel maps in the previous layer, while the 3rd layer connects to the full kernel maps in the 2nd layer. Besides, only the 1st ,2nd ,and 5th layers are followed by a max pooling layer. As for the 3 fully-connected layers, the first two layers connect all the neurons in the former layer, while the last layer is fed to a 1000-way softmax which produces a distribution over the 1000 class labels [9].

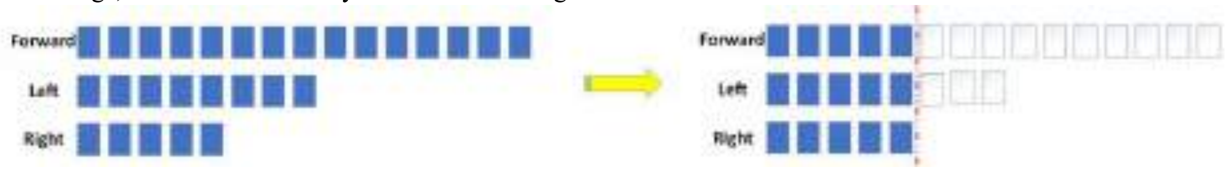

Fig. 1. One intuitive way is to compute the shortest array among *Forward*, *Left* ,and *Right*. Then cut off those longer two from the tail to achieve the data balancing.

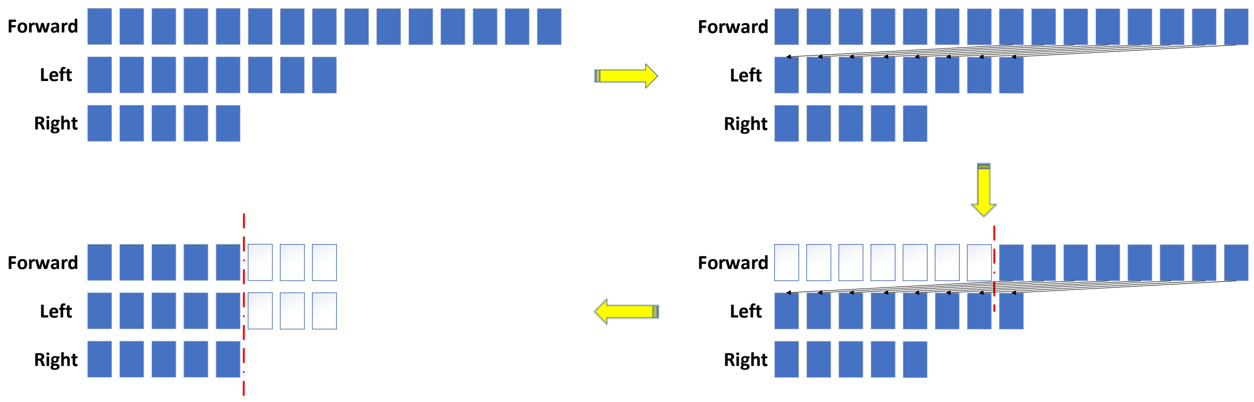

Fig. 2. Our thinking is to first abandon some elements in *Forward* from the head, then continue the work shown in Fig. 1. This is because driving in the beginning of our simulator is relatively easy; too many data recorded are valueless from the beginning.

AlexNet architecture has several advantages for training part. The first one is the ReLU (Rectified Linear Unit nonlinearity) activation function, whose output *f* as a function of input x is with $f(x)=\max(0,x)$, much faster than the traditional model $f(x)=(1+e^{-x})^{-1}$. Reference [9] indicates the 6 times of improvement in convergence with the **ReLU neurons** compared to the **tanh neurons**. Besides, the overlapping pool and dropout technology in AlexNet avoid overfitting efficiently. Hence, ReLU can help us train a much deeper neural network in a fast speed with a good performance.

In order to fit with our simulator when using the ReLU architecture, we first reshape our image to the size of the screen grabbed during the *Data Generation* mentioned before. Then we import the resized images and set *epochs* = 10 and *validation* = 500 for training our model based on the AlexNet. After training, our simulator teaches the vehicle to map the *input* images to the *target* operation.

D. Testing Model

The aim of the last section, the testing model, is to let the vehicle drive automatically in our simulator; thereby, we need to first educate the car how to control itself. Similarly, we still do not consider "Go backward". As for "Go forward", the computer just needs to press *W* and release *A* or *D* meanwhile. However, the thing is different when considering the other two controls. The correct control to make the car make a left turn is: press *W*, press *A*, release *D*, and then release *A* after t_{time}. The control for “Turn right” is similar. This is because the car can make a turn only when it has a forward speed. We finally set $t_{time} = 0.09s$ by experiment, for getting an ideal performance.

After training the model in the past section, the automatic vehicle has already established a mapping relationship between the scene grabbed and the operation of the car. In this section, users just set the car to the place where they start training their model and run the testing program. We then predict the operation of the car by comparing the scene grabbed now and the scene already trained earlier, remembered in the model.

It is obvious that the *prediction* is a ternary array, for which the sum of the three numbers is 1. We set a parameter, $turn_{threshold} = 0.75$, to indicate at what circumstance the car needs to make a turn. Assuming the prediction is $[a_0, a_1, a_2]$, if $a_0 \geqslant turn_{threshold}$ or $a_2 \geqslant turn_{threshold}$, then the computer will control the car to turn left or right, while it will keep going forward apart from these two situations until it is controlled to make a turn.

An improvement is additionally implemented in our simulator to mimic the functionality of “throttle” in a real car: the speed control. We extracted the speed from the dashboard, then if the speed is lower than $50km/h$, we make the computer press W 3 times for $t_{time} = 0.09s$ each; if the speed is lower than $60km/h$, we make the computer press W twice for $t_{time} = 0.09s$ each; and if the speed is lower than $70km/h$, we make the computer press W for $t_{time} = 0.09s$ once. Our experiments show that each t_{time} speed up can lead the vehicle to accelerate $10km/h$. Therefore, this is an efficient and easy way to parameterize the car’s speed neither too slow(less than $50km/h$) nor too fast(over than $80km/h$). The principle of operation and speed control is shown in Fig. 3.

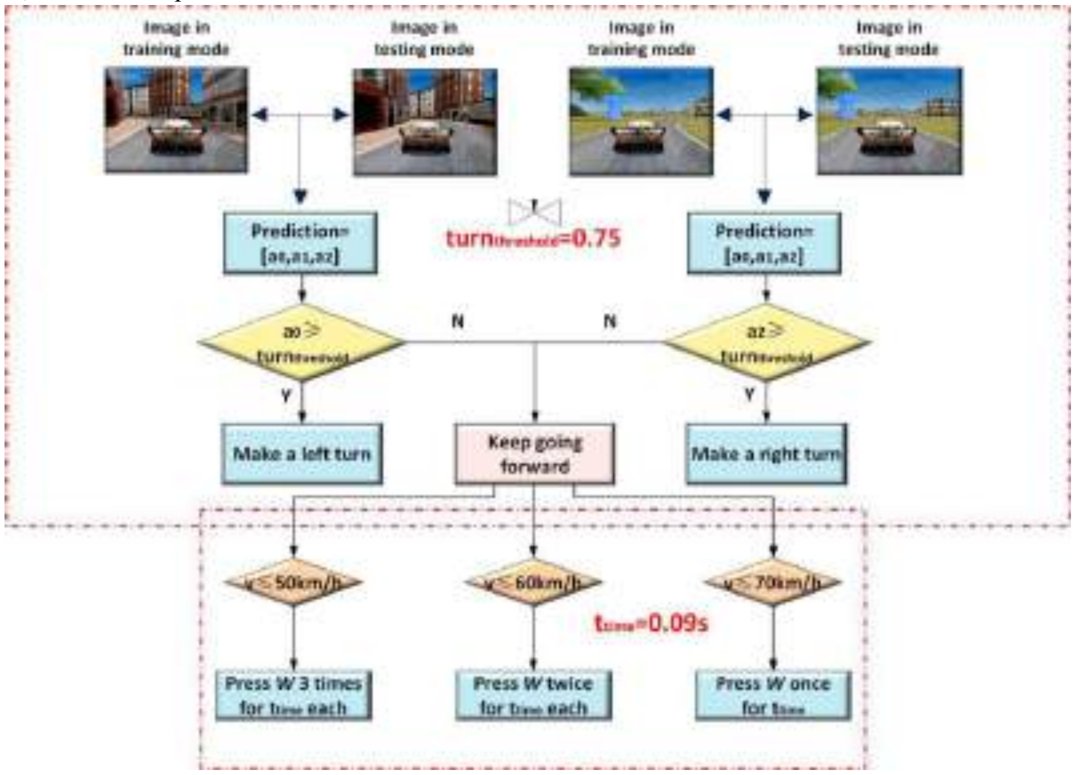

Fig. 3. The top box indicates the principle of when to make a turn for the car. The bottom box shows the principle of speed control. $turn_{threshold}$ and t_{time} are 2 critical parameters based on our experiments.

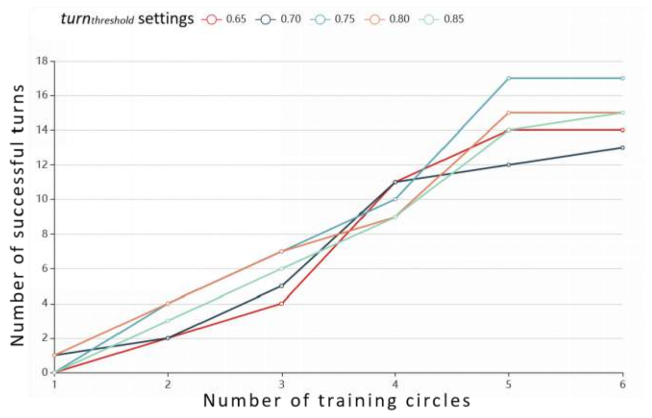

Fig. 4. The performance of our simulator when considering different number of training circles and different value of $turn_{threshold}$ settings.

In order to evaluate the performance of the platform, we primarily assess the number of successful turns of the car in our simulator. Of all the 18 bends, 5 are turning right and 13 are turning left. If the vehicle can make a turn without touching the lanes, it is considered as a successful turn. Fig. 4 shows the number of successful turns to the number of training circles in our simulator when setting a different $turn_{threshold}$ from 0.65 to 0.85, with a stride of 0.05. By experiments, 5 circles of training in the *Data Generation* section and $turn_{threshold}$ setting as 0.75 achieves a great performance with an accuracy of 94.4% approximately.

E. Implementation

Our simulator is propelled by Unity3D, programmable by C#, which is a cross-platform 3D technology available for the major operating system. Also, the game engine makes it possible to add obstacles and vehicles in the scenario. Moreover, it can also add different sensors or cameras on the vehicle for different perspectives, or make the traffic more chaotic to import more challenges. Therefore, users of our simulator are able to design their own favored traffic situations and test algorithms.

The core of our training and testing model is programmed by Python, which is efficient and universal. The *Tensorflow* library in Python facilitates to build deep neural networks, as well. Therefore, the source code is transportable, scalable, and intelligible on different platforms and operating systems.

IV. Flexibility And Scalability

The usability of our simulator is fully discussed in the former section. In fact, this simulator not only conducts data generating, training and testing, but also supports other plug-in algorithms. This section presents the successful combination of our simulation with lane detection algorithm and obstacle detection algorithm. We tried our best to make it as flexible and scalable as possible.

A. Application of Lane Detection Algorithm

The topic on lane detection in the field of self-driving has been deeply studied in the past decade, in addition, it is also one of the most significant and popular issues in this area.

In the previous tasks [2,3,8,10–13], researchers usually downloaded a video on vehicle driving from the Internet or used a DVR to record a piece of video, then ran the program for lane detection to find out the lanes on the road. This simulator not only renders road condition on the fly, but also it provides a screen grabbing API, so that video downloading becomes unnecessary any more. We apply the lane detection algorithm in our simulator directly, and the performance goes well. This produces a great convenience and high efficiency for researchers who want to test their algorithms on lane detection.

We used Python and OpenCV to detect the lanes on the road. Our algorithm does not care about the color of the image, as it first transfers the RGB image to a gray one. Moreover, as mentioned in *Lane Detection*, the top part of the image does not contain much valuable information in our lane detection task; therefore, a mask is needed to set a RoI (Region of Interest). We set a mask to do an XOR (Exclusive Or) operation with the original gray image and the RoI is in a shape of trapezoid at the bottom of the image. This region perfectly contains the critical information of the car and the front perspective, excluding unnecessary information.

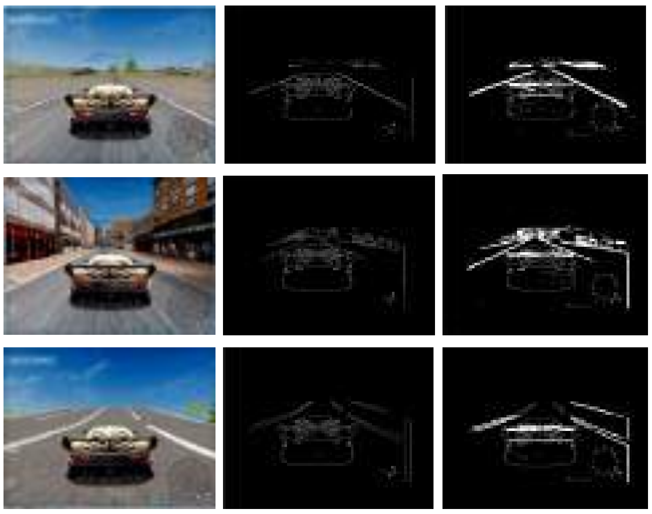
Fig. 5. Column 1 shows the original images in our simulator, emulating the reality. Column 2 indicates the gray images transferring from the original RGB images. Column 3 is the result after implementing Gaussian Blur algorithm and Canny algorithm.

After obtaining the processed image, Gaussian Blur and Canny algorithms are implemented to process the image in order to achieve image denoising and detect the edges in the image precisely. Gaussian Blur in OpenCV2 mainly targets to smooth the edges, or rough edges will occur caused by noises. We set the *kernel size* in Gaussian Blur to be 3, because the bigger the value of the *kernel size* is, the more obscure the image will be, also consuming much longer time to process. Canny is used to find the edges in the next step. The lower-threshold and the higher-threshold are two significant parameters in Canny. We set *threshold*1 = 200 and *threshold*2 = 300 by testing. The result of our experiment is shown in Fig. 5.

The final step is to use Hough Transfer to detect lines in the edge images, and then draw them out. We set the *threshold* = 180, *minLineLength* = 20 and *maxLineGap* = 15 to fit our simulator with a good performance. Fig. 6 shows our results of lane detection after running the Hough Transfer algorithm.

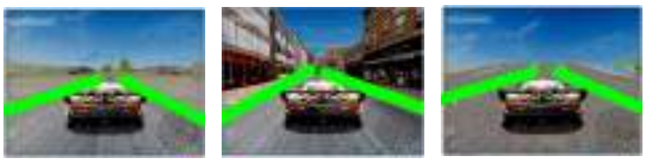
Fig. 6. Draw out the lanes by using Hough Transfer algorithm.

B. Application of Obstacle Detection Algorithm

Obstacle detection is a field studied even before the emergence of the automatic vehicles topic for its wide use in many aspects from image recognition to smart driving. The past works [4,10,16] mainly based on discrete data in a relatively safe environment; however, the real situation is totally different from those in the past works because of noises in the traffic. Therefore, our contribution is to test whether the algorithm can be applied in our simulator which mimics a real traffic world full of noise.

YOLO (*You Only Look Once*), a great deep learning framework, is applied in the obstacle detection algorithm in our simulator for real-time processing. YOLO is regarded fast and accurate because it only "looks" the image once to identify the bounding box and category of the target object; thus transferring the detection problem to a regression problem [15].

The training uses the sum of squared error loss, then it predicts an objectness value for each bounding box using logistic regression [15]. The application of YOLO in our system indicates that our simulator is a good platform to test and support other algorithms.

The result of implementing YOLO in our simulator is shown in Fig. 7.

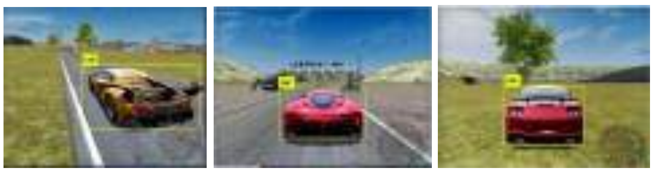
Fig. 7. Obstacle detection for the car by implementing YOLO in our simulator.

V. Conclusion and Future Work

Our achievement is successfully creating a simulator in automatic vehicles area as a pioneer. It is a hybrid, cross 3D and robust platform, orienting to both companies, researchers, and any other users who are interested in this field. This simulator can achieve the data generation, balancing data, training, and testing by using a *Tensorflow* framework in machine learning.

Additionally, various algorithms are able to be applied in our system, such as algorithms for lane detection and obstacle detection; hence, we provide a real-time rather than off-line platform to train their models and test their algorithms with many conveniences in the field of automatic vehicles. Besides, we also define an evaluation criterion on the approaches and performance in this field. Hopefully, our work could readily help more researchers to test and evaluate other similar works.

The results of this study are promising, yet there is room for more improvements. One of the identified issues is that multi-camera technology is not used in the current version of this simulator. Moreover, we also plan to add the object tracking technology in our work in the future to identify not only a category of "car" but also any particular vehicle, making it much closer to the real world. These are challenging tasks for ameliorating our simulation as for the future work.

REFERENCES

[1] C. Ajmi, S. E. Ferchichi, and K. Laabidi. New procedure for weld defect detection based-gabor filter. pages 11–16, 2018.

[2] M. Bertozzi and A. Broggi. Gold: a parallel real-time stereo vision system for generic obstacle and lane detection. IEEE Transactions on Image Processing, 7(1):62–81, 1998.

[3] F. Bounini, D. Gingras, V. Lapointe, and H. Pollart. Autonomous vehicle and real time road lanes detection and tracking. In 2015 IEEE Vehicle Power and Propulsion Conference (VPPC), pages 1–6, 2015.

[4] A. Dairi, F. Harrou, Y. Sun, and M. Senouci. Obstacle detection for intelligent transportation systems using deep stacked autoencoder and k -nearest neighbor scheme. IEEE Sensors Journal, 18(12):5122– 5132, 2018.

[5] EliteDataScience.https://elitedatascience.com/overfitting-in-machine-learning-how-to-prevent. Elite- DataScience.com, 2016-2018.

[6] Erico Guizzo. How google's self-driving car works. https://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works/, 2011.

[7] Q. P. He and J. Wang. Fault detection using the k-nearest neighbor rule for semiconductor manufac- turing processes. In IEEE Trans. Semicond. Manuf., volume 20, pages 345–354, 2007.

[8] O. O. Khalifa, A. H. A. Hashim, and A. A. M. Assidiq. Vision-based lane detection for autonomous artificial intelligent vehicles. In 2009 IEEE International Conference on Semantic Computing, pages 636–641, 2009.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolu- tional neural networks. 2012.

[10] P. Li, Y. Mi, C. He, and Y. Li. Detection and discrimination of obstacles to vehicle environment under convolutional neural networks. pages 337–341, 2018.

[11] X. Li, Q. Wu, Y. Kou, L. Hou, and H. Yang. Lane detection based on spiking neural network and hough transform. pages 626–630, 2015.

[12] Z. Q. Li, H. M. Ma, and Z. Y. Liu. Road lane detection with gabor filters. In 2016 International Conference on Information System and Artificial Intelligence (ISAI), pages 436–440, 2016.

[13] B. T. Nugraha, S. F. Su, and Fahmizal. Towards self-driving car using convolutional neural network and road lane detector. In 2017 2nd International Conference on Automation, Cognitive Science, Optics, Micro Electro- 173;Mechanical System, and Information Technology (ICACOMIT), pages 65– 69, 2017.

[14] D. Pritam and J. H. Dewan. Detection of fire using image processing techniques with luv color space. pages 1158–1162, 2017.

[15] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. CoRR, abs/1804.02767, 2018.

[16] L. Zhang, Q. Li, M. Li, Q. Mao, and A. Nüchter. Multiple vehicle-like target tracking based on the velodyne lidar, volume 8. 2013.

[17] Dr. Jan Becker. Toward Fully Automated Driving. https://higherlogicdownload.s3.amazonaws.com/AUVSI/3a47c2f1-97a8-4fb7-8a39-56cba0733145/UploadedImages/documents/pdfs/7-15-14%20AVS%20presentations/Jan%20Becker.pdf, 2014