Creating 3D Virtual Driving Environments for Simulation-Aided Development of Autonomous Driving and Active Safety

Arvind Jayaraman, Ashley Micks, Ethan Gross
MathWorks, Ford Motor Company

Abstract

Recreating traffic scenarios for testing autonomous driving in the real world requires significant time, resources and expense, and can present a safety risk if hazardous scenarios are tested. Using a 3D virtual environment to enable testing of many of these traffic scenarios on the desktop or cluster significantly reduces the amount of required road tests. In order to facilitate the development of perception and control algorithms for level 4 autonomy, a shared memory interface between MATLAB, Simulink, and Unreal Engine 4 can send information (such as vehicle control signals) back to the virtual environment. The shared memory interface conveys arbitrary numerical data, RGB image data, and point cloud data for the simulation of LiDAR sensors. The interface consists of a plugin for Unreal Engine, which contains the necessary read/write functions, and a beta toolbox for MATLAB, capable of reading and writing from the same shared memory locations specified in Unreal Engine, MATLAB, and Simulink. The LiDAR sensor model was tested by generating point clouds with beam patterns that mimic Velodyne HDL-32E (32 beam) sensors and is demonstrated to run at sufficient frame rates for real-time computations by leveraging the Graphics Processing Unit (GPU).

Introduction

As the automotive industry progresses toward autonomy, the need for simulation-based development and validation increases, as does the need for greater detail and volume in simulations. Full autonomy requires an unprecedented amount of trust placed in the vehicle's systems to safely handle a broad range of scenarios, and such trust requires extensive testing. Estimates are on the order of 100 million km and several hundred million euros for validation of autonomous systems using road tests alone [1]. These estimates, along with the dangers associated with testing specific scenarios, further motivates the use of simulation.

The systems to be simulated also go beyond vehicle dynamics alone, requiring sensor models in the loop with perception and control algorithms, to test all aspects of an autonomous vehicle or driver assist system. This includes the generation of synthetic camera data at the RGB level, synthetic LiDAR point clouds, and synthetic radar data, though this work focuses on the first two. The ability to annotate this data with ground truth information is also necessary on a large scale, both for use in evaluating the performance of perception systems, and for machine learning systems.

Methodology

This paper focuses on an integration of capabilities existing within a commercially available gaming engine and the MathWorks toolchain, using a shared memory interface to enable co-simulation between the two tools. It establishes a complete workflow for the simulation of vehicle perception systems in a 3D driving environment. The application of these capabilities can be extended to simulate and test control systems as well, since the exchange of information over shared memory is equally possible in both directions between the virtual environment and the algorithms to be tested.

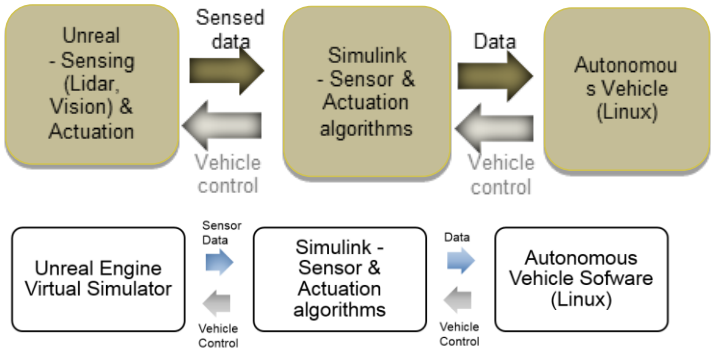

Figure 1. Block diagram of the simulator setup.

Software Tools Used

This paper focuses on the use of Unreal Engine 4 to create 3D virtual driving environments, and the MathWorks toolchain as the environment in which data is processed. Unreal Engine is a free, open source video game engine. As such, it was possible to build new functionality within this tool to support the export of data to MATLAB over shared memory, and to acquire data about the 3D environment necessary to generate synthetic camera and LiDAR data. The gaming engine provides an efficient means of creating 3D environments and dynamic scenarios through a drag and drop interface; it provides the option of either C++ scripting or the visual scripting language, Blueprints. Environments created in this method can achieve a realistic, detailed appearance and geometry for the generation of synthetic camera and LiDAR data. Meanwhile, the MathWorks toolchain can process this data and prototype perception

and control algorithms using MATLAB and Simulink. This includes toolboxes for computer vision, image processing, and LiDAR point cloud processing; therefore enabling an interface between the two software tools and allowing a full pipeline to be tested that includes these capabilities.

Shared Memory Interface

In order to exchange numerical data between MATLAB and Unreal Engine, a shared memory interface was developed. Shared memory allows multiple processes running on a single machine to simultaneously access the same location in memory. In order to access shared memory using a string identifier, a library of corresponding read/write functions were developed for both Unreal Engine and MATLAB. The utilities are provided in the form of an Unreal Engine plugin and MATLAB system objects, which provide a simplified interface for end-users to share arbitrary data between the two processes. The use of a MATLAB system object allows users to provide a common implementation for accessing shared memory in both MATLAB and Simulink. Using this shared memory interface, users can program the virtual simulator in Unreal Engine to send numerical data to MATLAB, wait for MATLAB to execute algorithms with this data, and finally read data within the simulator; thus facilitating full bi-directional communication for the simulation and visualization of dynamic systems.

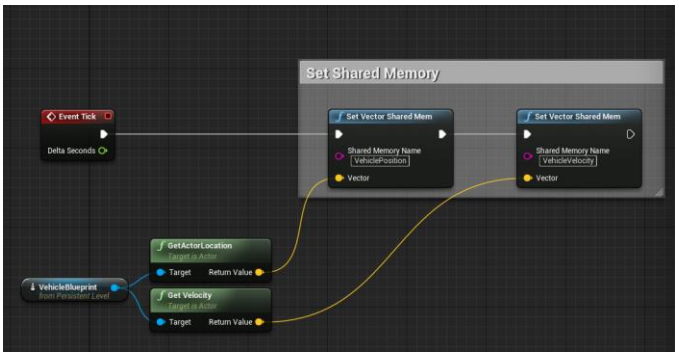

Figure 2. Write the vehicle position and velocity within the Unreal Engine to shared memory.

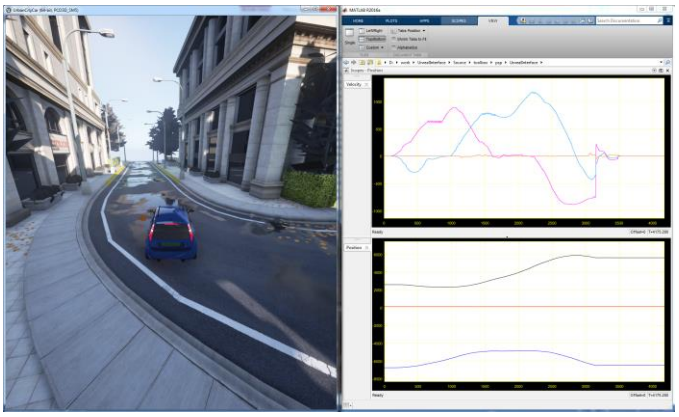

Figure 3. Visualizing the vehicle velocity and position transmitted from the virtual simulation environment in MATLAB.

Alternate Communication Methods Explored

The choice of using shared memory as the communication interface was necessitated by the need to efficiently exchange large amounts of data, such as high definition images from virtual cameras and distance measurements from virtual LiDAR sensors within the Unreal Engine project. With design iteration, two other popular data exchange methods, UDP and TCP/IP, were also explored. However, both methods were much slower in comparison to using shared memory. The shared-memory interface was the only interface that was fast enough to facilitate data-exchange rates that allowed the Unreal project to run at 30 FPS or faster while still being able to process camera and LiDAR data in MATLAB. Note that TCP/IP and UDP communication are still useful in cases where MATLAB and the Unreal Engine project is not running processes on the same machine.

Virtual Camera Setup

The Unreal Engine-based virtual simulator platform provides photo-realistic images which can be used to facilitate prototyping computer vision algorithms in MATLAB; these extract useful information such as vehicles, pedestrians, lanes etc. from images. To facilitate the workflow, we set up a virtual camera sensor in Unreal Engine using a Scene Capture 2D camera actor. The Scene Capture 2D actor is available with Unreal Engine and can be placed anywhere in the virtual driving environment. The images rendered by this actor are transmitted via the shared memory interface to MATLAB. The horizontal field of view angle, as well as image size and resolution, are adjustable in the Scene Capture 2D component. In addition, it is also feasible to add arbitrary post processing effects to the camera in order to model lens distortion effects often present in actual camera sensors. Image disturbances can also be introduced after transmission of the image to MATLAB. To facilitate operating on these images in MATLAB, the rendered images are transposed and the image format is changed; the memory layout of the image is column-major, with separate image planes for each RGB color channel.

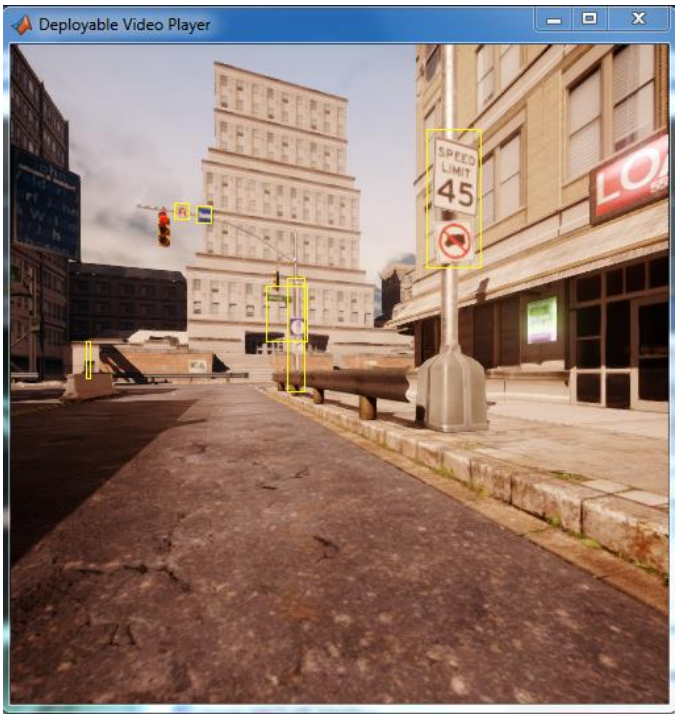

Figure 4. Example of the shared memory interface for sending images to MATLAB, alongside ground truth information to identify bounding boxes for objects of interest.

Virtual LiDAR Data Generation

For autonomous vehicle navigation, LiDAR is an important sensor that provides accurate distance measurements and reflectivity of objects. A virtual LiDAR sensor model that transmits distances and reflectivity of objects in the 3D scene to MATLAB is provided as part of the package. Below we describe the details of the sensor model.

Modeling Virtual LiDAR Distance Measurements

To model the sensor's distance measurements to objects in the scene, we repurposed a Scene Capture 2D camera's rendering pipeline in Unreal Engine, to output a texture image where each pixel location in the image contains the distance of the visible surface from the sensor. The depth image textures are then sampled at predetermined locations which correspond to the LiDAR's scanning pattern. This scanning pattern is fixed and can be determined once at initialization using the horizontal and vertical angular resolution that are part of the parameters of the sensor which can be adjusted in the model. In practice, we found that a Scene Capture 2D camera is only capable of reliably rendering scenes within a 120-degree field of view. To model a LiDAR sensor that covers a full 360-degree field of view, we set up three Scene Capture 2D cameras and combined the depth images from each camera. These components are brought together into a single 360 degree LiDAR actor, which is provided as part of the plugin.

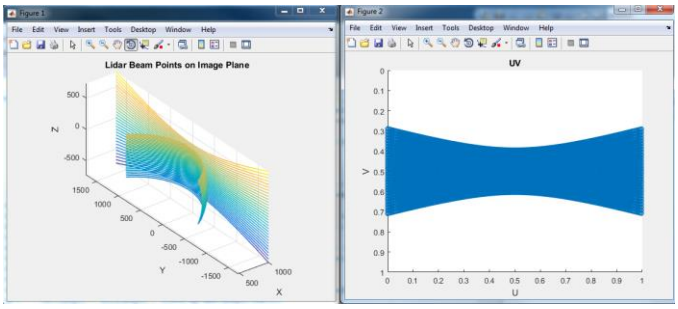

Figure 5. The figure shows the scanning pattern of the LiDAR sensor model in 3D and the image pixel locations at which we sample a depth image in order to model the beam pattern traced by the sensor. This plot corresponds to a sensor with horizontal resolution of 0.16 degrees, vertical resolution of 1.33 degrees, vertical field of view of 41.3 degrees, and horizontal field of view of 120 degrees.

Modeling Surface Reflectivity of LiDAR Sensors

Reflectivity of each beam is important to model as it may be used by the autonomous vehicle software. For example, surface reflectivity can be used to distinguish lane markings from asphalt since lane markings tend to possess higher reflectivity than asphalt in the LiDAR spectrum. However, reflectivity is also influenced by other factors, such as the direction of the beam relative to the surface it is reflected from, the properties of the surface's material, and the distance the LiDAR beam travels.

We model reflectivity of the surface to a LiDAR beam with the empirical Phong shading model [2] as the sum of diffuse and specular reflections of a surface. The key difference between LiDAR and the visible light electromagnetic spectrum is that the diffuse and specular reflection coefficients of each material need to be specified as properties of the material for the LiDAR's wavelength. The reflectivity is given by the sum of the diffuse and specular reflections. Our model assumes that transmitting and receiving elements of the LiDAR sensor are co-located at the virtual LiDAR sensor's position and that there is no ambient energy in the LiDAR spectrum.

$$R_{diffuse} = K_{diffuse} * (\hat{P} - \hat{C}).\ \hat{N}$$

$$R_{specular} = K_{specular} * \left(2\left(\hat{P}.\hat{C}\right)\hat{N} - \hat{C}\right).\hat{C}\ )^{\alpha}$$

$$R_{total} = R_{diffuse} + R_{specular}$$

Where,

$R_{diffuse}$= diffuse reflection

$R_{specular}$= specular reflection

$K_{diffuse}$ = diffuse reflection coefficient of the reflecting surface

$K_{specular}$ = specular reflection coefficient of the reflecting surface

$\hat{N}$ = the normalized surface normal vector of the reflecting surface at a given pixel location

α = the specular exponent of the surface

$\hat{P}$ = the normalized position vector of the reflecting surface

$\hat{C}$ = normalized position vector of the LiDAR sensor

R_{total} is the reflectivity of the surface

Due to time constraints, we could not find the reflectivity coefficients for typical materials such as buildings, foliage, asphalt, and cars based on observed values from an actual sensor for these materials. We approximated the diffuse reflection coefficient in the LiDAR spectrum by using the grayscale value of the material color in the visible light spectrum. The specular reflection coefficient of the materials for the LiDAR spectrum was assumed to be the same as that of the visible light spectrum. Our model also does not account for the decrease in reflectivity with increasing distances to surface, due to loss of energy when propagating through a medium. However, this information can be incorporated into the model once the reflectivity values are more accurately compared to that of an actual LiDAR sensor. Also, note that some real world LiDAR sensors rotate around their vertical axis to capture the full 360-degree horizontal field of view, while our model captures a full 360-degree field of view instantaneously in the virtual environment without any physical rotation.

The mathematical model for the LiDAR sensor was initially prototyped in MATLAB. However, in order to support running the virtual sensor model in real time, all computations were later redesigned to run on the GPU as part of the Unreal Engine plugin. The distance and reflectivity data obtained was then transmitted to MATLAB via the shared memory interface described above.

In practice, we found that transmitting the distances as floating point values in a Cartesian 3D coordinate system increases the amount of data that needs to be exchanged via the shared memory interface, thus hindering simulation speed. To avoid this issue, we quantized the distance and reflectivity output and encoded them into a 32-bit integer word before transmitting it to MATLAB. In MATLAB, the values are decoded and can be either used directly in spherical coordinates or converted back to a point cloud in Cartesian coordinates (by multiplying the distances for each beam with the corresponding unit vector along the beam direction).

Alternative Methods to Model LiDAR Sensors

Another technique explored was modeling the LiDAR sensor using ray tracing, which is a method that is already provided as part of the Unreal Engine programming environment. We found that performing ray tracing for a large number of LiDAR beams using this method is computationally very expensive. Furthermore, ray tracing capabilities that are available to programmers only run on the CPU, which causes the simulator to run slow (at 1-2 FPS on a Lenovo S20 workstation with an NVIDIA GTX 980 graphics card). Because real-time interaction with other actors in the virtual driving simulator is important, this method was not suitable for our application. We did not pursue using multi-threaded implementation of the ray tracing algorithm since the alternative technique described in our article met the needs.

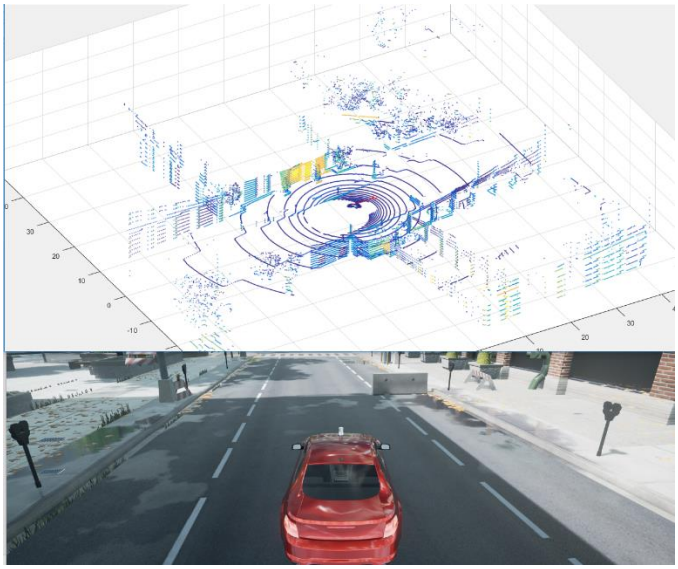

Figure 6. The figure shows the 3D scene and the corresponding LiDAR output viewed in MATLAB as a point cloud. This example shows a 360-degree field of view. Vertical and horizontal field of view and beam spacing are adjustable parameters. The reflectivity of each point is shown using a color map that varies from blue for a reflectivity of 0, to green for a reflectivity of 1.

Sharing LiDAR Sensor Data with Autonomous Vehicle Software

Ford's application of the MathWorks toolchain supports validation of autonomous driving software running in Ubuntu Linux. Multiple methods can be used to integrate the virtual LiDAR sensor data into the autonomous vehicle software. In our testing, we explored two methods: data transmission via UDP packets and direct binary log construction. The following is an overview and analysis of these two methods.

For an initial proof of concept, we decided to use a Windows 10 laptop (Razer Blade 2016) to run Unreal Engine, MathWorks Unreal Engine plugin prototype, and MATLAB in conjunction with an Ubuntu 14.04 laptop (Dell Precision M6800) running Ford's autonomous vehicle code. We connected both laptops via an Ethernet connection, configured a MATLAB script to send UDP packets, and configured the Ubuntu laptop to listen to the port we had established through MATLAB. Once this connection was made, we needed to precondition the raw sensor data received via shared memory from Unreal Engine by formatting the data correctly to meet the specs for a Velodyne HDL-32E LiDAR. This included:

- Creating a data structure of correct dimensions and data type
- Quantizing distance data to the nearest value set by spec (in this case, 2mm range resolution)
- Setting default values for out-of-spec data points (negative and extreme values)
- Looping continuously through the data to create and send UDP packets using MATLAB's UDP functionality

This setup allowed our Windows laptop to act as a LiDAR sensor, since it was connected via Ethernet and provided the exact same UDP

format our Velodyne LiDAR sensors use in autonomous vehicle navigation. This setup also took away the need for any post-processing code and allowed our AV code to read in the data directly. It is also possible to run both the Unreal Engine simulation and autonomous vehicle code together on the same Linux machine, in which case data packets can be generated in a similar manner, except without the need for a physical Ethernet connection.

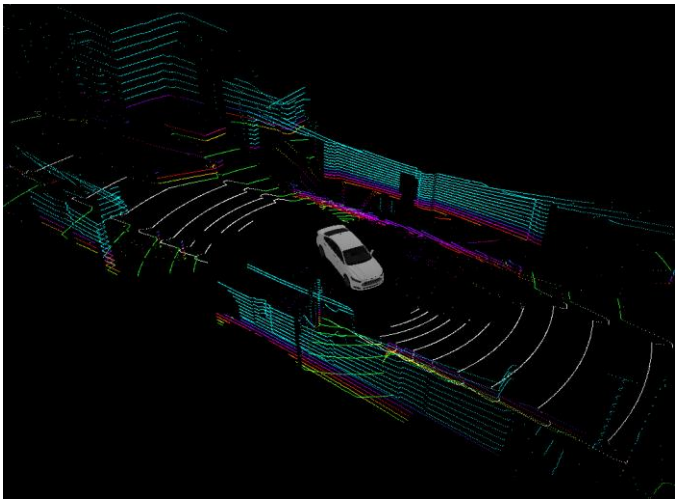

Figure 7. Synthetic LiDAR data that was transmitted from Unreal Engine to a viewer that exists alongside Ford's autonomous vehicle code.

For this proof of concept, the code was not yet optimized for speed. The data speed averaged 300 packets sent/sec. Because each UDP packet contains a set of 12 firing data sets, this is equivalent to 3600 sets per second. This data signifies we received approximately less than 1.5 revolutions per second. Our spec required 10 revolutions per second, meaning this method was too slow for real-time simulation on the laptop hardware used. Further optimizations could be made to reach our desired revolutions/sec on such hardware, but it was decided to achieve useful results in the meantime by creating logs that Ford's autonomous vehicle code could read, showing a scenario that was previously simulated. This leads to our second use case: binary log construction.

For binary log construction, we preconditioned the data in the same manner as we did for the interactive use case, where the autonomous vehicle code was in the loop. The only difference is that data was sent to a log rather than directly to the autonomous vehicle code. Before we preconditioned the data, we stored the raw sensor data directly into a binary data file to prevent slowdown for our log construction procedure. After creating the file, we created a parser that would parse this file and format all data as was described with the UDP packet process above. Once we created 12 sets of 32 firings with their associated rotation angle and a Velodyne data start identifier, we added the remaining information required by the struct that our autonomous vehicle code uses for LiDAR data packets. This included spoofing extra timing data that was included in our struct so the struct would be complete for the communication protocol and logging framework to read. Once the struct format was created, we constructed another layer to wrap struct information into the communication protocol format. One advantage of this use case was the ability to manipulate the log reading speed to run faster or slower than real time if desired. Finally, the data was stored into a binary data file as repeated messages, each containing their proper header information, struct information, and LiDAR data information. This log was then transferred (if necessary) to the machine running the autonomous vehicle code, which then read and played the log back.

Creating the logs in this manner allowed us to run faster, slower, or exactly at 10 revolutions/sec, by manipulating the timing to do so. This was a great advantage for resimulation; interactive simulation would be limited though to either slower than real time, or to running on computer clusters or other hardware more powerful than the machines used in this work.

Directions we intend to take to optimize the communication between Unreal Engine and AV code include:

- Revise the UDP packet processing code for higher performance.
- Unless the use case is to test the embedded hardware, avoid transmitting data between different machines; run both the sensor model simulation and the AV code on a single Linux machine or cluster.
- Bypass UDP by outputting data directly to a format used downstream of that by the AV code.

Once the LiDAR data is read by the AV code, it can then be used for obstacle detection, lane marking detection, localization, and other applications.

Future Work

With the release of the initial sensor models in the beta version of MathWorks interface to Unreal Engine, we developed proof of concept capabilities to model camera and LiDAR sensors. We plan to extend this capability to also model a radar sensor using a similar method to compute the surface reflection.

As part of future work on the camera sensor, the current model needs to be extended to offer additional sensor parameters that can be configured to match the settings of real world camera sensors. For instance, we need the ability to specify intrinsic camera parameters, such as focal length and principal point. We also plan to add fisheye and wide-angle lens distortion parameters. In addition, we intend to collect ground truth for objects viewed by the camera for use in the training of machine learning algorithms, with applications such as object detection and classification, or image segmentation.

For future work on the LiDAR sensor, we plan to compare the reflectivity values with that of an actual LiDAR sensor, which will aid in completing the empirical model used to calculate reflectivity. After estimating the values of the reflection coefficients of common materials in the LiDAR spectrum, we can enhance the tool to allow users to specify these properties for any material used in Unreal Engine. We also plan to model an alternative version that can take into account the physical rotation of the sensor about a vertical axis.

Summary/Conclusions

Overall, a workflow was successfully established and tested that provides an interface between a 3D virtual driving environment and vehicle perception systems related to autonomy or active safety. This

virtual environment was shown to be capable of generating a synthetic camera and LiDAR data that resembles data from real sensors, and is capable of communicating bidirectionally, via shared memory with algorithms in development.

References

1. Bartels, Arne. (2013, July 17). Online Services & Testing for High Automated Driving Functions. In *Testing, certification, and licensing*. Symposium conducted at the meeting of the Transportation Research Board, Stanford, CA.
2. B. T. Phong, Illumination for computer generated pictures, *Communications of ACM* 18 (1975), no. 6, 311–317.

Contact Information

Ashley Micks, 3200 Hillview Ave, Palo Alto 94304
Email: amicks2@ford.com, Ph: 650-646-6816

Arvind Jayaraman, 39555 Orchard Hill Place, Suite 280, Novi, MI 48375
Email: ajayaram@mathworks.com, Ph: 248-675-3302

Ethan Gross, 20000 Rotunda Dr, Dearborn, MI 48124
Email: egross16@ford.com, Ph: 313-410-6165

Definitions/Abbreviations

LiDAR	light detection and ranging
UDP	user datagram protocol
TCP/IP	transmission control protocol/internet protocol
GPU	Graphics Processing Unit
AV	autonomous vehicle
RGB	Red Green Blue (image)