Open-Source CiThruS Simulation Environment for Real-Time 360-Degree Traffic Imaging

Teo Niemirepo, Juuso Toivonen, Marko Viitanen and Jarno Vanne
Tampere University
Tampere, Finland
{teo.niemirepo, juuso.toivonen, marko.viitanen, jarno.vanne}@tuni.fi

Abstract — **This paper presents an open-source simulation environment for 360-degree traffic imaging. The environment is built on the openly available AirSim Windridge City Asset. In this work, the city is populated with custom autonomous vehicles and pedestrians. The vehicles navigate along a designed node map that can be manually placed on the roads according to the specified traffic regulations. The vehicles are also made to detect other vehicles, pedestrians, and traffic lights for simple collision avoidance and smoother traffic flows in intersections. The pedestrians follow a NavMesh placed on the walkable areas and stop at the traffic lights when crossing the streets. Weather effects, time-of-day, and rain distortion lens shader bring the environment more close to the reality. The whole system is built on top of free and self-made assets, making it easy to use, configure, and extend. The performance of the simulator exceeds 60 frames per second when run on NVIDIA RTX 2070 with Intel Xeon E5-2620 or equivalent hardware.**

Keywords— Open-source software, traffic simulator, scene population, traffic imaging

I. Introduction

Advanced driver-assistance systems (*ADAS*) have been an increasingly popular research topic in recent years [1], [2]. These systems are being developed to improve vehicle safety, driver behavior, and driving experience. Particularly, a cumulative number of modern vehicles contain video-based ADAS to detect other vehicles, pedestrians, and surrounding obstacles. The next-generation systems are also able to leverage *vehicle-to-everything* (*V2X*) communication to provide vehicles with more visual data.

In vision-based ADAS development, the simulation environment is a must-have for testing different parameter settings and verifying the operation before actual implementation. However, the existing traffic imaging simulators are either expensive or lacking in usability or modifiability, especially when specific camera positioning and image distortions are of interest.

This paper presents a tailor-made, open-source *See-Through Sight* (*CiThruS*) simulation environment for 360-degree traffic imaging. It is designed to facilitate the development of vision-based ADAS for next-generation vehicles. Fig. 1 depicts a snapshot of the open Windridge City Asset [3] which is used as a basis for our work. The asset is available in the Unity Asset Store and the 3D models for the vehicles are obtained from [4]-[7]. The environment is built in Unity.

The original Windridge City scene is void of life and it only has a clear day weather. The design goal was to add photorealistic dynamic features to the city in order to make camera-capture simulation look more realistic. The implemented features include self-driving vehicles, pedestrians, various weather effects, and different time-of-day lightings. Since our interest is not in the traffic simulation, the autonomously moving vehicles and pedestrians are made to follow predefined routes for the sake of lower complexity. However, this customizable and extendable setup still enables us to address wide range of use cases, from ideal to non-ideal traffic imaging conditions. The presented solution is available at **https://github.com/ultravideo/CiThruS-simulation-environment/**.

Fig. 1. Windridge City Asset [3].

This paper is organized as follows. Section II gives an overview of the previous work. Section III takes an in-depth look at the vehicles and pedestrians implemented in the scene. Section IV describes the environment effects. Section V explains the lighting features and Section VI concludes the paper.

II. Related Work

To the best of our knowledge, there are no ready-made, open-source, and easy-to-use simulation environments for 360-degree traffic imaging in the prior art. The existing open-source traffic simulators are lacking in realistic graphics like *SIRCA* (*SImulador Reactivo de Conduccion de Autómoviles*) [8], exclude other vehicles or pedestrians like TORCS (*The Open Racing Car Simulator*) [9], or include a lot of additional features not necessarily needed in traffic imaging [10].

The closest approach to ours is *CARLA* (*Car Learning to Act*) [10] open-source simulator for autonomous driving systems. CARLA is built on compute-intensive machine-learning algorithms and it uses highly detailed vehicle models

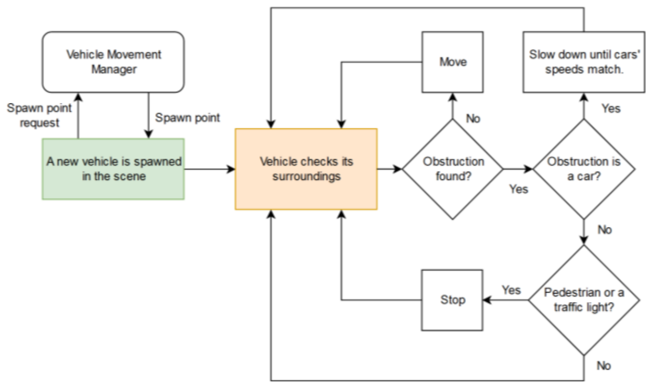
Vehicle Movement Manager
Spawn point request
Spawn point
A new vehicle is spawned in the scene
Vehicle checks its surroundings
Move
No
Obstruction found?
Yes
Slow down until cars' speeds match.
Yes
Obstruction is a car?
No
Stop
Yes
Pedestrian or a traffic light?
No

Fig. 2. AI state machine for vehicle control in the scene.

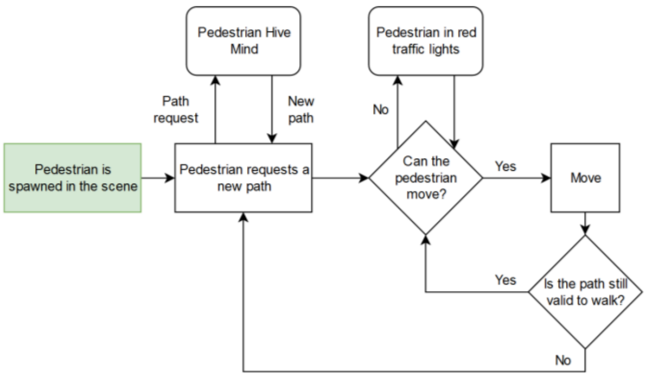
Pedestrian Hive Mind
Path request
New path
Pedestrian in red traffic lights
No
Pedestrian is spawned in the scene
Pedestrian requests a new path
Can the pedestrian move?
Yes
Move
Yes
Is the path still valid to walk?
No

Fig. 3. AI state machine for pedestrian control in the scene.

to simulate traffic as realistically as possible. However, this approach adds multiple layers of complexity and makes the simulator fall behind real-time performance.

Unlike CARLA, our simulator is able to attain real-time performance on high-end consumer-grade *graphics processing unit* (*GPUs*). Our approach is also more scalable, meaning that the number of vehicles and pedestrians can be practically selected freely without making any major performance compromises.

III. Operating Principles of Vehicles and Pedestrians

Fig. 2 describes the *artificial intelligence* (*AI*) state machine implemented for vehicle control in the scene. The goal was not to create a complete simulation model of a real-life city, but a more straightforward and light-weight control scheme is enough for the vehicle control.

A. Collision Prevention

The vehicles use an optimized array of Unity 3D built-in raycasts to monitor their surroundings. These rays reach the distance of two meters at maximum depending on the car type. An individual car uses a total of eight rays to check for other vehicles and an additional spherical sweep of rays for pedestrians, totaling to a couple of hundred of rays. This check is performed once per frame for every vehicle, which for this many raycasts is inefficient and is optimized by factoring the scene into different physics layers. This allows the raycasts only to check against specific objects in the scene and ignore the rest. The raycasts can collide only with vehicles and pedestrians.

Should any of the rays hit an obstruction in the way of the car, an appropriate action will be taken. If a ray hits another vehicle, first the speed of the car hit with the ray will be checked. If the speed is lower than the speed of the car casting the ray, but a non-zero value, the current car will begin to slow down. Since this is checked once per frame, there will come a point of time when the car casting the ray is going slower than the car in front. After a while, the current car will have slowed down enough so that none of the rays will hit anything and the car can begin to speed up again. If the next car is still going slower than the current car, the process begins anew.

The cars always try to prevent collisions with a car or a pedestrian. This means if a ray hits a stopped car the current car will be stopped as well in order to prevent collision.

The cars check for pedestrians by using a spherical sweep. In essence, a spherical sweep is an empty volume which is projected in front of the car. If the volume contains any pedestrians, a hit is registered and appropriate actions are taken according to Fig. 2.

Pedestrians themselves comply with the control scheme described in Fig. 3. It is even more light-weight and does not take into account anything except traffic lights. It is up to the cars to prevent collisions with anything moving in the scene.

B. Pathfinding

Fig. 4 depicts the node network the vehicles use to navigate the scene. Using state-of-the-art approaches, like machine learning and machine vision, would deteriorate the simulator performance. Instead, building a custom solution and manually placing nodes on the roads beforehand keeps the frame rate high.

The nodes are directional and enforce the possible turns a car can take. This eliminates unnecessary conflicts such as car driving on the wrong lane or turning in an illegal direction. For the rest, a full path control is left for the developer.

A node based approach is also able to detect allowed paths and speed limits without heavy algorithms. Should the need arise, speed limits could be programmed directly into the individual nodes. In that case, there would only exists a global speed limit to avoid any unnecessary complexity.

The pedestrians operate on a Unity built-in “NavMesh” based solution exemplified in Fig. 5. They are controlled by a script called the *Pedestrian Hive Mind*. It is able to command a huge number of individual humans. The *Hive Mind* calculates appropriate paths for the pedestrians and broadcasts them to all pedestrians at once. In addition, a single pedestrian can also request an individual path.

This kind of data -oriented approach has the benefit of managing many sub-objects efficiently and not generating a lot of garbage, which is customary for managed programming languages such as C# used in this project.

C. Integrating Vehicles with a Virtual Camera System

Cameras can be added inside vehicles for first person view and other possible configurations as illustrated in Fig. 6.

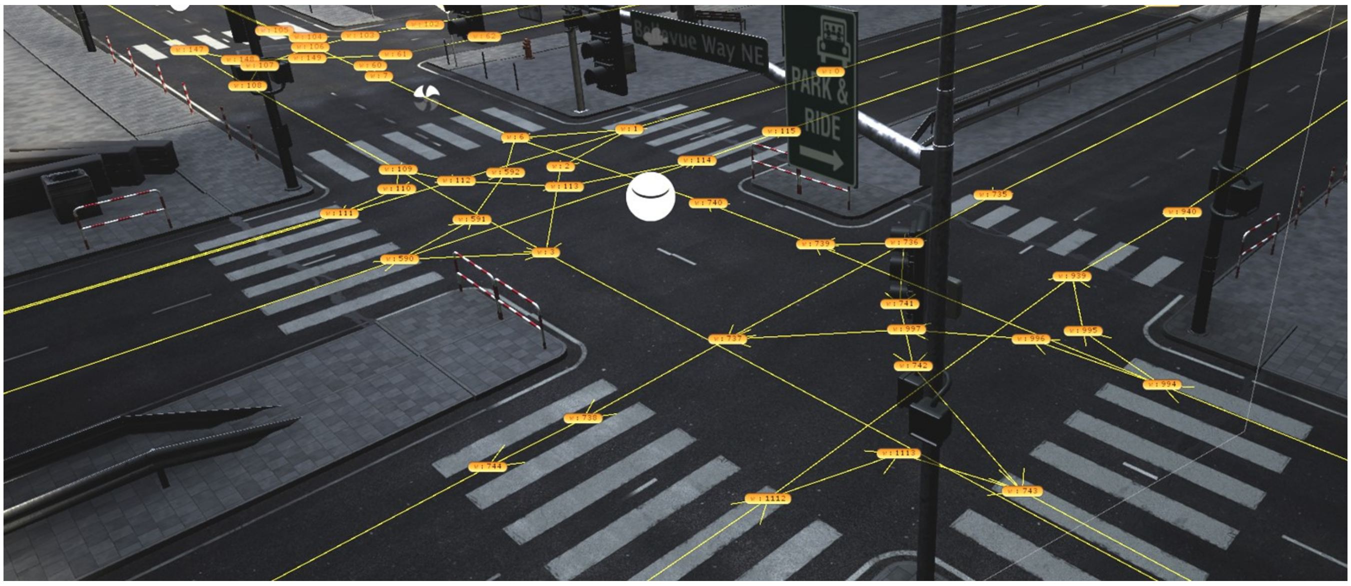

Fig. 4. Node network for vehicles.

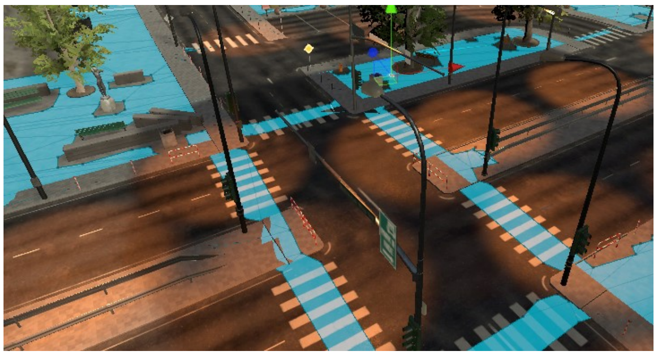
Fig. 5. NavMesh for pedestrians (blue regions).

Fig. 6. First person view from a car.

D. *Models for Vehicles and Pedestrians*

The 3D models for the cars were obtained from the Unity Asset Store [4]-[6] and the model for the truck from free3d.com [7].

The pedestrians were created using an open-source tool Make Human [11] for 3D characters. The animations were obtained from an online rigging tool and animation database called Mixamo [12]. Due to incompatibility between the animations and the models, some of the animations are not realistic enough at closer inspection. Nevertheless, this mismatch does not change the functionality of the scene so tweaking the animations to fit the models better was not considered important.

E. *Route Editor*

A route editor was implemented for editing existing vehicle routes and adding routes to new maps. The route editor was implemented as drag-and-drop Unity prefab, which makes it easy to include it in a new scene. The route editor has a brush mode for creating new paths (Fig. 4) and a deletion mode to remove paths. It writes the path data to disk.

IV. Environmental Effects

A simulation world with perfect cameras and always sunny weather is not in line with reality so it fails to offer interesting test cases for traffic imaging. We are especially interested in non-ideal conditions, e.g., test cases where there is water or dirt on the lens. For this purpose, different weather effects and times of day were implemented. These effects have no impact on the functionality of the vehicles or pedestrians in the scene.

A. *Daytime*

A Sunny daytime was the default setting in the Windridge City Asset. The setting was kept mostly as is, only the skybox was tweaked to better match with the other skyboxes used in different weather conditions. A skybox creates an illusion of the sky. It is a texture which is stretched across the box or sphere which indicates the scene bounds.

B. *Night Time*

In order to add a semi-realistic clear night to the scene, the skybox was changed to a starry sky and the fog color was tweaked to better suit the darker atmosphere. Street lights were also added to brighten up the scene and make it more realistic for a small city. The windows of the buildings are not illuminated for performance reasons but the cars turn on their headlights at night time.

C. *Weather Effects: Rain*

Rain was one of the most important weather effects to be added due to the visual distortions it causes. In addition to rain drops on lenses and car windshields, heavy rain also deteriorates visibility on multiple levels. This was simulated with a particle system where the amount of individual rain particles can easily be altered.

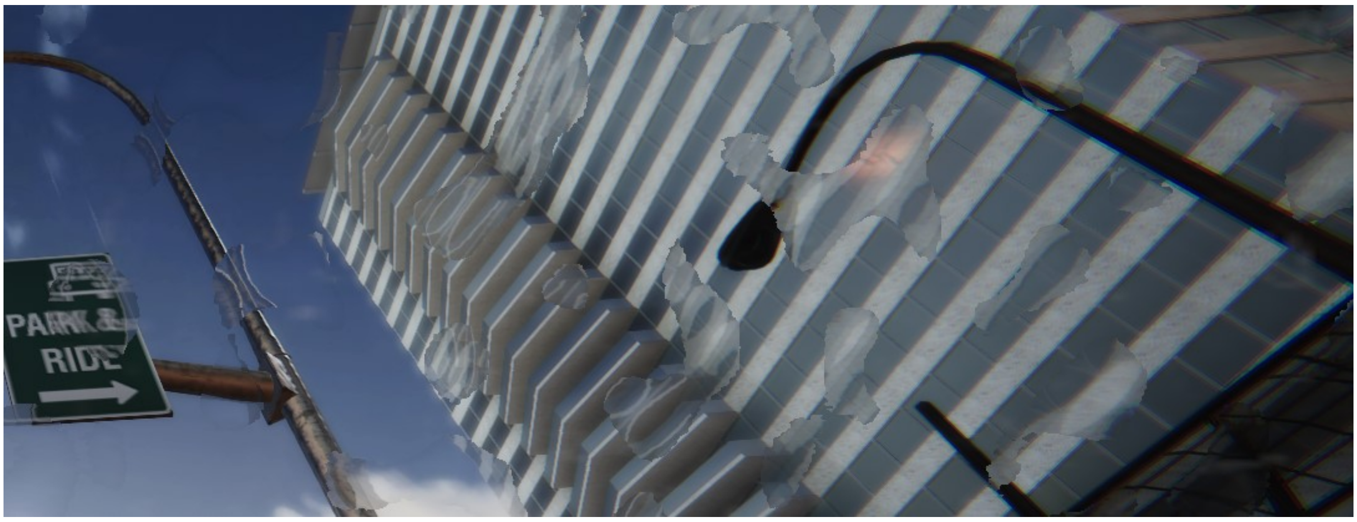

Fig. 7. Lens rain effect in the daytime.

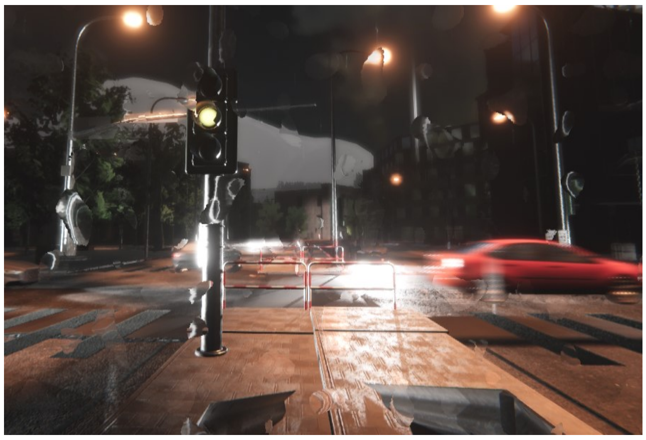

Fig. 8. Lens rain effect at night.

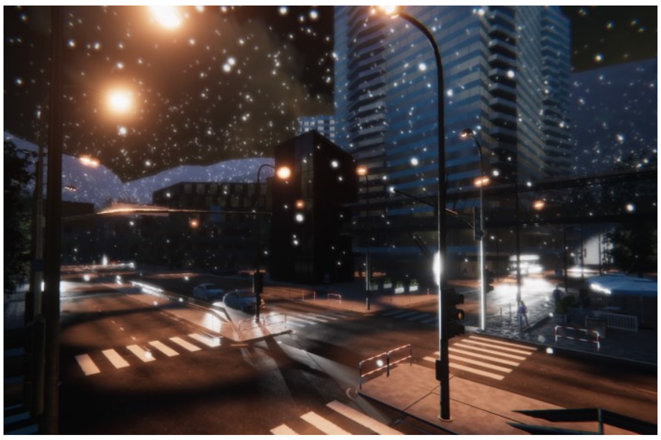

Fig. 9. Snow effect at night.

When creating the lens raindrop effect, visual distortion was emphasized in place of actual photorealism as shown in Fig. 7. An image-effect shader was used for this purpose. There are ready-made solutions available, but implementing a custom shader gives more control over the resulting effects. A tailored shader also provides easy access to and from our own controllers and other scripts.

The raindrops-on-lens shader was tweaked for the night – mode as shown in Fig. 8. It was observed that the effect was not pronounced enough during the night so it was made brighter by taking a weighted average of pixels under the effect and blending that with the original view.

D. *Weather Effects: Snow*

Heavy snowfall at night is one of the hardest driving conditions so implementing it was vital for our testing purposes. Fig. 9 depicts an example snapshot of the wintry weather.

The snow effect was implemented using Unity 3D particle system. To make it work better with multiple cameras, the effect was made global. This means that instead of a small particle system which follows the camera around, a single large particle system was used with hundreds of thousands of particles. This approach is not optimal performance wise but it is unlikely to be the bottleneck of the system. Furthermore, optimizing the effect is possible on demand.

The snow effect also alters the scene ambient lighting to better suit the atmosphere. A blue tint is applied during the day mode and a blue-gray tint during the night.

E. *Weather Effects: Fog*

Fog in the scene is implemented using Unity 3D “RenderSettings” and it can be changed at run time. Every weather effect makes use of the ability to change how the fog looks.

F. *Miscellanious Lens Effects: Dust and Dirt*

In the simulation scene, it is easy to work with a perfect camera and use it as a basis for testing, but it would not correspond to real-world testing conditions. This motivated us to add the ability to overlay dirt, dust, and scratches over the lens as shown in Fig. 10. These effects were achieved using an additional blending shader with a weighting factor that can be adjusted at run time.

V. Lighting

The system implements real-time lights instead of baked ones. This approach slightly sacrifices performance in favor of easy manipulation at run time, which is important for the day-night switching process and different weather effects.

For better performance, most lights do not cast shadows at all, do not contribute to the scene global illumination, and have limited range. This approach has a couple of drawbacks.

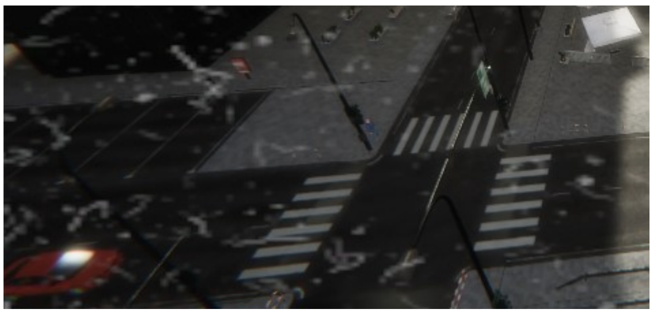

Fig. 10. Camera lens dust shader in use.

For example, passive street lights make the area surrounding them look unrealistic and dull. However, adjusting the global illumination settings for the night mode makes the scene look more realistic. For example, increasing the skybox's intensity multiplier helped to blend the scene better together.

Despite the aforementioned improvements, the environment still suffers from the "cheese-effect" illustrated in Fig. 11. That is, the scene looks like Swiss cheese, full of holes when viewed from above especially at night time. The effect looks quite strong when viewed directly, but it is not noticeable from the ground or inside a vehicle so the effect is not considered critical.

VI. Conclusions

This paper presented an open-source CiThruS simulation environment for 360-degree traffic imaging. We populated the open Windridge City Asset with vehicles and pedestrians that follow basic traffic rules to keep the traffic flow smooth. An easy-to-use route editor was also designed to allow users to adjust the node system for the vehicle path control. Furthermore, different weather conditions and times of day were added to meet diverse traffic imaging needs. Thanks to the low-complexity nature of our traffic controlling scheme, we were able to keep the simulation light-weight but still maintain high-quality graphics. Our simulator is able to sustain a stable frame rate of over 60 frames per second on high-end consumer-grade hardware, like NVIDIA RTX 2070 with Intel Xeon E5-2620 desktop computer or NVIDIA GTX 1060 with Intel Core i7-7700HQ laptop. The source files of the proposed simulator are available for download at https://github.com/ultravideo/CiThruS-simulation-environment/ with a permissive MIT license. Future plans for the system include adding LiDAR and other sensor simulations as well as camera rigging tools, for attaching cameras to the vehicles and stationary objects.

Acknowledgment

This work was supported in part by the European ECSEL project PRYSTINE (under the grant agreement 783190) and the Academy of Finland (decision no. 301820).

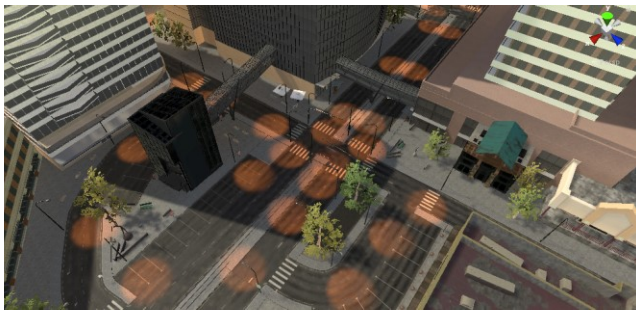

Fig. 11. Cheese-effect as viewed from the Unity editor.

References

[1] K. Bengler, K. Dietmayer, B. Farber, M. Maurer, C. Stiller, and H. Winner, "Three decades of driver assistance systems: review and future perspectives," *IEEE Intell. Transportation Syst. Mag.*, vol. 6, no. 4, Oct. 2014, pp. 6-22.

[2] S. Alvarez, Y. Page, U. Sander, F. Fahrenkrog, T. Helmer, O. Jung, T. Hermitte, M. Düering, S. Döering, and O. Op den Camp, "Prospective effectiveness assessment of ADAS and active safety systems via virtual simulation: a review of the current practices," *in Proc. Int. Tech. Conf. on the Enhanced Safety of Vehicles,* Detroit, Michigan, USA, June 2017.

[3] Windridge City Asset [online]. Available: https://assetstore.unity.com/packages/3d/environments/roadways/windridge-city-132222

[4] Low-poly Civilian Vehicle #5 [online]. Available: https://assetstore.unity.com/packages/3d/vehicles/land/low-poly-civilian-vehicle-5-124987

[5] 3D Low Poly Car For Games (Tocus) [online]. Available: https://assetstore.unity.com/packages/3d/vehicles/land/3d-low-poly-car-for-games-tocus-101652

[6] 4 Door Sport Car – Mobile [online]. Available: https://assetstore.unity.com/packages/3d/characters/4-door-sport-car-mobile-104177

[7] KamAZ 55111 3D Model [online]. Available: https://free3d.com/3d-model/kamaz-55111-28591.html

[8] S. Bayarri, M. Fernandez, and M. Perez, "Virtual reality for driving simulation - SIRCA," *Commun. ACM*, vol. 39, no. 5, May 1996, pp. 72-76.

[9] TORCS: The Open Racing Car Simulator [online]. Available: http://www.torcs.org

[10] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: an open urban driving simulator," *in Proc. Annual Conf. on Robot Learning*, Mountain View, California, USA, Nov. 2017.

[11] Make Human [online]. Available: http://www.makehumancommunity.org

[12] Adobe Mixamo [online]. Available: https://www.mixamo.com