Sensor simulation

Is AGXUnity a viable platform for adding synthetic sensors

Niklas Kallin

Niklas Kallin
VT 2018
Examensarbete, 15 hp
Supervisor: Eddie Wadbro
Extern Supervisors: Martin Servin, Anders Backman
Examiner: Pedher Johansson
Kandidatprogrammet i datavetenskap, 180 hp

Abstract

When developing algorithms for autonomous vehicles it is important to test several different scenarios many times. New and untested algorithms are prone to make errors which results in accidents. It is therefore preferred to use a simulation environment instead. Sensors used to determine the vehicle's position must then be modelled. This thesis answers the question whether adding sensor simulation to an existing simulation platform (AGXUnity) is viable, or if using other existing options is preferred. To reach the goal, a function-based sensor is developed and its accuracy tested. Its performance is determined by simulation in a standard scene. Tests showed that the sensor had acceptable accuracy and performance. The conclusion is that AGXUnity is a viable platform for sensor simulation.

Contents

1 Introduction

In the real world, many issues have to be considered when developing control systems for autonomous vehicles. Collisions with pedestrians can cause injury and running into objects could damage the vehicle. It can also be time consuming. Reproducing a specific scenario many times and with different traffic, objects, and weather scenarios is almost unfeasible [1]. The autonomous control of the vehicle is in large part dependent on sensors that monitor the environment and allow the control unit to act appropriately.

To remove safety issues and allow replication, the scenario can be simulated by a computer system. For a simulation to be realistic it needs to emulate sensors and allow them to read the 3D scene accurately. The most common method for this is ray tracing but also function and shader based methods exist [1],[2].

AGXUnity combines the AGX physics engine with Unity by a plug-in. A scene can be built using Unity and physics components from AGX can be added through a drop down menu. Algoryx, the developer of the AGX-engine is about to add the feature of sensor simulation to their solution. This could be achieved completely inside the AGX platform, or by utilizing an added 3D platform like AGXUnity. AGXUnity has many advantages to AGX alone and eases the creation of complicated scenes. It is therefore important to find out whether adding sensor simulation to the AGXUnity platform is an option.

The goal of this thesis is to find out if AGXUnity is a *viable* platform to offer customers in need of physics and sensor simulation. To this end a generic sensor will be implemented having only the three high-level properties range-, angle- and speed measuring. Criteria for evaluation will be the sensors accuracy in determining these three properties to an AGX geometry as well as its performance in a realistic simulation.

In Section 2, a short presentation of the state of sensor simulation is presented as well as an introduction to AGXUnity. The method of implementation is described in Section 3 and the results and conducted tests are described in Section 4. A discussion of the results and a conclusion of the thesis can be found in Section 5.

2 Background

This chapter starts by giving an overview of related sensor modelling approaches. An introduction to sensors and sensor simulations in general then follows. The chapter ends with a description of the simulation environment used in the thesis.

2.1 Previous work

A similar solution to AGXUnity, developed by the U.S. Army is VANE [3]. It was originally developed for unmanned ground vehicles. Its purpose is to produce sensor output that are easy to use and only uses a few determined input parameters. The idea behind their sensor models is that it should be general and not require a detailed characterization of each sensor. It uses ray casting as technique called the Quick Caster for creating ideal images of scenes.

Pro-Sivic [4] is an interactive physics driven simulation platform developed by IFSTTAR and made available by CIVITEC. It is primarily focused on optical sensors but also models e.g. LiDAR, radar, and GPS. It is a modular solution that creates a complete road environment including vehicles, infrastructure and the tested sensors. It is built up like a 3D simulator where different plug-ins can be added for different effects. Originally developed for embedded algorithm tests in road safety applications, it has later been used in other automotive areas as well as security and aeronautics industries.

An open source software that creates mesh clouds is GLIDAR [5]. It is based on OpenGL and works by reading in 3D meshes with the Assimp library and then converts these to point clouds. These clouds are primarily used for 3D sensors like cameras. In order to use physics, its result can be connected by TCP to a physics engine. In this way a custom sensor can measure distance to the point cloud and noise can be added as well.

Closely related solutions are those used in robot applications. ROS is mainly used for this purpose and its result can be visualized in its default tool RVIZ [6]. It is a modular system that can handle a number of programming languages like C++ and Python. GAZEBO is another robot simulation tool that is modular and ready for simulations [7]. It can create physics and sensor generation on a server side and has visualization and user interface on its client side. ROS and GAZEBOO can also interact. ROS for instance can use the latter instead of RVIZ for visualization.

A generic sensor was created by Deng et al. [1], having only high-level functionality. This is a function based approach using a combination of the cone and sphere equations. Each obstacle is then surrounded by straight lines which are tested for intersection with the cone and spherical surface. Low-level, FMCW type physical radar properties were then added on top of this generic sensor. A closed-loop adaptive cruise control simulation using Matlab/Simulink and iSim was used to test the sensor.

Another group [8] also creates a radar sensor by a combination of functional and physical methods. Like in the previous solution, a generic sensor is used as base that have distance and object detection capabilities. They use the cone equation only and measures the range from the origin to the center of an AABB. Different physical simulations are then added to create the FMCW radar properties. They use PanoSim to create a Matlab/Simulink simulation for testing.

2.2 Sensor simulation

There are a number of different sensors in the automotive industry. A few of them includes radar, which is gaining popularity in autonomous applications [9] and LiDAR as well as those based on computer vision, such as cameras and depth cameras. There are also accelerometers, gyros, and pressure sensitive sensors. A few different approaches exists when modelling sensors in simulations. To create the high-level sensor part, there are the ray tracing-, function- and shader based approaches [2],[8]. The physical characteristics of the sensor can then be added on top of this.

In the function based approach to model e.g., radar, a radio wave is represented as a geometric object. This object is usually in the form of a cone aimed in the direction of the object to be measured. The object itself can be represented as a simple bounding box for ease of computation. The actual measuring is then a matter of checking which parts of the object's bounding box that intersect the cone surface.

There are two common ways of adding physical characteristics. The most common [2] uses circuit software to create a radar wave, and ray tracing to form the propagation channel. Another form is signal based where clutter and noise are added artificially to the reflected radio signal, often using a function based sensor as base.

In complex simulations consisting of a large number of 3D traffic participants and sensors, ray tracing based sensor simulation is almost real time unfeasible [2]. For more realistic simulations, accuracy has to be sacrificed somewhat for efficiency. Shader and function based approaches has slightly lower precision than ray tracing, but is much faster in these scenarios. The functional approach has been around for longer than the shader based and is well tested.

2.3 AGX

Algoryx Simulation is a provider of software and services for visual and interactive physics based simulation. Algoryx currently provides three products: AGX Dynamics, Algoryx Momentum for the professional market, and Algodoo for the education market. The company is located in Umeå and is a spin off from Umeå university [10].

Of the three mentioned products, it is AGX Dynamics that is used in this thesis. AGX Dynamics is a multi-purpose physics engine for applications, such as simulators, VR applications, engineering, and scientific simulations [11]. Specific systems include robots, vehicles, and cranes. The software architecture is formed as a Software Development Kit (SDK). The core library is oriented around rigid multibody systems with nonsmooth dy-

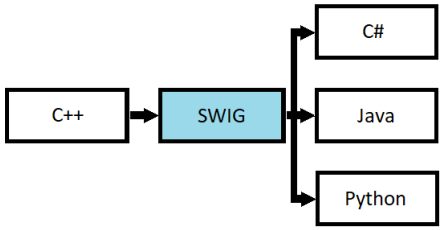

Figure 1: Overview of the SWIG wrapper tool.

namics and numerical time integration based on SPOOK [12]. AGX can be combined with several different softwares like Unity and CAD applications. It also has direct coupling to Matlab/Simulink.

2.4 Unity

Unity is a platform for creating 2D and 3D scenes. Scenes can be created with a minimal of coding by dragging and dropping so called *GameObjects*, e.g., boxes and cameras, into the *Scene* view or game *Hierarchy*. The Hierarchy is a table next to the scene window where GameObjects are listed. This gives an overview of all objects in the scene, and makes reordering and creation of child objects easier. A specific GameObject can also have *Components* added to it. These are e.g., *Colliders* that makes intersection possible, and *scripts* where C# code is used to add behaviour to the GameObject.

In *Edit mode*, objects can be added to the scene, moved around or configured in the *Inspector* window. Activating *Game mode* starts the simulation and any changes made here will disappear again when exiting.

2.5 Coupling between AGX and Unity

Unity is connected to AGX through SWIG. As seen in Figure 1, SWIG can translate between chosen languages, in this case C++ and C#. A C++ library called *AgxDotNet* contains all the relevant classes. On the Unity side, C# scripts control the behaviour of all objects in the scene. When using any object like the AGX *Collide box*, the script includes headers from AGX by translation of *AgxDotNet* classes through SWIG. An object like a box inherits from both AGX classes *Shape* and *Geometry*. The box script must also add Unity functionality, e.g., to make it visible, and moved around in the scene. Any sensor-script that will be added to the scene and utilize AGX must inherit Unity behaviour and include the AGX API through SWIG.

A scene therefore uses AGX specific objects, Unity specific objects, and a combination of them both. For instance, to create a script containing the sensor code, it must inherit *MonoBehaviour* from Unity and uses the AGX Collide box class, an AGXUnity script, for intersection tests.

3 Method

As mentioned in Section 2.2, function based sensors are preferred over ray tracing in more complex scenes. The principle for this implementation is therefore based on the works of Deng et al. [1] described in Section 2.1. The equation used here is simpler than in Deng and instead uses the equation in [8], which is without the frontal spherical surface.

The method creates straight lines around the object which are used to calculate intersections with the cone surface. Straight lines are simple and make intersections with surfaces like the cone more accurate. Because the number of lines can be adjusted, the resolution of the sensor can be optimized for either accuracy or performance. Moreover, the method needs only the vertices of the objects mesh to create the lines. Because of this, any object made with triangles and faces can be used. This includes not only AGXUnity objects, but any one with a mesh.

3.1 Lines

The lines are formed from the vertices of the objects mesh. The first line is created from two consecutive vertices, $(\mathbf{v}_0, \mathbf{v}_1)$. The last line is created from the vertices of the other side, $(\mathbf{v}_2, \mathbf{v}_3)$. More lines in between are then created by interpolation of the four vertices of the face according to Table 1 below.

To make sure all vertices are in the correct order, the dot product is used. If a face vertex is not sequential, the positions will be swapped. The lines are then stored as pairs of vertices, $(\mathbf{q}_0, \mathbf{q}_1)$. An object with lines visualized can be seen in Figure 2

3.2 Cone

The actual sensor is composed of a mesh[1] and the cone equation. The mesh in Figure 3 is in the form of a one sided cone in the positive z-direction with the length r. The cone equation 3.1 is adjusted to allow for different angles in horizontal (θ) and vertical (ϕ) axes.

$$\begin{aligned} \frac{x^2}{z^2 tan(\theta/2)^2} + \frac{y^2}{z^2 tan(\phi/2)^2} &\leq 1 \\ z &\leq r \end{aligned} \tag{3.1}$$

[1] http://wiki.unity3d.com/index.php?title=CreateCone

Table 1 Creation of the line vertices. $\mathbf{v}_n$ are the vertices of the face, $\mathbf{q}_n$ are the vertices of the finished line

Line number	1^{st} vertex ($\mathbf{q}_0$)	2^{nd} vertex ($\mathbf{q}_1$)
L_0	$\mathbf{v}_0$	$\mathbf{v}_1$
L_1	$\mathbf{v}_0 + \frac{\mathbf{v}_3 - \mathbf{v}_0}{N-1} * 1$	$\mathbf{v}_1 + \frac{\mathbf{v}_2 - \mathbf{v}_1}{N-1} * 1$
L_2	$\mathbf{v}_0 + \frac{\mathbf{v}_3 - \mathbf{v}_0}{N-1} * 2$	$\mathbf{v}_1 + \frac{\mathbf{v}_2 - \mathbf{v}_1}{N-1} * 2$
...	...	...
L_{N-1}	$\mathbf{v}_2$	$\mathbf{v}_3$

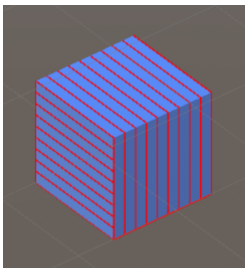

Figure 2: Visualization of 10 lines per face

3.3 Closest point on line

After the lines have been computed, they can be used to determine if the object is within the cone volume. First the inverse sensor transformation is applied to the line to be tested to move it to the sensor frame. There are three cases for a line. Both its vertices are inside the cone, one vertex is inside and the other outside or both vertices are outside of the cone. How a vertex is determined to be inside or not is described in Section 3.3.1. When both are inside, the question of whether or not the line segment is inside is trivial. The closest point on this line to the sensor origin is then determined by the equation

$$\begin{aligned} t &= -\mathbf{q}_0 \bullet \frac{\mathbf{q}_1 - \mathbf{q}_0}{|\mathbf{q}_1 - \mathbf{q}_0|^2}, \quad (0 \leq t \leq 1) \\ \mathbf{q} &= \mathbf{q}_0 + t(\mathbf{q}_1 - \mathbf{q}_0). \end{aligned} \tag{3.2}$$

This equation projects the line from the first vertex, $\mathbf{q}_0$, to the sensor origin onto the normalized line segment in question. This will create a perpendicular line from the origin to the line segment which is also the closest distance. The shortest point on the line segment, $\mathbf{q}$, is then found by going from $\mathbf{q}_0$ to this position. Since the line is infinite, but the segment is finite, the point $\mathbf{q}$ must be found between $\mathbf{q}_0$ and $\mathbf{q}_1$. The value of the parameter t must therefore lie between 0 and 1 and clamped otherwise.

If only one vertex is inside and the other outside, the vertex $\mathbf{q}_1$ must be the intersection of the line with the cone surface. An intersection test described in section 3.3.2 is then performed. The inner point $\mathbf{q}_0$ together with the intersection point is now tested for closest distance in the previous manner.

If both vertices are outside there can be two cases. The line is completely outside and the

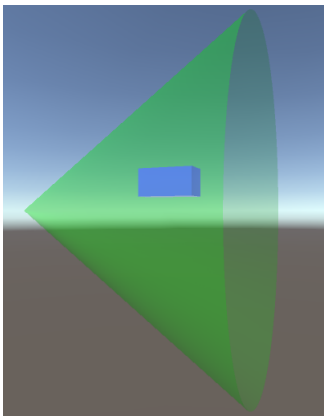

Figure 3: The cone mesh at 45 degrees with transparent color.

infinite line does not intersect the cone. In the other case the infinite line intersects the cone surface and it must be determined if the line segment crosses the cone. The intersection test is now used to get both vertices of the line. If the line segment crosses the cone, its vertices must lie outside of the intersections. Otherwise, both intersections and both vertices lie on separate sides. The dot-product between the vertices, $\mathbf{q}_n$, and an intersection point $\mathbf{i}_0$ is then used to determine if the segment is inside according to Equation 3.3

$$(\mathbf{q}_0 - \mathbf{i}_0) \bullet (\mathbf{q}_1 - \mathbf{i}_0) < 0. \tag{3.3}$$

If the sign is negative, the vertices lie outside of the intersections and the line segment crosses the cone. Otherwise vertices and intersections lie on opposite sides and the line segment does not cross the cone.

All the closest points on each line are collected in a list and used as described in 3.4 to find the one closest to the origin.

3.3.1 In-cone test

To determine if a vertex is within the volume of the sensor cone, Equation 3.1 is used. All values of the equation that are less than or equal to 1 is inside the cone. Since the equation is an infinite cone in both positive and negative z-direction it must be truncated. The z-value of the vertex must therefore lie between 0 and the length, r, of the sensor cone. When fulfilling these criteria, the vertex is flagged as inside.

3.3.2 Intersection test

To find the intersection with the cone surface, the coordinates x, y, z in Equation 3.1 is replaced by the three components of the infinite line $\mathbf{q}_0 + t(\mathbf{q}_1 - \mathbf{q}_0)$. The new expression is rearranged into a second degree equation of the form

$$At^2 + Bt + C = 0. \tag{3.4}$$

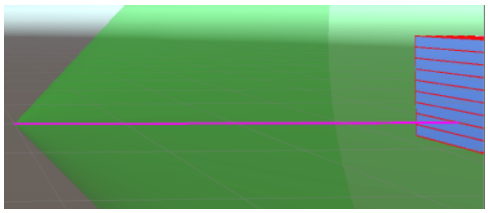

Figure 4: The closest point to the sensor origin visualized with a magenta colored line

The parameter t can then be calculated in the standard way with the equation

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}. \tag{3.5}$$

Most lines will not intersect and can be discarded without taking the square root by using

$$\delta = B^2 - 4AC \tag{3.6}$$

as a first test. If δ is less than zero, there are no real roots and no intersections. If δ is zero, the intersection is $-B/2A$. For δ larger than 0, there are two intersections that will be tested. If the point created from the tested root is within the visible cone length, it is added as an intersection. If the root leads to a point on the positive side of the cone, but farther away than the cone length, it is moved. This is achieved by sliding it along the infinite line until it is at the cone's length. This means it intersects the elliptic surface at the end of the cone. Another similar case is when the infinite line is parallel to the cone length. The intersection will then be on the cone surface as well as the frontal elliptic surface and the point is adjusted in the previous manner.

3.4 Shortest distance

After the closest point on each visible line has been saved, the shortest distance will be retrieved. A search loops through the list to find the points with the lowest squared magnitude. This point is then used as shortest distance from the sensor origin. It is then transformed back to its real world coordinate by applying the sensor world transform to the point. As seen in Figure 4, a line renderer then draws a line between the point and the real world origin of the sensor.

3.5 Calculation of angles

There are two different angles between the sensor origin and the closest point on the object. One is the angle in horizontal direction, called the azimuth angle, α . The other is the vertical angle called elevation, e. The azimuth angle is when seen from above, the arc-tangens of the x-axis and the z-axis in the Unity left handed coordinate system.

$$\alpha = arctan\left(\frac{x}{z}\right). \tag{3.7}$$

The elevation angle is then when seen from the side, the arc-tangens of the y-axis over the z-axis

$$e = arctan\left(\frac{y}{z}\right). \tag{3.8}$$

Extracting these values is only about taking out the x, y and z values from $\mathbf{q} = (x, y, z)$.

3.6 Calculation of speed

The speed is calculated from the second recording of the closest point. The magnitude of the difference between two consecutive points is divided by the actual time since last reading.

$$\frac{|\mathbf{q}_i - \mathbf{q}_{i+1}|}{t_{real}} \tag{3.9}$$

The solution does not take into account whether it is the same object as last reading or not. This could give a strange value in between objects.

4 Results

The tests are divided into three main parts. First the accuracy in determining range and angle to the obstacle is tested. The next test determines accuracy and variability in speed. Last is a stress test to determine how the sensor acts in a standard simulation with several obstacles and with varying number of lines.

4.1 Angles and range

To test the accuracy in range and angles a simple test was conducted where other variables are eliminated. The obstacle was positioned at $z = 1$ and $x = y = 0$. In this position, the sensor should give range, $r = 1$, azimuth angle, $\alpha = 0$ and elevation angle, $e = 0$. First, only three lines were used to see if the sensor gives the exact values. With three lines, two will be on each side of the face and the third perfectly centered in the middle. Results showed the range to be $r = 0.99999$ and the two angles, 0. To get a measure of the general accuracy of the sensor, the test was done in 4 different steps with increasing number of lines, from 2 to 1000.

As can be seen in Table 2, accuracy increases with higher number of lines. The error in elevation angle comes from the way the lines are drawn on the measured face. Since they are horizontal the error will be in the vertical direction. Maximum error in angle should be the angle from half the distance between two lines. This can be confirmed from the tangens value in Table 2. With two lines and with a face with side 1 in vertical direction the y-value should be 0, but the result in y-axis is 0.5. This relationship is confirmed with higher number of lines. The maximum error in angle can then be calculated from

$$\theta_{error} = arctan\left(\frac{extent_{1/2}}{(nLines-1)\cdot r}\right). \tag{4.1}$$

The error in range always have a lower value than the angle in Table 2. The angle is determined from the side in the triangle formed by the origin, the actual position and the error point. Range is the hypotenuse in this triangle and its error will be of smaller value. Therefore the angles can be used as an upper limit in error. Table 3 shows error in angle calculated from the above formula, at range $z = 10$ for increasing number of lines.

To get a confirmation of the sensors accuracy for a more random position, an obstacle was placed at $(5,0,7)$. It was then rotated by $arctan(5/7) = 35.53767$ degrees around the y-axis to align with the origin. With 1000 lines, range was measure to be 8.60233 and calculated value 8.60233. Azimuth angle was 35.53765 and calculated value 35.53768. Elevation angle was 0.004096 instead of 0 because of the error.

Table 2 Measure of a $1 \times 1 \times 10^{-5}$ obstacle at $(0,0,1)$ with different N:o of lines

N:o lines	Range	Elevation (e)	Azimuth (α)	Tan(e)
2	1.118025	26.56528	0	0.50000
10	1.001532	3.179862	0	0.055556
100	1.000003	0.2893734	0	0.0050505
1000	0.99999	0.02867683	0	0.000500505

Table 3 Values for error in angle from range $z = 10$ calculated from Equation 4.1

N:o lines	Error (Deg)
2	2.862
10	0.318
100	0.0289
1000	0.00287

4.2 Speed

A test with three different linear velocities was conducted to determine the speed accuracy of the sensor. The test lets an obstacle go from the origin to $z = 10$. The total number of speed values was saved and the average was computed. The linear velocity of the rigid body was set to $z = 0.25$, $z = 1$ and $z = 2$. The average speeds can be seen in Table 4

The difference in speed between two consecutive measurements was between $\pm 20\%$. The averages will therefore be closer to the real speed the more measurements recorded. A faster speed will give fewer measurements.

4.3 Frame rate

The accuracy of the sensor is dependent on the number of lines surrounding the obstacles. Number of lines in turn is limited by the systems frame rate in the simulation. To determine maximum accuracy, a scene used by Algoryx (Figure 5) was tested with varying number of obstacles and lines.

The frame rate for the scene without sensor was 56.2 fps. In Table 5, frame rates for 1, 5 and 10 obstacles can be found. For each of these three categories, 10, 100 and 1000 lines were tested.

Table 4 Average speeds measure by the sensor for different linear velocities

linear velocity	avg speed
0.25	0.249
1	0.988
2	1.946

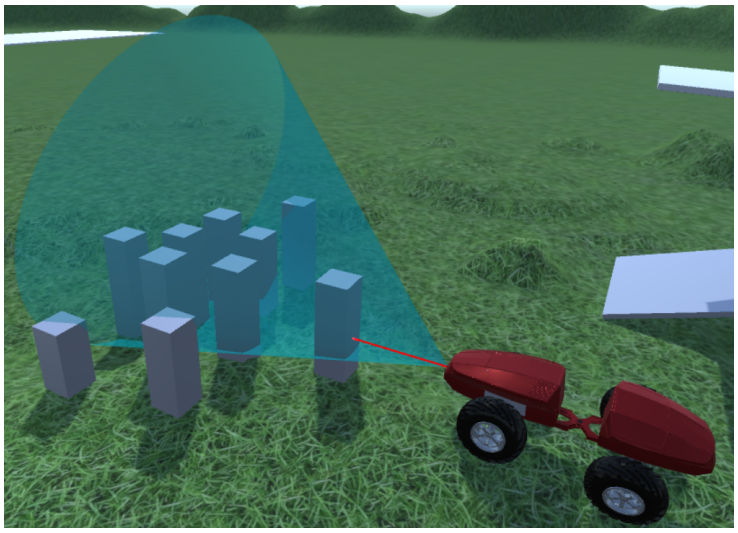

Figure 5: The simulated scene with 10 obstacles

Table 5 Frame rates for a scene with varying number of obstacles and lines

N:o obstacles N:o lines	1	5	10
10	55.2	53.8	55.5
100	54.4	52.8	54.4
1000	20.5	20.6	20.7

5 Discussion and Conclusion

The main question to answer in this section is whether or not AGXUnity is a *viable* platform for sensor simulation. To this end a generic sensor was implemented with only high-level functionality. Criteria for evaluation are the sensor's accuracy in measuring range, angle and speed of an obstacle together with its performance.

It was shown in Section 4 that if there is a line at a point, the sensor will give an exact value. Since lines have a distance between them, there will always be errors in measurement. What limits the accuracy is the frame rate drop that comes from increasing the number of lines in a simulation. Table 2 shows that at a distance of 10 meters and with 100 lines, the error in angle is 0.029 degrees for a 1×1 face. In comparison, the radar model in [13] has an error in angle of 0.1 over 10.65 meters. At this distance 100 lines is well within limits. The difference is that their model uses radar characteristics and doesn't show worse results at closer distance. At 1 meter, Table 2 shows an error of 0.29 degrees at 100 lines which is much worse. 1000 lines shows 0.028 degrees in error and is well within limits. At this number the frame rate of the test computer went from about 55 fps down to 20 fps. Even a faster computer could still struggle at this resolution. A number of lines in between 100 and 1000 would therefore be recommended both for accuracy and performance.

Speed was measured and presented in Table 4. The variability in values was high and probably dependent on the way time is measured. There are methods to compensate for time fluctuation between frames which could improve the implementation. As it stands right now, the variability between two consecutive speed measurements is too high to give an exact value. It can on the other hand be used to determine if an obstacle is moving or standing still. It can also be seen if the obstacle is approaching or leaving the sensor.

There are several improvements of the implementation that would increase accuracy and performance. The number of lines for instance could scale with the size of the object, or its numbers increase closer to the sensor origin. In [8], faces pointing away from the sensor is discarded first, as well as objects not in sight. This would mean to remove objects behind the sensor or to far from the sensor's range. This would require that an obstacle had a maximum determined size, since a very large object could have a center far from the sensor, but still have faces within the sensor cone.

By using lines and the cone equation instead of ray tracing there will be less precision but increased performance. Ray tracing is very accurate but performance heavy. Several implementations uses ray tracing or casting [3],[9]. This makes simulation of physical characteristics more accurate whether it be e.g., radar, LiDAR or sonar. The present implementation still allows for adding physical characteristics and is a good trade off between accuracy and performance. New hardware like the GeForce RTX 20 Series[1] allows for real time ray tracing. This could make it possible to use ray tracing based sensors with the same performance. Price of this hardware will fall in the future and software will be adapted to utilize

[1] https://www.nvidia.com/en-us/geforce/turing/

it. This change could make the present implementation obsolete, since higher accuracy will be achieved with retained performance. Despite this, older or cheaper hardware as well as many different software's could still need the present solution in the foreseeable future.

In conclusion the implemented generic sensor gives accurate measurements with good performance. Speed measurement could be improved, but shows direction and movement of the obstacle. These points together shows that AGXUnity is a viable platform for synthetic sensors.

References

[1] W. Deng, S. Zeng, Q. Zhao, and J. Dai, "Modelling and simulation of sensor-guided autonomous driving," *Int. J. Vehicle Design*, vol. 56, pp. 341–366, 2011.

[2] S. Wang, S. Heinrich, M. Wang, and R. Rojas, "Shader-based sensor simulation for autonomous car testing," in *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pp. 224–229, Sept 2012.

[3] C. Goodin, R. Kala, A. Carrrillo, and L. Y. Liu, "Sensor modeling for the virtual autonomous navigation environment.," *Sensors, 2009 IEEE*, pp. 1588–1592, 2010.

[4] M. Grapinet, P. D. Souza, J.-C. Smal, and J.-M. Blosseville, "Characterization and simulation of optical sensors," *Procedia - Social and Behavioural Sciences*, vol. 48, pp. 962–971, 2012.

[5] J. O. Woods and J. A. Christian, "Glidar: An opengl-based, real-time, and open source 3d sensor simulator for testing computer vision algorithms," *Journal of Imaging*, vol. 2, no. 1, 2016.

[6] D. Bagheri and S. Starke, "Ros and unity a comprehensive introduction." https://tams.informatik.uni-hamburg.de/lehre/2016ws/seminar/ir/doc/slides/DanialBagheri-ROS_and_Unity.pdf (visited 2018-04-20).

[7] J. Hsu and N. Koenig, "GAZEBO." http://gazebosim.org/assets/gazebo_roscon_2012-07009eb8e8a3d43fa7ff585ee562f67e.pdf (visited 2018-04-20), 2012.

[8] J. Guo, W. Deng, S. Zhang, S. Qi, X. Li, C. Wang, and J. Wang, "A novel method of radar modeling for vehicle intelligence," *SAE Int. J. Passeng. Cars â" Electron. Electr. Syst.*, vol. 10, pp. 50–56, 09 2016.

[9] M. Dudek, R. Wahl, D. Kissinger, R. Weigel, and G. Fischer, "Millimeter wave fmcw radar system simulations including a 3d ray tracing channel simulator," *Proceedings of the Asia-Pacific Microwave Conference*, pp. 1665–1668, 2010.

[10] Algoryx, "About Algoryx." https://www.algoryx.se/mainpage/wp-content/uploads/2014/10/About-Algoryx.pdf (visited 2018-04-20).

[11] Algoryx, "Agx dynamics." https://www.algoryx.se/products/agx-dynamics/ (visited 2018-04-20).

[12] C. Lacoursière, "Ghosts and machines: regularized variational methods for interactive simulations of multibodies with dry frictional contacts," *PhD thesis*, 2007.

[13] J.-J. Lin, Y.-P. Li, W.-C. Hsu, and T.-S. Lee, “Design of an fmcw radar baseband signal processing system for automotive application,” *SpringerPlus*, vol. 5, p. 42, Jan 2016.