

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

ApolloSim: A Lidar Simulator With Calibrated Sensor Noise

By
Gavri Kepets

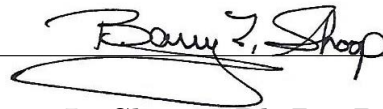
A thesis submitted in partial fulfillment of the requirements for the degree of Master of
Engineering

Advisor

Dr. Carl Sable

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART
ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.



Barry L. Shoop, Ph.D., P.E. - May 10th, 2024

Dean, Albert Nerken School of Engineering



Prof. Carl Sable - May 10th, 2024

Candidate's Thesis Advisor

Acknowledgment

I would like to extend my deepest gratitude to those who have made the completion of this thesis possible. Throughout my entire college career I have been incredibly fortunate with the amount of support I have received from family, friends, professors, and more.

A special thanks goes to Professor Carl Sable, my thesis advisor and professor, whose guidance and classes have been invaluable. It was always a pleasure to be in Professor Sable's classes, and I have learned an immense amount from him. I'd also like to thank Michael Giglia for always providing me with technical assistance and excellent advice.

I am extremely thankful to my family and friends. Thank you to my parents, who enabled me to be able to go to college, and have always been an incredible support system for me. Thank you to Abby, my amazing fiance, who has always been there for me no matter what and has been a constant source of support and happiness. Finally, I would like to thank my friends, especially Netanel and Ayden, for continually pushing me to excel. Our shared accomplishments at Cooper have been remarkable.

The journey has been challenging and rewarding, and I owe everything to those who have supported me throughout it.

Thank you - Gavri Kepets

Abstract

In this work, I present a light detection and ranging (lidar) sensor simulator that incorporates realistic sensor noise. Lidar sensors are used for various applications, ranging from autonomous vehicles to satellite imagery. Field testing a lidar presents multiple inconveniences and challenges; it is often costly, poses a risk of damage to the device, and is time-intensive. Therefore, the benefits of employing a lidar simulator are evident. Moreover, lidar data can be crucial for training large-scale deep learning models, which require massive datasets; generating this data in simulation is significantly quicker and more cost-effective than manual data collection.

I introduce a lidar simulation methodology that uses a real lidar sensor to calibrate the simulation, in order to maximize the accuracy of the simulation. The simulation utilizes sensor data from multiple benchmark materials to accurately replicate the noise in the data produced by the sensor. The results indicate that simulations calibrated with real sensor noise outperform those based on standard parametric approaches. The data generated by the simulation confirms that calibrating a lidar simulation with the target sensor is a viable and economical approach for rapid and precise lidar simulations.

Contents

Introduction	1
1.1 Motivation	1
1.2 Novel Approach to Lidar Simulation	3
Background Information	5
2.1 Lidar Sensor Functionality	5
2.1.1 Time of Flight	5
2.1.2 Lidar Sensor Specifications	6
2.1.3 Beam Emission	7
2.1.4 Point Clouds	8
2.1.5 Key Characteristics of Lidar Data	9
2.2 Simulation	10
2.2.1 Parametric vs Nonparametric Simulation	10
2.2.2 Synthetic Noise	11
2.3 Computer Graphics	12
2.3.1 OpenGL	12
2.3.2 Compute Shaders	13
2.3.3 Ray Tracing	14
2.3.4 Bidirectional Reflectance Distribution Models	15

Related Work	18
3.1 HELIOS++	18
3.2 A GPU-accelerated framework for simulating LiDAR scanning	19
3.3 BlenSor: Blender Sensor Simulation Toolbox and BlAInder Range Scanner	20
3.4 Lidar Simulation for Robotic Application Development	21
3.5 LiDARsim: Realistic LiDAR Simulation by Leveraging the Real World	22
3.6 Learning to Simulate Realistic LiDARs	22
Implementation	24
4.1 Simulator Overview	24
4.1.1 Simulator Pipeline	24
4.1.2 Technology Stack	25
4.2 Sensor Calibration	27
4.2.1 Sensor Calibration Overview	27
4.2.2 Interfacing with the Lidar	28
4.2.3 Sensor Calibration Implementation	29
4.3 Graphics Engine	32
4.3.1 Graphics Engine Overview	32
4.3.2 Beam Rendering	33
4.3.3 Calibration Data Rendering	34

4.4	Simulation Engine	35
4.4.1	Simulation Engine Overview	35
4.4.2	Simulation Engine Inputs	35
4.4.3	Lidar Beam Calculation	36
4.4.4	Synthetic Noise Calculation	38
4.4.5	BRDFs for Addressing Data Gaps	40
4.5	Application Walk-Through	43
4.5.1	Calibrating the Simulation	43
4.5.2	Configuring the Simulation	45
4.5.3	Running the Simulation	45
	Results and Evaluation	48
5.1	Evaluation Overview	48
5.2	Validating Key Characteristics	48
5.3	Improved Accuracy of ApolloSim Over Parametric Models	53
5.4	Efficacy of BRDFs in Addressing Data Gaps	55
5.5	Qualitative Analysis of ApolloSim	57
	Conclusions and Future Work	66
6.1	Conclusions	66
6.2	Future Work	68
6.2.1	Improving the Current Features of ApolloSim	68
6.2.2	Additional Features for ApolloSim	71

A	Additional Information	73
A.1	Why “ApolloSim”?	73
A.2	ROS 2 Node for Collecting Lidar Data	73
A.3	Propagation of Data from Calibration to Simulation	75
A.4	Compute Shader Pseudocode	77
A.5	The Cook-Torrance Model	79
A.6	The Fresnel Effect	81
	Bibliography	82

Introduction

1.1 Motivation

Light detection and ranging sensors, commonly known as lidar sensors, are used to measure the distances of surrounding objects using light. Lidar sensors have become incredibly useful and increasingly common in the fields of robotics, automation, and remote sensing. For example, lidar sensors can be used on autonomous vehicles or robots for environmental detection, on drones or satellites for geographic mapping, and even for mapping archaeological sites [1, 2, 3].

An important aspect of working with lidar sensors is testing the equipment, especially if the sensor is a key component of high-stakes systems like an autonomous vehicle's environmental detection system. It is essential to rigorously test both the lidar sensor and the vehicle itself before deploying them in real-world operations. The real world is unpredictable, and ensuring the functionality of a device as potentially dangerous as an autonomous vehicle is imperative. Testing a device can be an incredibly elaborate task; often, in lieu of testing a physical device, the device is tested in a computer simulation.

Simulation is an important part of testing in engineering. Simulations

exist for various scenarios, devices, and phenomena. Computers can simulate myriad scenarios, sometimes instantaneously, allowing the testing process to be relatively quick, economical, and extensive. In addition to testing, simulations can be useful for generating synthetic datasets to be used in machine learning models. The focus of this thesis is on lidar sensor simulation. The ability to simulate the data collected from a lidar sensor can be especially useful for testing devices that utilize lidar sensors as well as for training models that use lidar data.

Numerous lidar sensor simulators already exist [1, 4, 5, 6, 7], and each simulator has their own advantages and disadvantages. A major aspect of simulation is “noise”. Sensors collect data from the real world, and the real world is not perfect; therefore, “noise”, or unwanted, irrelevant, and possibly false information, is introduced to the sensor. Computer simulations can be inherently perfect, and have no sensor noise; therefore, often, synthetic noise is injected into the simulation in order for the simulation to more closely resemble the real world [8].

As mentioned above, lidar simulators can be incredibly useful for a variety of applications. The goal of this thesis is to develop a real-time lidar sensor simulator, called ApolloSim (A.1), with accurate synthetic noise; the quicker and more accurate a simulation is, the more useful it can be for testing purposes.

1.2 Novel Approach to Lidar Simulation

The concept of synthetic sensor noise for simulation is not new. Sometimes, the synthetic noise is based on numerous derived equations and approximations that are collected from the real world [4, 5], and other times, synthetic noise is data-driven, and learned through a machine learning or deep learning model [9, 10, 11]. This thesis focuses on a different kind of noise, which will be referred to as “calibrated noise”. Calibrated noise is derived from real data collected specifically with the sensor the user wants to simulate.

First, the user sets up their lidar in front of the material benchmark. The material benchmark includes a number of distinct and prevalent materials that the lidar may encounter. Each material interacts with light in different ways, causing the beams emitted by the lidar to react differently to each material, resulting in a unique signal for each material. After data is collected for each benchmark material, the noise that the sensor experiences with each one can be derived. After characterizing the noise in the data, ApolloSim will attempt to mimic it in simulation.

For example, the majority of a beam of light is reflected from a smooth, metallic surface, but scattered from a rough, craggy surface. Therefore, lidar sensors will receive a different signal from a brick wall than a shiny car door. Because the materials during calibration are known, the simulation will know

to make smooth, metallic surfaces return intense and complete signals, and to make rough, craggy surfaces return dim and incomplete signals.

This approach has the potential to provide complete and accurate sensor noise for a lidar simulation. Using calibrated noise bridges the gap between theoretical models and data-driven noise, promising significant improvements to lidar simulation accuracy. Using calibrated noise, this work aims to improve the realism of lidar simulations, bringing them one step closer to the real world.

Background Information

2.1 Lidar Sensor Functionality

2.1.1 Time of Flight

A lidar sensor uses light to determine the distances of objects around itself. In order to do so, the lidar sensor emits a laser beam and records how long it takes for the beam to return. Because the speed of light is known, and the time it took for the light to return is known, the distance the light traveled can be derived. Lidar sensors can emit hundreds of thousands of beams per second, allowing them to collect extensive information about their surrounding environment.

$$D = c \frac{t}{2} \tag{2.1}$$

As seen in equation 2.1, the distance between the sensor and an obstacle can be determined by multiplying the speed of light by half of the time it took for the laser to return back to the sensor [12].

2.1.2 Lidar Sensor Specifications

Lidar sensors come in all shapes and sizes. To name a few examples, terrestrial lidar sensors are used for gathering data on land, often from robots and autonomous vehicles, airborne lidar sensors are used for gathering data from drones or helicopters, and bathymetric sensors are tuned to gather data underwater [13].

Each sensor has its own set of specifications, which determine the capabilities of the sensor as well as the data that it can collect. Specifications such as working range and field of view determine the area that the lidar can see. Additionally, some sensors are two-dimensional, which means they can only see in the horizontal plane, while others are three-dimensional, which means they can see in the horizontal and vertical plane (it also means they have both a horizontal and vertical field of view). In order to capture a large field of view, the laser inside of a lidar sensor rotates, allowing the sensor to capture data from all around itself.

More often than not, lidar sensors contain more than one beam emitter, in order to collect as much data as quickly as possible. For example, the Velodyne Puck, as seen in figure 2.1, has sixteen laser emitters [14]. Lidar sensors also have a scan and sample frequency, which determine the amount of beams that are cast by the sensor. The scan and sample frequency are related; the scan frequency is the speed at which the lidar completes a full



Figure 2.1: The Puck Lidar, by Velodyne, is a three-dimensional lidar sensor with sixteen lasers. The Puck has a 360-degree horizontal angular range, a 30-degree vertical range, a 100-meter working range, and a sample frequency of about 600 KHz [14].

cycle of measurement across its entire field of view, which indicates how many times per second the sensor can scan its environment, while the sample frequency is the amount of beams generated by the lidar per second. For example, if a lidar has a 10 Hz scan frequency, it gathers and completes a complete cycle of data ten times per second. If that same lidar has a 32 KHz sample frequency, it means that it casts 32,000 beams per second, and 3,200 beams per frame [15].

2.1.3 Beam Emission

For each beam, a series of light pulses are actually emitted as opposed to a single, constant beam. A constant beam is easier for the sensor to detect and will travel further, but will be more difficult to determine exactly how long it took for the beam to return, as it lacks discrete timing markers. On the other hand, the emission of multiple, shorter light pulses enables

the lidar to accurately measure the travel time of the beam. Pulses can be differentiated by varying the wavelength, amplitude, and more. Additionally, multiple unique pulses can be used to capture environments at varying levels of detail [16]. The size and shape of the beam heavily influence the accuracy and capabilities of the sensor [17].

2.1.4 Point Clouds

The output of a lidar sensor is a collection of points, commonly known as a “point cloud”. Each point marks the location of a reflected lidar beam from an object or surface back to the sensor. Point clouds are the lidar sensor’s representation of its surrounding environment, as seen in Figure 2.2.

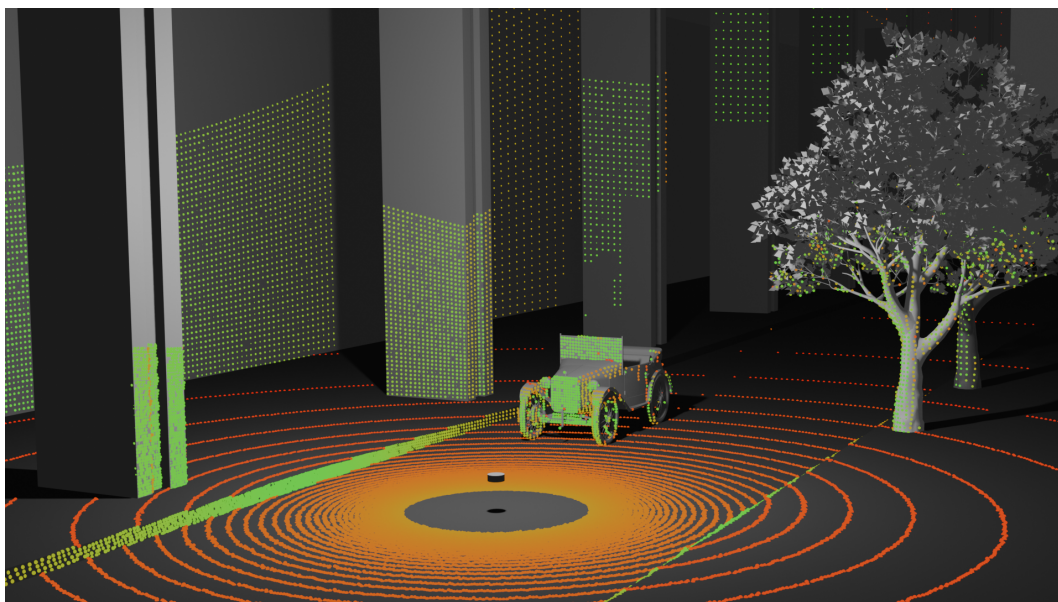


Figure 2.2: This is a render of a point cloud generated using BLAINDER, a lidar simulation tool for Blender [18]. The simulated sensor is a three-dimensional sensor with a scan frequency of 5 Hz and a sample frequency of 288 KHz. (3D Models supplied supplied by Sketchfab [19, 20, 21]).

In addition to the locations of points, point clouds often also include the

intensity of the backscattered light for each point. In Figure 2.2, points shaded in green represent stronger intensity values, whereas points shaded in red indicate weaker intensity values.

The location vector can be a two or three-dimensional vector, and the density and accuracy of these points depend on the operational parameters of the sensor, as detailed above in Section 2.1.2. For example, a lidar with a very high sample frequency will generate a high-density point cloud [22].

2.1.5 Key Characteristics of Lidar Data

There are a few key metrics that are often used to analyze a point cloud. Common metrics include raydrop and the distributions of intensity and distance values. Raydrop is when a lidar sensor emits a beam and never observes a response. Raydrop can happen if the nearest obstacle is out of the detectable range of the sensor, if the material the beam collides with does not reflect the beam back appropriately, or due to various environmental conditions such as fog, rain, or atmospheric conditions [9, 10]. In addition to raydrop, it is also important to consider the distribution of the point cloud data. Many lidar simulations use a normal distribution to sample synthetic noise [23, 4, 24]. In ApolloSim, the mean and standard deviations for distance and intensity values, as well as the raydrop rate, are important for generating synthetic noise.

2.2 Simulation

2.2.1 Parametric vs Nonparametric Simulation

Simulations are often either parametric or nonparametric. A parametric model or simulation is determined by a derived equation, based on real-world observations and approximations. Nonparametric models are typically data-driven and do not assume a predefined form for the relationship they are modeling.

For example, let's say there is a simulation that simulates the growth of a tree. A parametric solution employs a specific pattern or model to estimate the growth of a tree. An example of a parametric solution could be to employ a simple formula in which each year the tree grows two feet, as seen in Equation 2.2.

$$\text{Height} = \text{Age} * 2 \tag{2.2}$$

The parametric model is simple, and typically derived by someone who determined that trees, on average, grow two feet per year. The nonparametric solution, on the other hand, does not assume a specific growth pattern; instead, it may use historical data from numerous trees to model the growth curve. An example of a nonparametric model would be using a Decision Tree approach to predict the height of a tree based on a variety of factors, not just age. This method could consider data on sunlight, soil type, water availabil-

ity, and other factors that influence growth, creating a more complex model that can adjust to a wide range of conditions without a fixed formula.

As described above, parametric simulations are great for fast, understandable results, while nonparametric simulations are more suitable for complex and nuanced results [25, 26]. In comparing parametric and nonparametric lidar simulations, a parametric simulation may use specific equations to model the interaction of light with objects, often simplifying complex environments into manageable mathematical forms. Conversely, a nonparametric lidar simulation does not rely on defined models or equations but instead uses data-driven approaches to model interactions. This may involve training a model to generate point clouds given a specific environment [9, 10].

2.2.2 Synthetic Noise

Sensors in the real world are not perfect; the data that they collect always comes with “noise”, which is irrelevant, incoherent, or false data. In simulation, however, sensors are not prone to the same issues. Because a simulation aims to resemble the real world as closely as possible, synthetic noise must be added to the system. Due to the formulaic nature of parametric simulations, they often require synthetic noise. Synthetic noise can be either parametric, based on various relevant equations [8], or nonparametric, derived from a machine learning model, as mentioned in Section 2.2.2.

2.3 Computer Graphics

2.3.1 OpenGL

OpenGL is an application programming interface and programming standard for rendering graphics and images and is commonly used for video games, visualization, CAD software, and more. OpenGL is particularly powerful due to its ability to run commands on the graphics processing unit, or GPU, allowing for fast and efficient graphics rendering [27]. The OpenGL graphics pipeline consists of multiple stages, each of which is responsible for running various calculations on the GPU to render an image. At each stage, small programs known as shaders, perform the appropriate calculations for each stage. For OpenGL, these shaders are written in the OpenGL Shading Language, or GLSL [28].

The OpenGL pipeline mostly utilizes two primary shader stages, the vertex shader stage, and the fragment shader stage. The vertex shaders are responsible for processing every vertex in the scene, while the fragment shader is responsible for determining the color value for each pixel on the screen [29]. As seen in Figure 2.3, the vertex and fragment shaders work together to render a multi-colored triangle.

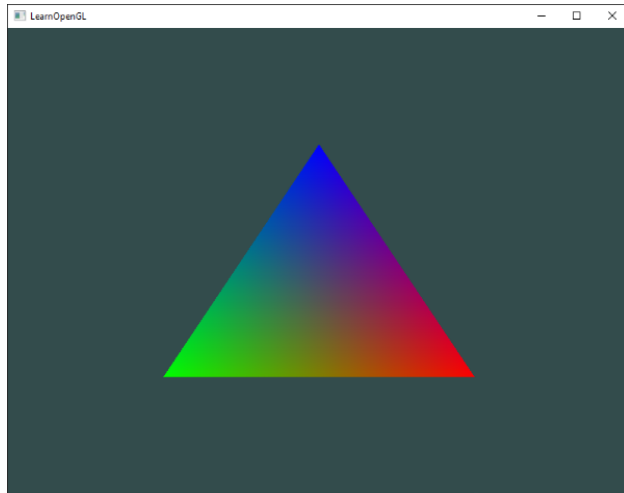


Figure 2.3: The vertex shader specifies three vertices that make up the triangle, while the fragment shader colors each vertex and all of the pixels in between [27].

2.3.2 Compute Shaders

A compute shader is one of the shader stages available in OpenGL. Most other shader stages serve a specific purpose; as mentioned above in Section 2.3.1, vertex shaders are responsible for processing vertices and fragment shaders are responsible for calculating colors. The compute shader, however, is different in that it serves no specific purpose. The advantage of a compute shader is its ability to run arbitrary calculations and leverage the processing power and parallelization of the GPU [30]. Therefore, compute shaders are incredibly useful for simulations [4, 31, 32]. In practice, a compute shader operates by processing data from an input buffer, and then generating results to an output buffer. For instance, in ApolloSim, the input buffer is populated with information about the sensor and environment, and the output buffer is populated with synthetic lidar data.

2.3.3 Ray Tracing

Ray tracing is a rendering technique that is often used to simulate the movement of light in an environment. Traditionally, many images are rendered through rasterization, which determines which color each pixel is based on objects in the scene. Rasterization also often employs lighting models that attempt to emulate realistic lighting [33]. Ray tracing, however, actually attempts to simulate the beams of light that reflect and refract throughout the scene and into the camera. To do this, a beam is cast from each pixel on the screen, as seen in Figure 2.4, and bounced around the scene; as the beam bounces through the scene, it collides with various objects, picking up colors along the way.

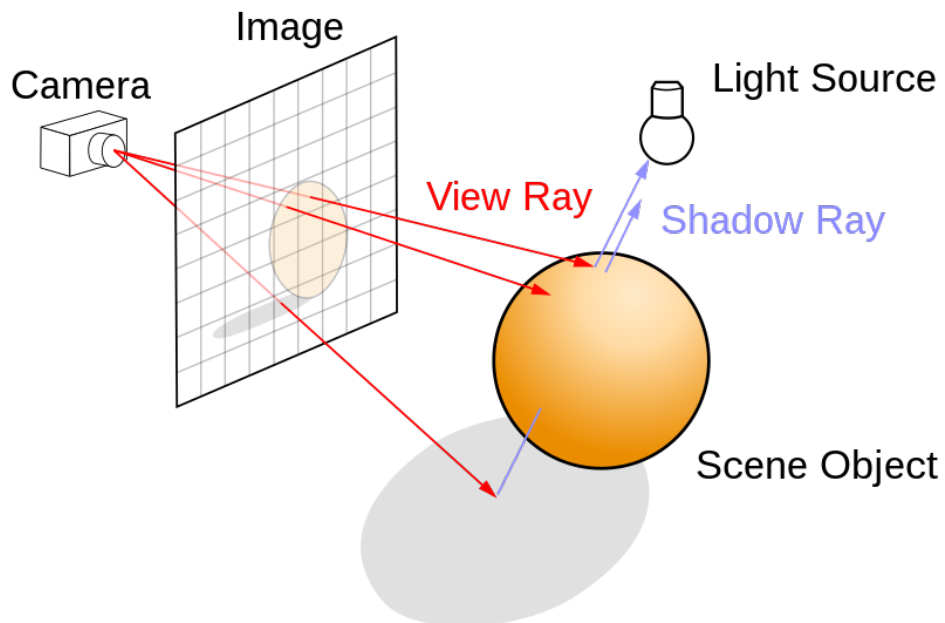


Figure 2.4: Beams of light are cast from each pixel and reflected off of items in the scene [34].

Eventually, the beams stop reflecting and a color is determined for each pixel. Although more expensive than traditional rendering techniques, ray tracing can vastly improve the realism of a virtual scene [35]. In the context of ApolloSim, ray tracing techniques are used to simulate the movement of light beams emitted by the lidar sensor.

2.3.4 Bidirectional Reflectance Distribution Models

A bidirectional reflectance distribution model, or BRDF, is an equation used to determine how much light reflects off of an opaque surface. BRDFs are often used in computer graphics applications and are incredibly useful for physics-based or photorealistic rendering. The fundamental concept of a BRDF is simple. Given an incident angle, ω_i , and a reflected angle, ω_r , return the percentage of light that is reflected.

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} \quad (2.3)$$

In the BRDF equation (Equation 2.3), $\frac{dL_r(\omega_r)}{dE_i(\omega_i)}$ represents the ratio of reflected radiance to incident irradiance. $dL_r(\omega_r)$ is the differential radiance, which is the amount of light power that is emitted or reflected by a surface given a reflected angle ω_r , while $dE_i(\omega_i)$ is the differential irradiance, the amount of light power that is incident on a surface given an incident angle ω_i . The BRDF equation is incomplete without a reflectance model, which is

how the radiance ratio is calculated [36].

There are numerous reflectance models available, and each one is useful for different materials and scenarios. For example, the Lambertian model, shown in Equation 2.4, is a simple model used for ideal matte materials that are assumed to scatter light equally in every direction. Therefore, the Lambertian model only accounts for the incident angle (L) and reflected angle (N) of the beam, as well as the color C and initial intensity I_L of the light [37].

$$B_D = L \cdot N C I_L = \cos(\alpha) I_L \quad (2.4)$$

The Lambertian model is capable of representing a simple, ideal material, and of course, reflectance models can become much more sophisticated. The Cook-Torrance model is an example of a microfacet model, which considers the surface of a material to be composed of many tiny, flat facets, which each reflect and obscure light. The model incorporates many physics approximations of how light interacts with these types of surfaces to account for the microfacets.

$$r_s = \frac{D * G * F}{4 * (\vec{n} \cdot \vec{l}) * (\vec{n} \cdot \vec{v})} \quad (2.5)$$

The Cook-Torrance model, as shown in Equation 2.5, models the complex interaction of light with surfaces through three primary components: the microfacet distribution (D), which accounts for the orientation and density

of microfacets; geometric shadowing (G), addressing the occlusion of light as it interacts with these microfacets; and Fresnel reflectance (F), quantifying reflectance due to the Fresnel effect (see Appendix A.5 and Appendix A.6 for more information). There are many different reflectance models, and each has its own use cases [38].

Related Work

3.1 HELIOS++

In a paper called “Virtual laser scanning with HELIOS++: A novel take on ray tracing-based simulation of topographic full-waveform 3D laser scanning” [1], a virtual laser scanning simulator (a lidar simulator), called HELIOS++, is introduced. It is an open-source, general purpose lidar simulation tool, which is capable of performing static and mobile terrestrial lidar simulation and aerial lidar simulation. The creators of HELIOS++ explain that users can use HELIOS++ to create datasets, test their robot algorithms, as well as determine the requirements for their lidar sensor. Winiwarter et al. stresses the importance of a realistic simulation environment in addition to a robust lidar simulation. While HELIOS++ does an excellent job at implementing various lidar sensors for myriad scenarios, it only uses random, parameterized noise for the sensor data.

3.2 A GPU-accelerated framework for simulating LiDAR scanning

In “A GPU-accelerated framework for simulating LiDAR scanning” [4], Lopez et al. introduce a lidar simulation tool that utilizes a GPU, using OpenGL compute shaders, for the computation. They present a parameterized lidar simulation tool that is specifically built to generate large datasets for training neural networks. For the virtual environment, they created their own virtual forest from scratch, and labeled the scene so that the data can be used for semantic segmentation. Additionally, they use a BRDF for each model in the scene in order to determine the intensity of the backscatter from the beam. They explain that they use a different reflection model for each object in the scene, depending on which BRDF model is most suitable. For example, they used the Cook-Torrance model, a BRDF model designed for both specular and diffuse materials, for buildings in their scene, as buildings consist of both diffuse and specular materials. Another example is the Oren-Nayar model, which is designed for rough, diffuse materials, is applied to tree canopies to accurately represent their textured and diffuse characteristics.

3.3 BlenSor: Blender Sensor Simulation Toolbox and BIAInder Range Scanner

BlenSor is an extension for Blender, a widely used 3D computer graphics software; BlenSor is capable of simulating various sensors within Blender, including lidar sensors [5]. BlenSor excels in simulating realistic sensor properties, such as sensor noise and physical effects such as reflection, allowing for realistic data generation. A major point Gschwandtner et al. make is that the sensor simulation is closely related to ray tracing techniques used in computer graphics. Additionally, they explain how they have implemented parametric noise using a normal distribution. The mean and standard deviation of the normal distribution is dependant on the material of the object.

BIAInder is another Blender-based simulator. BIAInder was specifically developed for generating labeled point cloud datasets for semantic segmentation models. The advantage of using BIAInder is the ability to quickly generate large labeled point cloud datasets, as opposed to manually labeling an existing dataset. The synthetic noise in BIAInder accounts for various physical phenomena, such as atmospheric conditions and reflection and refraction. A Gaussian distribution is used to sample the various parametric noise models they employed [18].

3.4 Lidar Simulation for Robotic Application Development

A PhD paper titled “Lidar Simulation for Robotic Application Development: Modeling and Evaluation” [24] describes a new lidar and robotics simulator meant for a robotics course at Carnegie Mellon. Tallavajhula describes the architecture of the simulator, providing a detailed description of how they structured the simulator. The simulator is split into three parts: sensor modeling, scene generation, and simulator evaluation. This simulator takes a parametric approach to the sensor models, collects data from the real world, and then uses distribution regression to match the parametric model to the real world data.

A key difference between this paper and ApolloSim is how each approach handles synthetic noise. “Lidar Simulation for Robotic Application Development: Modeling and Evaluation” uses a non-parametric approach based on noise observed in a variety of environments. This approach is different than that of ApolloSim’s, which uses a parametric approach to synthetic noise based on noise observed from a variety of materials.

3.5 LiDARsim: Realistic LiDAR Simulation by Leveraging the Real World

In a paper about a simulator titled “LiDARsim”, a virtual lidar sensor simulator that uses a data-driven, deep learning approach to synthetic sensor noise [9]. LiDARsim utilizes a traditional ray casting approach to lidar simulation, but then introduces noise with a deep learning model. The authors observed that real lidar data typically had about 10% fewer points than the simulated lidar; this is due to raydrop, which is when a beam is cast by the sensor but never returns. Raydrop is introduced to the simulation via a deep learning model that can predict when a ray is likely to be dropped. The resulting data showed that LiDARsim significantly outperformed Carla [39], a popular open-source simulator for autonomous driving research. This proves that leveraging real data is imperative in closing the gap between simulation and real-world environments.

3.6 Learning to Simulate Realistic LiDARs

The paper “Learning to Simulate Realistic LiDARs” introduces a novel method for converting RGB images into lidar data, addressing the challenge of accurately simulating lidar sensors. The authors highlight that traditional simulators often fail to capture crucial aspects such as intensity values and raydrop, due to the complex nature of environmental factors like material

reflectance and intricate geometries. They explain that many simulators use basic lidar models that generate simple point clouds through raycasting, which often does not include the nuances and intricacies of real lidar data. More specifically, many basic lidar models do not properly simulate ray drop or intensity values. This paper proposes a data-driven approach capable of simulating nuanced details and complex lidar data. Given an RGB image, their model can predict which rays will be dropped as well as the intensity values of the backscattered rays that are not dropped. The goal of this paper was to overcome the limitations of existing simulators by offering accurate predictions to raydrop and intensity values, which are essential for realistic lidar simulations [10].

Implementation

4.1 Simulator Overview

4.1.1 Simulator Pipeline

ApolloSim consists of three sequential processes: sensor calibration, lidar data calculation, and a visual display. The pipeline of ApolloSim, shown in Figure 4.1, starts with the calibration process. The calibration process allows the user to gather data using their target sensor to calibrate ApolloSim. First, a lidar sensor is placed in front of a benchmark material in order to collect data. The user can collect data from multiple benchmark materials. The data is parsed and organized so that the simulation engine can process it. Various key metrics, such as the average beam intensity, are calculated by the simulation engine. After the noise for each material is characterized, the engine can begin to simulate a virtual environment. Once the virtual environment is determined, ApolloSim can calculate the collision points of the beams emitted from the lidar with the various objects in the environment, allowing for the creation of a synthetic point cloud. Using the calibration data, ApolloSim can mimic the noise in the real data in the simulated data. After the point cloud is calculated, the graphics engine renders a visual rep-

resentation of the virtual environment, as well as the beams emitted by the lidar. Each stage of the pipeline will be discussed in detail in further sections.

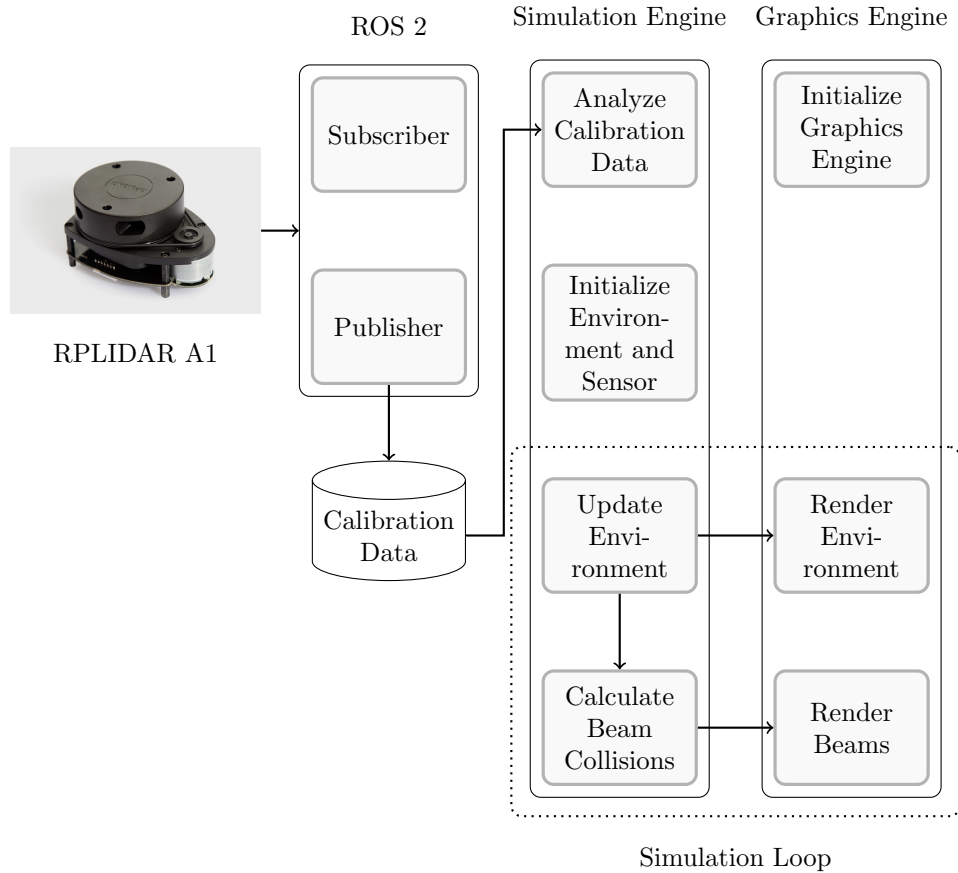


Figure 4.1: A detailed overview of the ApolloSim Pipeline

4.1.2 Technology Stack

ApolloSim is mainly implemented in a language called Odin. Odin is a high-performance, simple, and data-oriented programming language. A key advantage of Odin is its integration of graphics APIs, such as OpenGL and Vulkan; very minimal setup is required to work with these APIs compared to C++. In addition to Odin, the OpenGL shading language, GLSL, is

utilized for computation on the GPU. OpenGL shaders are written in GLSL; ApolloSim’s graphics engine uses vertex and fragment shaders to render the virtual environment and compute shaders to calculate the lidar beams. The beam calculation is expensive and similar to a ray tracing algorithm, which typically requires parallelized computation on a powerful device such as a GPU.

The Slamtec RPLIDAR A1 sensor, as depicted in Figure 4.2, was used for evaluating ApolloSim. ApolloSim utilizes ROS 2 [40], a robotics library, to interface with the sensor and collect data from it. The ROS 2 portion of ApolloSim consists of a Python package capable of collecting the lidar data and publishing it to a file, as well as the RPLIDAR ROS 2 SDK [41], which is used to connect to the sensor.



Figure 4.2: The Slamtec RPLIDAR A1 is a 360-degree, two-dimensional lidar with a scanning range of 0.15 to 12 meters, and a 3960Hz sample frequency [42].

4.2 Sensor Calibration

4.2.1 Sensor Calibration Overview

The novelty of ApolloSim is the process in which it adapts its synthetic noise to any sensor. In order to generate accurate synthetic noise, the simulation is calibrated by a real sensor. The sensor is positioned in front of a benchmark material and captures data, which can be used to determine how the sensor responds to the given material. As depicted in Figure 4.3, the lidar sensor records data at multiple incident angles.

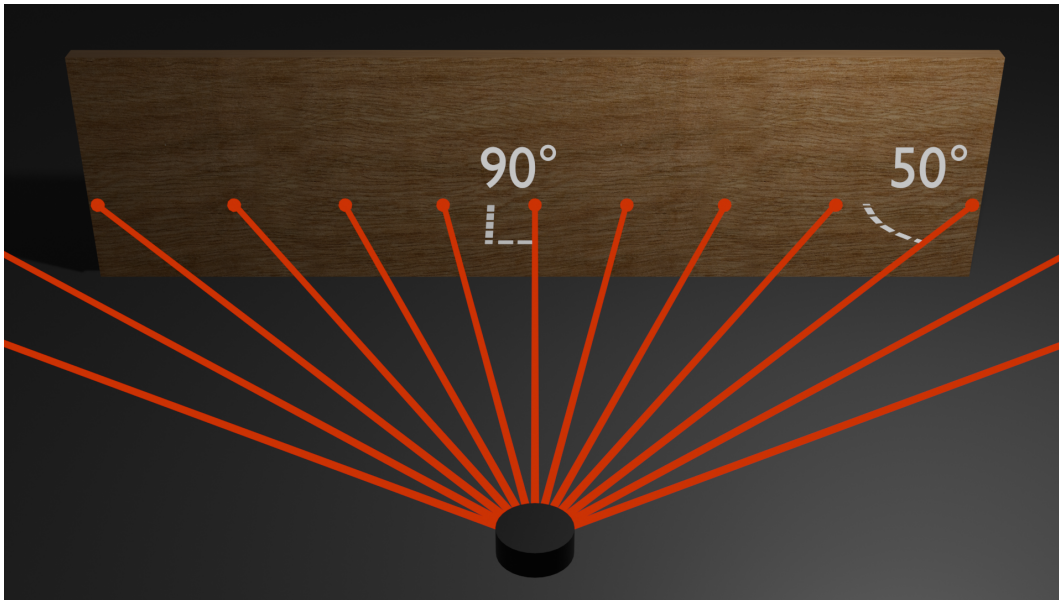


Figure 4.3: A render of a simulated lidar and wooden benchmark material. The beams (red lines) are emitted from the lidar (black cylinder) and collide with the benchmark material (wooden block) at various incident angles.

During the calibration process, information about how the sensor interacts with that material at each angle is analyzed. Each material is associated with a series of angles; for every angle, there is a corresponding set of points,

which are comprised of distance and intensity values. Using this data, we can calculate the standard deviation and mean of both distance and intensity values across all angles. This information defines how the lidar sensor reacts to each specific material at various angles of incidence.

4.2.2 Interfacing with the Lidar

The first phase of calibrating the simulation is collecting data from a lidar sensor, specifically the RPLIDAR A1, which was selected for experimental purposes. The Slamtec RPLIDAR A1 is a two-dimensional, 360-degree lidar sensor. The Slamtec RPLIDAR software development kit for ROS 2 was used to interface with the sensor. The RPLIDAR SDK includes a ROS 2 node, which records all incoming lidar data to a ROS 2 public data channel, also known as a topic. Then, another ROS 2 node simply listens to the topic and writes the relevant data, such as intensity values and distances, to a text file (as described in Appendix A.2). The lidar only needs to run for a few seconds to collect a sufficient amount of data for the calibration process. In addition to writing the lidar data to a text file, the ROS 2 node also spawns a live, overhead display of the data, as seen in Figure 4.4; this display helps the user in setting up their calibration environment.

When the user launches the ROS 2 node, they are prompted for the name of the benchmark material they are calibrating, the distance between the sensor and the material, and the width of the material. After this information

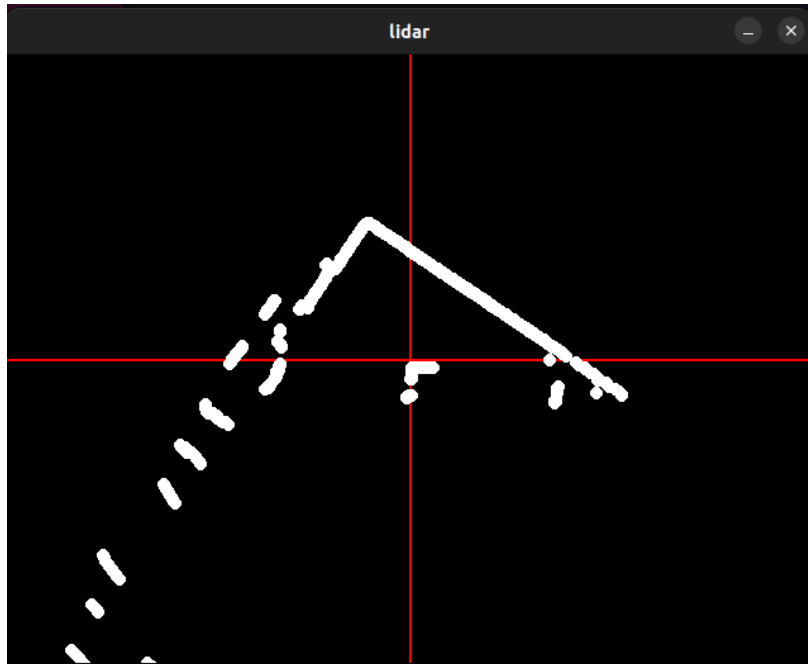


Figure 4.4: The lidar is positioned at the center of the screen, where the two red guidelines intersect. The white dots are points in the point cloud that are gathered by the sensor.

is provided, the live display adds guidelines to show which parts of the data intersect with the benchmark material.

4.2.3 Sensor Calibration Implementation

In order to collect data on how the sensor interacts with a specific material, the user runs a calibration script. The calibration script is given the a few inputs specified by the user. First, the user must specify the details of the benchmark material, as listed in Table 4.1. The user can specify these details in a small GUI, as shown in Figure 4.6.

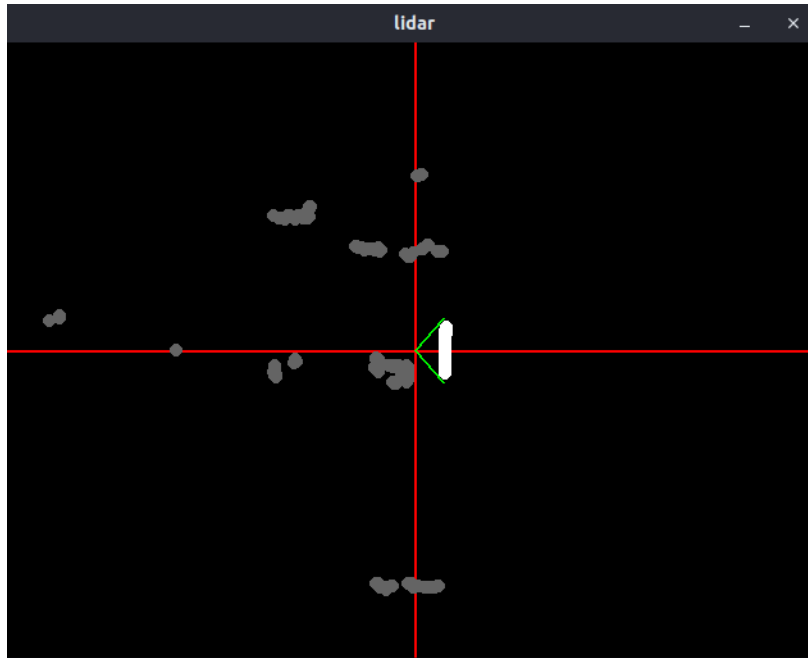


Figure 4.5: The green lines show the field of view of the sensor that senses the benchmark material. The white points intersect with the benchmark material and will be used for the calibration, while the grey points are the rest of the points that are subsequently dropped.

Input	Description	Data Type
Material Name	The name of the material that the user is calibrating for	string
Benchmark Distance	The distance between the sensor and the benchmark material	float
Benchmark Width	The width of the benchmark material	float
BRDF Type	The type of BRDF to use: either Oren-Nayar or Cook-Torrance	enum
Roughness	Roughness index of the material	float
Index of Refraction	Index of refraction of the material	float

Table 4.1: Inputs relevant to Benchmark Material for Sensor Calibration

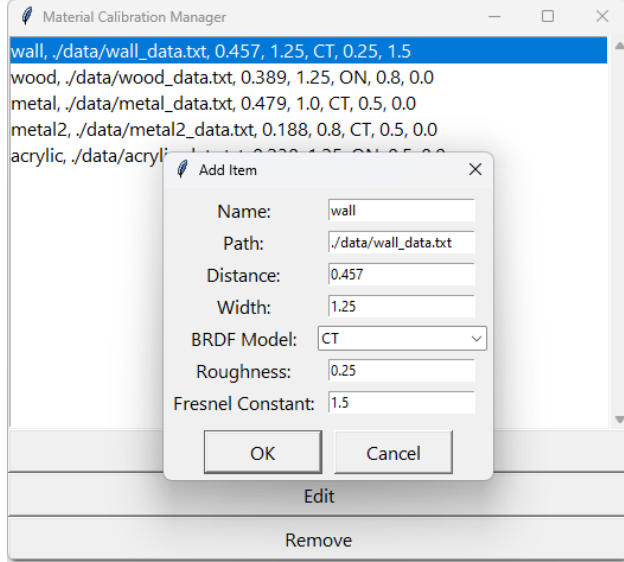


Figure 4.6: The user can use a small tool to add, edit, and remove calibration materials.

The width and distance must be specified so that the calibration process can determine which beams are hitting the benchmark material. The valid beam angles are calculated by determining the maximum angle at which the lidar’s beams are still making contact with the benchmark material. In Equation 4.1, the data from the lidar is filtered to only include the beams that connect with the benchmark material, where w is the width of the benchmark material and d is the distance between the sensor and the benchmark material.

$$\text{Valid Angles} = \{\theta \in \text{Angles} \mid \theta < \arctan(\frac{w/2}{d})\} \quad (4.1)$$

Once the irrelevant angles are filtered out, the corresponding lidar data is dropped. The remaining lidar data is compiled into a list of a custom data structure named `AngleData`, which contains important information about the

beams cast at a specific angle, as presented in Table 4.2.

Attribute	Description	Data Type
Angle	The angle of the beam for which the data represents	float
Intensities	List of beam intensity values collected at this angle	float array
Mean Intensity	Mean of the beam intensities	float
Intensities Standard Deviation	Standard deviation of beam intensities	float
Distances	List of beam distances collected at this angle	float array
Mean Distance	Mean of beam distances	float
Distances Standard Deviation	Standard deviation of beam distances	float
Drop Rate	The percentage of beams that never return to the sensor at this angle	float

Table 4.2: The `AngleData` Data Structure

A simple algorithm iterates through every beam cast by the lidar and creates an `AngleData` object for each unique angle; after every beam is sorted, the means, standard deviations, and drop rates are calculated for each angle. This data will be used later on in order to determine how the beams interact with each material in the simulation.

4.3 Graphics Engine

4.3.1 Graphics Engine Overview

To provide visual feedback for the simulation, `ApolloSim` has a graphics engine that can render the lidar sensor and its surrounding environment in real time. Additionally, the engine is able to display the beams that are cast by the virtual lidar sensor. The graphics engine relies on the OpenGL APIs

and shaders for rendering. While quantitative analysis is important in a lidar simulation, a visual, qualitative analysis is helpful as well.

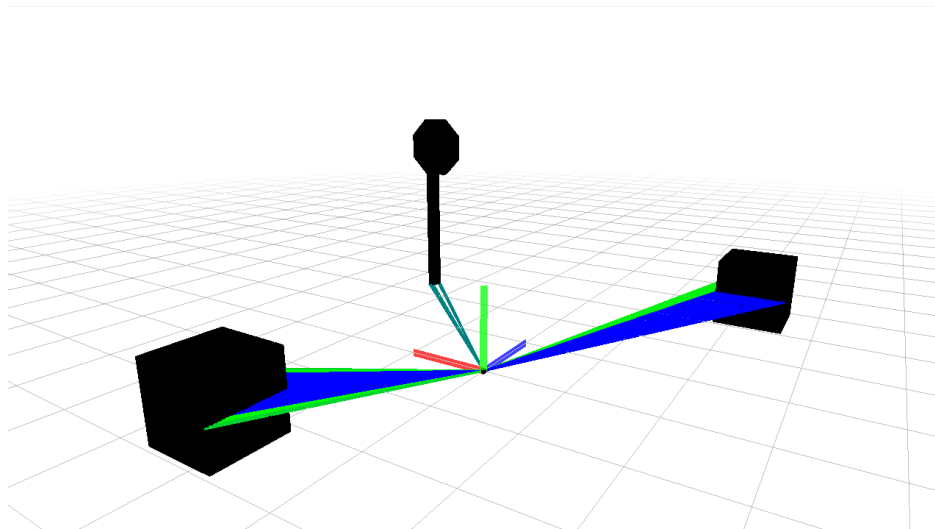


Figure 4.7: A screenshot of ApolloSim’s graphics engine. The small cylinder in the middle represents the lidar sensor, and the environment is made up solely of two cubes and a stop sign.

The graphics engine is relatively bare-bones, with support for a few essential features. The engine offers support for three-dimensional meshes that are either primitive shapes or custom geometry, as well as custom textures. The user has the ability to adjust their viewpoint and navigate through the virtual environment, enabling them to observe the simulation from any preferred perspective.

4.3.2 Beam Rendering

The graphics engine is provided with a list of points that belong to the point cloud generated by the virtual lidar. Each beam is rendered as a line, where the first point is the origin of the beam, the lidar, and the second point

is the collision point of the beam. These beams are rendered as lines that range from green to blue, where green represents a weaker intensity, and blue represents a stronger intensity.

4.3.3 Calibration Data Rendering

In addition to the real-time virtual environment render, ApolloSim also has a feature that allows the user to view the calibration data for a certain material. When ApolloSim is launched with the argument `viewer` and a file path, the engine will analyze the specified file and display a static view of what the calibration data. This allows the user to see a visual representation of how the sensor interacts with a specific material.

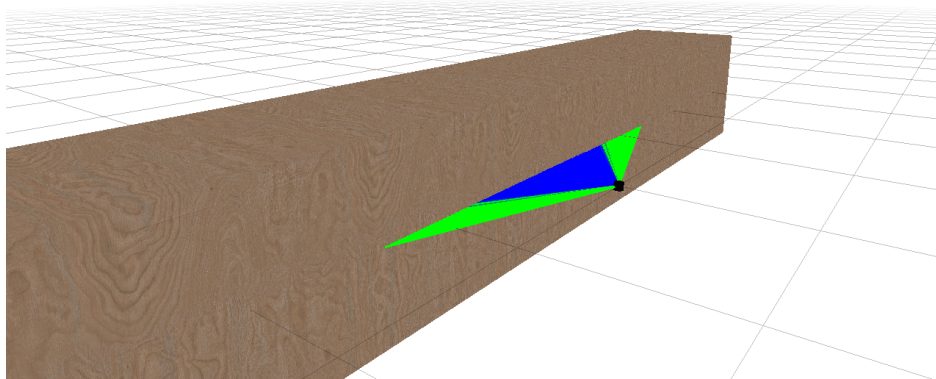


Figure 4.8: The static wall represents the benchmark material, and the beams represent the data generated by the calibration.

The color of each beam is determined by its mean intensity value for a specific angle. The cube is positioned at the same distance from the virtual sensor as the actual real sensor is from the benchmark material, allowing the user

to explore the data for a benchmark material.

4.4 Simulation Engine

4.4.1 Simulation Engine Overview

In ApolloSim, the simulation engine is responsible for calculating synthetic lidar data. This process begins after calibration data is acquired, at which point the simulation engine employs a virtual model of the lidar in a virtual environment to generate lidar data. The inputs to the simulation engine are the lidar sensor specifications and the virtual environment, while the output is a point cloud. The virtual lidar sensor is configured based on the user's real lidar sensor, while the virtual environment can be programmatically configured to resemble an environment in which the lidar will be tested.

4.4.2 Simulation Engine Inputs

The user must input the specifications of the lidar sensor, as listed in Table 4.3. For example, if the user was testing with a Slamtec RPLIDAR A1, the virtual sensor would be programmed to have the same specifications of a 0.15-meter to 12-meter working range, a 360-degree angular range, a scan frequency of 5.5 KHz, and a sample frequency of 3960 Hz. The sensor specifications are important for determining the quantity and positions of the beams cast by the virtual lidar.

In addition to the lidar specifications, the user must specify an environ-

Input	Description	Data Type
Scan Frequency	Total complete environment scans per second	integer
Sample Frequency	Total beams cast per second	integer
Angular Range	Maximum span of angles detectable by the sensor	float
Working Range	Maximum and minimum distance reliably detectable by the sensor	float[]
Sensor Data Dimensions	Whether the sensor collects one, two, or three-dimensional data	int

Table 4.3: Parameters for the virtual lidar sensor

ment in which to test the lidar. Unlike many graphics engines that support real-time manipulation of three-dimensional environments or environment setup via configuration files, ApolloSim requires environments to be configured through code. The user can add primitive shapes as well as custom models in the Wavefront .obj file format.

4.4.3 Lidar Beam Calculation

Lidar Beam Calculation Overview

In order to generate a point cloud, the engine must calculate every beam cast by the lidar and where they intersect with the environment. Some lidar sensors are capable of emitting over one million beams per second [43]; therefore, an efficient computation method is necessary. Parallelizing beam calculation is the primary method for making ApolloSim more efficient. In order to parallelize these calculations, all beams are calculated on the computer’s GPU. A compute shader, as described in Section 2.3.2, is responsible for these calculations.

Compute Shader Buffers

The first step in the lidar beam calculation process is gathering the relevant data, as all the data for these calculations will eventually be stored in a buffer that can be accessed by the compute shader. Important data such as details about the environment, the directions and quantity of the beams cast by the lidar, material data, and more, are sent to the compute shader for calculation. Once all the relevant data is gathered, it is stored in a few buffers, as listed in Table 4.4, that can be accessed by the compute shader. The compute shader reads those buffers, and can now perform calculations on the GPU. The compute shader loops through every beam direction in order to calculate where it intersects with the environment; for more information, see Appendix A.4.

Lidar Beam Intersection Point Calculation

The points at which the beams intersect with the virtual environment are what make up the point cloud generated by the virtual lidar. In order to calculate these points, an algorithm similar to ray tracing is employed. Every beam that is cast by the lidar is checked against every item in the scene, and the closest point of intersection between the beam and the environment is determined to be the intersection point. Once the intersection point is found, the next step is to find the intensity of the beam's backscatter. This is where the calibration data becomes important; using the material of the object the

Input	Description	Data Type
Simple Geometry	A list of all basic geometric shapes in the scene, including details about their material and transformation matrix	SimpleGeometry[]
Complex Geometry	A list of all complex geometric shapes in the scene, including details about their material and transformation matrix	ComplexGeometry[]
Directions	List of directions at which the lidar emits a beam	vec3[]
Vertices	List of all vertices in the scene	float[]
Indices	List of all vertex indices in the scene	int[]
Materials	List of all materials used in the scene	Material[]
Seeds	List of random numbers to be used as seeds for sampling distributions	float[]
AngleData	List of information about the lidar calibration, including every angle for every material	AngleData[]
Output	Description	Data Type
Output	A list of vectors that contain the positions and intensities of the points in the point cloud	vec4[]

Table 4.4: The compute shader accepts a number of inputs and generates an output. The inputs are required to calculate the lidar beams, while the output is essentially a point cloud.

beam intersected with, along with the angle at which they intersected, the compute shader can use the calibration data to determine the intensity of the beam.

4.4.4 Synthetic Noise Calculation

Once each point is calculated, synthetic noise must be injected into the data. As displayed in Figure 4.9, adding noise consists of potentially assigning a new intensity value and a new location vector for each point.

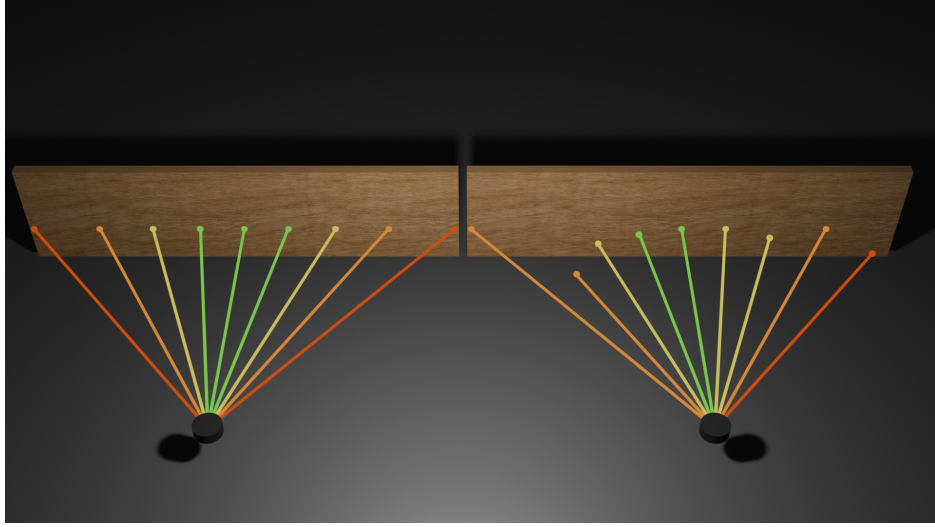


Figure 4.9: The data from the lidar on the left has no noise at all; each location vector and intensity value is exactly as expected. The data from the lidar on the right is noisy; some of the locations and intensities of the points are incorrect.

Calculating the new distance value is simple. The distance value determined by the parameterized lidar model is modified by adding a random value. The random value is determined by the incident angle and material that the beam collides with. Using the standard deviation of the distances for the incident angle and material, a random value is sampled from a normal distribution and added to the distance, as shown in Equation 4.2.

$$D_{noisy} = D_{parameterized} + \mathcal{N}(0, \sigma_{angle,material}) \quad (4.2)$$

Calculating the new intensity value is similar. The intensity value is determined by the mean and standard deviation of intensity values for a given incident angle and material. The intensity value is also sampled from a normal distribution, as shown in Equation 4.3.

$$I_{noisy} = \mathcal{N}(\mu_{angle,material}, \sigma_{angle,material}) \quad (4.3)$$

For instance, consider that at an incident angle of 30 degrees on a wooden surface, the lidar captures the following set of distance values: [1.0, 1.1, 0.9, 1.25], along with a matching set of intensity values: [0.75, 0.7, 0.8, 0.0]. Now in simulation, suppose a beam collides with a wooden material at 30°, with a distance of 5.0. Given the standard deviation of distance values to be 0.129, the standard deviation of intensity values to be 0.327, the mean of intensity values to be 0.563, and the drop rate to be calculated as 0.25, the new distance would be $5.0 + \mathcal{N}(0, 0.129)$, the new intensity value would be $\mathcal{N}(0.563, 0.327)$, and the beam would have a 25% chance of never returning due to ray drop.

While calibrating materials, only a subset of all incident angles can be measured, as shown in Figure 4.10. In simulation, there is a possibility of a beam colliding with a surface at an incident angle that was not measured during the calibration process. If this occurs, the BRDF model that was specified by the user in the configuration file for the material is used to determine the intensity of the beam.

4.4.5 BRDFs for Addressing Data Gaps

To calculate the intensity of backscattered light using a BRDF model, the angle of incidence must be determined. The angle of incidence is calculated

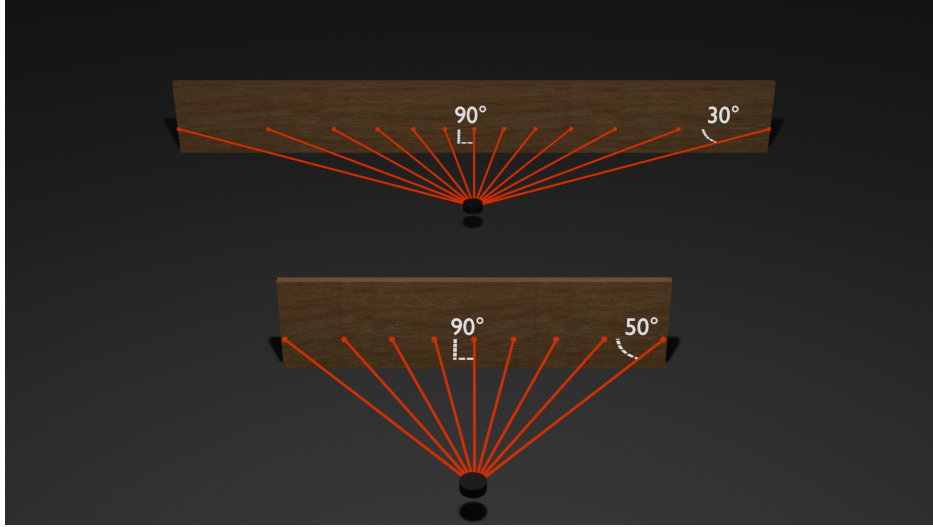


Figure 4.10: This render shows two benchmark materials of varying lengths; there are more collisions between the lidar beams and the wider benchmark material, and therefore, more angles of incidence.

given the light vector (L) and the normal vector of the surface (N), as shown in Equation 4.4

$$\theta_i = \arccos(\vec{L} \cdot \vec{N}) \quad (4.4)$$

The Oren-Nayar reflectance model, presented in Equation 4.5, accounts for the reflected angle (θ_r), the incident angle (θ_i), the azimuthal angles ($\phi_r - \phi_i$), and the roughness of the material (σ) [44].

$$L_r(\theta_r, \theta_i, \phi_r - \phi_i; \sigma) = \frac{\rho}{\pi} E_0 \cos \theta_i \left[C_1(\sigma) + \right. \\ \left. \cos(\phi_r - \phi_i) C_2(\alpha; \beta; \phi_r - \phi_i; \sigma) \tan \beta + \right. \\ \left. (1 - |\cos(\phi_r - \phi_i)|) C_3(\alpha; \beta; \sigma) \tan\left(\frac{\alpha + \beta}{2}\right) \right]$$

where

$$C_1 = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33} \tag{4.5}$$

$$C_2 = \begin{cases} \frac{0.45\sigma^2}{\sigma^2+0.09} \sin \alpha & \text{if } \cos(\phi_r - \phi_i) \geq 0 \\ \frac{0.45\sigma^2}{\sigma^2+0.09} (\sin \alpha - (\frac{2\beta}{\pi})^3) & \text{otherwise} \end{cases}$$

$$C_3 = 0.125 \left(\frac{\sigma^2}{\sigma^2 + 0.09} \right) \left(\frac{4\alpha\beta}{\pi^2} \right)^2$$

$$\alpha = \text{Max} [\theta_r, \theta_i]$$

$$\beta = \text{Min} [\theta_r, \theta_i]$$

Fortunately, for the case of a two-dimensional lidar sensor, the Oren-Nayar model can be simplified since the azimuthal angle difference is always 0, and the viewpoint is essentially located in the same place as the light source. Therefore, the Oren-Nayar model can be simplified to Equation 4.6. Now, the intensity of light that is backscattered from a diffuse material can be calculated.

$$L_r(\theta_i, \sigma) = |\cos \theta_i|(C_1(\sigma) + C_2(\sigma) \sin \theta_i \tan \theta_i)$$

where

$$\begin{aligned} C_1 &= 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33} \\ C_2 &= \frac{0.45\sigma^2}{\sigma^2 + 0.09} \end{aligned} \tag{4.6}$$

The Cook-Torrance model, which is useful for materials with a specular reflection, is defined in Equation 2.5 and discussed in Appendix A.5. Similar to the Oren-Nayar model, the Cook-Torrance model takes into account the angle of incidence, material roughness, and index of refraction to determine backscattered light.

4.5 Application Walk-Through

This section details the step-by-step process of using ApolloSim. For this example, two materials were configured for simulation: wood and epoxy.

4.5.1 Calibrating the Simulation

The first step for using ApolloSim is gathering calibration data. To do this, the user places their sensor in front of a benchmark material, as shown in Figure 4.11.

Then, they can run the calibration script by launching the ROS 2 node.

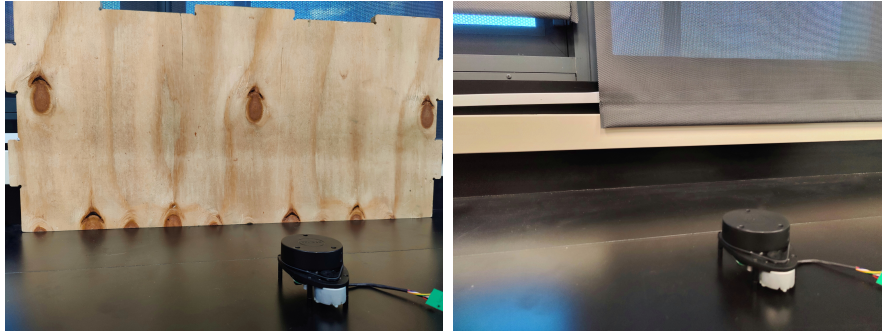


Figure 4.11: The image on the left shows the lidar in front of a benchmark material made of plywood, while the image on the right shows the lidar in front of the epoxy resin tabletop material.

The calibration script will prompt the user regarding information about the benchmark material, as shown in Figure 4.12.

```
gavri@gavri-linux:~/Code/ApolloSim/ROS2_WS/src/py_pubsub$ ros2 run py_pubsub lidar_data
Enter your material name: wood
Enter your material width: 1.0
Enter your material distance: 0.5
```

Figure 4.12: The user can specify the name, distance, and width of their benchmark material.

The calibration script then opens a visual representation, shown in Figure 4.13, of what the lidar is sensing, with green guidelines to show the user which points are expected to be colliding with the benchmark material and will be used for calibration. These guidelines are calculated using the width and distance values collected from the user, as displayed in Figure 4.12.

The sensor should run for a few seconds in order to get a good sample

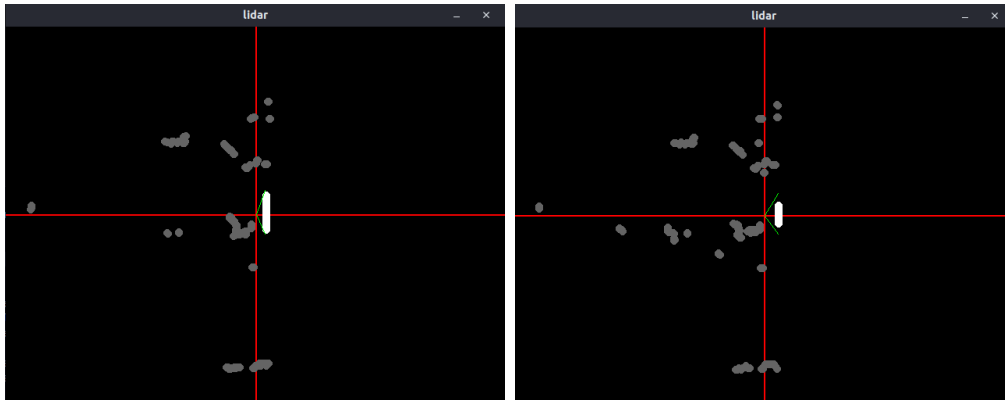


Figure 4.13: The left image is the live view of the lidar in front of the wooden benchmark material, while the right is of the epoxy benchmark material

size. Once the user decides they have gathered enough data, they can close the script, and a text file with the data is generated. The user may repeat this process for as many materials as they'd like.

4.5.2 Configuring the Simulation

In order for ApolloSim to know which calibration data to use, the user must provide a configuration file. The user can run the configuration file editor GUI and enter the epoxy and wood material configurations, as displayed in Figure 4.14. Once the configuration file is made, ApolloSim can now calibrate the simulation to properly represent those materials in the simulation.

4.5.3 Running the Simulation

Finally, the user can launch the simulation. There are two modes for ApolloSim: the viewer and the simulation. The viewer, as mentioned in Section 4.3.3, allows the user to see the calibration data for an individual

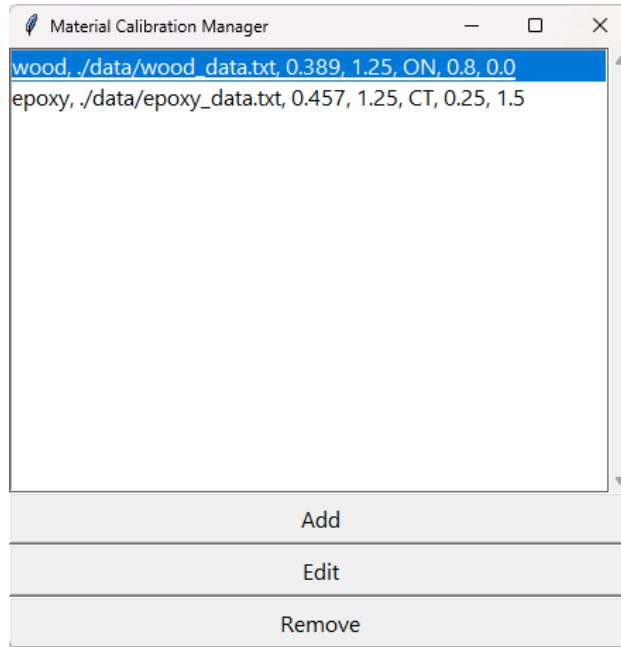


Figure 4.14: The user can enter the wood and epoxy materials into the configuration editor material. The viewer can simply be launched by adding `viewer <material name>` to the arguments while launching ApolloSim. In this case, the user can run this command: `odin run . -- ./data.config viewer wood` to see the calibration data for the wood material, and this command: `odin run . -- ./data.config viewer epoxy` to see the calibration data for the epoxy material. This allows the user to verify that their calibration data makes sense. Figure 4.15 shows the viewers for the wood and epoxy materials.

After verifying the calibration data, the user can also launch the simulation. For now, the environment must be built by modifying the code of ApolloSim; thus, in this case, a pre-built example scene is tested. The example scene shown in Figure 4.16 consists of one wooden cube and one epoxy cube. Once the user launches the simulation, they can move the camera

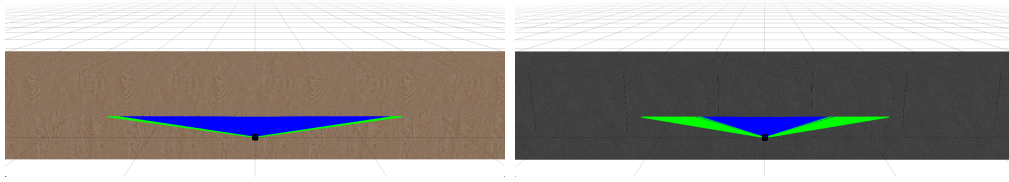


Figure 4.15: The user can view what their calibration data looks like. Of course, the rough, diffuse wood material has a much better response at larger incident angles than the shiny, specular epoxy material.

around the scene and observe the lidar data in real time.

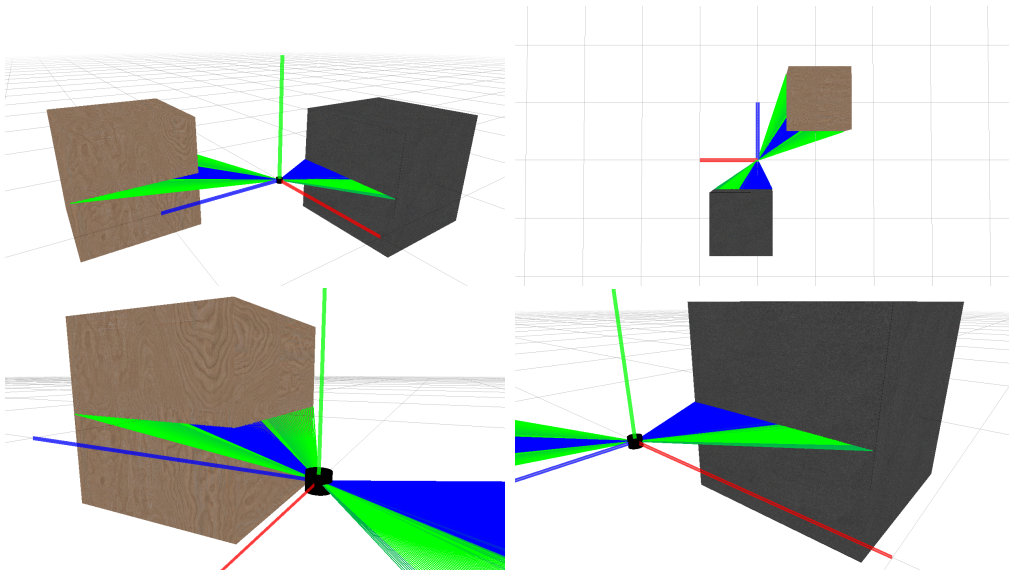


Figure 4.16: These four screenshots of ApolloSim show the simulation environment at various viewing angles and positions.

Results and Evaluation

5.1 Evaluation Overview

A lidar sensor simulation tool would be useless if it doesn't properly emulate the behavior of a lidar in the real world. To prove the authenticity of ApolloSim, several tests have been conducted. Each test shows that the data that ApolloSim generates closely resembles the data that is output by the sensor. As mentioned in Section 4.1.2, the RPLIDAR A1 was used for calibrating and evaluating ApolloSim.

5.2 Validating Key Characteristics

As described in Section 2.1.5, there are a few key characteristics that define the lidar data: raydrop rate, mean intensity, intensity variance, and distance variance. To validate that ApolloSim can properly generate data similar to the calibration data, five materials were calibrated and tested in simulation. As shown in Figures 5.1, 5.2, 5.3, and 5.4, the mean and standard deviation of intensity values, the raydrop rate, and the standard deviation of the distance values all experience similar distributions in the real lidar measurements and the simulation data.

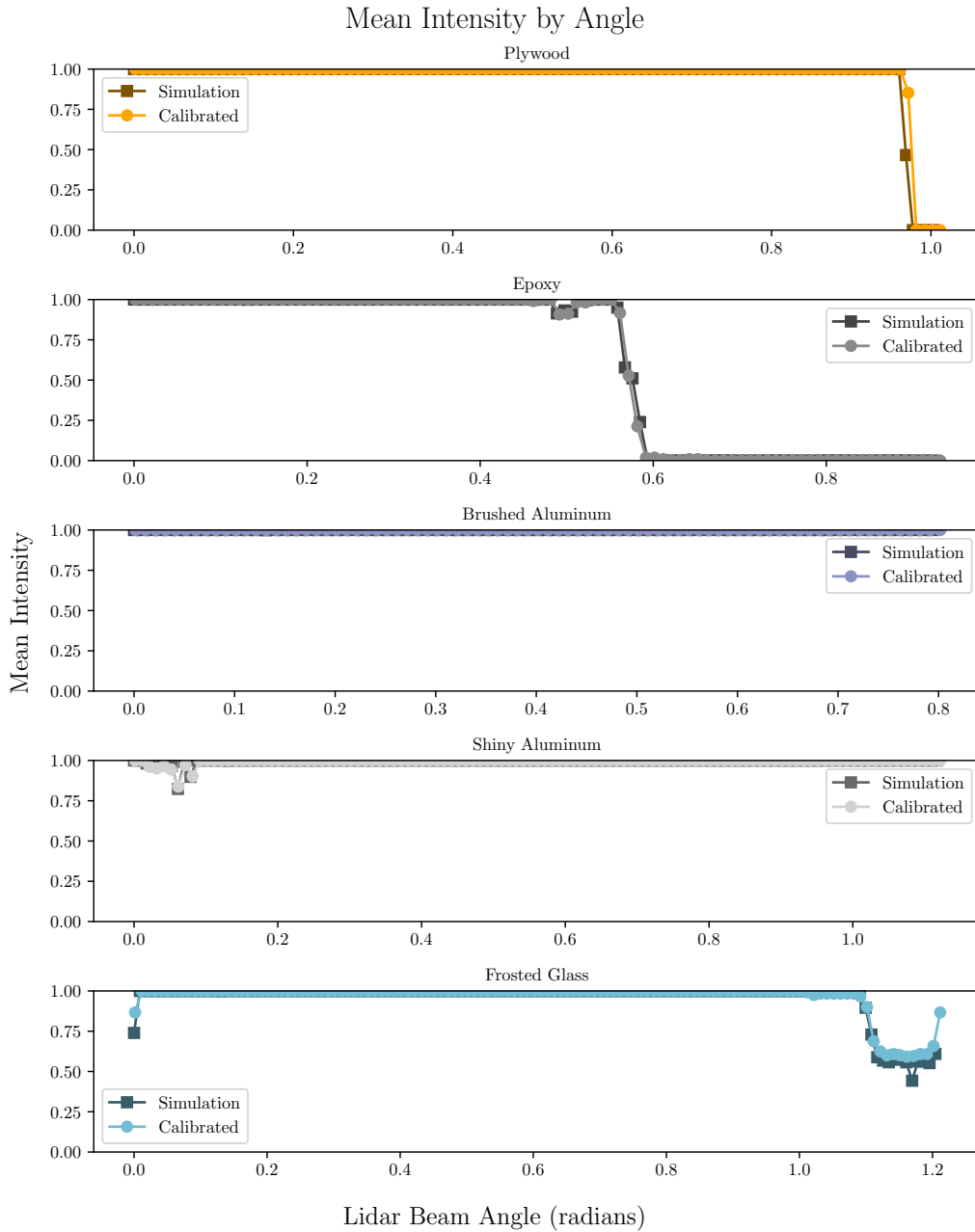


Figure 5.1: For each material, the distribution of intensity values across every angle of measurement is similar. As expected, at high incident angles, the intensity values generally drop.

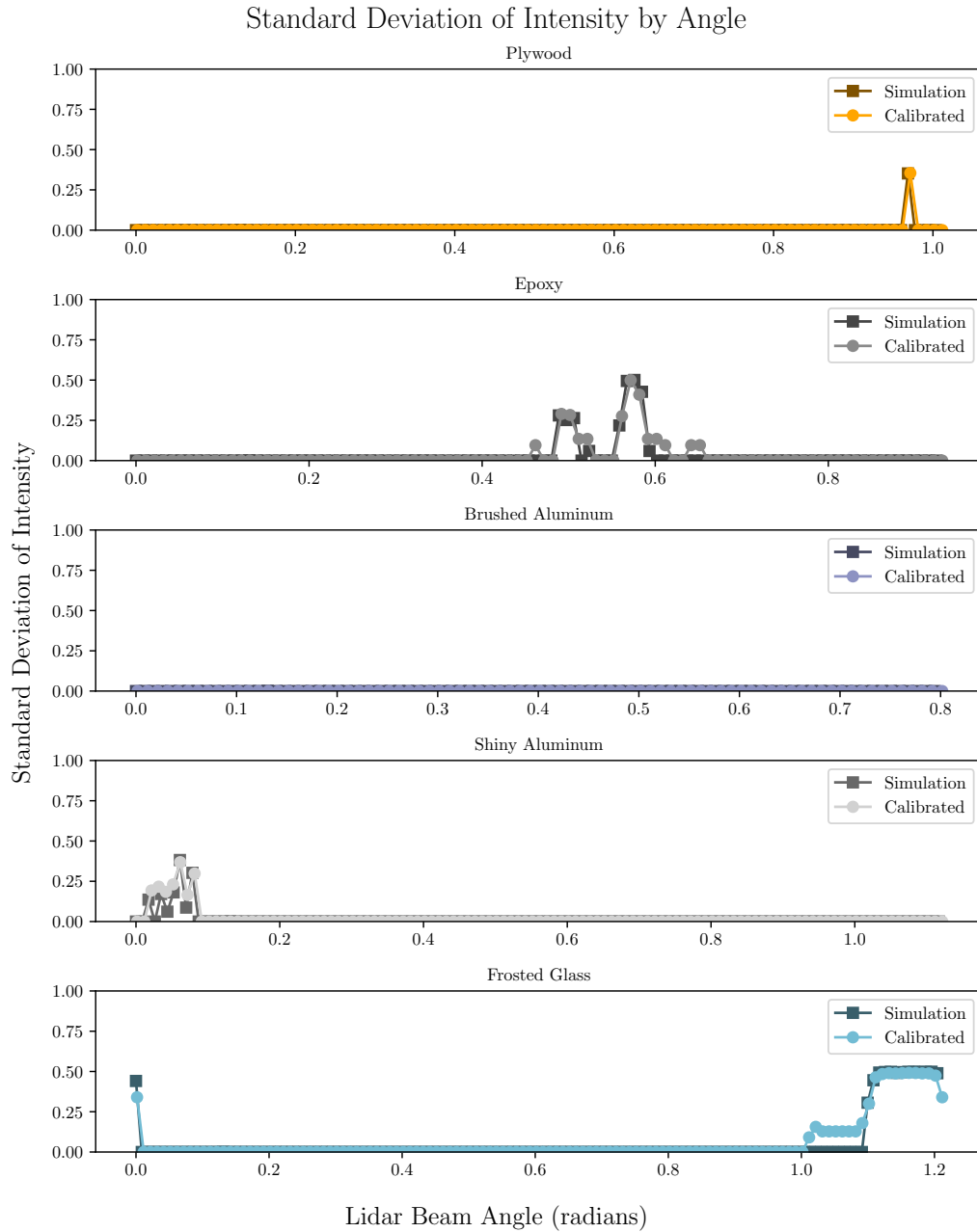


Figure 5.2: For each material, the distribution of standard deviation values for the intensity values across every angle of measurement is similar.

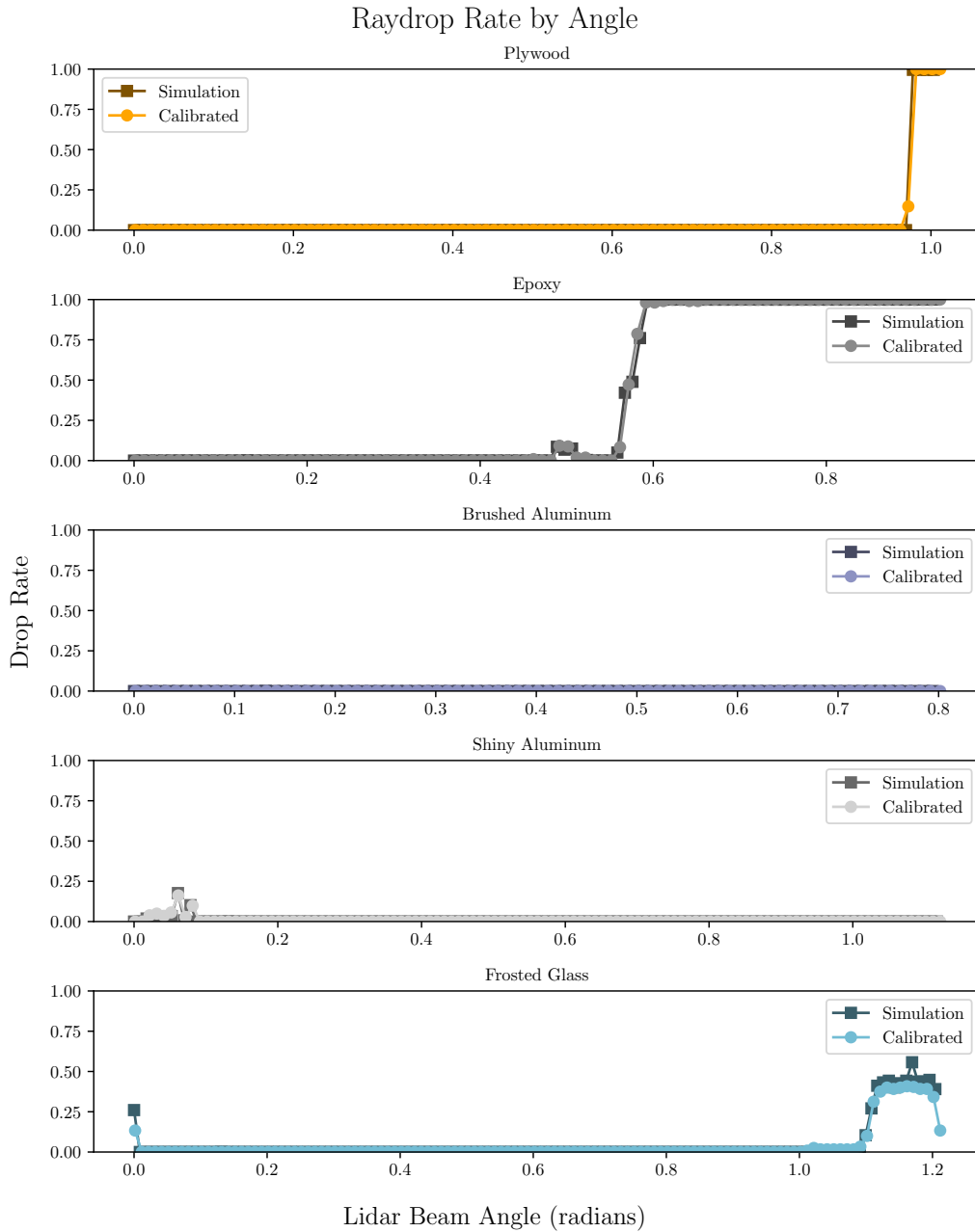


Figure 5.3: For each material, the distribution of drop rates across every angle of measurement is similar.

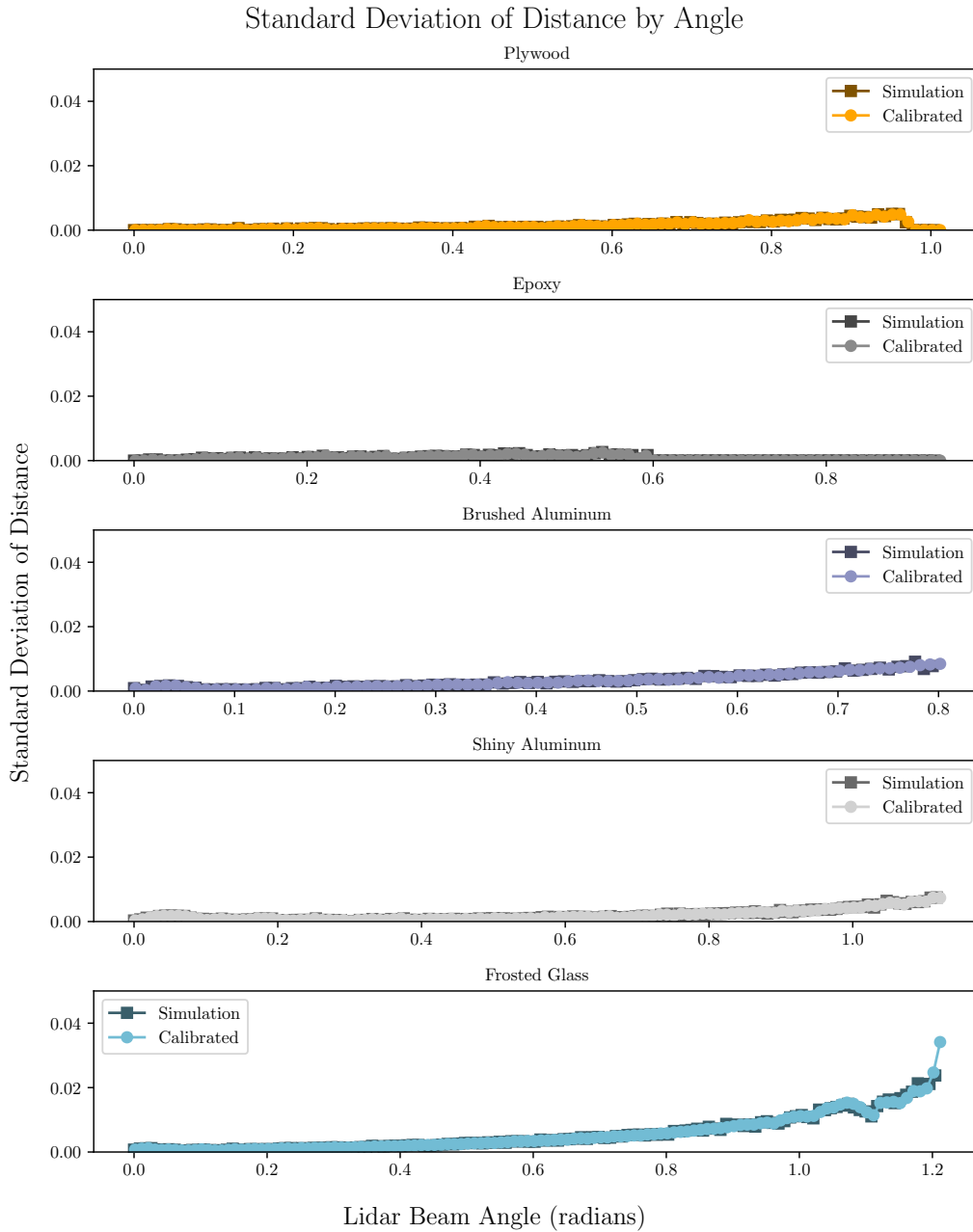


Figure 5.4: For each material, the distribution of standard deviation values for the distances across every angle of measurement is similar. The distance readings become more volatile as the incident angle increases. For the Plywood and Epoxy, the variance drops to zero when the incident angle gets too high as the rays are consistently dropped.

5.3 Improved Accuracy of ApolloSim Over Parametric Models

As mentioned in Sections 2.1.5 and 2.2.2, many lidar simulations depend on parametric models to generate realistic synthetic noise. Some parametric models apply a Gaussian distribution to randomly adjust intensity and distance values, along with raydrop [1, 23]. Other models might utilize BRDFs to simulate interactions between the lidar sensor and various materials [4]. ApolloSim incorporates BRDFs along with calibration data to generate realistic sensor noise. Figure 5.5 shows a comparison between the plywood and epoxy benchmark materials against their corresponding BRDF models. The plywood material was compared to a simulation using the Oren-Nayar BRDF, as plywood is a rough, diffuse material, while the epoxy material was compared to a simulation using the Cook-Torrance BRDF, as it is a diffuse material with a specular component.

Figure 5.5 displays a comparison between two simulation types: a standard BRDF simulation yielding values between 0 and 1, and a binary BRDF simulation yielding values of either 0 or 1. Given that the lidar sensor used to test ApolloSim only reads binary intensity values, it is important to look at both simulation types. The binary simulation should show a stronger resemblance to the calibrated noise, however, the continuous BRDF simulation offers insight into its distribution. Given a sensor with the ability to produce

continuous intensity values, the calibrated values would be more likely to have a stronger correlation with the standard BRDF simulation.

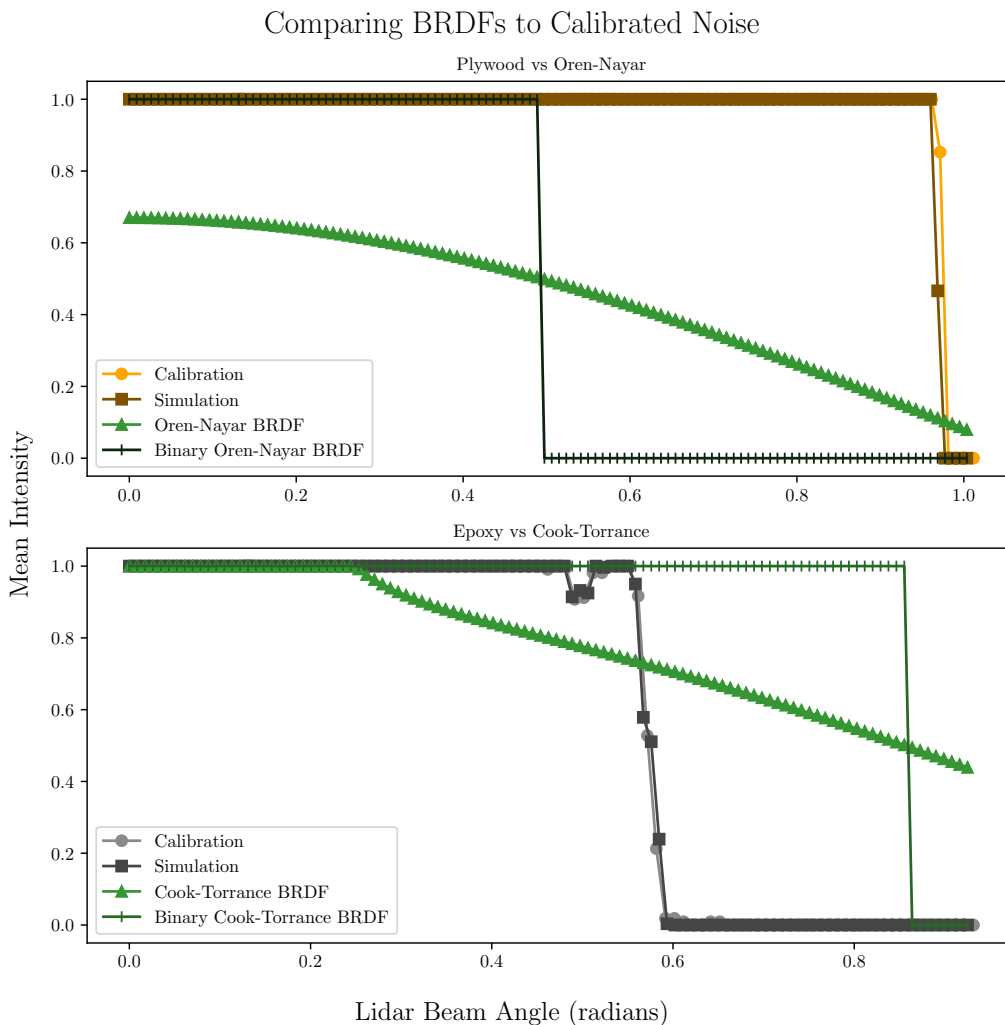


Figure 5.5: ApolloSim’s noise resembles the noise in the real world much closer than the simulations that just use BRDFs. The roughness and index of refraction values were chosen based on photoscans provided by Quixel Megascans, a public, high-quality 3D scan library [45].

Discrepancies between the BRDF simulation results and actual sensor data appear for two reasons. The first potential issue is the selection of incorrect BRDF parameters (in this case, roughness and index of refraction) for the materials being tested. Ideally, testing materials with precisely known BRDF

parameters would help; this can be done by either using materials from a BRDF database [46] or by measuring those parameters manually. The second issue might be the intrinsic noise present in the lidar sensor itself. If a traditional BRDF simulation were to be implemented, an additional model would be required to simulate the intrinsic noise of the sensor.

ApolloSim aims to fix both of these issues with calibrated noise. As illustrated in Figure 5.5, ApolloSim demonstrates superior performance compared to a simplistic BRDF-based approach, as the calibrated noise also accounts for the intrinsic noise of the sensor. That being said, BRDFs are still incredibly powerful for modeling light transfer, and ApolloSim utilizes them to address gaps in the calibration data, as discussed in Section 4.4.5.

5.4 Efficacy of BRDFs in Addressing Data Gaps

A benchmark material is used to calibrate ApolloSim for simulating that specific material. ApolloSim collects information about how the lidar sensor reacts to the material at multiple different incident angles. Ideally, this calibration data would include all incident angles, ranging from zero to ninety degrees. However, as depicted in Figure 4.10, it would be nearly impossible to collect data on all incident angles. In order to fill the gaps in the data, a BRDF simulation model is used. Figure 5.6 demonstrates that using a BRDF model is a viable way to fill in the gaps in the calibration data.

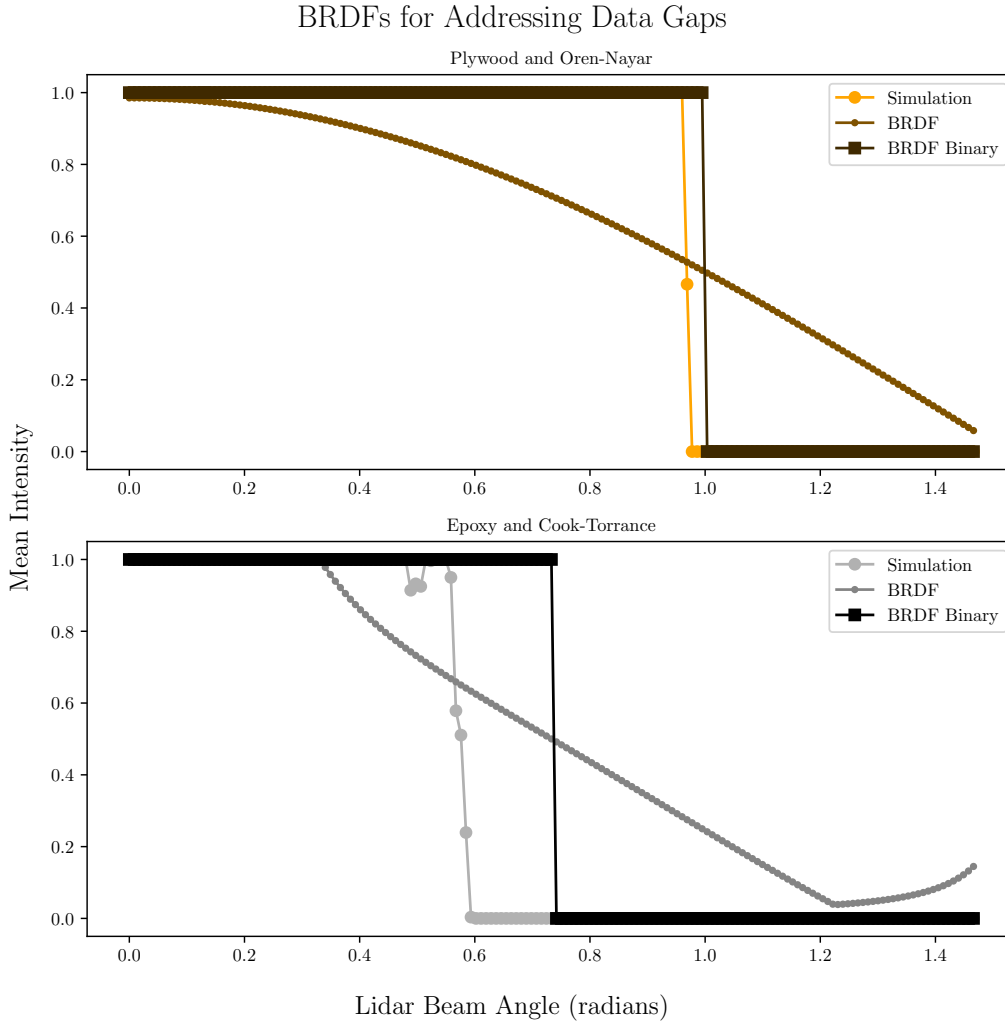


Figure 5.6: The binary BRDF distributions match ApolloSim’s distribution relatively closely, which means that a BRDF model could be used to address missing incident angles in the calibration data. Additionally, the slight rise at the end of the BRDF curve for the epoxy material is attributed to the Fresnel Effect (see Appendix A.6).

While Section 5.3 shows that ApolloSim’s calibrated noise performs better than a standard BRDF simulation model, ApolloSim still uses BRDFs to fill in data gaps. The key difference between just using BRDFs and using them in conjunction with calibrated noise is the selection of BRDF parameters. For Figure 5.6, the BRDF parameters for each material were chosen such that

they resemble the distribution of the calibration noise, instead of choosing parameters based on third-party data.

5.5 Qualitative Analysis of ApolloSim

Qualitative Tests Overview

The results presented above quantitatively demonstrate that ApolloSim can successfully generate accurate lidar data. Because point clouds are intuitively comprehensible in three-dimensional space, a qualitative analysis was also conducted to further demonstrate ApolloSim’s accuracy. In these qualitative tests, two test environments were set up and captured by the lidar sensor. Then, the same environments were recreated in simulation, and captured by a virtual lidar sensor. The real data has been visually compared to the synthetic data.

One of the major challenges while testing a lidar simulation is creating an accurate representation of the real world [1, 24]. Even if the model for simulating the sensor is perfect, if the virtual environment around it is inaccurate, the resulting data will look wrong. Recreating a realistic environment from scratch is an entirely different project on its own. Because of this limitation, simple objects were arranged into simple shapes in order to ensure that the virtual recreation of the environment could be as accurate as possible. Two distinct tests were performed to compare the synthetic data with actual data. These tests employed boards made of wood and foam arranged in var-

ious configurations; the wood and foam materials were calibrated as shown in Figure 5.7.

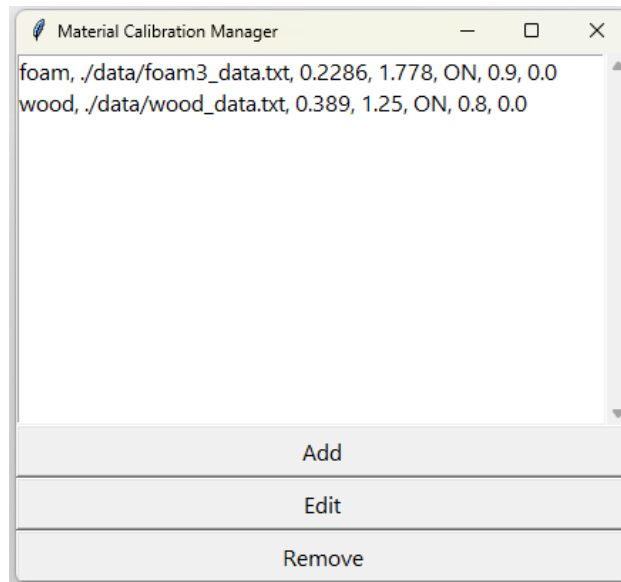


Figure 5.7: Wood and foam were configured for the qualitative tests.

Figure 5.8 shows the calibration data for the foam material, and interestingly, every measured incident angle has an average intensity of exactly 100%. The sensor did not experience any raydrop with the foam material, even at extreme incident angles of close to 75 degrees.

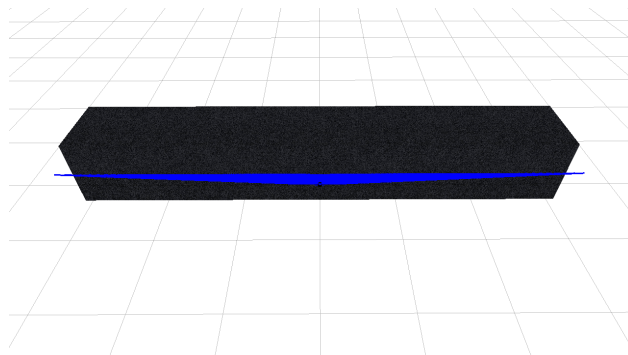


Figure 5.8: The foam material did not cause the sensor to drop any rays.

Qualitative Test 1

As displayed in Figure 5.9, the first test used three foam pads arranged in a zigzag shape as the test object. This was recreated as a virtual test environment in ApolloSim using two cubes, as shown in Figure 5.10. Measurements of the real environment were taken to ensure that the simulated version was as accurate as possible. Finally, Figure 5.11 shows the data from real environment.

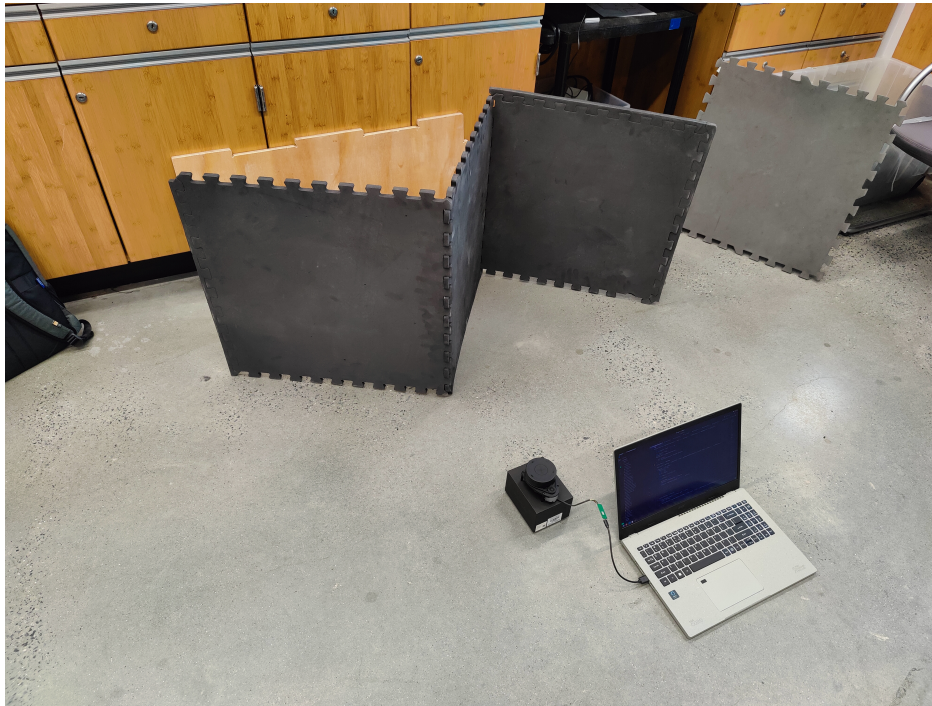


Figure 5.9: Three foam pads were arranged in a zigzag shape for the first qualitative test.

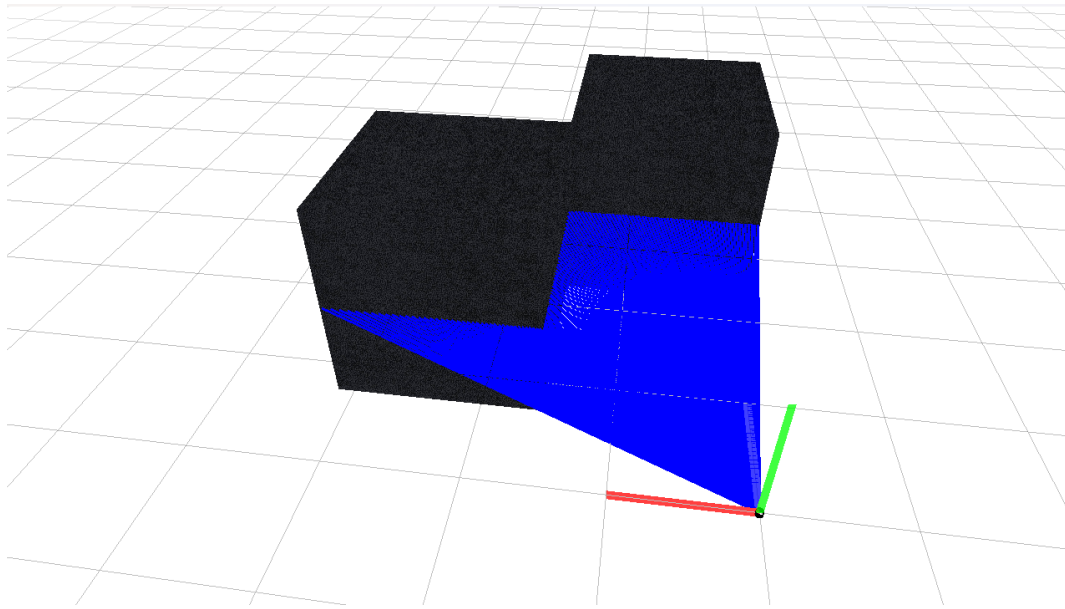


Figure 5.10: Two cubes were arranged to mimic the configuration of the three foam boards.

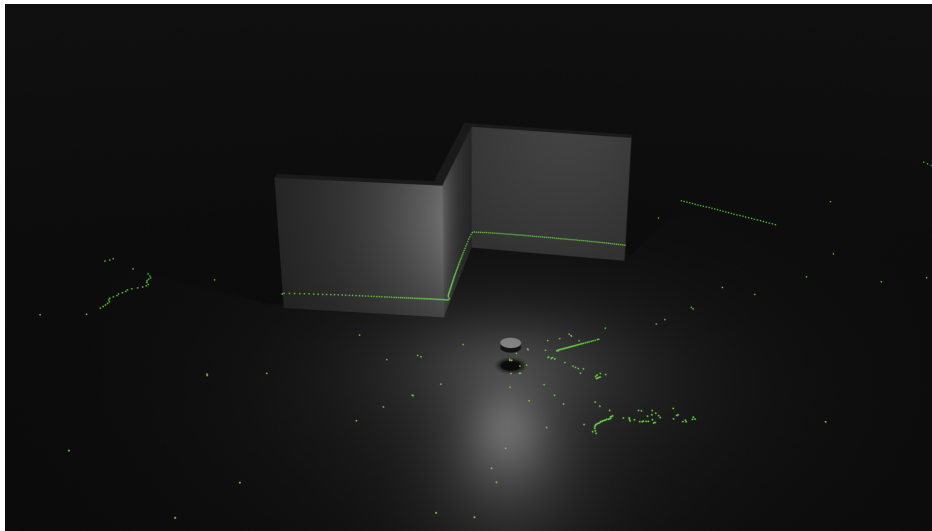


Figure 5.11: A render of the data gathered in the first test.

Finally, the data from the real world and ApolloSim were compared in Figure 5.12. While there are some minor discrepancies between the two datasets, the overall patterns are consistent. The differences in the data are likely due to the fact that the virtual environment recreated in ApolloSim

does not perfectly mimic the environment in the laboratory.

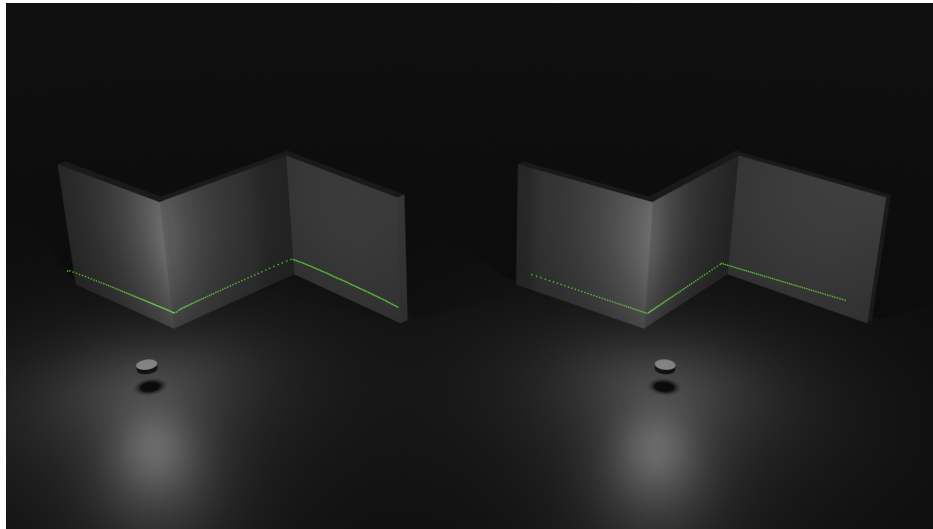


Figure 5.12: The left side shows the data from the real world, and the right side shows data generated by ApolloSim. The data accurately represents the zigzag shape, and has a similar pattern where points get more sparse at higher incident angles.

Qualitative Test 2

The second test used two foam boards and a wooden board, arranged in the shape of a “U”, as shown in Figure 5.13. This environment was also created in ApolloSim using three cubes, as shown in Figure 5.14. Finally, the data from the real world is shown in Figure 5.15.

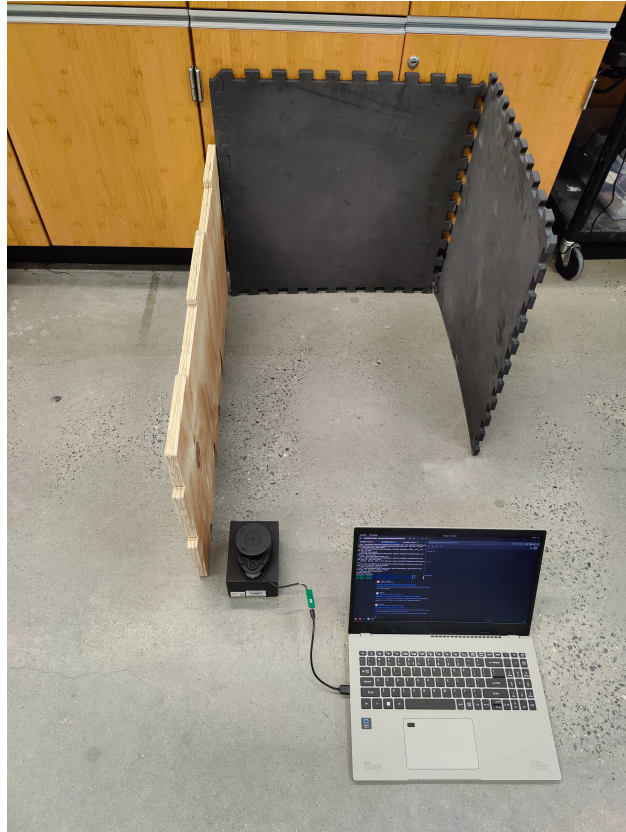


Figure 5.13: The “U”-shaped structure created for the second qualitative test.

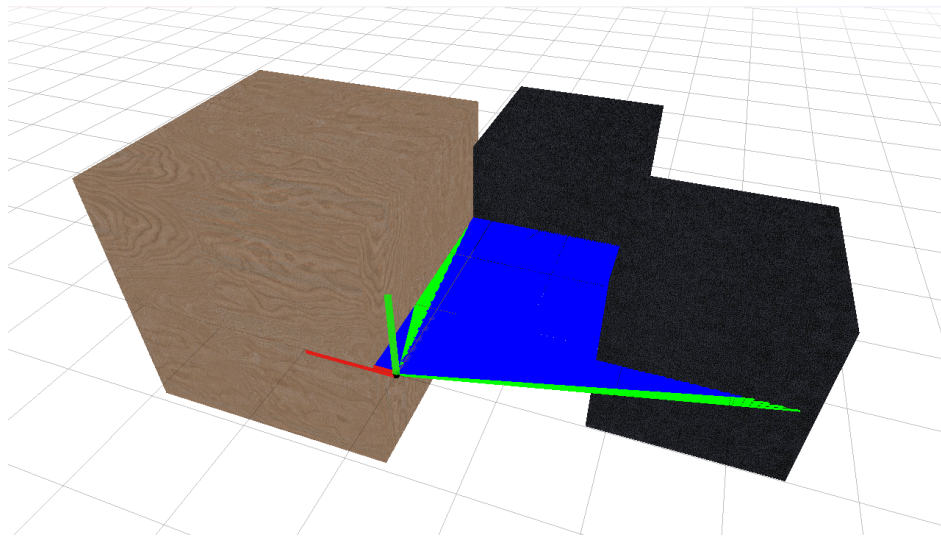


Figure 5.14: Three cubes, one made of wood and the other two made of foam, were used to recreate the shape of the test environment.

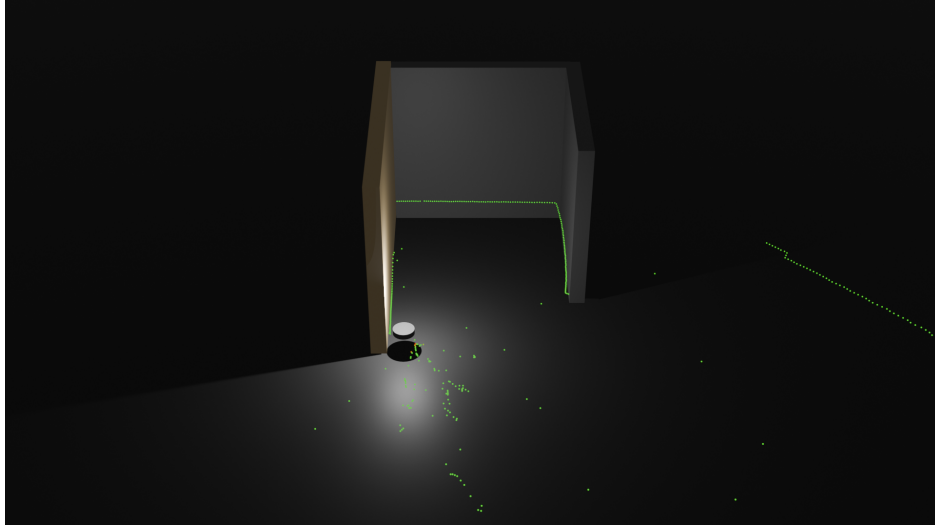


Figure 5.15: A render of the data gathered in the second test.

The data from the real world and the data generated by ApolloSim were compared, as shown in Figure 5.16. Once again, there are some minor discrepancies that are likely present due to an unfaithful recreation of the real-world test environment. For example, in the real-world data, the points intersecting with the foam panel on the right side clearly indicate that the foam panel is not a perfectly flat object; however, in simulation, the foam panel was represented as a perfectly flat object. Therefore, the real-world data has a small curve in it, while the simulation data is perfectly straight, as demonstrated in Figure 5.17.

The similarities in the data highlight ApolloSim’s ability to accurately simulate how a lidar sensor interacts with various objects and materials. For example, the incident angles at which the sensor experiences raydrop on the wooden material are extremely similar in the simulation and the real world. As noted in Section 4.4.5, ApolloSim employs a BRDF for larger incident

angles that were not accounted for during the calibration process. The consistency in raydrop between the simulated and real-world data confirms the accuracy of both the calibrated noise as well as the effectiveness of the BRDF method for filling in data gaps.

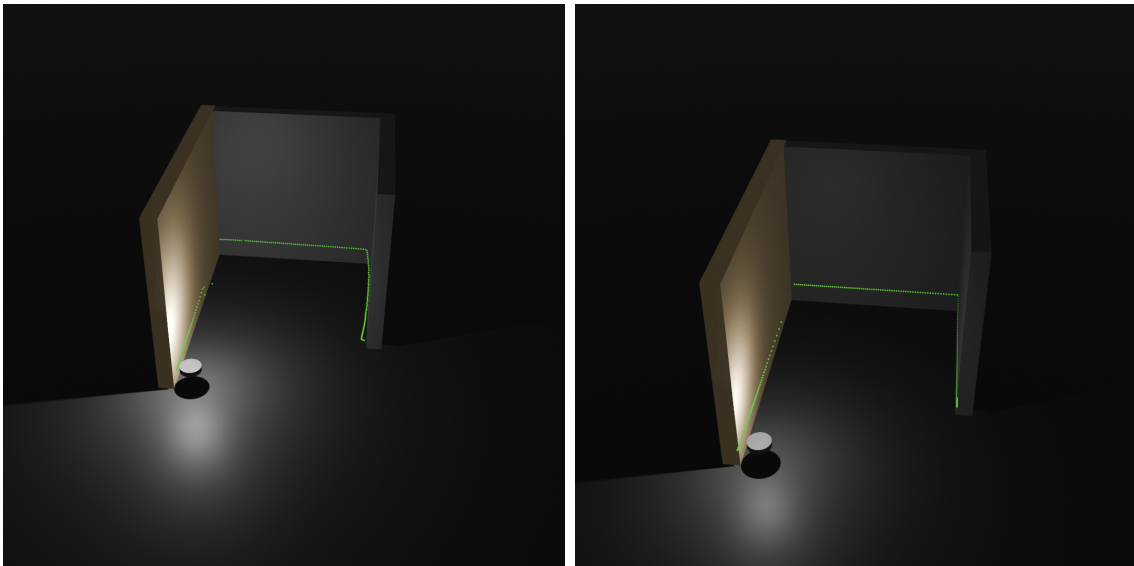


Figure 5.16: The data from the real world is shown on the left, while the data generated by ApolloSim is on the right.

Overall, the qualitative tests confirm that ApolloSim can accurately create synthetic point clouds for complex environments with multiple materials. While the data is not perfectly accurate, a major factor for the discrepancies has to do with imperfect virtual environments.

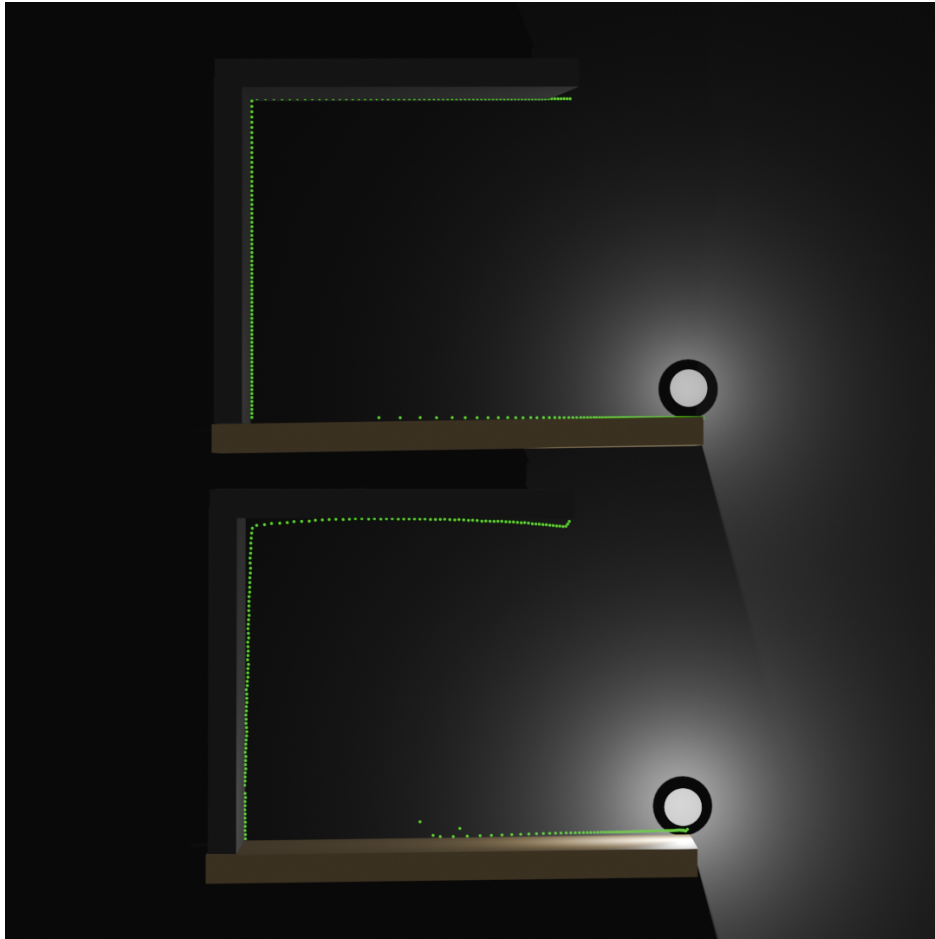


Figure 5.17: The real data, on the bottom, has a slight curve, while the simulated data, on top, is perfectly straight. Additionally, on the wooden material, the simulated and real-world data demonstrate raydrop in the same locations.

Conclusions and Future Work

6.1 Conclusions

In this thesis, I have introduced a novel lidar simulation methodology that is designed to enhance the accuracy and realism of synthetic lidar data through data-driven calibration. Unlike other methodologies that rely on either purely parametric approaches or complex nonparametric models such as those based on deep learning, ApolloSim fits in the middle and uses data-driven parametric models. It offers a nuanced simulation process that is not only more accurate than conventional parametric models but also more accessible and less resource-intensive compared to deep learning alternatives.

ApolloSim is composed of three integral components: calibration, simulation, and visualization. Users can calibrate their target sensor with selected benchmark materials that they would like to simulate. Then, they can set up a virtual environment to test the virtual lidar, and then finally, they can observe their simulation run in real-time.

ApolloSim performs well against traditional parametric models that use BRDFs to simulate light rays, indicating that choosing standard BRDF parameters may not suit the target sensor. However, ApolloSim does use BRDFs to fill the gaps in the calibration data. More specifically, by us-

ing a careful selection of BRDF parameters tailored to the characteristics of the target sensor, ApolloSim can accurately fill gaps in calibration data using BRDF models.

Compared to other lidar simulations that primarily focus on either advanced parametric or nonparametric noise models, ApolloSim carves out a distinctive niche. ApolloSim has the capacity to generate data-driven noise more quickly and with fewer resources than other data-driven approaches while also delivering data with greater accuracy than standard parametric simulation techniques. A data-driven simulation such as ApolloSim can be crucial for various testing environments, particularly in the development of autonomous vehicles. Autonomous vehicles often encounter diverse and complex scenarios, such as navigating among complex geometries and materials found on buildings or other cars. ApolloSim offers a detailed method for predicting and analyzing the performance of specific target lidar sensors, enhancing the precision of sensor evaluations.

To evaluate ApolloSim, the simulation underwent calibration for five materials, and the resulting data was analyzed to verify that the simulation closely resembled real life. The results of the experiments with ApolloSim suggest that using an actual lidar sensor to calibrate a lidar simulation can significantly improve the accuracy of the synthetic data generated by the simulation. While ApolloSim is not without limitations and the potential for further enhancement remains, the rigorous testing of its core concept has

conveyed the validity of this approach.

6.2 Future Work

To effectively enhance the performance and user experience of ApolloSim, several key aspects of ApolloSim have been identified for potential improvement and several new features have been formulated. These enhancements aim to refine both the calibration and simulation aspects of ApolloSim, ensuring greater accuracy, efficiency, and versatility in future iterations.

6.2.1 Improving the Current Features of ApolloSim

Graphics Engine Integration

ApolloSim's graphics engine was developed from the ground up using Odin and OpenGL. Building the engine from scratch allowed for complete control over how ApolloSim worked. However, as the primary aim of this thesis was not to delve into the intricacies of building a graphics engine, ApolloSim was left with a bare-bones graphics engine. Consequently, the engine lacks several advanced features, most notably shading and easily configurable environments. Given these limitations, it would be worth integrating ApolloSim into a more sophisticated, pre-existing engine like Blender or Unreal Engine. These engines offer superior optimization and a wealth of features that would greatly enhance ApolloSim's user experience and functionality.

Automation of Calibration Process

The current calibration process is clunky and prone to error. The distance between the lidar sensor and the benchmark material, the width of the material, and the angle of the material with respect to the lidar sensor are among the measurements that are susceptible to human error. The ability to automatically detect the width and distance of the benchmark materials would greatly improve the speed and accuracy of calibration. Another major improvement would be to allow the user to calibrate multiple materials at once, by specifying to the calibration script where each benchmark material is relative to the sensor.

Fixing Overfitting

ApolloSim relies heavily on calibration data to generate synthetic noise. This introduces the risk of overfitting the simulation to a specific piece of wood, instead of the specific type of material. For instance, if the wooden material used in my experiments features a unique dent or blemish, the simulation might inaccurately propagate these anomalies. This would mean that the noise profile for all wood is tailored to a particular specimen instead of the material as a whole. This would be more limiting when the user only has access to substandard benchmark material samples. Additionally, the data may be overfitted due to the possibility of the sensor reacting differently to a material at various distances. A potential solution could be to use multiple

benchmark materials at various distances to represent a single material type within the simulation, which would give a better representation of the average response to that material. Additionally, smoothing the calibration data curves could aid in achieving a more generalized representation of materials, thereby improving ApolloSim’s accuracy across a broader range of scenarios.

Path Tracing for Beam Calculation

In ApolloSim, the intensity value of a simulated beam is determined solely by its collision point. For instance, if a beam collides with a metal object, ApolloSim calculates the intensity based on the properties of that metal. However, in reality, light reflects off the metal surface, scatters, and may bounce several times before potentially returning to the sensor, introducing noise to the signal. Simulating this complex behavior accurately would be computationally expensive. Nonetheless, techniques like Monte Carlo Path Tracing [36], a widely used ray tracing method, offer a way to approximate this process. In addition to reflecting off of surfaces, light also refracts through surfaces. Refraction is currently not simulated in ApolloSim, but could be an important aspect to generating lidar data in environments with transparent, refractive materials such as glass and water. If refraction were to be implemented, a bidirectional scattering surface reflectance distribution function, also known as a BSSRDF [36], would be used instead of a BRDF in order to model how light reflects and refracts for various materials.

6.2.2 Additional Features for ApolloSim

New Lidar Types

Currently, ApolloSim is equipped to calibrate and simulate two-dimensional, terrestrial lidar sensors. To extend ApolloSim’s capabilities to three-dimensional lidar sensors, the calibration process must be modified to account for azimuthal angles. Moreover, ApolloSim could potentially support more than just terrestrial lidar sensors. Adding support for aerial, satellite, and even bathymetric lidar sensors could significantly broaden ApolloSim’s versatility. However, adapting the calibration process for these sensors could be a challenge.

Simulating Unknown Materials

ApolloSim is currently confined to only simulating materials that have been calibrated by the user. Implementing the functionality to simulate unknown, uncalibrated materials would significantly increase the adaptability of ApolloSim. Ideally, this would involve melding data derived from the calibration phase with existing BRDF models to estimate the sensor’s response to new materials. For example, if a user calibrated their sensor on a concrete wall and wishes to simulate a stone wall, the sensor’s response to the concrete could serve as a baseline dataset for simulating rough, diffuse materials. By examining the differences in the BRDF parameters between concrete and stone, one can adjust the baseline dataset to approximate the characteristics

of the stone wall. This would allow for a more generalized calibration process; instead of having to calibrate for every material in existence, one would calibrate for a few unique materials and then use that as a baseline for all other similar materials.

Material Identification

Finally, an intriguing potential capability of ApolloSim would be its ability to enable sensors to identify materials based on their response patterns. ApolloSim possesses data on how the sensor reacts to various materials; this information could potentially be utilized to identify unknown materials based on their response to the sensor. Given this potential feature, ApolloSim would be able to produce datasets that could be used to train models that can recognize materials given a point cloud.

Appendix A: Additional Information

A.1 Why “ApolloSim”?

ApolloSim is heavily influenced by Helios++ [1]; Helios is the Greek god of the sun, and this project was implemented in a language called Odin, the Norse god. I decided to stick to the mythical god theme and went with Apollo, the Greek god of the sun, light, truth and prophecy because ApolloSim ultimately simulates light and people use simulations to predict the future.

A.2 ROS 2 Node for Collecting Lidar Data

As discussed in Section 4.2.2, data is collected from the RPLIDAR A1 via ROS 2 nodes and topics. First, the RPLIDAR ROS 2 SDK [41] is used to launch a node that publishes a packet of data each time the lidar sensor completes a full revolution. This packet of data is structured as a list of angles, intensity values, distance values and some metadata, as shown in Figure A.1. Of course, this data is unique to the RPLIDAR A1, as other lidar sensors may have more data or a different structure.

The data packets generated by the RPLIDAR SDK are published to a topic called `\scan`. A custom ROS 2 node was built to subscribe to and

```
Timestamp: 1707162652.862409387
  start_angle: -3.1241393089294434
end_angle: 3.1415927410125732
  angle_increment: 0.008714509196579456
range_min: 0.15000000596046448
range_max: 12.0
ranges: array('f', [inf, inf, 3.635999917984009...])
intensities: array('f', [0.0, 0.0, 47.0...])
```

Figure A.1: Raw lidar data

read the data from this topic to gather calibration data. In addition to just collecting the data, the node launches a live view of the lidar data, as described in Section 4.2.2. The lidar data is filtered such that only beams that collide with the benchmark material are included, as explained in Equation 4.1. Finally, the data is parsed and written to a text file, where each line is an angle, an intensity value, and a distance value. Figure A.2 displays the structure of the calibration data for ApolloSim.

```
0.11900000274181366,47.0,-3.1241393089294434
0.11999999731779099,47.0,-3.115424799732864
...
0.5120000243186951,47.0,-0.0043450165539979935
0.5040000081062317,47.0,0.004369492642581463
...
0.11800000071525574,47.0,3.1328782942146063
0.11900000274181366,47.0,3.1415928034111857
```

Figure A.2: Processed lidar data for ApolloSim. The pattern is `distance, intensity, angle`.

A.3 Propagation of Data from Calibration to Simulation

The primary focus of ApolloSim is its use of calibration data in simulation. The calibration data is analyzed by ApolloSim before being put to use in the simulation engine. First, the calibration data is collected and stored in a text file, as described in Appendix A.2. Then, the configuration file, as specified in Section 4.5.2, is parsed and stored in a data structure called `MaterialInput`, as displayed in Table A.1.

Attribute	Description	Data Type
Material Name	The name of the material (e.g. wood, epoxy, metal)	String
File Path	File path of calibration data	String
Distance	Distance between the benchmark material and lidar sensor	float
Width	Width of the benchmark material	float
BRDF	The chosen BRDF for the material	integer
Roughness	Roughness of the material	float
Index of Refraction	Index of refraction of the material	float
Material ID	Each material is assigned a unique ID	int

Table A.1: The `MaterialInput` data structure is used to store the materials that have been calibrated by the user.

Once the configuration file is parsed and there exists a `MaterialInput` object for each calibration file, ApolloSim parses the calibration data in order to calculate the key characteristics of each material and organizes the data into `AngleData` objects, as described in Section 4.2.3. Each material is stored as a `MaterialData` object, as described in Table A.2.

Attribute	Description	Data Type
Material Name	The name of the material (e.g. wood, epoxy, metal)	String
Material ID	Unique ID of the material	int
Angles Data	List of <code>AngleData</code> data structures	<code>AngleData[]</code>
Mean Intensity	Overall mean of intensity values	float
Standard Deviation of Intensity	Overall standard deviation of intensity values	float
Standard Deviation of Distance	Overall standard deviation of distance values	float
BRDF	The chosen BRDF for the material	integer

Table A.2: The `MaterialData` data structure is used to store the materials and their calibration data.

Finally, in order to send the calibration data to the GPU, the materials must be simplified and flattened into one array. A new data structure called `GPUAngle`, as seen in Table A.3, is specifically used to aggregate all of the data in order to store it in a buffer for the compute shader. Along with the data for each angle, a list of `GPUMaterial` objects, as shown in Table A.4, is sent to the GPU. That way, the compute shader can cross-reference a `GPUAngle` object with a `GPUMaterial` object in order to gain information about the performance of the lidar at a specific angle for a specific material.

Attribute	Description	Data Type
Angle	The incident angle (degrees) associated with this data	float
Material ID	Unique ID of the material	int
Mean Intensity	Mean of intensity values	float
Standard Deviation of Intensity	Standard deviation of intensity values	float
Standard Deviation of Distance	Standard deviation of distance values	float
Drop Rate	Percentage of rays dropped	float

Table A.3: The `GPUAngle` data structure is used to flatten the calibration data into one array that is easily accessible by the compute shader.

Attribute	Description	Data Type
ID	Unique ID of the material	integer
BRDF	The BRDF model used for this material	integer
Roughness	The roughness of the material	float
Index of Refraction	The index of refraction of the material	float
Real Material	A flag to indicate if the material is based on real calibration data	boolean

Table A.4: The `GPUMaterial` data structure is used to store the details about each material.

A.4 Compute Shader Pseudocode

The locations and intensity values of points in the synthetic point cloud generated by ApolloSim are calculated using a compute shader, as discussed in Section 4.4.3. Algorithm 1 shows the pseudocode for this process. In this process, each beam is checked to see if there is an intersection with an object in the scene; if there is, then the corresponding output is set.

Algorithm 1 Loop through each beam

```

1: procedure MAIN
2:   directions  $\leftarrow$  list of beam directions as vec3s
3:   output  $\leftarrow$  corresponding output variables as vec4s
4:   for  $i \leftarrow 0$  to length of directions do
5:     intersection  $\leftarrow$  point where the beam intersects with an object
6:     direction  $\leftarrow$  directions[ $i$ ]
7:     for  $j \leftarrow 0$  to length of objects in scene do
8:       object  $\leftarrow$  objects[ $j$ ]
9:       new intersection  $\leftarrow$  getIntersectionResult(direction, object)
10:      if distance(new intersection) < distance(intersection) then
11:        intersection  $\leftarrow$  new intersection
12:      end if
13:    end for
14:    output[ $i$ ]  $\leftarrow$  intersection
15:  end for
16: end procedure

```

The existence of an intersection between each beam and each item in

the scene has to be checked. Algorithm 2 shows the pseudocode for that process. The first step is to find the closest intersection point between the sensor and an object in the scene, using the `getIntersection` function. The `getIntersection` function that is used in Algorithm 2 varies depending on the object; for primitives, such as spheres and cubes, implicit equations are used, as commonly done in ray tracing algorithms [38]. For complex meshes, a different algorithm is employed, which checks each triangle in the mesh for an intersection. The `getIntersection` function returns a `IntersectionResult` data structure, which is detailed in Table A.5.

Attribute	Description	Data Type
Intersects	Does this beam intersect with the object?	bool
Point	The point of intersection	vec3
Intensity	Intensity of the backscattered beam	float
θ_i	Incident angle of the intersection	float
Uses BRDF	Does this angle require a BRDF or does it use calibration data?	bool

Table A.5: The `IntersectionResult` data structure is used to store the details regarding an intersection between a beam and an object in the scene.

As described in Section 4.4.4, depending on the incident angle of collision with the object, ApolloSim either utilizes the calibration data or a BRDF to calculate the intensity. The calibration data is only used if the incident angle of the collision is within the range of angles that are part of the calibration data. Otherwise, the intensity is calculated using a BRDF function; depending on the material, different BRDF functions are utilized.

Algorithm 2 Find intersection for a given beam

```
1: procedure FIND INTERSECTION
2:   direction  $\leftarrow$  beam directions as vec3
3:   object  $\leftarrow$  object to check beam intersection for
4:   material  $\leftarrow$  material of object
5:   result  $\leftarrow$  getIntersection(beam, object)
6:   if result.intersects then
7:     incident angle  $\leftarrow$  incident angle of intersection
8:     AngleData  $\leftarrow$  getClosestAngle(incident angle)
9:     if incident angle = AngleData.angle then
10:      mean  $\leftarrow$  AngleData.meanIntensity
11:      stdev  $\leftarrow$  AngleData.standardDevIntensity
12:      result.intensity  $\leftarrow$  sampleNormalDistribution(mean, stdev)
13:     else
14:      result.intensity  $\leftarrow$  BRDF(incident angle, material)
15:     end if
16:     stdev  $\leftarrow$  AngleData.standardDevDistance
17:     point noise  $\leftarrow$  sampleNormalDistribution(0, stdev)
18:     result.point  $\leftarrow$  result.point + point noise
19:   end if
20:   return result
21: end procedure
```

A.5 The Cook-Torrance Model

The Cook-Torrance model, as presented in Equation 2.5, is a great BRDF model for rough materials with a specular component. Examples of this type of material include metals, plastics, and even polished wood. Cook-Torrance is composed of three main components: the microfacet distribution (D), which accounts for the orientation and density of microfacets; geometric shadowing (G), addressing the occlusion of light as it interacts with these microfacets; and Fresnel reflectance (F), which accounts for reflection due to the Fresnel Effect [38].

The Cook-Torrance model typically uses the following inputs: \vec{l} , the direction of the light, \vec{v} , the direction from the collision point to the position of the viewer (for a lidar sensor, the viewer and light direction are opposites, since the lidar observes the beam of light from approximately where it was cast), and \vec{n} , the normal of the surface. The microfacet distribution term, D , defines the shape of the specular highlight. Popular choices for the microfacet distribution include Blinn-Phong, Beckmann, and GGX. The GGX distribution, as seen in Equation A.1, is very popular and is the chosen distribution for ApolloSim, as it produces a more accurate specular distribution than the previous distributions at little extra cost [47].

$$D(\vec{h}) = \frac{\alpha^2}{\pi((\vec{n} \cdot \vec{h})^2(\alpha^2 - 1) + 1)^2} \quad (\text{A.1})$$

The geometric attenuation term G , is the geometric attenuation of the surface, which accounts for the attenuation caused by nearby microfacets, as microfacets may occlude light from or reflect light to other microfacets. The Schlick model is a popular choice for the geometric attenuation [47, 48], as shown in Equation A.2 [49].

$$\begin{aligned} k &= (\text{roughness} + 1)^2 \\ G_1(\vec{v}) &= \frac{\vec{n} \cdot \vec{v}}{(\vec{n} \cdot \vec{v})(1 - k) + k} \\ G(\vec{l}, \vec{v}, \vec{h}) &= G_1(\vec{l}) * G_1(\vec{v}) \end{aligned} \quad (\text{A.2})$$

Finally, the Fresnel reflectance term F is the reflection of light due to the

Fresnel Effect. The most common approximation for the Fresnel Effect is Schlick's Approximation, as seen in Equation A.3 [49].

$$\begin{aligned} F &= F_0 + (1 - F_0) * (1 - (\vec{v} \cdot \vec{h})) \\ F_0 &= \frac{(n - 1)^2}{(n + 1)^2} \end{aligned} \tag{A.3}$$

A.6 The Fresnel Effect

The Fresnel Effect causes increased reflection at larger angles of incidence for specular materials, particularly dielectrics. Intuitively, the higher the angle of incidence, the less light should be reflected back to the viewer. However, when light strikes a surface at a steep angle, the properties of the surface, most notably the index of refraction, cause a change in the light's electric field, which leads to more light being reflected and less being refracted or absorbed [50, 51].

Bibliography

- [1] L. Winiwarter, A. M. E. Pena, H. Weiser, K. Anders, J. M. Sanchez, M. Searle, and B. Höfle, “Virtual laser scanning with HELIOS++: A novel take on ray tracing-based simulation of topographic 3D laser scanning,” Jan. 2021, issue: arXiv:2101.09154 arXiv:2101.09154 [cs, eess]. [Online]. Available: <http://arxiv.org/abs/2101.09154>
- [2] Y. Li and J. Ibanez-Guzman, “Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems,” *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, Jul. 2020. [Online]. Available: <https://ieeexplore.ieee.org/document/9127855/>
- [3] O. Risbøl and L. Gustavsen, “LiDAR from drones employed for mapping archaeology – Potential, benefits and challenges,” *Archaeological Prospection*, vol. 25, no. 4, pp. 329–338, Oct. 2018. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/arp.1712>
- [4] A. Lopez, C. J. Ogayar, J. M. Jurado, and F. R. Feito, “A GPU-Accelerated Framework for Simulating LiDAR Scanning,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–18, 2022. [Online]. Available: <https://ieeexplore.ieee.org/document/9751040/>

- [5] M. Gschwandtner, R. Kwitt, A. Uhl, and W. Pree, “BlenSor: Blender Sensor Simulation Toolbox,” in *Advances in Visual Computing*, G. Bebis, R. Boyle, B. Parvin, D. Koracin, S. Wang, K. Kyungnam, B. Benes, K. Moreland, C. Borst, S. DiVerdi, C. Yi-Jen, and J. Ming, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6939, pp. 199–208, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-24031-7_20
- [6] “Gazebo.” [Online]. Available: <https://gazebo.org/about>
- [7] “Outsight 3D LiDAR Simulator.” [Online]. Available: <https://www.outsight.ai/>
- [8] “Adding Synthetic Noise.” [Online]. Available: <https://www.nmr.mgh.harvard.edu/PMI/toolbox/Documentation/pminoise.xhtml>
- [9] S. Manivasagam, S. Wang, K. Wong, W. Zeng, M. Sazanovich, S. Tan, B. Yang, W.-C. Ma, and R. Urtasun, “LiDARsim: Realistic LiDAR Simulation by Leveraging the Real World,” Jun. 2020, issue: arXiv:2006.09348 arXiv:2006.09348 [cs]. [Online]. Available: <http://arxiv.org/abs/2006.09348>
- [10] B. Guillard, S. Vemprala, J. K. Gupta, O. Miksik, V. Vineet, P. Fua, and A. Kapoor, “Learning to Simulate Realistic LiDARs,” Sep. 2022, issue: arXiv:2209.10986 arXiv:2209.10986 [cs]. [Online]. Available: <http://arxiv.org/abs/2209.10986>

- [11] J. Zhang, F. Zhang, S. Kuang, and L. Zhang, “NeRF-LiDAR: Generating Realistic LiDAR Point Clouds with Neural Radiance Fields,” Jan. 2024, issue: arXiv:2304.14811 arXiv:2304.14811 [cs]. [Online]. Available: <http://arxiv.org/abs/2304.14811>
- [12] Weng, Qihao, *Advances in environmental remote sensing: sensors, algorithms, and applications*, first paperback edition ed. Boca Raton: CRC Press, 2017, oCLC: 982649332.
- [13] “What is LiDAR and How Does It Work?” [Online]. Available: <https://www.jouav.com/blog/what-is-lidar.html>
- [14] “Velodyne Lidar Puck.” [Online]. Available: <https://velodynelidar.com/products/puck/>
- [15] V. Mazzari, “What is LiDAR technology?” [Online]. Available: <https://www.generationrobots.com/blog/en/what-is-lidar-technology/>
- [16] “Lidar.” [Online]. Available: <https://www.neonscience.org/data-collection/lidar>
- [17] B. Jutzi and U. Stilla, “Range determination with waveform recording laser systems using a Wiener Filter,” *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 61, no. 2, pp. 95–107, Nov. 2006, number: 2. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S0924271606001080>

- [18] S. Reitmann, L. Neumann, and B. Jung, “BLAINDER—A Blender AI Add-On for Generation of Semantically Labeled Depth-Sensing Data,” *Sensors*, vol. 21, no. 6, p. 2144, Mar. 2021. [Online]. Available: <https://www.mdpi.com/1424-8220/21/6/2144>
- [19] JarlBallin89, “Maple Tree.” [Online]. Available: <https://sketchfab.com/3d-models/maple-tree-68bea58fd9a549a99cfa5d1c739c97a8>
- [20] DrCG, “Building No 6 form Tokyo Otemachi Building Pack.” [Online]. Available: <https://skfb.ly/oyP8X>
- [21] Krzysztof Stolorz, “1929 BMW 3/15 (Dixi, LP).” [Online]. Available: <https://sketchfab.com/3d-models/1929-bmw-315-dixi-lp-df5748546cdb429ba3c5038697e4a4d4>
- [22] GISGeography, “What is a Point Cloud?” [Online]. Available: <https://gisgeography.com/point-cloud/>
- [23] M. Gschwandtner, R. Kwitt, A. Uhl, and W. Pree, “BlenSor: Blender Sensor Simulation Toolbox,” in *Advances in Visual Computing*, G. Bebis, R. Boyle, B. Parvin, D. Koracin, S. Wang, K. Kyungnam, B. Benes, K. Moreland, C. Borst, S. DiVerdi, C. Yi-Jen, and J. Ming, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6939, pp. 199–208, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-24031-7_20

- [24] Abhijeet Tallavajhula, “Lidar Simulation for Robotic Application Development: Modeling and Evaluation,” p. 14724730 Bytes, 2018, artwork Size: 14724730 Bytes Publisher: Carnegie Mellon University. [Online]. Available: https://kilthub.cmu.edu/articles/Lidar_Simulation_for_Robotic_Application_Development_Modeling_and_Evaluation/6720428/1
- [25] “Difference between Parametric and Non-Parametric Methods.” [Online]. Available: <https://www.geeksforgeeks.org/difference-between-parametric-and-non-parametric-methods/>
- [26] Georgios Nanos, “Differences Between a Parametric and Non-parametric Model.” [Online]. Available: <https://www.baeldung.com/cs/ml-parametric-vs-non-parametric-models>
- [27] J. d. Vries, *Learn OpenGL - Graphics programming: Learn modern OpenGL graphics programming in a step-by-step fashion*. Erscheinungsort nicht ermittelbar: Kendall & Welling, 2020.
- [28] Patricio Gonzalez Vivo and Jen Lowe, *The Book of Shaders*. [Online]. Available: <https://thebookofshaders.com/>
- [29] “Shader.” [Online]. Available: <https://www.khronos.org/opengl/wiki/Shader>

- [30] “Compute Shader.” [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader
- [31] H. Va, M.-H. Choi, and M. Hong, “Real-Time Cloth Simulation Using Compute Shader in Unity3D for AR/VR Contents,” *Applied Sciences*, vol. 11, no. 17, p. 8255, Sep. 2021. [Online]. Available: <https://www.mdpi.com/2076-3417/11/17/8255>
- [32] A. Junker and G. Palamas, “Real-time Interactive Snow Simulation using Compute Shaders in Digital Environments,” in *International Conference on the Foundations of Digital Games*. Bugibba Malta: ACM, Sep. 2020, pp. 1–4. [Online]. Available: <https://dl.acm.org/doi/10.1145/3402942.3402995>
- [33] Brian Caulfield, “What’s the Difference Between Ray Tracing and Rasterization?” [Online]. Available: <https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/>
- [34] Henrik, “Ray trace diagram.” [Online]. Available: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg
- [35] *Ray Tracing Gems*. Springer Nature, 2021, oCLC: 1328776805.
- [36] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: from theory to implementation*, third edition ed. Cambridge, MA: Morgan Kaufmann Publishers/Elsevier, 2017, oCLC: ocn936532273.

- [37] “Lambertian reflectance.” [Online]. Available: https://en.wikipedia.org/wiki/Lambertian_reflectance
- [38] Ian Dunn and Zoe Wood, *Graphics Programming Compendium*. [Online]. Available: <https://graphicscompendium.com/index.html>
- [39] “CARLA Simulator.” [Online]. Available: <https://carla.org/>
- [40] “ROS 2 Humble.” [Online]. Available: <https://docs.ros.org/en/humble/index.html>
- [41] “SLAMTEC LIDAR ROS2 Package.” [Online]. Available: https://github.com/Slamtec/slidar_ros2
- [42] Slamtec, “RPLIDAR A1.” [Online]. Available: <https://www.slamtec.ai/product/slamtec-rplidar-a1/>
- [43] “Velodyne Lidar ULTRA Puck,” 2019. [Online]. Available: https://velodynelidar.com/wp-content/uploads/2019/12/63-9378_Rev-F_Ultra-Puck_Datasheet_Web.pdf
- [44] M. Oren and S. K. Nayar, “Generalization of Lambert’s reflectance model,” in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques - SIGGRAPH '94*. Not Known: ACM Press, 1994, pp. 239–246. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=192161.192213>
- [45] “Quixel Megascans.” [Online]. Available: <https://quixel.com/megascans>

- [46] W. Matusik, H. Pfister, M. Brand, and L. McMillan, “A data-driven reflectance model,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 759–769, Jul. 2003. [Online]. Available: <https://dl.acm.org/doi/10.1145/882262.882343>
- [47] Brian Karis, “Real Shading in Unreal Engine 4.” [Online]. Available: <https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf>
- [48] B. Walter, S. R. Marschner, H. Li, and K. E. Torrance, “Microfacet Models for Refraction through Rough Surfaces,” *Rendering Techniques*, p. 12 pages, 2007, artwork Size: 12 pages ISBN: 9783905673524 Publisher: [object Object]. [Online]. Available: <http://diglib.eg.org/handle/10.2312/EGWR.EGSR07.195-206>
- [49] C. Schlick, “An Inexpensive BRDF Model for Physically-based Rendering,” *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, Aug. 1994. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1111/1467-8659.1330233>
- [50] Greg A. Smith, “Fresnel Equations,” *The University of Arizona Wyant College of Optical Sciences*. [Online]. Available: <https://webs.optics.arizona.edu/gsmith/Fresnel.html>
- [51] “Fresnel equations.” [Online]. Available: https://en.wikipedia.org/wiki/Fresnel_equations

ProQuest Number: 31295713

INFORMATION TO ALL USERS

The quality and completeness of this reproduction is dependent on the quality and completeness of the copy made available to ProQuest.



Distributed by ProQuest LLC (2024).

Copyright of the Dissertation is held by the Author unless otherwise noted.

This work may be used in accordance with the terms of the Creative Commons license or other rights statement, as indicated in the copyright statement or in the metadata associated with this work. Unless otherwise specified in the copyright statement or the metadata, all rights are reserved by the copyright holder.

This work is protected against unauthorized copying under Title 17, United States Code and other applicable copyright laws.

Microform Edition where available © ProQuest LLC. No reproduction or digitization of the Microform Edition is authorized without permission of ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346 USA