Radar Sensor Plugin for Game Engine Based Autonomous Vehicle Simulators

Radar Sensor Plugin for Game Engine Based Autonomous Vehicle Simulators

Erdal Yilmaz

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

June 2020

Abstract

Simulations play an essential role in developing autonomous vehicles and verifying their safe operation. They enable research in sensor fusion with synthetic data and allow low-cost experimentation with different design decisions, like the number, location, and specifications of various sensors on vehicles. Apart from industrial simulation tools, researchers have been using game engine based simulators, mainly to generate training data for artificial intelligence systems and to test their decision making in virtual worlds. These simulators currently support camera and lidar sensors but lack a physics-based radar implementation. Automotive radars that serve for advanced driver-assistance systems today are evolving into imaging radar systems for autonomous vehicles. Therefore it is critical to be able to simulate them in the same environment together with other sensors. In this thesis, we aim to develop a generic radar sensor plugin using a raytracing method and to integrate it into one of the game engine based autonomous vehicle simulators.

Contents

List of Figures

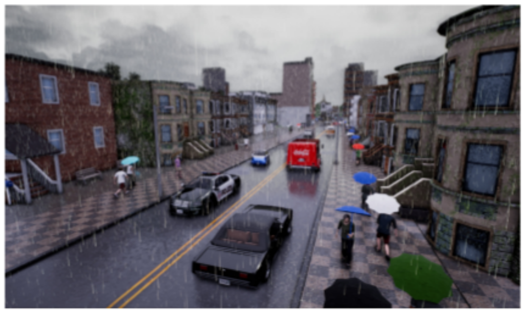
List of Tables

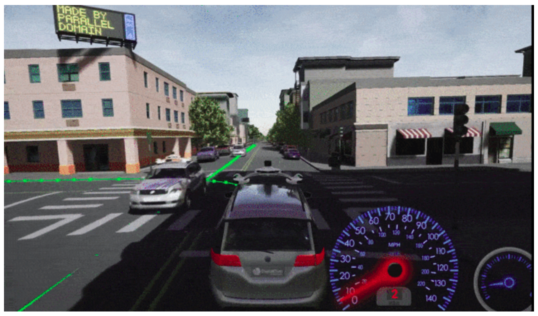
Chapter 1: Introduction

1.1. Motivation

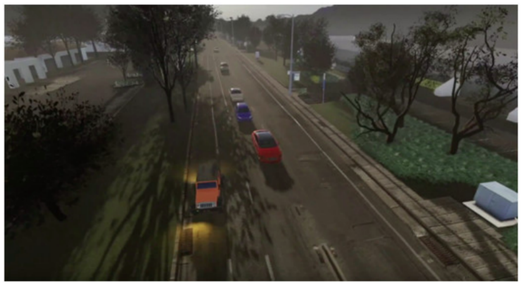
Autonomous driving companies need to demonstrate the safety of their products by driving with their systems for millions of miles [1]. Any changes in hardware or software implementation will require more testing on the road. Simulations provide a low-cost alternative to road tests and can expose the system-under-test to complex, repeatable scenarios [2]. The overarching goal is to verify the correctness of decisions made by an artificial intelligence subsystem. The functionality of this system depends on the fusion of high quality and rich sensory information [3].

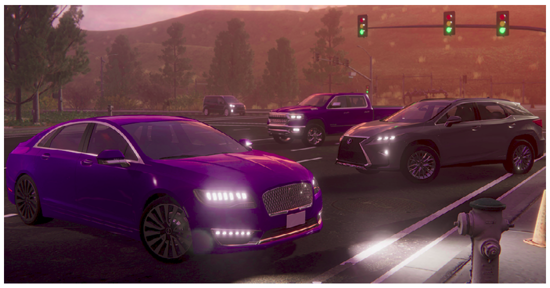
The development of environmental perception sensors is a costly industry-wide effort which requires modeling and simulation to improve sensor and system design, and our understanding of the data collected from road tests. In effect, simulations are virtual labs where we can test experimental sensors and environmental models with plugins. Many proprietary simulators are available from multiple vendors to aid sensor system research and development [4]; however, open-source tools impact research productivity more by allowing modifications and enhancements. From a scientific point of view, they also allow repeatability and comparison of results. Software modularity and well-defined interfaces for plugins are critical to orchestrating an ecosystem of open and closed-source tools together. In our application context, Robot Operating System (ROS) provides a good example [5, 6].

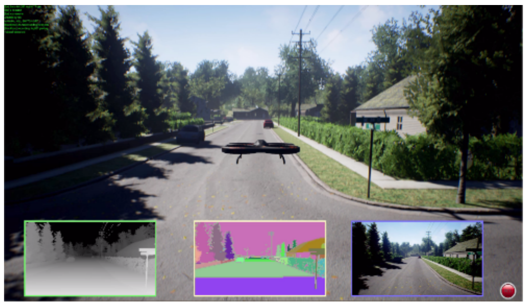

(a) CARLA

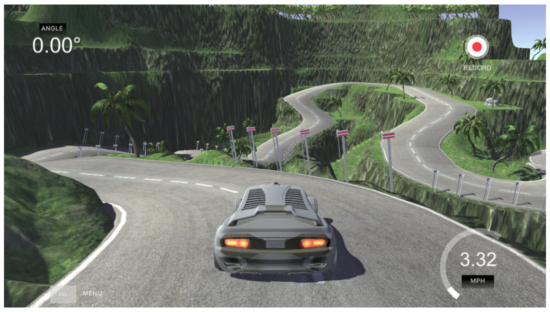


(b) DeepDrive


(c) LGSVL Simulator


(d) Baidu Apollo Simulator


(e) AirSim



(f) Udacity Simulator

Figure 1.1: Game engine based autonomous vehicle simulators. CARLA and DeepDrive are based on Unreal Engine. LGSVL Simulator and Baidu Apollo Simulator are based on Unity game engine. AirSim simulates quadcopters and cars, it can work with both Unreal and Unity. Udacity Simulator, which is based on Unity, was developed as a teaching tool.

In recent years, academic and industrial researchers released multiple simulators for autonomous vehicle research based on popular game engines. [7, 8, 9, 10]. These simulators spawn numerous actors within a predefined environment and allow the user or an algorithm to control their motion (Fig4.5). It's also possible to record multimodal sensor data for training deep neural networks and experiment with sensor fusion architectures for more accurate and robust object classification, visual-inertial odometry [11], or reinforcement learning [12].

A game engine mainly organizes user interface, physics engine, and rendering [13]. 3D games provide visual feedback to the users by simulating camera models and allow them to control camera positions and orientations. Therefore, for autonomous driving applications, rendering quality based on the physics of light is highly relevant. Game engines employ rasterization techniques, which involve shader programs running on graphics processing units (GPU). The resulting images are approximations and not physically accurate. There is another rendering technique called raytracing, which traces a high number of light rays between sources and cameras within a scene, producing photorealistic results [14]. Unfortunately, raytracing for high-quality images was quite slow until recently. Today, with improved GPU architectures, it's possible to get real-time raytracing (RTRT) in game engines (Fig.1.3).

To generate useful synthetic data for our applications, we should apply physically-based rendering for all perception sensors: cameras, lidars, and radars. The last two are active ranging sensors based on the propagation of self-generated electromagnetic waves within the environment and their reflections from objects. Therefore, we can apply similar techniques used for visible light with modifications to the material properties based on the frequency of operation. Inherently, lidar simulations can use low-density raytracing because of the spatial coherence of laser beams. On the other hand, radars emit waves in a wide angular span, and the response of materials can

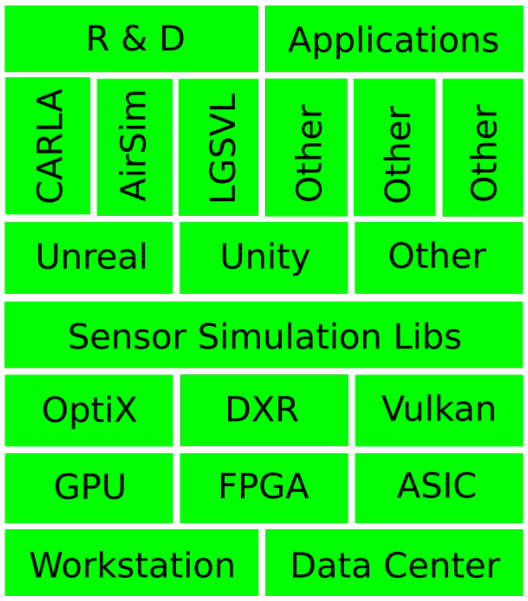


Figure 1.2: Sensor simulation stack

Figure 1.3: Unreal Engine RTRT

Figure 1.4: Unity Engine RTRT

be drastically different at microwave frequencies.

Current simulators have implemented various camera imperfections like lens distortion. They also provide lidar implementations utilizing a raycasting mechanism within the game engine framework, and currently, they lack a physics-based radar sensor. Implementing a radar simulator for a particular game application or a game engine would potentially tie the source code to a specific framework, limiting its portability. Ideally, the rendering engines should utilize lower-level APIs for all perception sensors and use real-time raytracing for simulations. This opens up an opportunity to develop software libraries that can function as engine plugins. Recently introduced technologies DXR and Vulkan raytracing provide necessary foundations [15].

In Figure 1.2, we depict the context of this thesis as “Sensor Simulation Libs”. Our software integration targets leftmost vertical (OptiX + Unreal Engine + CARLA) with a future goal to move towards the right. Note that this is not a technology dependency stack, but a visualization for how to bring various tools together to serve the business needs which could be either R&D efforts or direct applications.

1.2. Outline

The thesis is organized as follows:

- In Chapter 1, we provide the motivation for the thesis project.
- In Chapter 2, we review the main components of the thesis: automotive radars, raytracing, and game engine based simulators.
- In Chapter 3, we describe software architecture relating to the plugin and its integration; we also point out software design decisions and some implementation details.
- In Chapter 4, we demonstrate the capabilities and discuss the limitations of the simulator and plugin with use cases and benchmarks.
- In Chapter 5, we provide a summary and conclusions.

Chapter 2: Background

In this chapter, we review working principles of radar, describe FMCW and TDM-MIMO concepts, highlight difficulties in electromagnetic wave simulations in the automotive radar context, and motivate raytracing as an alternative approach. We then describe raytracing with OptiX framework and provide an overview of game engine based autonomous vehicle simulators.

2.1. Radar

Radar stands for *radio detection and ranging.* It was initially developed as a military technology and later found use in many application contexts. It uses scattered electromagnetic waves (EM) to extract information about the components of the environment they travel in, mostly for detection and classification of objects of interest, and estimating their positions and velocities.

A radar system consists of one or more transmitting and receiving antennas, and signal processing systems. With transmitting antennas, it sends EM waves at certain frequency towards direction of interest and collects scattered waves using receiving antennas. If transmitting and receiving antennas are co-located, it's called a *monostatic* radar, if they are separated it's called *bistatic* (Fig.2.1).

Received signals first go through analog stages, then digitized with converters, and processed based on their modulation schemes. The amplitude of the received

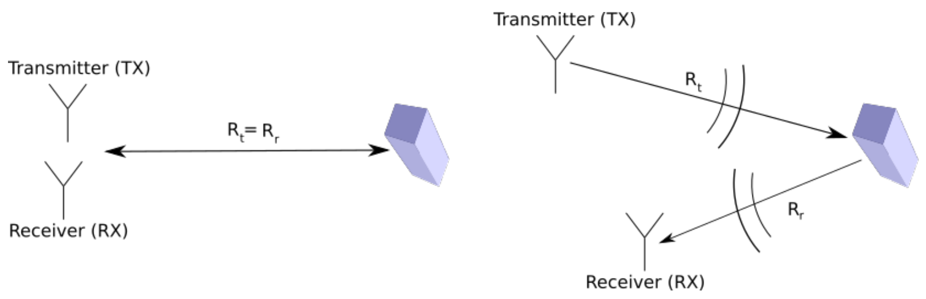


Figure 2.1: Monostatic and bistatic radar

signal mainly depends on the reflection properties of the materials of target objects, and its frequency shifts due to Doppler effect.

Considering the monostatic case, if we send continuous waves towards the object, they are reflected back with frequency shift $\delta f/f = -2v/c$, where v is the relative velocity of the object wrt to the antenna, and c is the speed of light. We call this operation mode *continuous wave* (CW) mode. The received power P_r depends on the transmitted power P_t, the antenna gains G_r and G_t, the wavelength λ, the distance R of the object and its radar cross section (RCS) σ:

$$P_r = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 R^4} \tag{2.1}$$

RCS is an orientation-dependent measure of reflectivity, and defined as:

$$\sigma = \frac{\text{Power reflected to receiver per unit solid angle}}{\text{Incident power density}/4\pi} \tag{2.2}$$

$$\sigma(\theta_o, \phi_o, \theta_i, \phi_i) = \lim_{R\to\infty} 4\pi R^2 \frac{|\vec{E}_s(\theta_o, \phi_o)|^2}{|\vec{E}_i(\theta_i, \phi_i)|^2} \tag{2.3}$$

where $\vec{E}_s$ and $\vec{E}_i$ are the electric fields of scattered and incident waves, (θ_o, ϕ_o) and (θ_i, ϕ_i) are the observation and incidence directions in target's local coordinates. It is generally reported relative to $1m^2$ in decibels: $\sigma_{dBsm} = 10\log_{10}(\sigma/1m^2)$. RCS plays an important role in the detectability of targets and reducing it is a system design objective in military applications [16].

The scattering characteristics of targets depend on the frequency f of wave. For a smooth target of size L and a wave with wavelength $\lambda = c/f$, there are three characteristic regions:

- $\lambda \gg L$: (Rayleigh) Low-frequency regime with small phase variations across the body, resulting in mostly isotropic scattering patterns.
- $\lambda \approx L$: (Mie) Resonance regime where all parts of the body contribute to scattering, resulting in oscillatory behavior in σ vs. λ.
- $\lambda \ll L$: (Optical) High-frequency regime with σ strongly angle dependent. Mostly isolated points are responsible for the scattering peaks.

The best illustration for these regimes is the cross section of a spherical object. In Figure 2.2, for spherical raindrops of radius a, we show the ratio of RCS to optical cross section πa^2 at 77GHz, where $k = 2\pi/\lambda \approx 1.6mm^{-1}$. We assumed the relative permittivity for water at this frequency is around $10 - 20j$. Notice, in the optical regime, for large raindrops $ka > 10$, $\sigma \approx \pi a^2$.

In the radar equation 2.1, we can control the antenna gains, transmitted power and the wavelength. If the object and its orientation is also known, σ, we can provide an estimate of its range; unfortunately neither is known. To recover the range, rather than CW operation, we send pulses, which echo back from the object and received after a duration $\delta t = 2R/c$. While these operation modes provide us with R and v, characterizing the object requires more information. If the object itself is moving and its orientation is changing due to a combination of translation and rotation, from observations of its radar cross section we can get its features. Similarly, for stationary distributed objects, we can scan the direction of interest or move the antennas to get

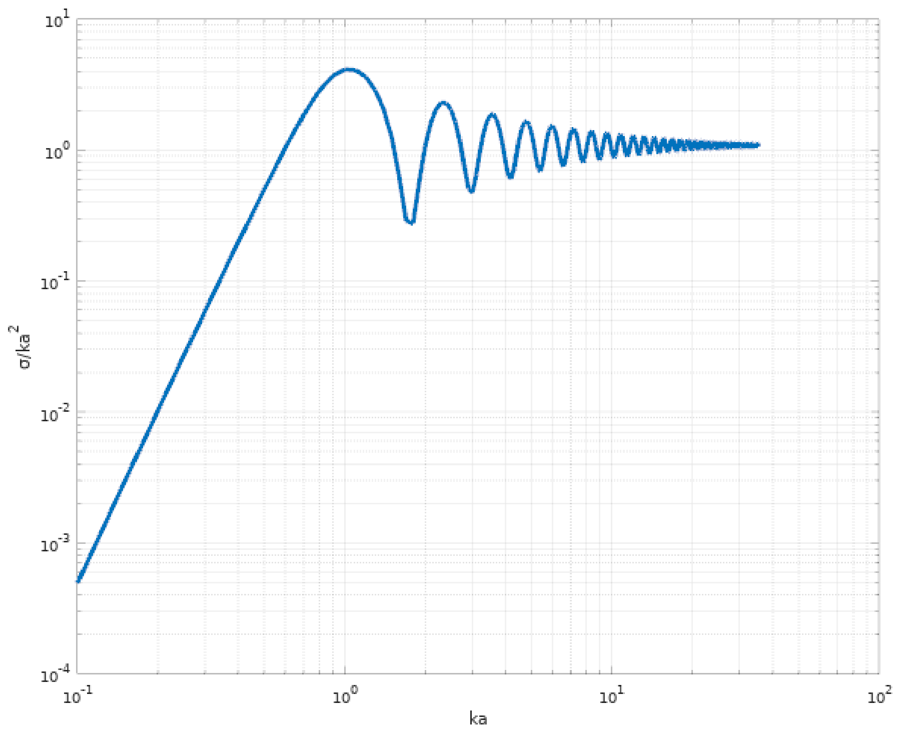


Figure 2.2: Scattering from spherical raindrops

cross section information from different directions. All these measurements can be combined to form an image.

2.2. Automotive Radar

Since the early experiments in 1970s, automotive radars have been under development and in use since 1999 for active safety and driver assistance functions [17]. Using well-known radar processing methods it's possible to measure the distance, azimuth angle and radial velocity of objects around the vehicle [18]. Today, with an increasing focus on autonomous driving, automotive radars are going through significant transformations to satisfy a new set of requirements for environmental perception. For example, they will generate dense point clouds, and be used in ego-motion estimation, radar-based localization and object classification [19]. Object detection, classification and tracking form a central application theme for all perception sensors. Researchers have successfully applied machine learning for these tasks with automotive radar data alone [20, 21, 22, 23, 24]. While it is possible to extract this information from individual sensors, fusing data from multiple sensors can produce more accurate results. Novel sensor fusion systems introduced in recent years mostly rely on deep neural networks; there are multiple works fusing radar data with cameras and lidars [25, 26, 27, 28, 29]. An important factor enabling these type of applications is the availability of datasets collected with radar sensors with improved resolutions [30, 31, 32]. The previous generation of automotive radars used 24GHz ($\lambda = 12.5mm$) frequency, and the current generation is based on 77GHz ($\lambda = 3.9mm$). With smaller wavelengths, we can achieve better resolutions.

Even though automotive radars have been deployed for two decades now, there are still challenges in modeling these sensors, including multipath propagation, separability of close objects, and the sensitivity of RCS [33]. It is also difficult to accurately simulate them. To our knowledge, there is no synthetic radar dataset available yet.

In the next two sections, we introduce some concepts used in automotive radars today, which we use in our simulations.

2.2.1 FMCW Radar

Frequency Modulated Continuous Wave (FMCW) is a common modulation scheme used in radars. In a particular form of it, the frequency is ramped up periodically at T_0, called *chirping* (Fig.2.3). We define a linear chirp $f(t) = f_c + Kt$ with three parameters: The frequency f_c around which this modulation happens is called the *center frequency*, the ramp rate K defines its slope, and $B = K\Delta T$ is the bandwidth. The duration of the chirp is $\Delta T = T_0 - T_R$, where T_R is the *reset time* between successive chirps. We can express the signal in complex form: $s(t) = e^{j\phi(t)}$, where $\phi(t) = \int 2\pi f(t)dt$ is the phase.

Consider a monostatic radar with 1 transmitter (TX) and 1 receiver (RX). When the signaled is echoed back from a target at distance R, it is delayed, $s(t-\tau)$, by an amount $\tau = 2R/c$ and its frequency is shifted by $f_d \approx 2f_c v/c$, where v is the relative radial velocity of the target and c is the speed of light. At the receiver, the returned signal is mixed with the transmitted signal and filtered to get components upto an IF bandwidth. Due to the shape of signal (Fig.2.3), for a stationary target, we observe a beat frequency at $f_b = K\tau$; therefore we can measure R in term of f_b: $R = Kc/2f_b$. The IF signal is sampled during the ramp at sampling frequency f_s, and we get $n_{max} = \Delta T f_s$ samples, where n is the sample index. By looking at the 1D-FFT of this signal segment, we can idenfity f_b for range estimation.

For a moving target $f_b = K\tau + f_d$ which couples R and v, and requires another independent measurement for velocity. The periodicity of the signal helps measuring the changes from chirp to chirp. For example, for a non-accelerating target, we can assume $R_p = R_{p-1} + vT_0$, where p is the chirp index. After $p_{max} = P$ chirps, we accumulate $n_{max} \times p_{max}$ samples. If we lay them out on a 2D grid, we can define a *fast time* with steps $1/f_s$ in n-direction, and also a *slow time* with steps T_0 in p-direction. After approximations, the sampled IF signal can be expressed as [18]:

$$d(n,p) \approx \exp\left\{ j2\pi \left[\left(K\frac{2R}{c} + f_d \right) \frac{n}{f_s} + f_d p T_0 + f_c \frac{2R}{c} \right] \right\} \tag{2.4}$$

An analysis with more terms is provided in [34]. Due to term $f_d p T_0$, using a 1D-FFT in p-direction, we can find f_d as a measure of v.

If we have one TX sending the chirped waveform, and n_{RX} RXs, for each RX we get $n_{max} \times p_{max}$ samples. Stacking these samples, we form a *radar cube* (Fig.2.4a). If the RXs are positioned as a uniformly spaced array (Fig.2.4b), depending on the direction of the target on the plane of observation, each RX receives an extra phase-shifted signal, $\delta\varphi = (2\pi/\lambda)\Delta_{RX}\sin\phi$, where Δ_{RX} is inter-RX spacing. With these additions, we can approximate the samples in the radar cube as:

$$d(l,n,p) \approx \exp\left\{ j2\pi \left[\left(K\frac{2R}{c} + f_d \right) \frac{n}{f_s} + f_c l \frac{\Delta_{RX}\sin\phi}{c} + f_d p T_0 + f_c \frac{2R}{c} \right] \right\} \tag{2.5}$$

where l is the RX index. With a 1D-FFT in l-direction we can estimate $\sin\phi$. The maximum angle that can be observed with a uniform linear array (ULA) of RXs is given by: $\sin^{-1}(\lambda/2\Delta_{RX})$.

For multiple targets, there will be multiple echoes at different amplitudes, coming from different directions, each shifted in time and frequency. The overall equation for the cube becomes:

$$d(l,n,p) = \sum_q \alpha_q d_q(l,n,p) + \xi_q(l,n,p) \tag{2.6}$$

where q is the target index, α is the return amplitude and ξ is additive noise.

The resolutions for range and velocity are give by: $\delta R = c/2B$, $\delta v = \lambda/2PT_0$. The frequencies in l-direction are uniformly spaced, but the arcsin results in an angle-dependent angular resolution $\delta\phi = \lambda/l_{max}\Delta_{RX}\cos\phi$. For $\Delta_{RX} = \lambda/2$ and $\phi = 0$: $\delta\phi = 2/l_{max}$. And theoretical field of view is: $-\pi/2 < \phi < \pi/2$.

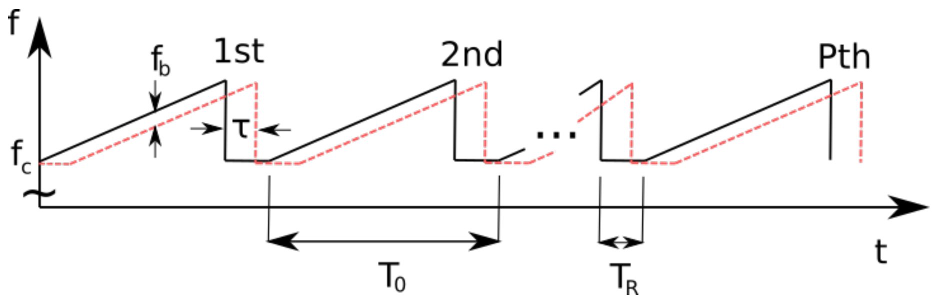
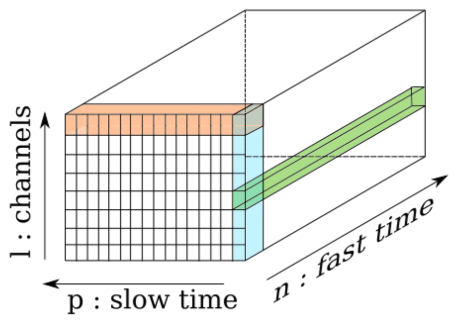
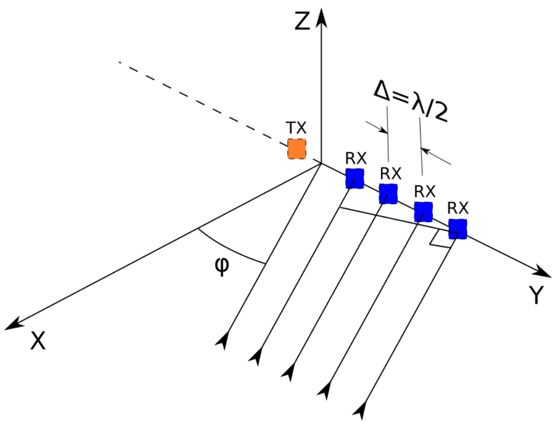


Figure 2.3: FMCW radar concept



(a) Radar cube concept



(b) Uniform Linear Array (ULA)

2.2.2 TDM-MIMO Radar

Increasing the number of RXs results in higher number of angle bins and better angular resolutions. However, there is a cost associated with additional TX/RX antennas. Time division multiplexing (TDM) with multiple TXs provides a solution. By adjusting the distance between the TX antennas to $\Delta_{TX} = n_{RX}\lambda/2$, and alternating the chirp sequence (Fig.2.5), the phases of the received signals at the RXs become equivalent to an arrangement with a single TX and $n_{TX}n_{RX}$ virtual RX antennas [18, 34] (Fig.2.6). Using this method, it is possible to create virtual antenna arrays that is not resticted to a line, and therefore can also provide elevation θ information. Different types of antenna arrangements and modes of sequencing the chirps will require specific processing; therefore for this thesis we will restrict our virtual antenna array to y-axis and a round-robin TX chirp sequence. (Fig.2.7).

In Table2.1, we provide example settings for a medium range and a short range radar system. With TDM, the total number of channels for the radar cube becomes $l_{max} = n_{TX}n_{RX}$, and the number of total chirps $P = n_{TX}p_{max}$.

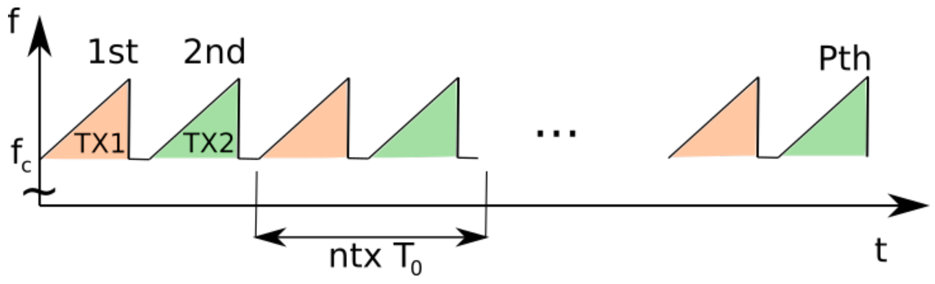


Figure 2.5: TDM chirp sequencing

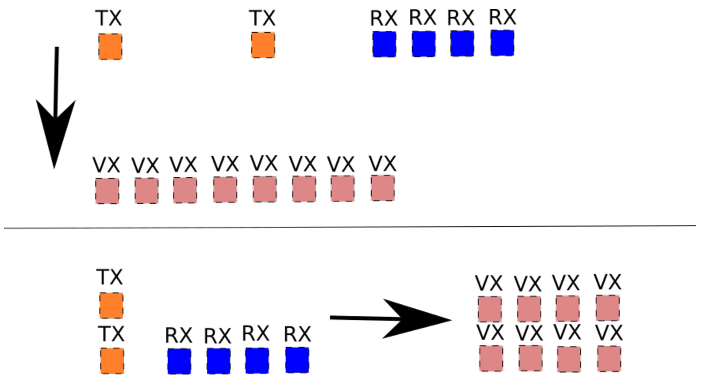


Figure 2.6: Virtual antenna arrays

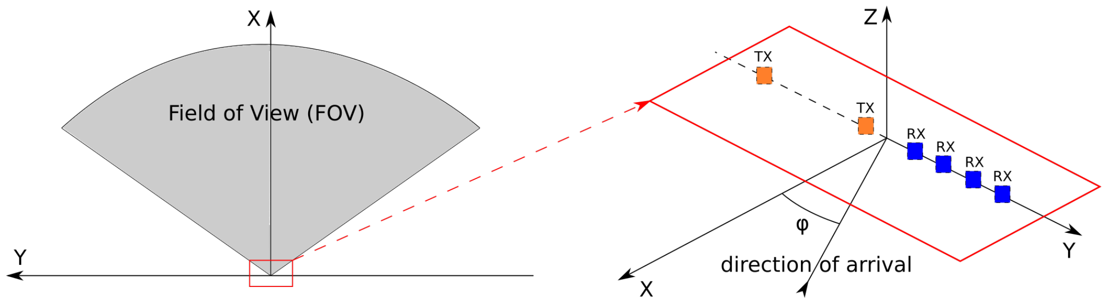


Figure 2.7: Radar field of view and MIMO antennas

2.3. Radar Simulation

Automotive radar simulations require electromagnetic wave solutions at millimeter wavelengths. Traditional techniques for solving Maxwell's equations involve partial differential equation solvers or integral equation solvers, such as finite element method, finite difference time domain, method of moments, and fast multipole method [35]. In the automotive context, space-time discretization employed with

Parameter	**Symbol**	**Radar1**	**Radar2**	**Units**
Number of TXs	n_{TX}	2	4	units
Number of RXs	n_{RX}	4	16	units
Ramp rate	K	10	50	THz/s
Pulse period	T_0	36	48	μs
Reset time	T_R	4	16	μs
Sampling Frequency	f_s	16	32	MHz
Center Frequency	f_c	77	77	GHz
Bandwidth	B	0.32	1.60	GHz
Repetitions	P	2×64	4×32	units
Maximum range	R_{max}	120	48	m
Maximum rel. vel.	v_{max}	13.5	5.07	m/s
Range resolution	δR	46.9	9.37	cm
Doppler resolution	δv	0.42	0.32	m/s
Angle resolution	$\delta\phi$	14	1.8	deg
Cube height	l_{max}	8	64	units
Cube width	p_{max}	64	32	units
Cube depth	n_{max}	512	1024	units

Table 2.1: Example radar settings

these techniques would result in solutions requiring very high memory and processing times. Since we are dealing with short wavelengths compared to the size of the simulation domain, a combination of approximate methods in geometric and physical optics is particularly helpful. In this section, we review these methodologies and the tools we use for our work.

2.3.1 RCS Simulation

Comptutation of radar cross section requires numeric integration of far-zone scattered fields; ignoring induced magnetic currents, we write electric $\vec{E}_s$ and magnetic fields $\vec{H}_s$ as:

$$\vec{E}_s(r,\theta,\phi) \approx \frac{-jk\eta}{4\pi r} e^{-jkr} \int_V \vec{J}(\vec{r}')e^{jk(\hat{r}\cdot\vec{r}')}dV' \tag{2.7}$$

$$\vec{H}_s \approx \frac{\hat{r} \times \vec{E}_s}{\eta} \tag{2.8}$$

where $\vec{J}$ is the induced electric current density due to the incident field, $k = 2\pi/\lambda$ is the wavenumber, $\vec{r'}$ is the local coordinate on the object, $\hat{r}$ is the direction of observation, and η is the impedance of material.

For perfect electric conductors (PEC), the fields can not penetrate into the body and reflect from the surface, therefore the volume integral turns into a surface integral with induced currents on the surface. For dielectric and magnetic materials, the fields penetrate into the body and the computations become more demanding. For simplicity we ignore magnetic materials in this thesis.

Matrix based methods solve these volume and surface integral equations by a mesh discretization. However, the number of degrees of freedom increases significantly with larger volumes and mesh details, and the application of these methods become infeasible. Therefore we turn to a collection of raytracing approximations called *microwave optics* [16], which include *geometric optics* (GO), *physical optics* (PO), *geometric theory of diffraction* (GTD) and *physical theory of diffraction* (PTD). We only use GO and PO in the thesis.

GO refers to using specular rays to approximate wavefronts and is based on a set of postulates:

- $\lambda \ll R$, where R is the wavefront curvature, meaning if we zoom in enough, the waves look like plane waves.
- Level sets of equal phases along the rays represent the wavefront.
- Rays travel in straight lines.
- Polarization can only change with reflection and refraction and is a constant along a ray.
- Multiple neighboring rays form a tube, with constant energy flux (Fig.2.8).

PO uses GO to compute surface currents on mesh primitives, then computes the far-zone fields, Eqn.2.7, to approximate RCS.

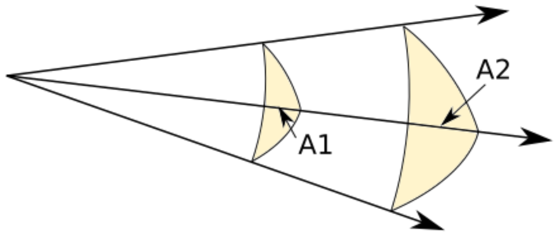


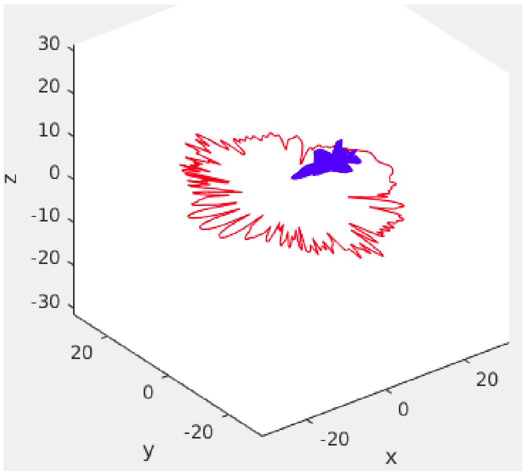
Figure 2.8: Raytube concept. Total energy passing through A_1 and A_2 are the same.

Tools. We identified three easy-access scattering simulation tools. (1) RaytrAMP [36] supports monostatic RCS of objects with PEC material, written in C++, but needs to be built on Windows OS. (2) RaiderTracer [37] is MATLAB based, free, allows multibounce effects, but closed-source. (3) POFACETS [38] is also MATLAB based, has a good set of examples and documentation. Therefore we decided to focus on POFACETS.

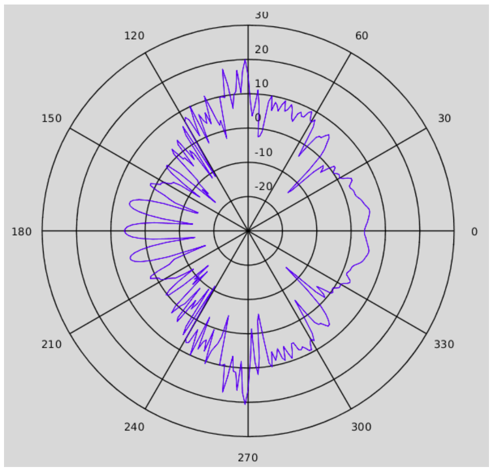
POFACETS. is a collection of MATLAB scripts with a GUI, and provides basic CAD models. It can compute monostatic and bistatic RCS of triangle-faceted models using PO approximations (Fig2.9). For material model, it can either use surface resistivity, or explicitly provided reflection coefficients for parallel and perpendicular polarizations. It also supports a simplified ground reflection model for additional scattering. For numerical integration of radiation integrals, it uses a semianalytical method based on [39], which requires two parameters for a Taylor series approximation. Selection of these parameters affects the results, and sometimes causes unwanted spurs. PO method itself also gets less accurate for observer directions away from the specular direction.

Partly because of its availability, some multiobjective optimization applications used it [40, 41]. Recently, it was also used to generate a dataset of time-varying RCS of rotating objects for classification [42].

It does not explicitly implement GO with rays, only iterates over facets and checks if they are exposed to the incident field by using facet normal. Therefore, unlike RaiderTracer, it doesn't provide multi-bounce effects, and not reliable except for simple geometies.



(a) RCS of an F-35 model at 300MHz



(b) Polar RCS plot at $\theta = 90$deg

Figure 2.9: Example monostatic RCS plot with POFACETS

2.3.2 Raytracing

In raytracing, we test a set of line segments, called rays, for intersection with a set of primitives (Fig.2.10). When an intersection is detected, we use the information from the hit point and the primitive to check some rules we set, and decide what to do next, which could involve branching, generating new rays recursively, computing and storing data or something else.

Raytracing of visible light has been successfully applied to static scenes to generate photorealistic rendering results under different lighting conditions. It is also applicable to other domains of physics where a ray representation is valid.

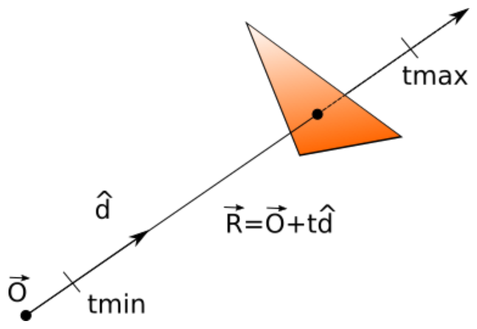


Figure 2.10: Ray-triangle intersection

Shooting and bouncing rays (SBR) is a raytracing approach to EM simulation, initially introduced in [43] for RCS computation of partially open cavities. It starts with a set of rays propagating towards the opening of a cavity, which is made of PEC and internally covered with a dielectric layer of certain thickness. GO is used to compute multiple, polarization dependent reflections of rays entering the cavity, together with adjustments to the wavefront curvature based on the reflector geometry. At the exit from the cavity, PO approach is employed to compute the backscattered radiation.

For better results, we have to use a higher number of rays, in return increasing the computational work. It is possible to accelerate both raytracing and physical optics portions of the SBR method using general purpose programming of the GPUs [44, 45, 46, 47].

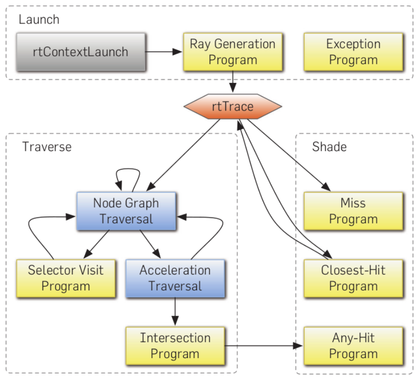
While there is significant flexibility of what the raytracing rules might be, there is a common structure to all raytracing programs. NVIDIA developed OptiX raytracing engine [48, 49] which has a software development kit freely available to developers. With raytracing cores in current GPUs, hardware acceleration enables real-time raytracing. Several researchers reported implementations of SBR method benefiting from this framework [50, 51], and recent activity in automotive radar simulations also indicates the use of SBR method and OptiX framework [52, 53, 54].

OptiX defines a state diagram for the lifetime of a ray and a set of user-defined programs that are executed on certain conditions (Fig.2.11a). A *context* is a collection of user-defined programs, variables and buffers, that contain information about primitives, lighting, material properties, etc. We can trace multiple types of rays within a particular context. Each ray type is assigned to a ray generation program that defines origin, direction, start time and end time for each ray. A launch is a single run of ray generation programs from an entrypoint, which could be a pin-hole camera. Rays can carry user-defined data called payload. Ray generation program can initialize the payload and call the rtTrace function to start tracing.

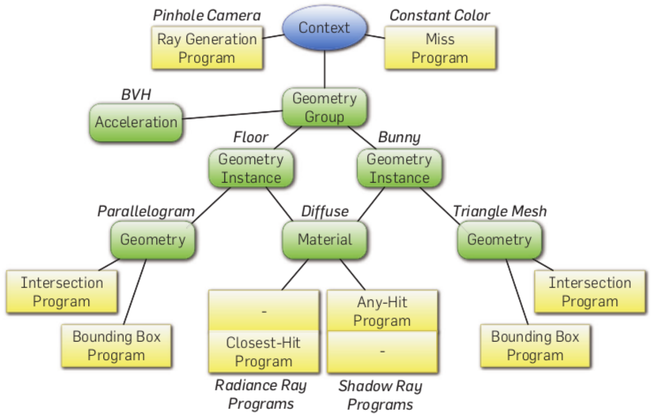
The scene information is stored as a tree structure (Fig.2.11b). Nodes can be geometries, materials, groups of geometries or acceleration structures. When a ray is traced, it's checked for intersection with the primitives of geometry instances. An intersection program is bound for each geometry instance. The SDK comes with default programs for ray-triangle intersections. What happens when a particular ray-type and material pair is programmed with hit programs. They are called after intersections are reported. A ray may intersect with multiple primitives and in many applications the closest one is the most relevant. This particular intersection is handled by closest hit program. It may also happen that a ray does not intersect with anything, in that case a miss program is executed.

2.3.3 Game Engine Based Simulators

Game engines are highly complex and optimized software frameworks [13]. They provide a development environment for game designers with advanced level editors, and game applications use their programmability with APIs. Game levels contain multiple game objects, sometimes called actors. They may have 3D represen-



(a) OptiX call graph



(b) OptiX context

Figure 2.11: NVIDIA OptiX raytracing framework [49]

tations with a mesh and materials associated with textures, and the game designer defines their behavior. A physics engine updates the transform of each active game object based on the laws of mechanics and can provide animation effects by moving vertices or segments of the mesh with respect to the other parts. A rendering engine uses the mesh and transform information of each game object, decorated with textures, to create a scene and displays it as visual feedback to the player. Since two separate tasks, simulation and visualization, use the same data simultaneously, to maintain real-time performance, they are handled by different threads, and data sharing between them is defined and managed by the game engine framework.

There are multiple game engines available, targeting various platforms and employing different programming languages. The two most popular game engines are Unreal Engine by Epic Games [55] and Unity by Unity Technologies [56]. Both of them allow the creation of 3D levels and let the players control the motion of a camera with keyboard and mouse, to navigate the 3D level to perform game-related tasks.

In recent years, 3D games got attention from researchers working on autonomous vehicle development. Games provide excellent simulated environments for software-in-the-loop and hardware-in-the-loop testing. Also, for perception research, they enable capturing camera and other sensor data to train neural networks. The ground truth information is directly available and can be used to label these synthetic datasets automatically. We can evaluate the performance of computer vision tasks with specific sensor specs and arrangements. We can record user input and train neural networks to control the vehicle. Using a software agent as the user, we can also research reinforcement learning. Since games allow multiple players, they are suitable for multi-agent simulations, and we can check the prediction and decision-making performance of the system during complex, repeatable scenarios. This experimentation feature has enormous value for system designers. In particular, rare events are unlikely to appear in real-world data collection, but we can systematically generate them in simulation.

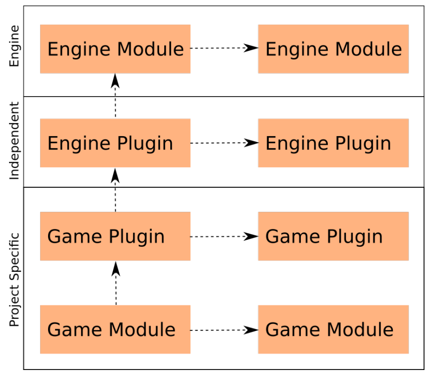
Researchers at Microsoft released AirSim as a drone and car simulator using Unreal Engine [8]. Intel Labs, TRI, and CVC created CARLA simulator, also based on Unreal [7]. LG Silicon Valley Lab released LGSVL simulator using Unity [9]. There are reviews available, including a few other simulators [57, 58, 59]; we will only highlight these three popular projects.

Unreal Engine. by Epic Games is a source-available commercial software [55]. Access to their source code is possible with free developer registration on their website and joining Epic Games organization on Github. They provide binaries for Windows and Mac systems and instructions for building on Linux with clang toolchain. Unreal Engine is highly modular and can be extended with modules and plugins (Fig.2.12a). CARLA and AirSim are examples of game applications, and they contain Carla and AirLib game plugins, respectively.

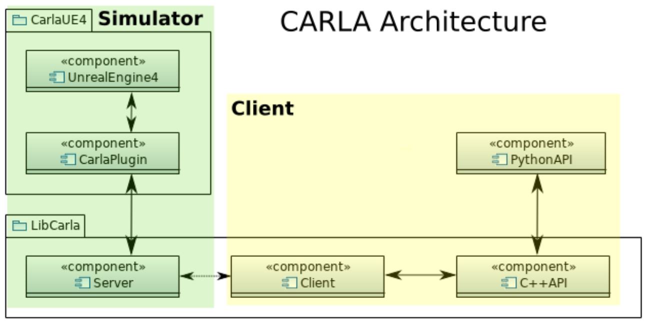
CARLA. (Car Learning to Act) is an autonomous vehicle simulator with a focus on urban environments [60]. It is open source under MIT License and available on Github. It provides free assets of buildings, vehicles, pedestrians, infrastructure elements, etc., and comes with multiple predesigned towns. It emphasizes multi-agent aspects of complicated urban driving and provides autopilot vehicles and pedestrian controllers. It has a client-server architecture with a Python API (Fig.2.12b). It allows multiple clients to connect to the simulator and perform actions. With the API, we can change agents' behavior, generate traffic, control the weather, etc. CARLA comes with a full suite of autonomous driving related sensors: camera, depth camera, lidar, radar, IMU, and GPS. It is possible to extend this sensor suite. It can import maps generated by standard tools and perform traffic scenario simulation. It can also work with ROS, making it available to a plethora of robotics algorithms and applications developed earlier.

AirSim. was initially developed for drone simulation to study reinforcement learning [61]. It is also open-source with MIT License. It comes fully equipped for hardware-in-the-loop testing with PX4 controller. It can work with cars, has experimental support for Unity and wrappers for ROS. Similar to CARLA, it provides a client-server architecture with a Python API. It features multiple sensors: camera, barometer, IMU, GPS, magnetometer, distance sensor, and lidar. AirSim defines sensors as a third-party library plugin, called AirLib. Unlike Carla, this plugin is less coupled to Unreal Engine.

Unity. by Unity Technologies is also a commercial game engine with a different business model than Unreal [56]. It is also cross-platform and used in multiple different industries. In Unity, we write scripts in C# and use MonoDevelop to compile them. We can introduce external code in C++ via native plugins. Similar to Unreal, its API is well-documented and available online. LGSVL and Baidu Apollo



(a) UnrealEngine modularity



(b) CARLA components

Figure 2.12: Unreal Engine module/plugin concepts and CARLA architecture

simulators are based on Unity. The latter was not publicly released at the time of this writing, so we only use LGSVL simulator and simple scenes in Unity editor.

LGSVL Simulator. by LG Silicon Valley Lab is a more recent simulator with a focus on simulation-as-a-service for commercial R&D and applications [62]. It is open-source and available on Github, but has a custom user license. It comes with a few vehicle and town models, and a full sensor suite for autonomous driving with a sensor plugin interface. It is suitable for software-in-the-loop and hardware-in-the-loop testing, synthetic data generation, and V2X infrastructure simulation. It can be used to create digital-twins of real-world locations. Like other simulators, it has ROS bridge functionality and can work with autonomous driving stacks like Autoware.

Chapter 3: Radar Sensor Plugin

In this chapter, firstly, we clarify our goals and motivate multiple use cases to identify minimum software requirements. Then, we define a software architecture satisfying these requirements. We provide high-level details of software design and implementation compatible with the defined architecture. Lastly, we provide examples of how to integrate this plugin to various game engines and simulators.

3.1. Software Requirements

Our main goal is to develop a radar sensor simulator integrated into multiple autonomous vehicle simulators via plugin interfaces. As discussed in Chapter 1, we would like our simulator to be portable between game engines. This implies that we have to extract primitive data from the engines and homogenize it before exposing it to our radar simulator. These extra steps taken impose performance penalties and may dictate certain design decisions, like the use of shared memory. In addition, we plan to create multiple radar units in the game and provide radar processing capabilities to each of them. Therefore, on a personal gaming PC, it can quickly become impossible to design a real-time system.

Another drawback working outside of the game engines is the difficulty of development in the absence of sophisticated utilities like game editors. At a minimum, we have to reconstruct a 3D world from game data, visualize it, and navigate it. These extra features will also ease debugging. Since there will be multiple software components that have to work together, we have to test them independently before any integration efforts. At the interfaces of every major component in the system, which makes significant amounts of data transformations, we have to dump intermediate data and replay it to test the next component.

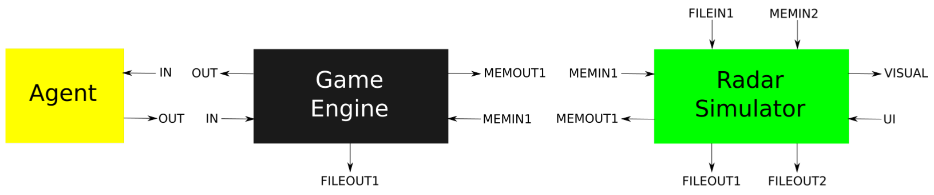


Figure 3.1: Abstract blocks of the overall system

To discuss specific use cases, let us introduce abstract block diagrams of components of the overall system (Fig.3.1). *Agent* can be a human or some software directly interfacing with the *GameEngine*. *GameEngine::IN* could be user inputs from keyboard, mouse, or API calls. *GameEngine::OUT* could be a display or some other data provided as feedback to the *Agent*. *GameEngine::FILEOUT1* port is used for saving game data. Memory input/output ports of *GameEngine* and *RadarSimulator* symbolize shared memory read/write operations. *RadarSimulator::FILEIN1* is used for material databases, predefined configurations, antenna patterns, etc. *RadarSimulator::FILEOUT1* symbolizes recording homogenized game data, whereas *RadarSimulator::FILEOUT2* symbolizes saving plots, raw and processed radar cubes. *RadarSimulator::MEMIN2* is used for replaying homogenized data recordings. In light of these definitions, we created a table of essential use cases (Table 3.1) and listed requirements (Table 3.2).

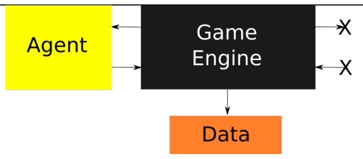
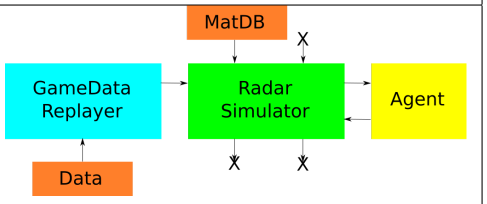
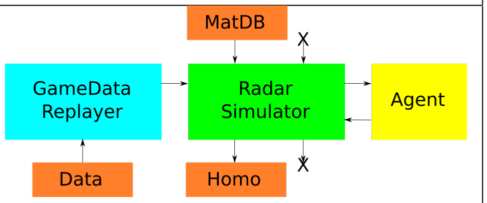
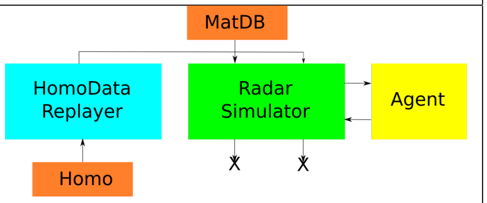
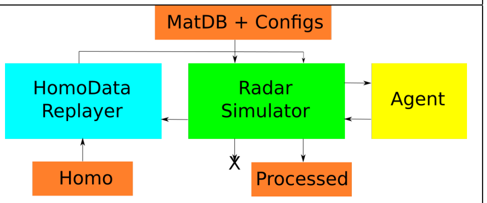
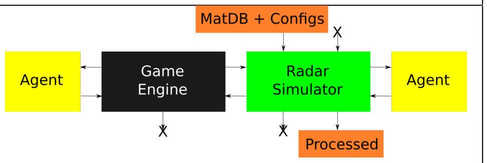
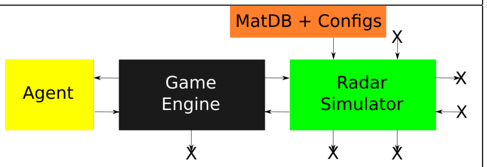
UseCase1	Save game data	
UseCase2	Replay game data	 MatDB X GameData Replayer Radar Simulator Agent Data X X
UseCase3	Record homogenized data	 MatDB X GameData Replayer Radar Simulator Agent Data Homo X
UseCase4	Replay homogenized data	 MatDB HomoData Replayer Radar Simulator Agent Homo X X
UseCase5	Raytrace and process data	 MatDB + Configs HomoData Replayer Radar Simulator Agent Homo X Processed
UseCase6	Integrated Radar Simulator	 MatDB + Configs X Agent Game Engine Radar Simulator Agent X X Processed
UseCase7	Headless Radar Simulator	 MatDB + Configs X Agent Game Engine Radar Simulator X X X X X

Table 3.1: Use case block diagrams

R.1	We should be able to record raw game data required for radar simulation.
R.2	We should be able to replay saved game data.
R.3	We should be able to visualize replayed game data.
R.4	We should be able to record, replay and visualize homogenized game data.
R.5	We should be able to define a radar with different antenna configurations.
R.6	We should be able to set sweep configurations for a radar.
R.7	We should be able to spawn radars in the game.
R.8	We should be able to attach radars to vehicles.
R.9	We should be able to visualize what each radar sees.
R.10	We should be able to explore the field of view of each radar visually.
R.11	We should be able to construct a radar cube using raytracing.
R.12	We should be able to perform FFTs on the radar cube.
R.13	We should be able to create range-angle and range-doppler plots.
R.14	We should be able to overlay these plots on the visualization.
R.15	We should be able to send processed data back to the vehicle simulator.

Table 3.2: List of requirements

3.2. Software Architecture

Based on the use cases and requirements, we defined an initial architecture for *RadarPlugin* and *RadarSimulator*. In Fig.3.2, we show their internal layers referenced to UseCase6. We have the following responsibilities for each layer:

- *RadarObserver*: Identify all relevant objects within the game that are of interest for radar simulation and communicate them to *RadarBridge*.
- *RadarBridge*: Put object data on shared memory or dump to filesystem. Also, receive processed data from shared memory and distribute it to individual sensors in the game.
- *GameDataReceiver*: Copy object data from shared memory.
- *SimulationDataManager*: Create internal representations of game objects.
- *RaytracerDataManager*: From internal representations, generate raytracer specific data to be consumed in simulation. Put these in shared memory or dump them to the filesystem.
- *RaytracerSimulator*: Perform raytracing using radar configurations and antenna patterns, generate radar cube, and process it to create range-angle and range-doppler plots. Put results on shared memory or dump to filesystem.
- *UI + Visualization*: Provide an OpenGL based viewport and user interactions with mouse and keyboard.

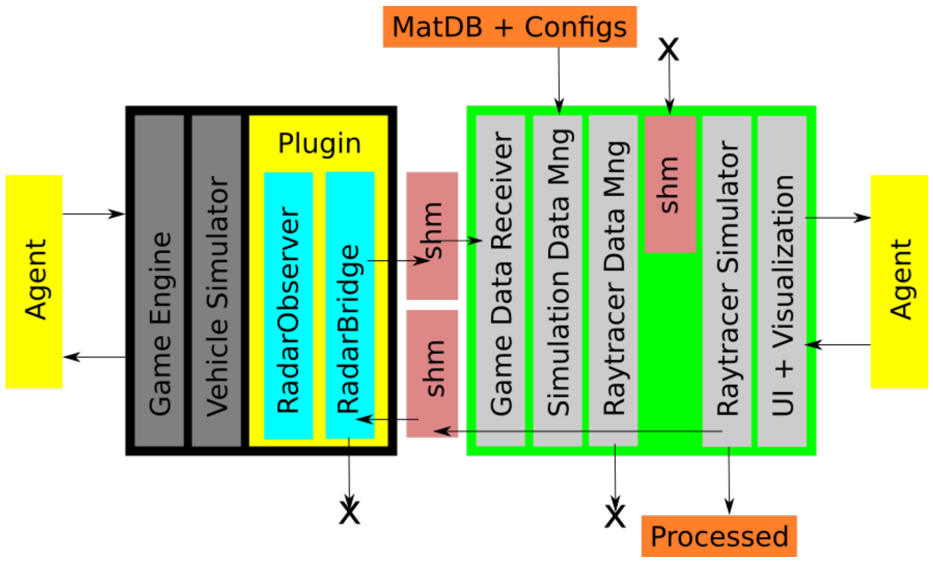


Figure 3.2: Software architecture diagram

3.3. Software Design

3.3.1 RadarPlugin Design

Our goal with *RadarPlugin* is to create an interface between our simulator and the vehicle simulator. Inherently it has to know about both the game engine and the vehicle simulator. Specifically for Unreal Engine, we want this to be a game plugin, like *Carla* itself, and be a part of the game project *CarlaUE4*. Considering we want to have our simulator externally developed, we decided to define it as a third party library plugin. We call this library *libRadarSim.so*, which is generated from the software components in the green block diagram (Fig.3.2).

RadarObserver should be unique, be created when the game starts, and should survive until the end. At every tick, it should go through all objects and keep track of newly spawned or removed objects. It should also wait for the physics engine to finish to have an accurate representation of the world.

RadarBridge should serialize mesh, material, and transform data of every object *RadarObserver* requests. It should also parse multi-radar processed data received from *RaytracerSimulator* and provide any data packet to the radar sensor it belongs to.

To satisfy **R.1**, *RadarPlugin* also dumps game data into a folder for each tick and follows a particular file naming convention.

3.3.2 RadarSimulator Design

Based on the use cases and the architecture, we decided to split *RadarSimulator* into two modules. The first module *GameMeshViewer* is responsible for data transformations and creates a 3D full scene mesh which we can visualize and navigate (Fig.3.3). It can be run independently with a *GameDataReplayer* as in UseCase2. These satisfy **R.2** and **R.3**. It can also save full scene mesh, velocities, and sensor transforms to partly satisfy **R.4** in UseCase3.

The second module is *RaytracerSimulator*, which is slightly more complicated because it has the responsibility to satisfy all remaining requirements (Fig.3.4). Firstly, it completes **R.4** by providing a *FullSceneReplayer*, which reads recorded data from the filesystem and puts them into the shared memory as input to the simulator (UseCase4). It also has a more complex visualization and user interface design due to multiple radar support.

In the end, application radars will be defined by the *Agent* using a script. Sweeps can be configured by indicating a prototype name from a preconfigured list known by the simulator, or individually by overriding default parameter values (**R.6**). Since this information originates at the game engine side, it has to be communicated as part of *RadarState*. Similarly, antenna pattern filenames have to be specified for each antenna. The pattern files should contain gain values for a grid of points on the unit sphere (**R.5**).

The vehicle simulator needs proxy objects for each radar in the simulator. They do not have to define a mesh but should be constructed as attachable elements. Therefore we need to modify the vehicle simulator's source code to support our plugin (**R.7**, **R.8**).

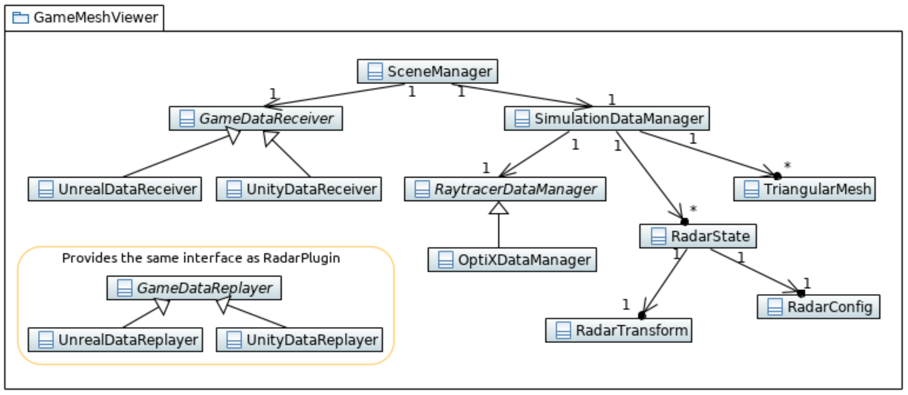


Figure 3.3: Simplified class diagram for GameMeshViewer module

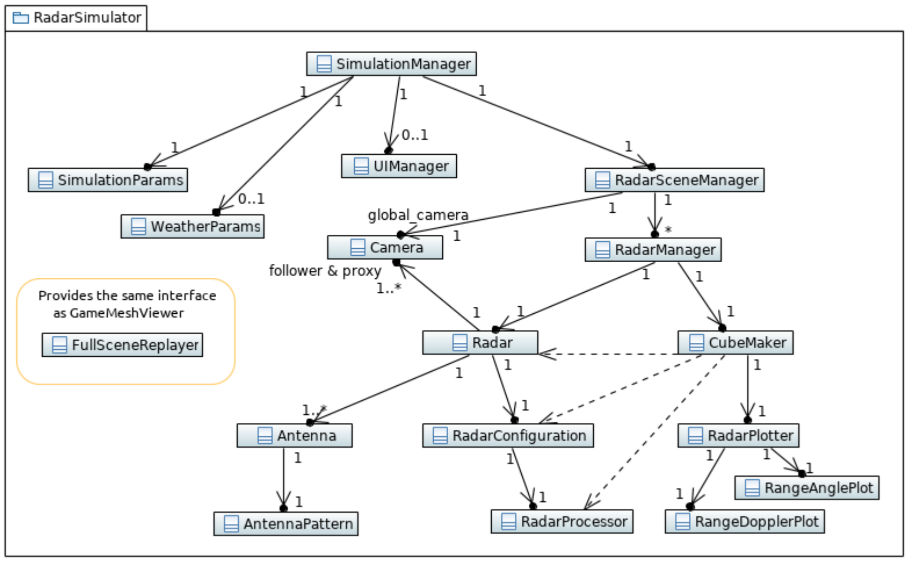


Figure 3.4: Simplified class diagram for RaytracerSimulator module

3.3.3 Visualization and UI Design

OptiX samples implement a simple window using OpenGL Utility Toolkit (GLUT) library. They utilize mouse for zooming in/out and arcball rotation. Ray-tracing results are copied over to a 2D texture buffer for display. They also provide SDK utility functions to overlay text. We did not need a graphical user interface and decided adding more functionality to these with keyboard shortcuts would be enough.

Firstly, to provide navigation within the game environment, we need to add translations and rotations to the raytracing camera. Translations can be axis-aligned with the global coordinate system or the camera's local coordinates. For rotations, it is more intuitive to use local coordinates by implementing yaw, pitch, and roll motions. We used QWEASD keys for translation as they are common in 3D games. For rotations, we used HOME, END, and arrow keys.

When radars are added to the game, we should be notified and see a list of available units to investigate. With multiple radars in the game, we need some functionality to select and/or cycle between them. Numeric keys are appropriate for this requirement. When the user selects a radar, we can move the camera directly to display its point of view. Since a radar's field of view can be visualized better from the top, we can assign another camera that follows the radar. We can allow the user to lock or unlock this follower camera to the radar. Also, when a vehicle is removed from the game, any radar attached to it will go out of scope. If a particular radar on that vehicle was selected at that instant, we should automatically fall back to a global camera view. We either have to define a default initial global camera location or keep track of that last coordinates of the global camera. Based on all these, we decided to define two cameras per radar, a proxy and a follower, and a single global camera. A follower initially starts locked at the radar like the proxy. Keypress L changes its locked state, and camera motion keys can be used to move it around the vehicle.

We should also display other useful information on the screen, like the position of the camera and the selected radar unit. See Figure 4.15 for implementation.

3.4. Software Implementation

We developed everything on a personal gaming PC with NVIDIA GeForce RTX 2060 Super GPU. We mainly used `clang` as the compiler, `googletest` for testing, a mixture of `CMake` and `make` based workflows, `OpenCV` for plotting, a mixture of `Boost` library, and some recent C++ additions. We also used `CUDA` and `thrust` library, mainly for signal processing. Below we highlight domain level details of our implementation.

3.4.1 Coordinate Systems

We decided to implement everything on a flat earth, in a right-handed coordinate system with the gravity vector in -z direction. Game engines have different conventions for handedness and even default gravity direction (Fig.3.5). For an agnostic mesh viewer, either camera coordinates and motion semantics have to be adjusted, or the whole mesh has to be represented in a common coordinate system. The former solution does not penalize performance and is helpful during the debug phase. The latter requires updates to vertex positions, translations, and quaternions; it is helpful during the replay phase and for engine-independent tasks. Since we update internal mesh representations only at spawn time, its performance penalty is not significant, except for the first tick during which the whole map requires an update. The software component responsible for game data capture and feedback has to provide these functionalities.

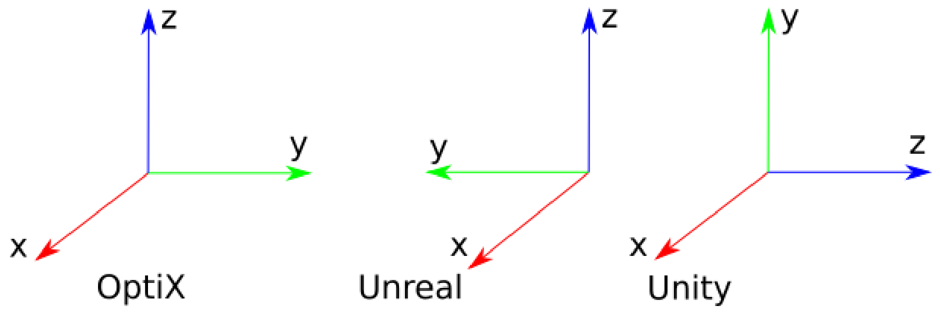


Figure 3.5: We converted Unreal and Unity coordinates into a common system

3.4.2 Antenna Patterns

Numerous antenna designs exist for 77GHz radar systems [63]. For our application, we only need to understand how we should assign a power level and polarization to our rays in ray generation program, and how we can compute each ray's contribution to the radar cube on the receive-side.

The total power radiated from an antenna, P_0 is distributed over the unit sphere:

$$P_0 = \int_{S^2} U(\theta, \phi) d\Omega, \quad d\Omega = \sin\theta d\theta d\phi \tag{3.1}$$

where the distribution function $U(\theta, \phi)$ has units of Watts. We can compare an antenna to an isotropic one with the same total power $U_0 = P_0/4\pi$, by defining *directive gain*:

$$D(\theta, \phi) = \frac{U(\theta, \phi)}{U_0} \tag{3.2}$$

Directivity is the maximum of directive gain $D_0 = \max_{S^2} D(\theta, \phi)$, and *antenna gain* takes into account all nonidealities $G_0 = eD_0$, where $0 < e < 1$ is called *efficiency*.

Far from the antenna, the E-field takes the form:

$$\vec{E} = \vec{p}(\theta, \phi) \frac{e^{-jkr}}{r} \tag{3.3}$$

where $\vec{p}$ is the field pattern. Also, E-field and power per unit area are related by:

$$U(\theta,\phi) = \frac{|\vec{E}|^2}{2Z_0} r^2 = \frac{|\vec{p}|^2}{2Z_0} \tag{3.4}$$

where Z_0 is the impedance of free space.

When we generate a set of ray directions on the sphere, we have to partition the total power P_0 between the rays by assigning spherical patches A_i to each ray i:

$$P_i = \int_{A_i} U(\theta,\phi) d\Omega, \quad \sum_i P_i = P_0, \quad \cup_i A_i = S^2 \tag{3.5}$$

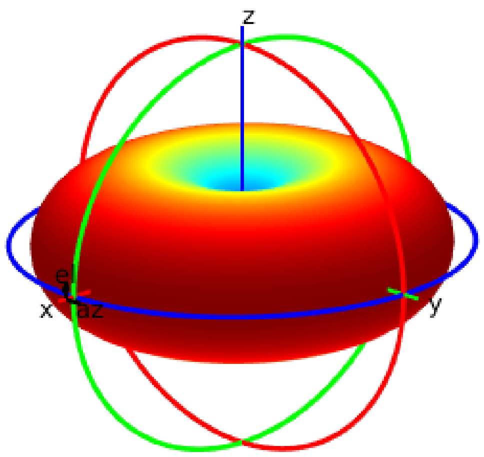
Ray sampling from a uniform distribution with latitude-longitude or octahedral concentric mapping is common in practice. Even though these methods would statistically achieve their goal, they stretch the sampling domain [64]. We prefer the ray-to-ray distance between neighbors almost constant. We noticed the use of Fibonacci lattice is more appropriate in this case [65, 54]. Since Fibonacci lattice divides the full solid angle almost uniformly, we can make a first-order approximation to Eqn.3.5 by assuming the power density is constant over the patch:

$$P_i \approx U(\theta_i,\phi_i) \int_{A_i} d\Omega = \frac{4\pi}{2N+1} U(\theta_i,\phi_i) \tag{3.6}$$

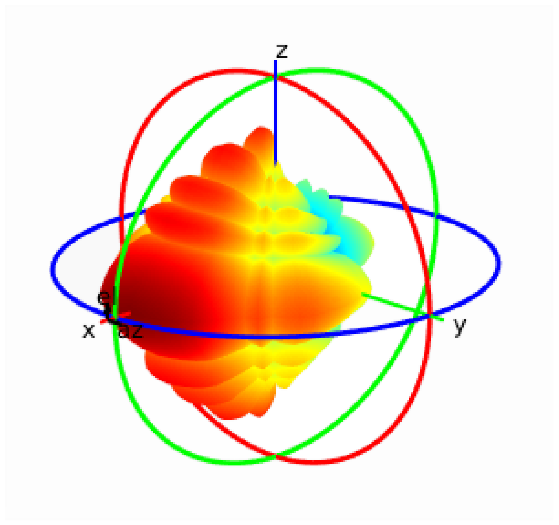
We usually indicate the polarization of an antenna with H and V symbols, corresponding to horizontal and vertical polarizations, respectively. If the radiated waves have electric field directions mostly in the elevation direction, we call the antenna vertically polarized; if the electric field is in the azimuth direction, it is horizontally polarized. Based on TX and RX polarizations, we may have, HH, HV, VH and VV type systems. In the automotive context, we will assume microstrip patch antennas are used in VV setting.

Practical antenna patterns can be quite complicated, and considering the effects of bumpers [52], it is best to utilize measurement results of the patterns as inputs, for example, on a uniform grid of θ and ϕ values. We assumed the user would provide us antenna patterns for each TX and RX antenna. There are also some standard antenna file formats available, like MSI and PLN, but they specify patterns only in equatorial ($\theta = \pi/2$) and longitude cuts ($\phi =$ boresight). MATLAB antenna toolbox uses them to approximate on a full grid.

Assuming we have $U(\theta_j, \phi_k)$ on a grid, we decided to apply bilinear interpolation to compute $U(\theta_i, \phi_i)$. Depending on the use case, other internal representations of the pattern can also be used, like a set of spherical harmonic coefficients [66, 67], with an argument that only some of these coefficients are sensitive to large frequency sweeps. It requires fitting measurement values to spherical harmonics basis, but currently presents unnecessary complexity for our project.



(a) Dipole antenna pattern



(b) Artificial antenna pattern

Figure 3.6: Example antenna patterns. On the left, the radiation pattern of a dipole antenna, and on the right, an artificially generated high directivity antenna pattern, similar to antenna patterns of automotive radars. The colors represent power in dB.

3.4.3 Antenna Transforms

Antenna gain is defined within antenna coordinate frame A. When we generate a ray in A, we need to know its origin and direction in the world coordinate frame W. Likewise when we receive a signal in W, we have to convert its direction back to the corresponding receive antenna's coordinate frame in order to assign reception gain correctly.

We can define a multi-antenna radar system with transforms from each antenna coordinate frame to the radar coordinate frame, $\mathbf{T}^{RA}$. From the game engine, we can get the transform of a vehicle $\mathbf{T}^{WV}$ or a radar unit within the world directly, $\mathbf{T}^{WR}$. Also, we have to specify how a radar is mounted on the vehicle $\mathbf{T}^{VR}$. Overall, we have a chain of transformations:

$$\mathbf{T}_{kji}^{WA} = \mathbf{T}_{k}^{WV}\mathbf{T}_{kj}^{VR}\mathbf{T}_{ji}^{RA} \tag{3.7}$$

where i, j, and k are antenna, radar, and vehicle indices, respectively. In general, each transform $\mathbf{T}$ can be represented as a 6 degrees of freedom matrix in homogenous coordinates:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.8}$$

where $\mathbf{R}$ is a rotation matrix and $\mathbf{t}$ is a translation vector. Game engines usually provide a unit quaternion $\mathbf{q}$ representation instead of $\mathbf{R}$, and library functions to get 16 float values for $\mathbf{T}$.

3.4.4 Material Properties

We need material properties to compute reflected and scattered fields. Frequency-dependent impedance of a material medium is given by Eqn.3.9, where ω is the

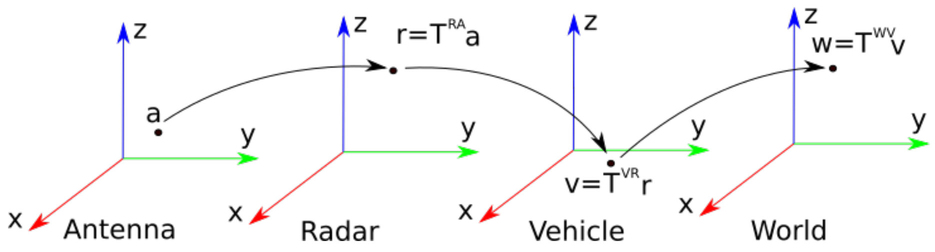


Figure 3.7: Transforms between antenna, radar, vehicle and world coord systems

angular frequency of the electromagnetic wave, and μ, ϵ, σ are the permeability, permittivity, and conductivity of the material, respectively. For a perfect electric conductor, as $\sigma \to \infty$ then $Z_{PEC} \to 0$. For a non-conductive material as $\sigma \to 0$ then $Z_{NC} \to \sqrt{\mu/\epsilon} = Z_0\sqrt{\mu_r/\epsilon_r}$, where $Z_0 = \sqrt{\mu_0/\epsilon_0} \approx 376.73\,\Omega$ is the impedance of free space, $\mu_r = \mu/\mu_0$ and $\epsilon_r = \epsilon/\epsilon_0$ are the relative permeability and permittivity of the material. The propagation constant is given by $\gamma = \alpha + i\beta = \sqrt{j\omega\mu(\sigma + j\omega\epsilon)}$; for a conductive material $\sigma > 0$, it has a real part which corresponds to an exponential power loss.

$$Z = \sqrt{\frac{j\omega\mu}{\sigma + j\omega\epsilon}} \tag{3.9}$$

When a wave comes to an interface between two different materials, some of the incoming power is reflected and the rest is transmitted (Fig.3.8). Reflection and transmission coefficients for perpendicular and parallel polarized waves are different and are given by equations in Table3.3. Also, boundary conditions result in geometric constraints: $\sin\theta_i = \sin\theta_r$ and $\sqrt{\epsilon_1\mu_1}\sin\theta_i = \sqrt{\epsilon_2\mu_2}\sin\theta_t$. These are known as Snell's laws.

From the game engines, we extract a surface mesh representation of the geometry; hence we only have triangles to assign materials to, but not tetrahedrons with a material volume. This poses an important problem when defining the transmission of waves between two volumes separated by a triangular interface. Therefore we had to

	Reflection Coefficient	Transmission Coefficient
TE	$\Gamma_{\perp} = \frac{\eta_2 \cos\theta_i - \eta_1 \cos\theta_t}{\eta_2 \cos\theta_i + \eta_1 \cos\theta_1}$	$\tau_{\perp} = \frac{2\eta_2 \cos\theta_i}{\eta_2 \cos\theta_i + \eta_1 \cos\theta_t}$
TM	$\Gamma_{\parallel} = \frac{\eta_2 \cos\theta_t - \eta_1 \cos\theta_i}{\eta_2 \cos\theta_t + \eta_1 \cos\theta_i}$	$\tau_{\parallel} = \frac{2\eta_2 \cos\theta_i}{\eta_2 \cos\theta_t + \eta_1 \cos\theta_i}$

Table 3.3: Reflection and transmission coefficients for TE & TM waves

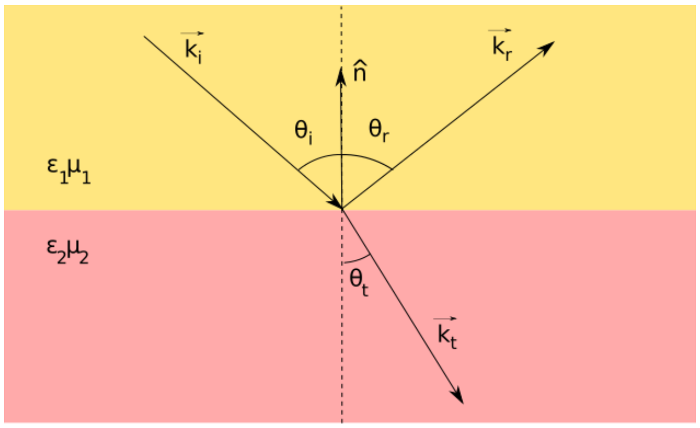


Figure 3.8: Reflection and refraction of plane EM wave at interface

make some assumptions and simplifications: incoming volume is always air and the triangles represent one of the material arrangements shown in Fig.3.9. We exposed ϵ_r, μ_r, σ, and film thickness t to material properties of triangles, and decided to adjust properties of air based on the weather conditions.

If the application considers only a single frequency or the material properties do not change considerably within the sweep bandwidth, another approach would be to provide reflection and transmission coefficients for various material stacks. We did not want to take this approach since it would require a lookup table for other frequencies. Providing a small set of thickness values for material stacks together with their permitivities, permeabilities and conductivities, allows wider frequency applications as long as those values aren't too sensitive to frequency. We believe a great majority of relevant materials in the field can be mapped into our simplified set. Also, extending this set requires minor code changes.

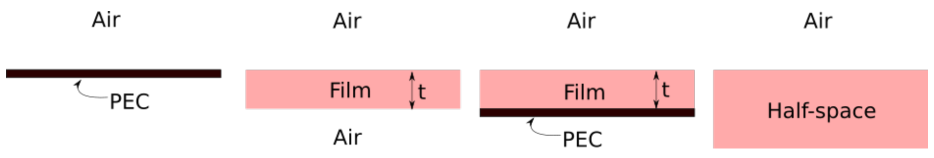


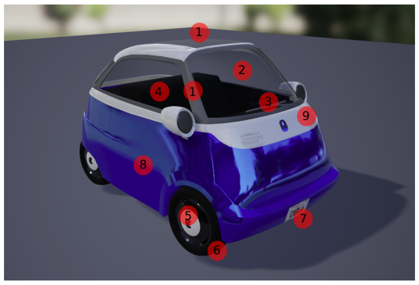
Figure 3.9: Simplified material models

In the simulations, we tag metals as PEC. If they have some coating, we use a dielectric layer with thickness t, which could be any thick paint, water layer, etc.

For transparent materials, we define a single dielectric layer with both sides air. The thickness for automotive glass is between $3 - 5mm$ with $\epsilon_r \approx 6.75$ and loss tangent $\sigma_c/\omega\epsilon \approx 0.003$, resulting in a transmission loss around 2-5dB at 77GHz depending on incidence [68].

We consider anything else as dielectric, which fills the other half-space. For more complex material compositions, parameters can be selected for an equivalent surface resistivity.

Vehicles. There are different types of vehicles available in the game: cars, trucks, motorcycles, bikes, minibusses, etc. They are our first-class citizens, and their accurate mesh and material representation is critical. When extracting the game data, we selected a high level-of-detail (LOD) mesh. They all define a skeletal mesh composed of components joint by a specific structure and allows the physics engine to create animations. When the wheels are animated, reflections from their updated mesh can automatically encode micro-doppler effects [69]. In the figure, we show the model of a small car captured from the game editor. We see enough detail to allow material composition. In Table 3.4, we listed the tagged materials. The model is defined for appearance and texture selections are not necessarily related to material properties. For example, both the roof and the pillar are set as plasticgray material.



Tag	Part	Material
1	Roof & Pillars	Metal
2	Windshield	Glass
3	Wiperblades	Metal
4	Internals	Rubber
5	Wheels	Metal
6	Tires	Rubber
7	License plate	Metal
8	Lower body	Metal
9	Upper body	Metal

Table 3.4: BMW Isetta model materials

However, the pillars should be made of metal. Similarly, wiper blades should contain both metal and non-metal parts. It is crucial to go over different models and update their materials, particularly the body should be set to metal. For the thesis, we are not much concerned about the remaining details.

Road, Buildings, Infrastructure. We assume the road is asphalt with $\epsilon_r \approx 2.6$ [70], the pavement and buildings are made of concrete with $\epsilon_r \approx 4.9$ [71]. Ideally, the road should result in diffuse reflections, which we did not consider in the thesis. We also did not consider the road state: dry, wet, and icy road conditions have different backscattering characteristics [70]. Other elements in the environment also reflect radar signals. For example, infrastructure elements on the side of the road are, in general, made of metals.

Pedestrians. When considering pedestrians, we need to capture the reflectivity of human skin and garments correctly. In-depth studies exist on the millimeter-wave response of human skin [72, 73]. The reflection coefficient of dry skin at 30C, 77GHz, is reported around -5.15dB [72]. An artificial mannequin design for radar testing [74] uses multilayer materials to mimic the overall reflection coefficient, assumed to be around $-4.7dB$. Based on the simplified model for the RCS $\sigma_{ped} \approx |\Gamma|^2\sigma_{ped}^{PEC}$, we can compute an effective dielectric layer covering the PEC material.

Like vehicles, pedestrians are also represented with skeletal mesh components and support animations. Therefore we may capture micro-doppler signatures while they are walking [75, 76].

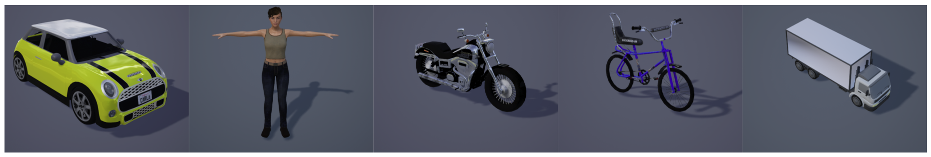


Figure 3.10: Examples of CARLA actors with skeletal mesh components

Weather. Adverse weather is an important problem for autonomous driving and demands a robust perception system that we can only construct by involving all sensor modalities [77]. Even though cameras and lidar are vulnerable to fog, rain, and snow, in general, radar gets much less affected by them. As indicated by experiments, fog has little to no effect on radar [78], so we decided to focus on rain, which results in significant changes in the material properties of air, the road surface, and even directly on the sensor [79].

Carla 0.9.6 contains *precipitation* and *precipitation deposits* in WeatherParameters specified as float values between 0 and 100. *precipitation deposits* modify the road surface, but they do not introduce a mesh layer. Since our data extraction did not include dynamic materials, we currently can not handle it. We could add a random behavior for the road material that affects the hit program. However, we decided not to create material-specific hit programs and branch within a single hit program based on the material properties. So, we do not have a way to identify the road surface, other than explicitly checking for specific parameters, which would couple code to the material database. Therefore we decided to use only *precipitation*. First, we assumed a linear scale with a full physical scale $100 mm/h$ rainfall corresponding to the value 100, and followed the rain scattering model in [78]:

$$\sigma_{rain} = \eta V, \quad \eta = \int_0^\infty \sigma(\lambda, D) N_D(D) dD \tag{3.10}$$

where η is rain reflectivity, V is observation volume, $\sigma(\lambda, D)$ is backscattering from a single drop of diameter D, and $N_D(D)$ is the raindrop distribution. For Marshall-Palmers drop-size distribution:

$$N_D(D) = N_0 e^{-\Lambda D}, \quad N_0 = 0.08cm^{-4}, \quad \Lambda = 41I^{-0.21}cm^{-1} \tag{3.11}$$

where I is rainfall rate in mm/h. We used the same reference for Mie scattering [80], numerically integrated Eqn.3.10 and plotted in Fig.3.11 for comparison, and created a table $\eta(I)$ to use within our simulations. Then, we interpolated η based on *precipitation*, and for each range-angle bin, we computed an estimated σ_{rain} to add corresponding power values to the processed cube. Rainfall direction is related to wind direction and intensity, but we did not want to add too much complexity yet and assumed rainfall downwards; hence the doppler dimension for the rain clutter was filled with the radial component of relative velocity, projected on the radar plane.

3.4.5 Data Communication

During our simulations, multiple processes and threads share or exchange data over shared memory with pointers. We implemented a semaphore-based locking mechanism between data producers and consumers. We have memory regions allocated for all mesh vertices, indices, materials, transforms, vertex velocities, and radar configuration and data. We use each memory region for data serialization following a specific format.

To reduce the data transfer between the game engine and the simulator, at

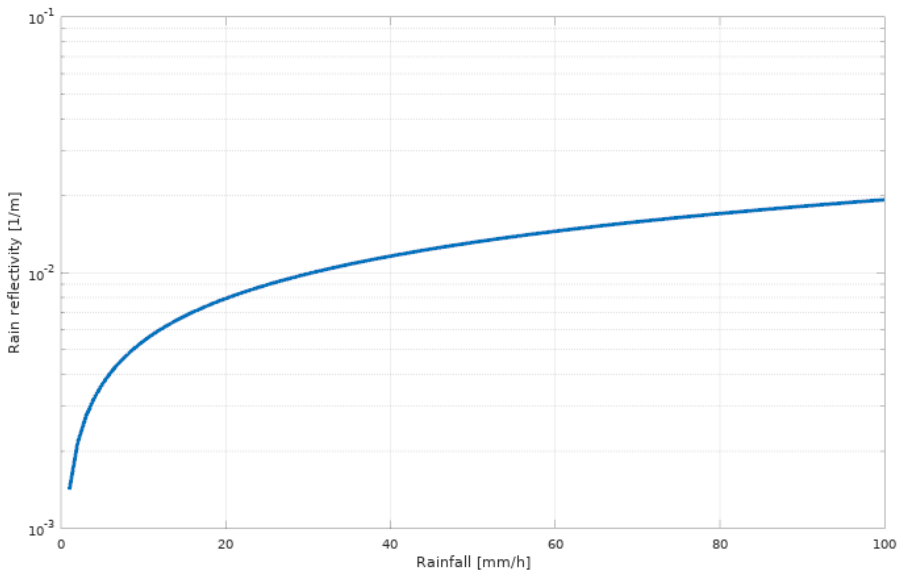


Figure 3.11: Rain reflectivity (1/m) as a function of rainfall (mm/h)

every tick, we only put newly spawned actors mesh data to the shared memory. Since there are internal representations of the world on the radar simulator side, we only need to transfer transforms for every actor. For animated skeletal meshes, we still need to transfer all mesh data.

3.4.6 Raytracing

After reviewing raytracing-based radar simulator implementations reported by others [45, 54, 52], we decided how to proceed with our work.

The original SBR method [43] defines a bounding surface for a cavity and computes the radiation integral on this surface. At the exit, from a set of rays representing a volume trace, an effective area is computed. At each bounce, based on the surface curvature, the wavefront area is updated. From a geometric perspective, it is well suited for cavity problems with twice-differentiable surfaces. In our applications, we can consider the vehicles as cavities since there will be some transmission of waves through the glass; however, if we take into account the inner material composition of the vehicles and the size of the glass-covered area, this is not necessary.

Another item to consider is the ray generation. Ref. [45] highlights three approaches based on how much mesh information is used to assign ray directions. The first one is a sensor-centric, mesh-blind approach similar to [54]. Since it is observational, it can help us understand the effects of parameters used in the ray-tracing method itself. The other two involve the mesh to improve accuracy by directing rays towards objects or triangle centers. After the first bounce, we still need an approach to handle the remaining bounces based on new hit positions. Assuming the first interaction will result in higher contributions to the received power than others, this method may be suitable for investigating the limits of detectability. Its performance may degrade and fluctuate based on the density of mesh in the game. The other approach they considered is directing rays on an implicit mesh on the bounding boxes of objects, which is influenced by the SBR cavity method and addresses the mesh-dependent performance issue.

A fluctuating performance may cause issues when we run the simulator together with the game engine; therefore, we decided to take the first approach and did not inform the ray generation about the mesh of the environment and followed [54], as noted in the antenna pattern discussions.

Considering the number of rays GPUs can handle today, we can make the solid angle assigned to a ray tiny. Therefore we can treat the ray as a cone rather than having a spherical cap. Let us say we have $2N + 1$ Fibonacci rays, the area of spherical cap assigned to a ray is given by:

$$A = 2\pi R^2(1 - \cos\alpha) = \frac{4\pi R^2}{2N + 1} \tag{3.12}$$

where 2α is the apex angle. When N is large, we can approximate: $\alpha \approx \sqrt{2/N}$. For

$N = 1M$, $\alpha \approx 0.08$ degrees; $N = 20k$ can probe better than 1.5 degree resolution. Furthermore, if we consider the ratio of spherical cap height to the radius of travel, $h/R = 1 - \cos\alpha$, depending on how much phase error tolerance we can accept, it is possible to compute the minimum rays required:

$$\Delta\phi_{max} = \frac{2\pi}{\lambda} h \approx \frac{R_{max}}{N} \tag{3.13}$$

For $\lambda \approx 4mm$, $R_{max} = 200m$, and phase error of 1% of a full cycle, we have to use at least $N = 5M$ rays. Given that GPUs support multi Gray/sec raytracing capabilities, this is not a concern. So, we decided on a ray model where each ray represents the axis of a cone.

Next, we considered reflections of the ray-cone. If we compute the radius of cone base: $r = R\sin\alpha \approx R\sqrt{2/N}$, at $R = 200m$ and $N = 5M$, $r \approx 13cm$ comparable to the range bin and generally much larger than triangle side lengths. By sampling the space without considering mesh details, our ray will hit a single triangle within that solid angle. Applying SBR requires extracting the curvature estimates using the vertices around the hit area and updating ray-cone parameters. Instead, we decided $R = 200m$ is far enough to assume geometry and material errors, and considered the ray hits an area equal to the projection of the cone-base, with the material of the hit position. If the mesh around this far-hit location is dense and contains multiple materials, then radar would give fluctuating and noisy signals. We assumed this was acceptable and by-passed the details required for SBR. As a result, our ray-cones hit flat triangles, where both principal surface curvatures, $\kappa_1$and κ_2 are 0, and they reflect without divergence as ray-cones with the same solid angle and mirrored origin (Fig.3.14). This approach also allows us to apply our PO code developed for triangles directly. When a ray hits a much larger triangle than its projected base, we assume

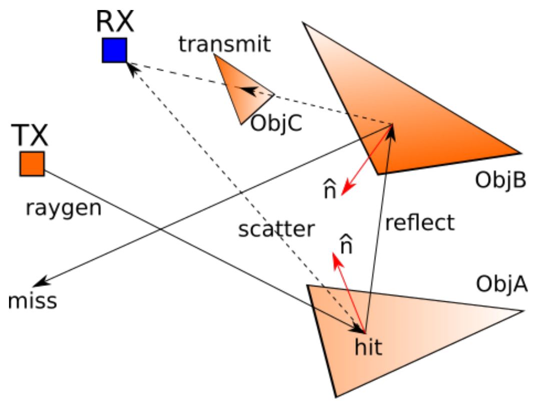


Figure 3.12: Example rays

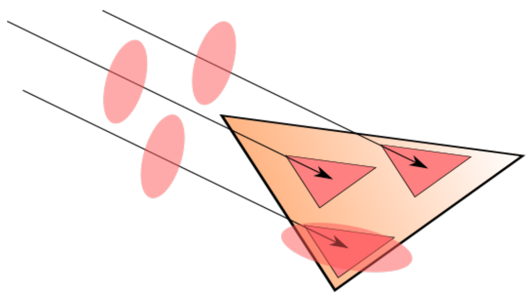


Figure 3.13: Triangle aberration

it hits a similar triangle with a scaled area (Fig.3.13). Therefore we did not have to define a numerical integration over the whole area. There is also an ambiguity on how we should apply integration if multiple rays hit the same triangle. Our approach implicitly defines a new triangulation based on hit sampling, instead of using the original geometry. If the ray hits close to the sides, an effective scaled triangle may fall partly outside of the original triangle, and if the projected base is larger than the original triangle, the overall area may cover it as a whole and some more on the same plane. We call these resolution-limited effects *triangle aberration.* If the mesh contains big elongated triangles or at hit with low elevation angles, we may encounter issues with this method.

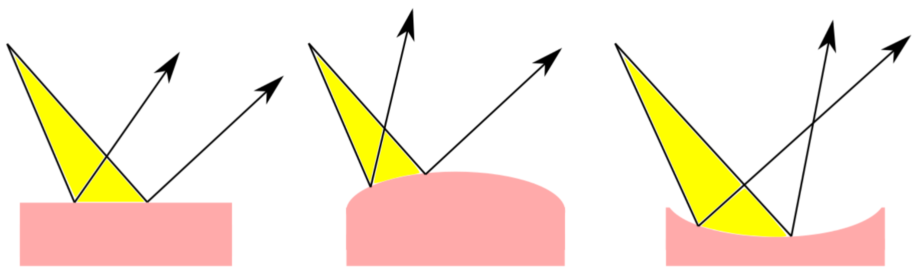


Figure 3.14: Reflection of a ray-cone from $\kappa_{1,2} = 0$, $\kappa_{1,2} > 0$ and $\kappa_{1,2} < 0$ surfaces

3.4.7 Data Processing

After filling the radar cube with sampled values of the return signals, we need to process it to detect objects in the scene. As mentioned in the previous chapter, there are different signal processing approaches depending on the requirements from the overall radar system. For completeness, we decided to implement the primary FFT processing, create canonical Range-Angle and Range-Doppler plots, and send them back to the game client. Plots are helpful in testing and debugging the software.

There are multiple libraries available for signal processing on GPUs, including: `cuFFT`, `NPP`, `Thrust` and `ArrayFire`. We implemented FFTs with `cuFFT`, and for the remaining parts used vanilla `CUDA` and `Thrust` functions. For reducing the 3D cube into a 2D image, we used maximum power value along the collapsed dimension. We then converted the range-angle image into a polar form. Implementing thresholding with CFAR to get detections and target tracking are beyond the scope of the thesis and can be done on the client-side.

3.4.8 CARLA Integration

At the beginning of our project, we fixed our development versions of CARLA and UnrealEngine to 0.9.6 and 4.22.3, respectively. During the course of the project, both tools added new features and capabilities. CARLA introduced a radar sensor defined by horizontal and vertical fields of view, range, and points per second. The implementation of this sensor is based on random raycasting within Unreal framework, acts as a high-level abstraction of a radar, and does not contain any details of radar design, underlying physics or signal processing (Fig.3.15). Nonetheless, CARLA project successfully leverages many ecosystem tools, thereby provides the best playground for our experiments.

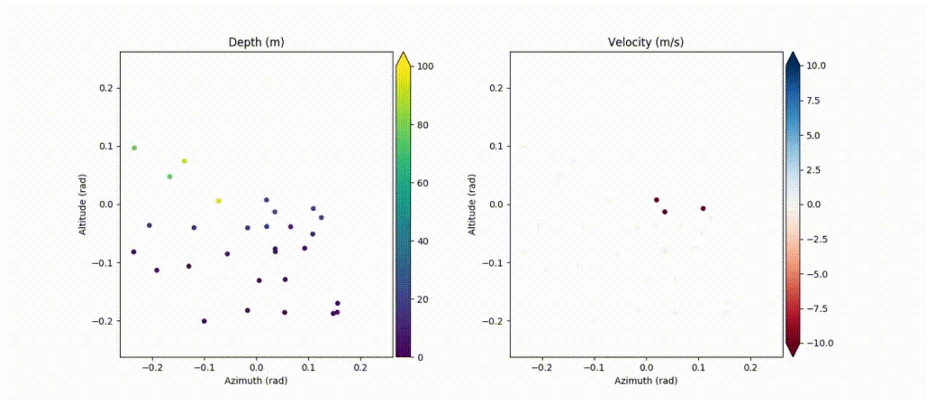


Figure 3.15: CARLA 0.9.7 includes a point-cloud radar implementation [81]

To integrate our third-party plugin, we followed their documentation, where they provide detailed instructions for adding a *SafeDistanceSensor* as an example [82]. First, we defined a new Actor class *ImagingRadar*, which is as a proxy of our radar sensor within the game. It is responsible for providing radar configuration to our simulator and expects processed data in return. An *ImagingRadar* object directly communicates with the *RadarBridge* and queries whether there is any data packet addressed to its UUID. Since there is a singleton *RadarBridge* in our current implementation and all *ImagingRadar*s either expect or ignore processed data, we decided to communicate only during their tick events. Otherwise, this situation calls for an *Observer design pattern* [83].

We wanted to be flexible regarding the contents of processed data; therefore defined our serialization on top of a raw data buffer, which is already serialized by *LibCarla*. In Fig.3.16, we show the layout of the data frame provided by *libRadarSim* to *RadarBridge* over shared memory and the data packet provided by *RadarBridge* to individual *ImagingRadar* objects over heap memory. As a result, a simple extra deserialization step on the receiving end is required during sensor callback. Currently, by default, we provide range-angle and range-doppler plots together with processed radar cube all in the same packet. This can be easily changed or extended by adjusting

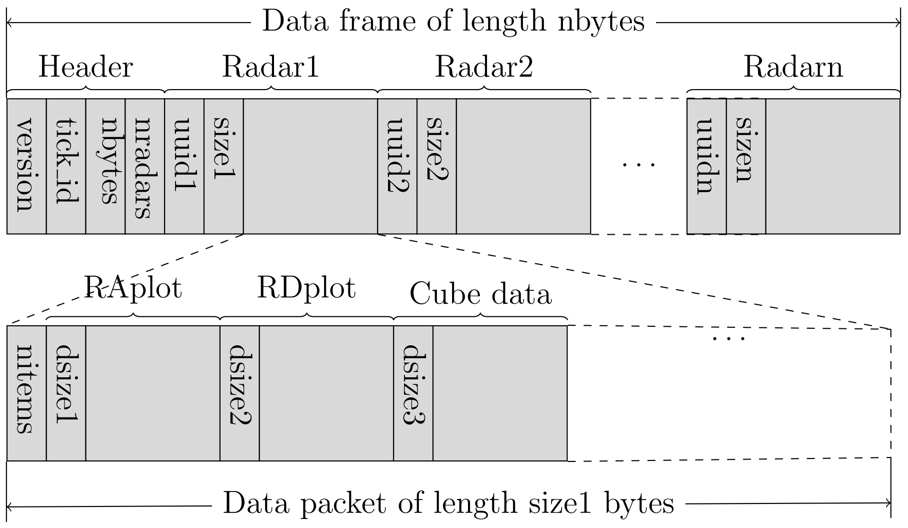


Figure 3.16: Radar data serialization

the number of fields in the header without any recompilation of plugins. Since the output data types and their contents are dependent on the implementation of signal processing and the intended application, we decided to leave the packet definition widely open and designed a very thin API to any external signal processing clients. For example, to get a point-cloud representation, our basic FFT implementation requires an additional CFAR detection step as post-processing. The resulting number of points may not be constant, and they can be grouped and associated with particular objects that can be tracked. These steps are common in radar processing, and their results can be included in the data packet.

Lastly, we exposed our radar measurement to *PythonAPI* as an array and registered our sensor to *LibCarla*, which completes the implementation of full data cycle starting from a Python client, going through CARLA application, radar plugin, raytracing simulation and returning from the same path to the Python client (Fig.3.17)

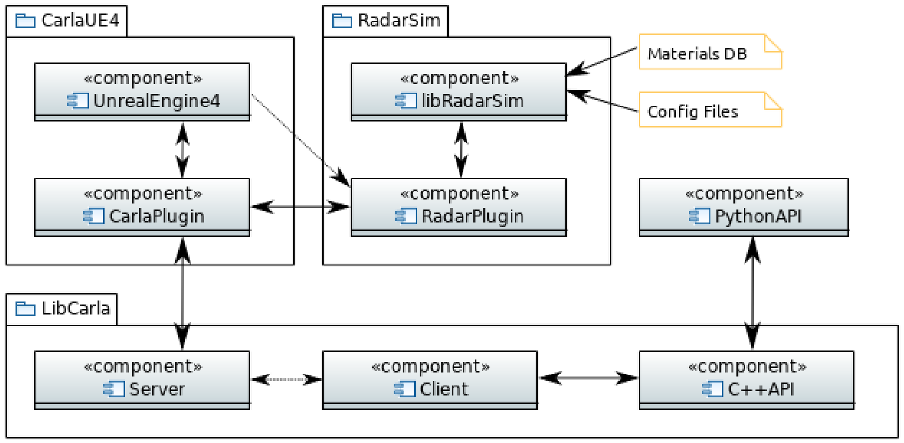


Figure 3.17: CARLA + RadarSim: Components of integrated system

3.4.9 AirSim Integration

AirSim was also developed in Unreal Engine; therefore, our `GameMeshViewer` should work out of the box. We only needed to copy our code to the Plugins directory to trigger a build. For our plugin to start working, `RadarObserver` should be placed at the current level or spawned with C++ API.

For `ImagingRadarTool` to work, we have to make modifications to AirLib, another third-party sensor library. It exposes a C++ API, uses rpclib for server-client interaction, and also had Python wrappers. We followed their Lidar implementation to create the required proxy class `ImagingRadar` with the same functionality in CARLA.

3.4.10 Unity Engine and LGSVL Integration

In Unity, game objects may have multiple behaviors attached as scripts. We converted RadarObserver and RadarBridge to C# scripts. The program workflow in each is the same as in their C++ equivalents. However, to reuse the data communication functionality of RadarBridge, which is mostly independent from the engine, we used the native plugin approach and loaded it as a dll file. We wrote wrapper functions in RadarBridge.cs to call RadarBridge library functions. RadarObserver and the mesh information extraction sections of RadarBridge are engine dependent; we followed Unity documentation to implement them.

As in the Unreal plugin, these behaviors should be instantiated in the game application. We put a dummy object in the scene called RadarBall and attached RadarObserver and RadarBridge behavior to it. We also created an ImagingRadar script that searches for the RadarBall to communicate with the RadarBridge. With these three behaviors, our plugin can work with any Unity application. To integrate with LGSVL we followed their specific sensor plugin instructions and adapted our ImagingRadar to include ROS writers that publish image messages of range-angle and range-doppler plots.

Chapter 4: Results

Since we developed our plugin library independent from a game engine or an application, it is possible to create lightweight simulators for different purposes. In this chapter, we introduce our standalone applications and plugin. We benchmark our RCS simulator on standard shapes and discuss the effects of mesh size and ray density. Then we show our application to navigate the game environment. Lastly, we describe our imaging radar application and show the results of plugin integration with multiple vehicle simulators.

4.1. RCS Tool

POFACETS provides a good starting point to understand RCS computations. We fixed our working version to 4.3. Since we had other mesh viewers already, its GUI functionality was not required; we directly used scripts that enable automation for testing and allows us to use Octave. We refactored the code for readability, which uses global variables and no structs while passing many related arguments to functions. Next, we profiled it and decided to convert facet computation to a mex file. This conversion gave us a speed factor of $\times 7$, and GPU implementation without any optimization resulted in $\times 20K$ effective speed up. Then, we created file format converters between its facet format and .obj files. With these modifications, POFACETS turned into a tester to cross-check our results as we developed our code for the GPU.

All facets in POFACETS are triangles. To compute the RCS of a complex object, it goes through all facets and adds their contribution. It first checks whether a facet is illuminated from the incidence direction based on the vertex winding normal. Then it transforms the incident field to local coordinates of the facet and computes scattered fields in the observer direction using an integral formulation. To replicate this behavior, we decided to implement a raycasting method for hit detection and computed the radiation integrals on the scaled facets corresponding to rays' hit locations, as discussed in the previous chapter.

We created a set of parallel rays directed towards the object over a rectangular region, which is large enough to cover the object's projection (Fig.4.1). For every computation in the azimuth direction, we rotated the rays' origins and directions. In effect, the rectangular region is an RCS camera. We also placed a pinhole camera at the same location and performed regular light raytracing in another context for visualization. By default, pinhole camera follows the observer direction, and stepping the azimuth angle creates a simple rotation animation. During the scan, we also plot computed RCS values overlayed on the viewport and print them on stdout to save for external plotting. After the scan is complete, the user can move the model around and capture the RCS result with the model (Fig.4.2). They can also provide the polarization of the incident field and its fixed direction for bistatic RCS computation using command line arguments.

4.1.1 Benchmarks

NASA Almond. The most common shape used in computational electromagnetics RCS benchmarks is NASA almond (Fig.4.3a), which was designed as an experimental test body to assess the performance of anechoic chambers [84]. Its surface is

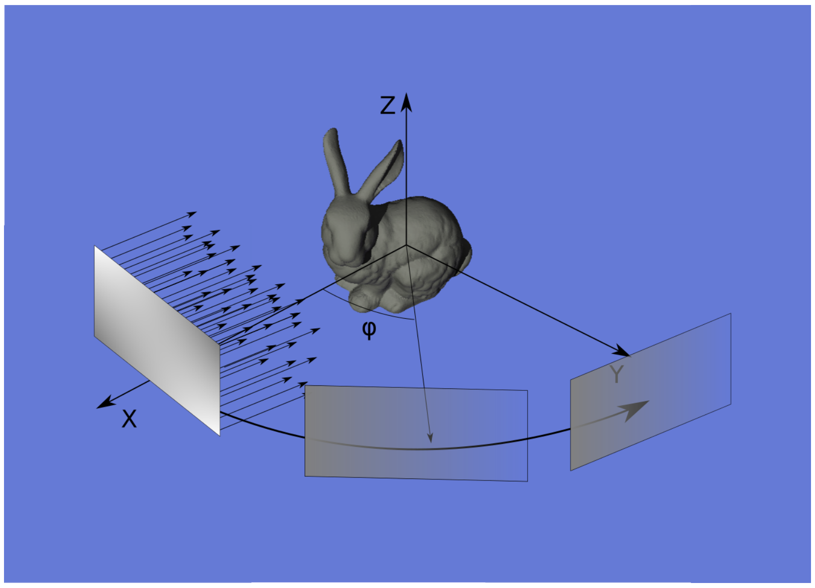
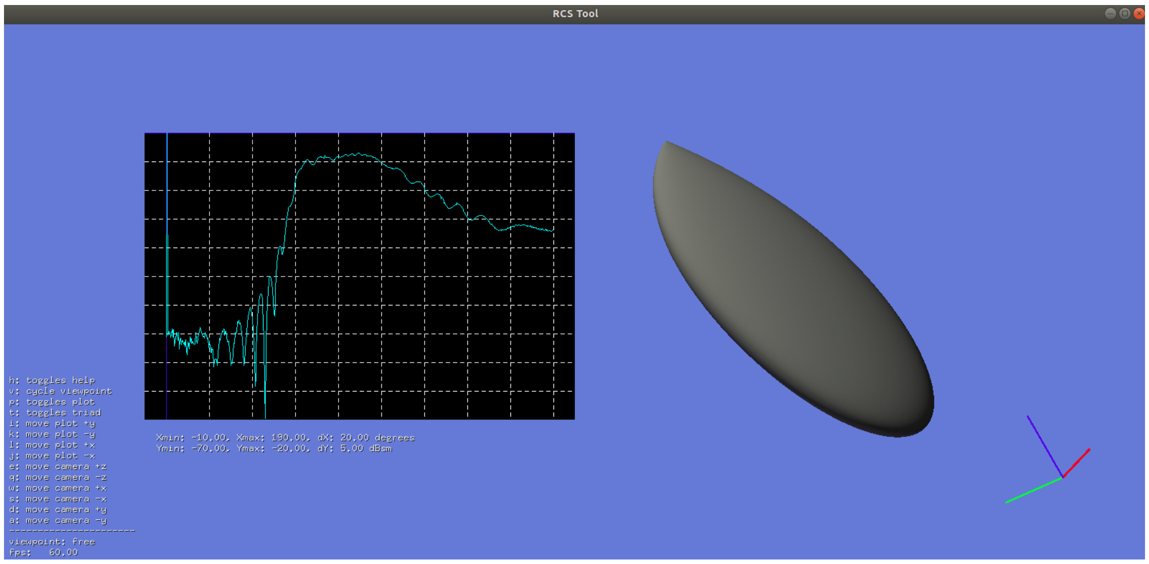


Figure 4.1: RCS Tool concept



```
h: toggles help
v: cycle viewpoint
p: toggles plot
t: toggles triad
i: move plot +y
k: move plot -y
l: move plot +x
j: move plot -x
e: move camera +z
q: move camera -z
w: move camera +x
s: move camera -x
d: move camera +y
a: move camera -y
----------------------
viewpoint: free
fps:   60.00
```

```
Xmin: -10.00, Xmax: 190.00, dX: 20.00 degrees
Ymin: -70.00, Ymax: -20.00, dY: 5.00 dBsm
```

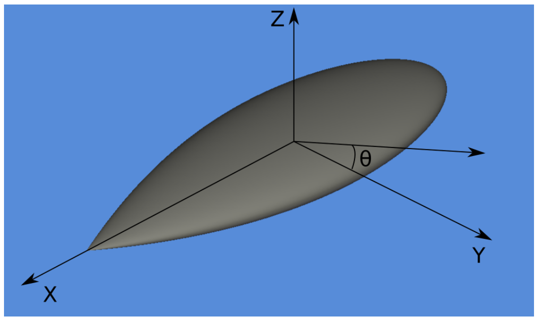
Figure 4.2: RCS Tool application

described with a parametric equation set given in Table 4.1. $L = R_2 + R_1 \sin\alpha$ is the body length. The constants satisfy the equations: $A_2/A_1 = B_2/B_1 = (R_2/R_1)^2 = 1 - \cos\alpha$ and $2 \le A_2/B_2 \le 20$. The original almond has size $L = 9.936\,\text{in} \approx 25.24\text{cm}$.

Since it has a convex shape, considering a single ray-triangle intersection will

$x \geq 0$	$x \leq 0$
$y = A_1\left(\sqrt{1-\left(\frac{x}{R_1}\right)^2} - \cos\alpha\right)\cos\theta$	$y = A_2\sqrt{1-\left(\frac{x}{R_2}\right)^2}\ \cos\theta$
$z = B_1\left(\sqrt{1-\left(\frac{x}{R_1}\right)^2} - \cos\alpha\right)\sin\theta$	$z = B_2\sqrt{1-\left(\frac{x}{R_2}\right)^2}\ \sin\theta$

Table 4.1: NASA almond parametric equations



(a) NASA Almond

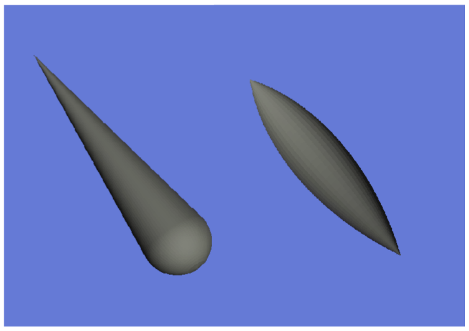
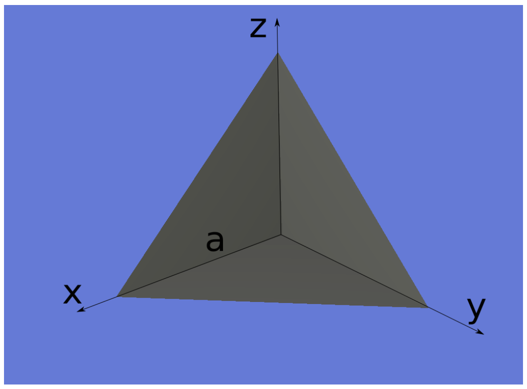
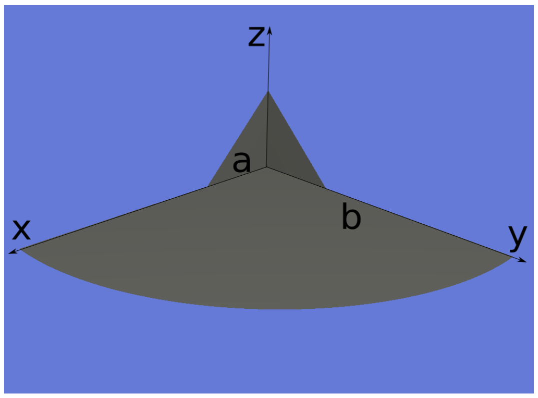

(b) Cone-Sphere and Single-Ogive

Figure 4.3: Example benchmark shapes

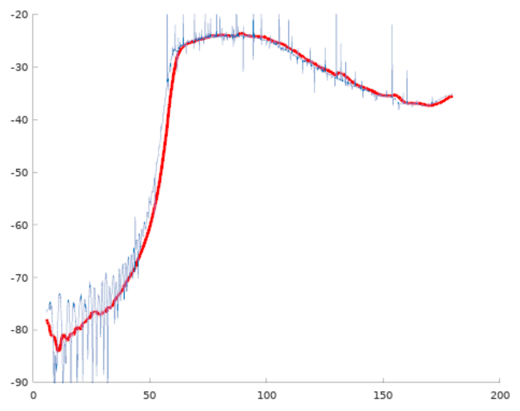


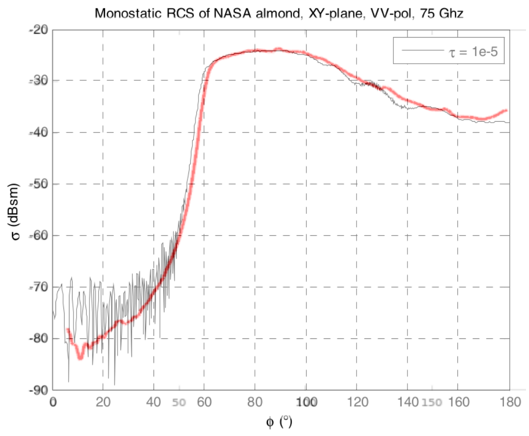
(a) Trihedral corner reflector



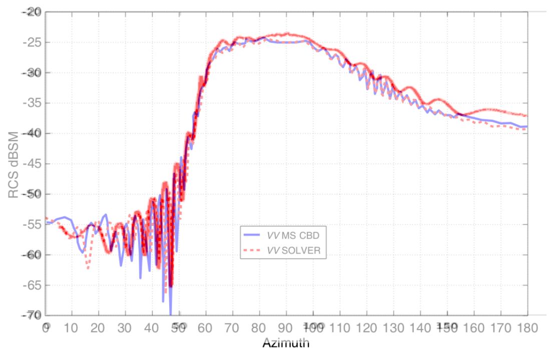
(b) Modified trihedral with bottom plate

Figure 4.4: Example corner reflectors

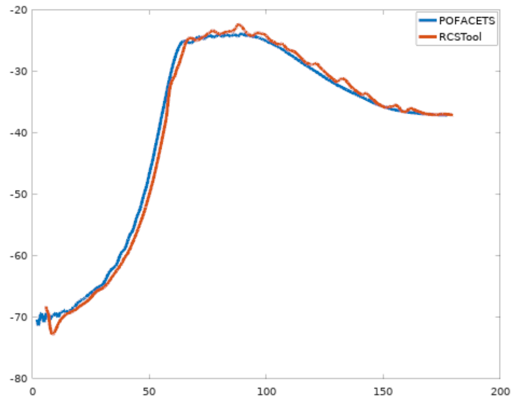

(a) Almond RCS at 75GHz after filtering



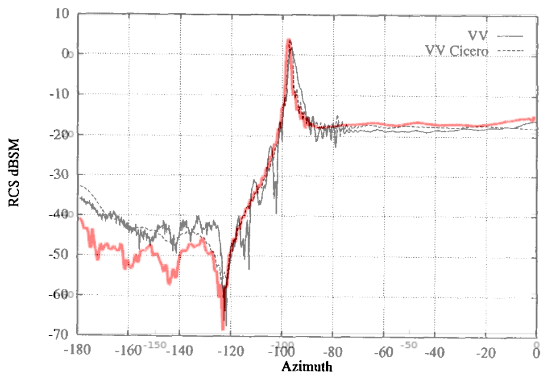
(b) Comparison with [85] at 75GHz



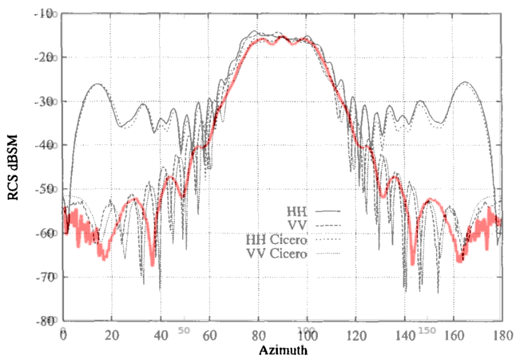
(c) Comparison with [86] at 21GHz



(d) Comparison with POFACETS at 21GHz



(e) Comparison with [87] at 9GHz



(f) Comparison with [87] at 9GHz

Figure 4.5: RCS results and comparisons with publications. Top-left: Spikes occur due to semianalytical integration region. Top-right: Our result is overlayed in red color. We also see variations around he tip which is filtered out while removing the spikes. Mid-left: Our result is using a low density of rays that matches MoM computations. Mid-right: POFACETS, with a tighter phase bound, does not generate spikes and matches well to our computation. Bottom-left: Cone-sphere, Bottom-right: Single-ogive. All plots are VV.

be sufficient for RCS computation; therefore we can use both POFACETS and RCS Tool. Our wavelength of interest is around $\lambda \approx 3.9$mm. Since $\lambda \ll L$, we are already in the high-frequency region for scattering and expect accurate results from the PO tools. Most of the published results were for $f < 10GHz$, but we identified two papers with MoM simulation results at 21GHz and 75GHz [86, 85]. Computing the RCS by solving the matrix problem takes hours; with RCSTool, we can scan the almond in less than a minute. However, the PO approximation, and particularly the integration method we use, has some drawbacks. In Figure 4.5a, we show the output of RCSTool in blue, which contains spikes; we adopted the integration scheme in [39], which uses a Taylor series expansion for the phase. We need to adjust the number of terms included in the series and the size of the region. For demonstration, we instead low-pass filtered the values, shown in red color. In Figure 4.5b, we overlayed our results on the published plot [85]. We see a good match, even for the filtered-out values between 0-60 degrees. This region corresponds to the conical part of the almond, and the oscillatory values can be physical rather than numerical artifacts. If we look at results for 21GHz, in Fig. 4.5c, the same region does not have a smooth RCS; MoM computations are in blue and dashed red. On the other hand, in Fig. 4.5d, we see POFACETS predicts smoother values. Our overlayed results, shown in red, for the same simulation, have only one different parameter, the grid density of rays is 15 times more on the right. We have an explanation for this contradictory result; with more rays we lost physical accuracy. PO approximation only considers a particular facet to compute the scattered fields, and the facet has to be large enough, 2.5 to 4.5 times the wavelength according to POFACETS documentation. With a fine-refined mesh this is no longer true; our almond had average facet side of 0.75mm, and $\lambda @ 21GHz \approx 14.3mm$. Using a low density of rays, the implicit triangle per ray becomes larger, and accuracy improves. These considerations will be important in game engine computations where we will adjust ray-limited resolution for radar simulation, creating an implicit level-of-detail (LOD). Also in the game mesh, for almost all objects of interest, the triangle sizes will be greater than $1cm(\approx 2.5 \times \lambda@77GHz)$.

Cone-sphere and Single-ogive. There are a few other benchmark shapes proposed in [87, 88]. In Figure 4.3b, we show two of them: a cone-sphere and a single-ogive. Their parametric equations and RCS results for a few frequencies are presented in [87]. These objects are $L_1 \approx 689$mm and $L_2 = 254$mm in length and have diameters $D_1 \approx 149$mm and $D_2 = 101.6$mm. We simulated them at 9GHz, where $\lambda \approx 33.3$mm; therefore, the first one is relatively closer to the optics regime. In Figure 4.5e and 4.5f, we overlayed our results on plots from [87]. As we noticed for the almond, we can not get a good match for regions around sharp tips; PO is not a good approximation when the surface normal and the direction of incidence are quite dissimilar. For the single-ogive, the region outside $70 - 110$degrees is highly sensitive to the ray density and not reliable.

Corner reflectors. POFACETS and RCS Tool are limited in applications since they do not consider multiple reflections; POFACETS provides a ground reflection option that only applies to a particular setting. These tools can not predict the RCS for a trihedral corner reflector in Fig.4.4a. At the boresight:

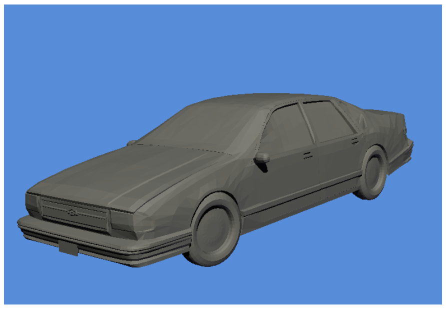
$$\sigma = \frac{4\pi a^4}{3\lambda^2} \tag{4.1}$$

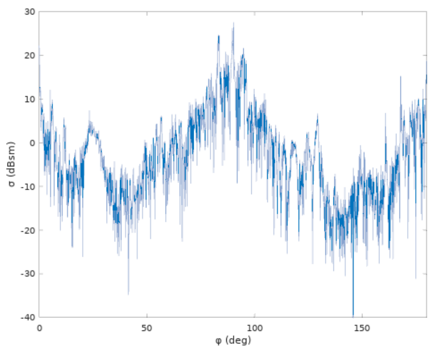
where a is the side length. For $a = 10$cm, at 21GHz, $\sigma = 3.123$dBsm; however, PO without any reflections results in $\sigma = -21.6$dBsm, which is much smaller, highlighting the importance of considering multiple reflections. In the real world, the radar may encounter many concave corners like this one.

For low grazing angles, Sandia researchers designed a modified trihedral reflector shown in Fig. 4.4b. For $a = 15$cm and $b = 50$cm, they simulated the design with a raytracing method at 10GHz($\lambda = 3mm$) and showed a good match with their geometric predictions [89]. We will come back to this example in the ImagingRadarTool section.

RCS of Vehicles and Pedestrians. Test models have simple shapes and smooth variations of RCS. However, our main objects of interest are vehicles and pedestrians. In Fig.4.6b we plotted RCS of a car model, with PEC material, which looks acceptable.

We also simulated a PEC pedestrian model (Fig.4.7a), and subtracted 4.7dB based on [74]. As we expect, the RCS data is very noisy (Fig.4.7b). When we plotted the histogram of it (Fig.4.7c), we get a distribution very similar to [90], with average -4.3dB rather than -8.1dB. These variations are normal with the effects of clothing.


(a) Car model



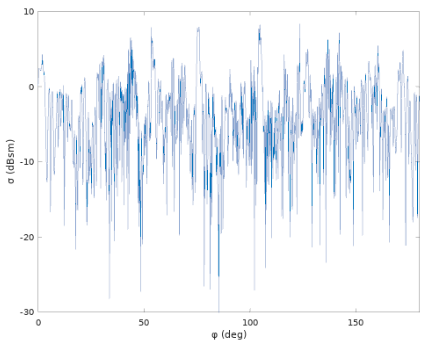
(b) Car RCS

Figure 4.6: RCS of a car model

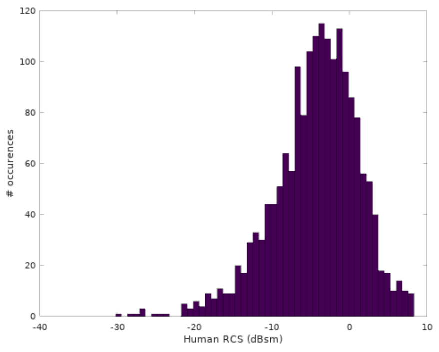
4.2. Game Mesh Viewer

Next, we worked on our first use case, recording game data. At every tick, RadarBridge puts mesh and transform data into folders on the filesystem or directly

(a) Pedestrian



(b) Pedestrian RCS



(c) RCS histogram

Figure 4.7: RCS of a pedestrian and its histogram

on the shared memory. To inspect this data, we developed another application, GameMeshViewer (Fig.4.8-4.9). By providing a command-line argument, the user can make it work online with the game engine or replay recorded data. Depending on which game engine the data originates from, it is scaled, rotated, and translated in a conventional right-handed coordinate system. Using the delta time for the tick, we compute a velocity field on the vertices based on previous transforms of the same mesh objects. Game data also contain materials for every triangle; however, we do not need texture information for radar simulation. For better visualization without texture, we assigned random color values to triangles.

The full scene for the game can be huge, and it is a concern for performance reasons. We mainly used the default Town03 map in CARLA during experiments. At the initial tick, we export around 7,000 objects. Transformed meshes of all these objects are combined to form the global scene mesh. OptiX organizes this mesh with an acceleration structure for performance, and GameMeshViewer runs raytracing with a pinhole camera model. We have about 5.4M vertices, 4.1M triangles, and 300 materials in the initial scene. It takes 80ms launch time for XGA resolution (1024x768). After spawning around 400 new actors with detailed mesh, we reach 11M triangles and the launch time doubles. We are using a GeForce RTX 2060

Actor	**Triangles**	**Vertices**	**Materials**	**Avg size (mm)**
Pedestrian (child)	9428	5817	9	14.8
Pedestrian (adult)	23500	19294	16	17.3
Bicycle	3968	2972	2	47.1
Car	19539	16258	13	79.9
Minibus	18541	16015	18	89.7
Truck	15980	10110	15	106
Jeep	19453	15494	15	89.6
Motorcycle	22694	20173	5	32.2
Building1	10728	16185	4	410
Building2	32870	34500	4	1062

Table 4.2: Mesh statistics for a sample of actors

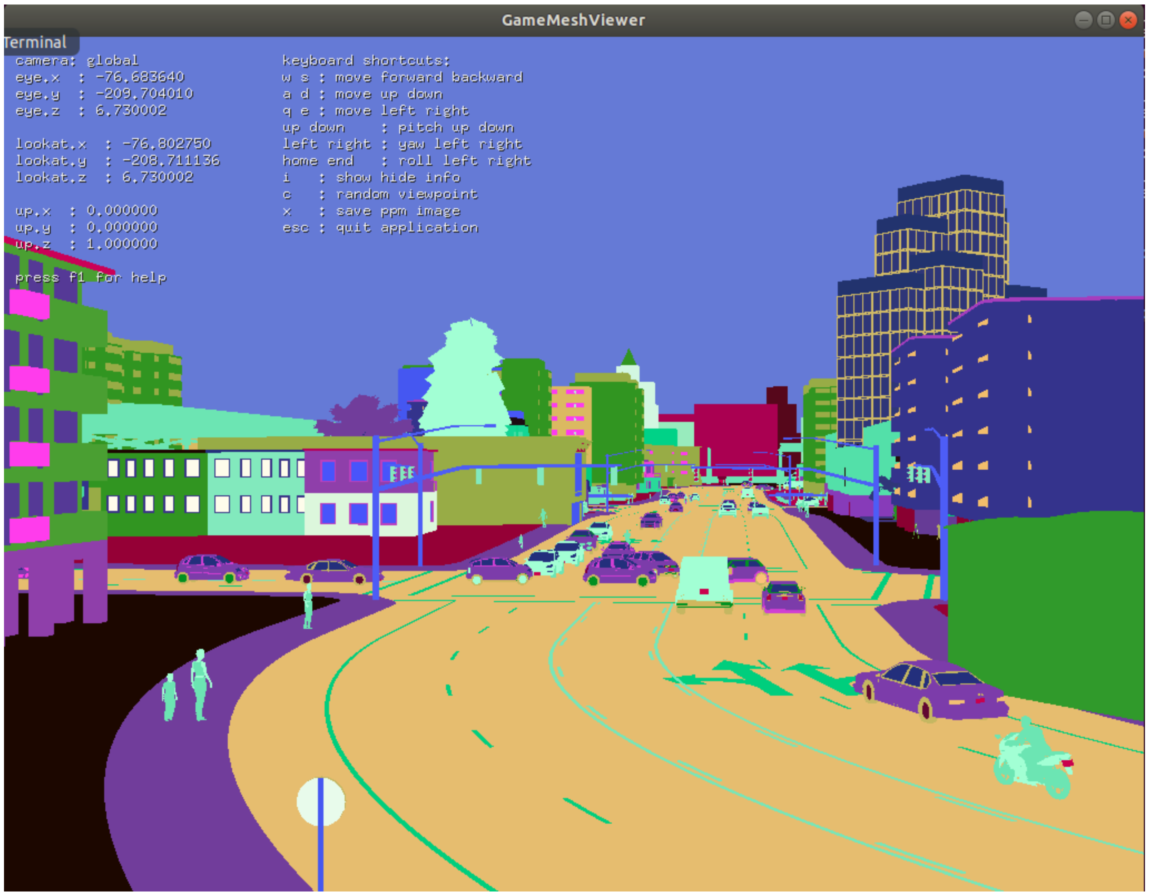
Super, which is capable of 6G rays per second. The GPU memory usage increases about 1GB with the actors, so our limitation is computational only. Without any performance optimizations, compute utilization is around 30%. While acceptable for viewing the world, raytracing in the global mesh would degrade the fps.

GameMeshViewer is agnostic to radar sensor objects' existence; it can work as a partially implemented plugin inside AirSim or LGSVL.

Unreal Engine Editor provides a statistics display about primitives. We can read which objects are in the game, how many actors exist, the type of mesh they use, the number of triangles, and memory usage for each item. Based on our RCS discussions, we would like to know more about the mesh details. We explored saved game data and looked at statistics for different types of objects and their material composition. We show some examples in Figure 4.10 and Table 4.2.

4.3. Imaging Radar Tool

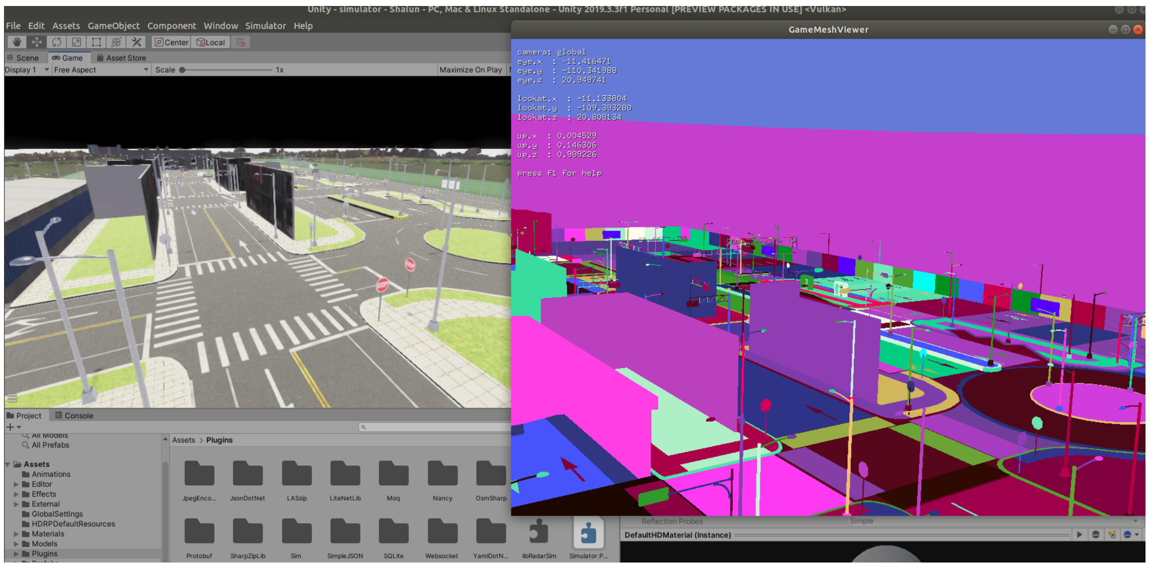
Next, we developed the radar raytracing code and enhanced the mesh viewer to switch between multiple cameras, a global camera and two cameras for each radar. Using the classes we designed for the plugin, we created another tool that can run



```
camera: global                keyboard shortcuts:
eye.x  : -76.683640           w s : move forward backward
eye.y  : -209.704010          a d : move up down
eye.z  : 6.730002             q e : move left right
                              up down    : pitch up down
lookat.x  : -76.802750        left right : yaw left right
lookat.y  : -208.711136       home end   : roll left right
lookat.z  : 6.730002          i   : show hide info
                              c   : random viewpoint
up.x  : 0.000000              x   : save ppm image
up.y  : 0.000000              esc : quit application
up.z  : 1.000000

press f1 for help
```

Figure 4.8: GameMeshViewer application displaying recorded CARLA data

together with the game engine or independently from recordings. For each radar, data processing automatically creates range-angle and range-doppler plots and puts them on the shared memory for RadarBridge to distribute to proxy radars in the game. We also display these plots on the screen as textures for the selected radar unit. In Fig.4.15, we show a pair of examples corresponding to the scene in Fig.4.11. We are using Radar2 prototype with a maximum range 48m in Table 2.1. We annotated some of the objects in the scene. These plots are intermediate results between raw data and point clouds, but they contain more relevant information, which makes them useful for machine learning applications. We also optionally dump unprocessed and processed radar cubes to the filesystem for the same reason.


```
camera: global
eye.x  : -11.416471
eye.y  : -110.341988
eye.z  : 20.949741

lookat.x  : -11.133804
lookat.y  : -109.393280
lookat.z  : 20.808134

up.x  : 0.004529
up.y  : 0.146306
up.z  : 0.989226

press f1 for help
```

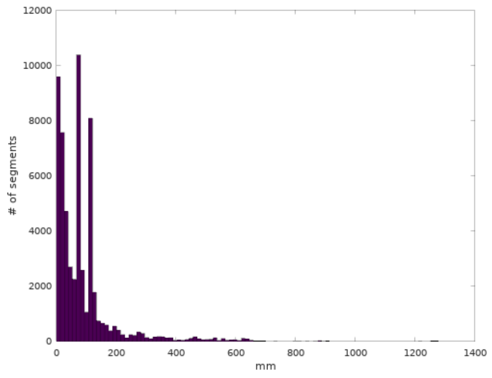
Unity - simulator - Shalun - PC, Mac & Linux Standalone - Unity 2019.3.3f1 Personal [PREVIEW PACKAGES IN USE] <Vulkan>
File Edit Assets GameObject Component Window Simulator Help
GameMeshViewer
Scene Game Asset Store
Maximize On Play
Project Console
Assets > Plugins

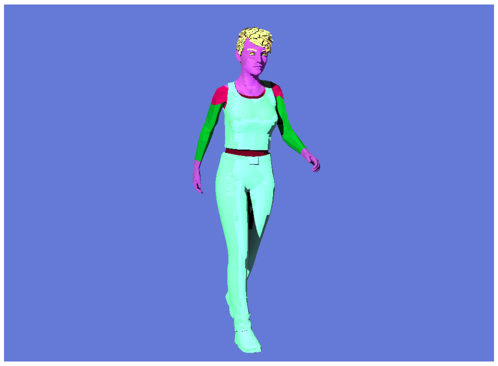
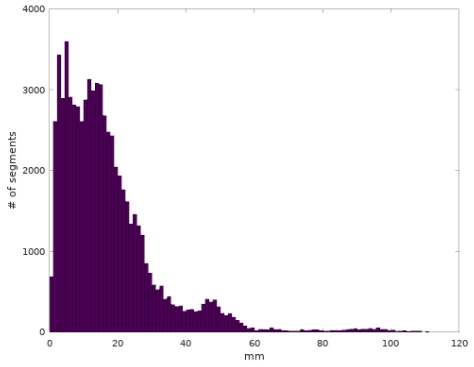
Figure 4.9: GameMeshViewer displaying LGSVL Simulator's Shalun Environment

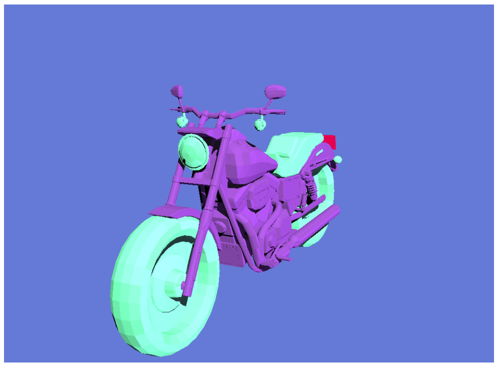
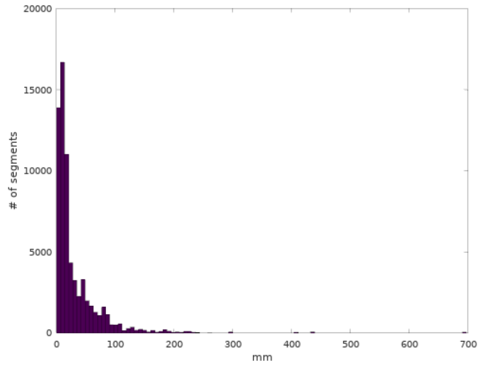
While developing radar raytracing contexts, we decided to filter the primitives by distance to the sensor. Having the full scene in each context is unnecessary; it would cost too much memory and increase the time for intersection checks. Since radar signals may bounce multiple times, there is a possibility that objects behind the sensor might also be needed. We defined a maximum radius for each radar to construct a spherical context with the sensor at the center.

Corner reflectors. Unlike the RCS tool, where we performed raycasting and looked for compatible results with POFACETS, we can perform multiple reflections with the radar tool. We need at least three reflections for the corner reflectors.

We created an empty scene and placed a corner reflector (Fig.4.4a) at the origin. Then, we oriented our radar at the boresight direction ($\theta = 54.7$ and $\phi = 45$) and placed it more than $R \gg 10a$ distance to enable paraxial approximation, and angular position with antenna gains close to unity ($G_t \approx 1, G_r \approx 1$). Under these conditions, we can read the value from the cube, proportional to the received power P_r, and compute RCS using Eqn.2.1. We use a sphere as a reference target, $\sigma = \pi a^2$.



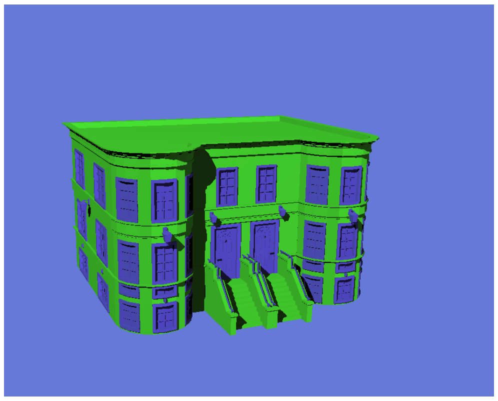
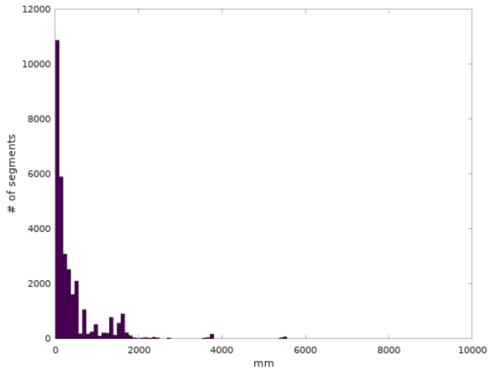
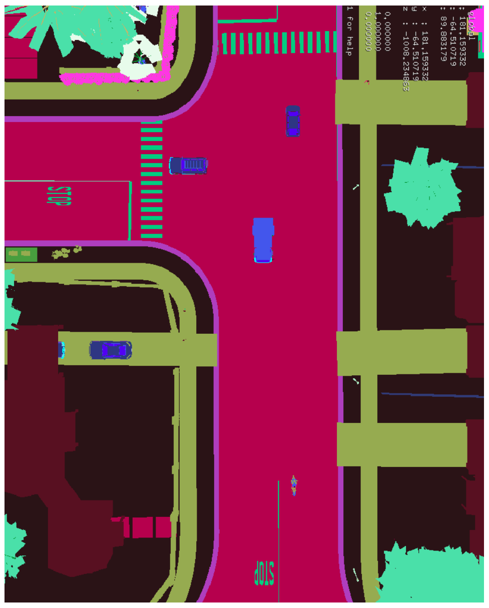
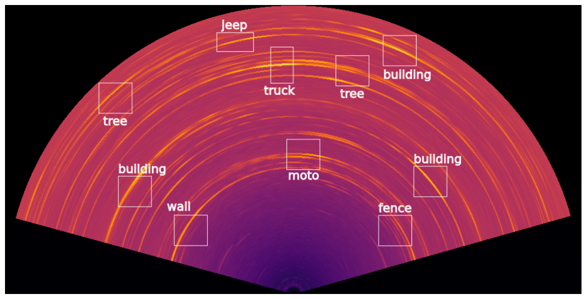



Figure 4.10: Random-colored meshes and their histograms of triangle side-lengths


(a) Topview



(b) Range-Angle plot

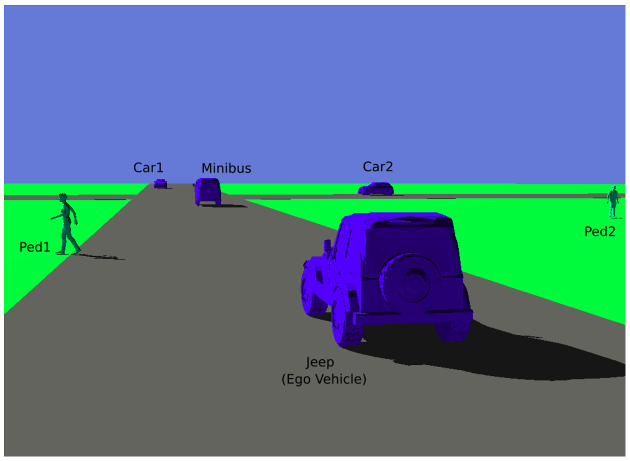
Figure 4.11: Example scene

There are multiple problems with this setting, however. The reflected fields are not captured by the antenna; since they are parallel to the incoming direction and the solid angle is small. The scattered fields are off from the parallel direction and have different phases dependent on R. Since the target is far away, the density of rays is small, which adds to numerical uncertainty. To alleviate some of these problems, we defined a method of ray zooming. Instead of a uniform lattice on the sphere, we generate rays on the spherical rectangle $[\theta_{min}, \theta_{max}] \times [\phi_{min}, \phi_{max}]$. First, we used a uniform grid of size $\lfloor \sqrt{N} \rfloor \times \lfloor \sqrt{N} \rfloor$. We select N based on the hit area, so that PO approximation and our numerical scheme works the best. For $a = 10cm$, $\lambda = 3.9mm$, $R = 10m$, we need around 1.15deg angular field of view in both azimuth and elevation. And to get hit areas around $4\lambda^2$, we need discretization around 0.045deg, leading to $N \approx 625$. Raycasting with RCS tool results in $\sigma \approx -21.37dBsm$, which is about the cross section of a sphere with $r = a/2$. Our zoomed grid approach resulted in $-12dBsm$, which is still too small compared to the actual value of $14.4dBsm$. As expected, when we increased N the results got even worse. To solve this problem, we need to keep the hit area constant but increase the density of rays, which we can implement with a Monte-Carlo approach. We chose the base-area of ray-cones $4\pi\lambda^2$ and used $N = 10k$ rays. Our new estimation was $6.8dBsm$, which is 17.5% of the actual RCS. Due to triangle aberration, we effectively offset the edges outwards. With geometric approximations, we can compute the RCS of the implicit target with $a_{eff} \approx a(1 + \lambda/a)$, resulting in $15.072dBsm$. This is actually an upper bound. Next, we changed $R = 5m$ and computed $\approx 21dBsm$. Clearly, this is above the upper bound and indicates an error source. Sandia modified reflector has about 15x area of this trihedral reflector. We adjust N to accommodate the same ray intensity. Also, we use a reference reflector with $a = 15cm$ at $R = 20m$. Based on [89], we expect 2.5dBsm higher RCS value at the same boresight direction, around 23.95 dBsm. Our simulations resulted in 20.78 dBsm instead. These are significant improvements over scattering with raycasting. However, since we are not using parallel rays, when the power is calculated at the receive antennas, there are extra phases depending on the ray lengths and scattering angle offsets, which we believe are the main sources of error here.

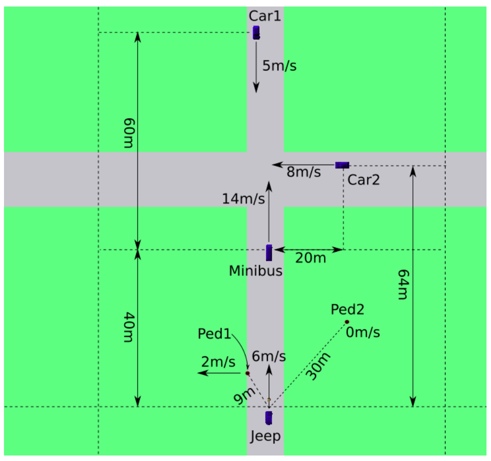
Scene Description Format. For testing the tool with more complex scenes, we defined a scene description format that brings together multiple obj files. We specify their positions and orientations with a translation and a quaternion, and also their velocities with a translational velocity and angular velocity. After parsing this file, we generate a combined obj file for the scene and a corresponding velocity-field file. With scripting, it is possible to generate a set of obj files and velocity files, forming an animation. Even though this is restricted to solid-body motions, it is still good enough for testing.

In Figure 4.12, we show an example scene generated with this scene description utility and corresponding range-doppler plots. We placed a mid-range radar (Radar1 configuration in Table 2.1) to the front-middle location of a jeep, at 0.5m height. We have five actors (2 pedestrians, 2 cars and 1 minibus) in the scene. In Figure 4.12b, we show their positions and velocities. We included three range-doppler plots: Fig.4.12c is for the static scene to identify different objects by their distance to the ego vehicle. Fig.4.12d shows as the scene is running with the jeep moving north with 6m/s speed. We directly see the minibus moves away with +8m/s, and Car1 gets closer at speed around -11m/s. For the other actors, we have to project their relative velocities to the ego vehicle in the radial direction. Ped1 and Ped2 are at positions (7.07, 6.07)m and (21.21, -21.21)m, where +x-axis is north, +y-axis is west, and the origin is at the radar. We compute their relative velocities as: -3.25m/s and -4.24m/s, respectively. Similarly, Car2 is at (64,-20)m, and its relative velocity is -8.11m/s. All these numbers are compatible with the middle plot. In Fig.4.12e, we show the same scene with the jeep standing still. Therefore, Ped2 is at 0m/s, and Ped1 is moving away with speed +1.3m/s. Car1 and Car2 are getting closer with -5m/s and -2.4m/s, respectively. Since the minibus is moving away with +14m/s and our radar has maximum velocity around 13.5m/s, we see an aliasing effect on the negative side. In such cases, we have to track the object on multiple frames for correct velocity estimates.

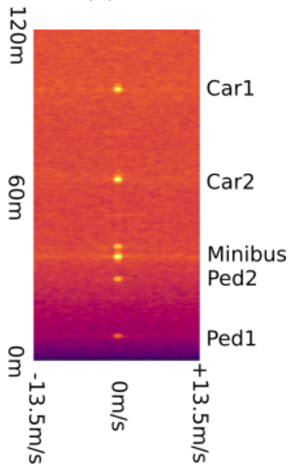
In Figure 4.13, we show an urban setting with multiple buildings, vehicles, and pedestrians. We used settings for a short-range radar (Radar2 in Table2.1). Fig.4.13c is an annotated range-angle plot corresponding to the top view in Fig.4.13b. In this case, the radar plots are slightly difficult to interpret due to complexity for various reasons. Firstly, we did not implement processing methods to suppress sidelobes from the FFT; therefore, we can misinterpret high sidelobes, particularly in the angular direction, as false targets with lower return signal. Secondly, we used 5 reflections for each ray; multiple reflections can result in ghost targets. Thirdly, the color-mapping



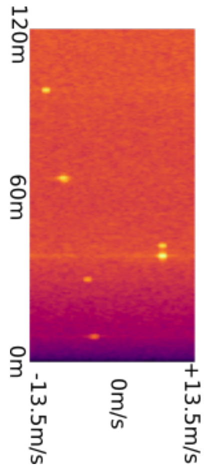
(a) Scene from the road



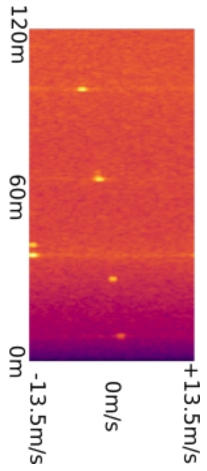
(b) Scene from the top



(c) RD (static scene)



(d) RD (egovel= 6m/s)

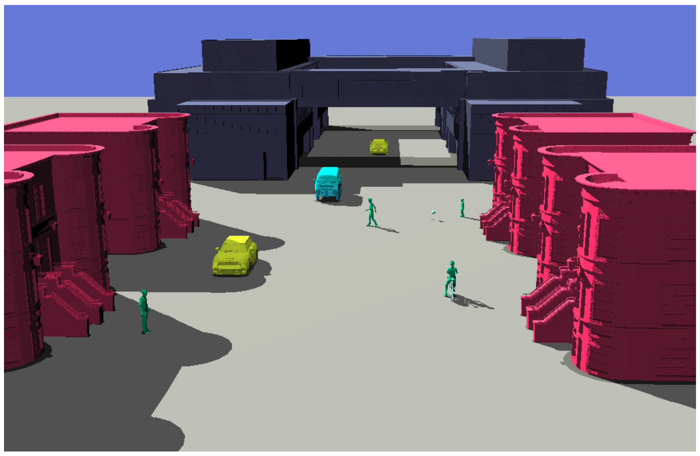


(e) RD (egovel= 0m/s)

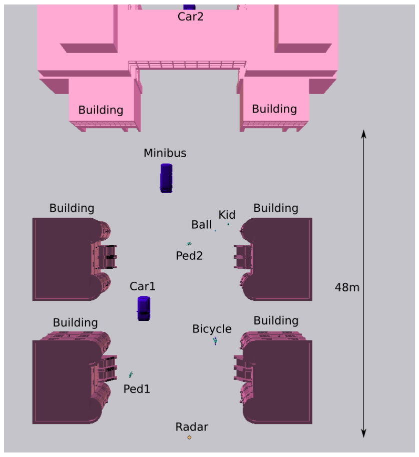
Figure 4.12: Range-Doppler examples using scene description utility

process itself distorts our perception. We use R^4 power correction so that the targets far away do not fade out. The colormap is dynamically scaled using the current minimum to maximum levels. We convert values for range-angle in rectangular grid (Fig.4.13d) to circular grid (Fig.4.13c). In this process, we introduce extra samples, which are interpolations of measured values. Since the angular resolution decreases as $|\phi| \to \pi/2$, these samples form wider arcs. They do not add extra information but affect our perception. Correct interpretation requires sidelobe suppression methods, filtering, peak detections, and tracking. These are client-side tasks and beyond the scope of this thesis.

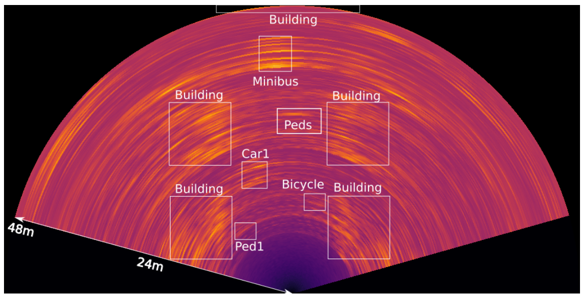
Detectability is another problem. The kid and the ball do not clearly appear in the range-angle plot. This could be related to the building that falls around the same range-angle bins, and also, for example, the ball not getting hit by enough rays. Pedestrians mostly walk slowly and orthogonal to the radial direction; therefore, we do not see them in the range-doppler plot (Fig.4.13e), which is captured with Radar1 settings. Car1 is parked, the minibus is approaching, and Car2 is going away. Our ego vehicle is not moving, and most of the scene appear at 0m/s as a line.



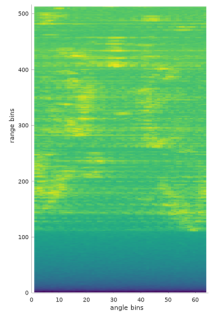
(a) Scene from the road



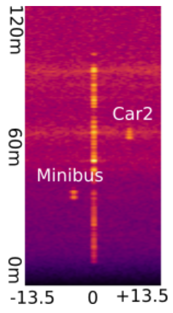
(b) Scene from the top



(c) Range-Angle plot



(d) Range-Angle



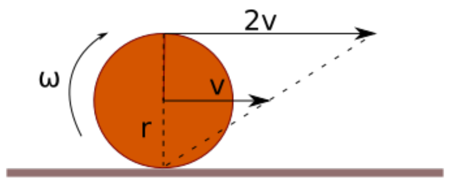
(e) Range-Doppler

Figure 4.13: An urban setting using scene description utility

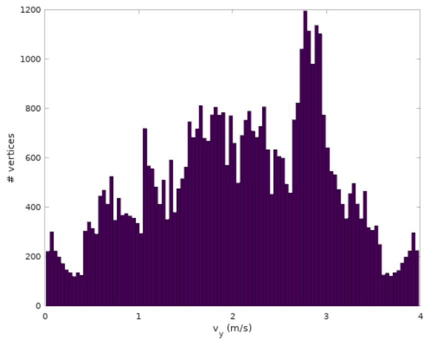
<u>Rolling wheel</u>. We simulated a wheel with tire (Fig.4.14a) in rolling without slipping motion. In this motion, the contact point's velocity is zero, and the top of the tire moves at twice the center's speed (Fig.4.14b). The overall diameter of the tire

is around 0.57m, and we used center speed 2m/s, resulting in angular velocity 7.02 rad/s. In Figure 4.14c, we plotted the histogram of vertex velocities in the rolling direction. We observed the wheel from right and behind, at about 30deg. This angle results in a cosine scaling of the maximum speed on the histogram from 4m/s to 3.5m/s. When we plot the FFT output for range-doppler, we see a single peak with long skirts (Fig.4.14d). The doppler bins range from -5.07m/s to 5.07m/s for Radar2 settings. Without a threshold value, it is not meaningful to indicate an observed velocity range, but the peak includes $[0, 3.5]m/s$ range.

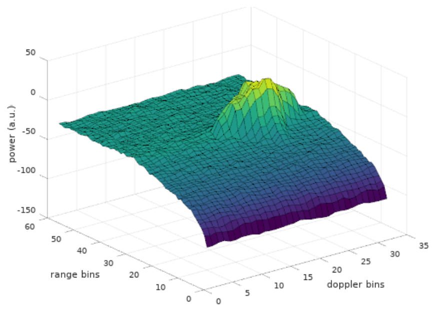
(a) Wheel and tire model



(b) Rolling without slipping



(c) Vertex v_y histogram



(d) Range-Doppler

Figure 4.14: Rolling wheel model

<u>Using with CARLA</u>. CARLA has a startup script that accepts command-line parameters. We extended these parameters with: `--radar_plugin`, `--headless_plugin`, and `--[0|1][0|1][0|1]`. We can run CARLA with Unreal Editor and integrated

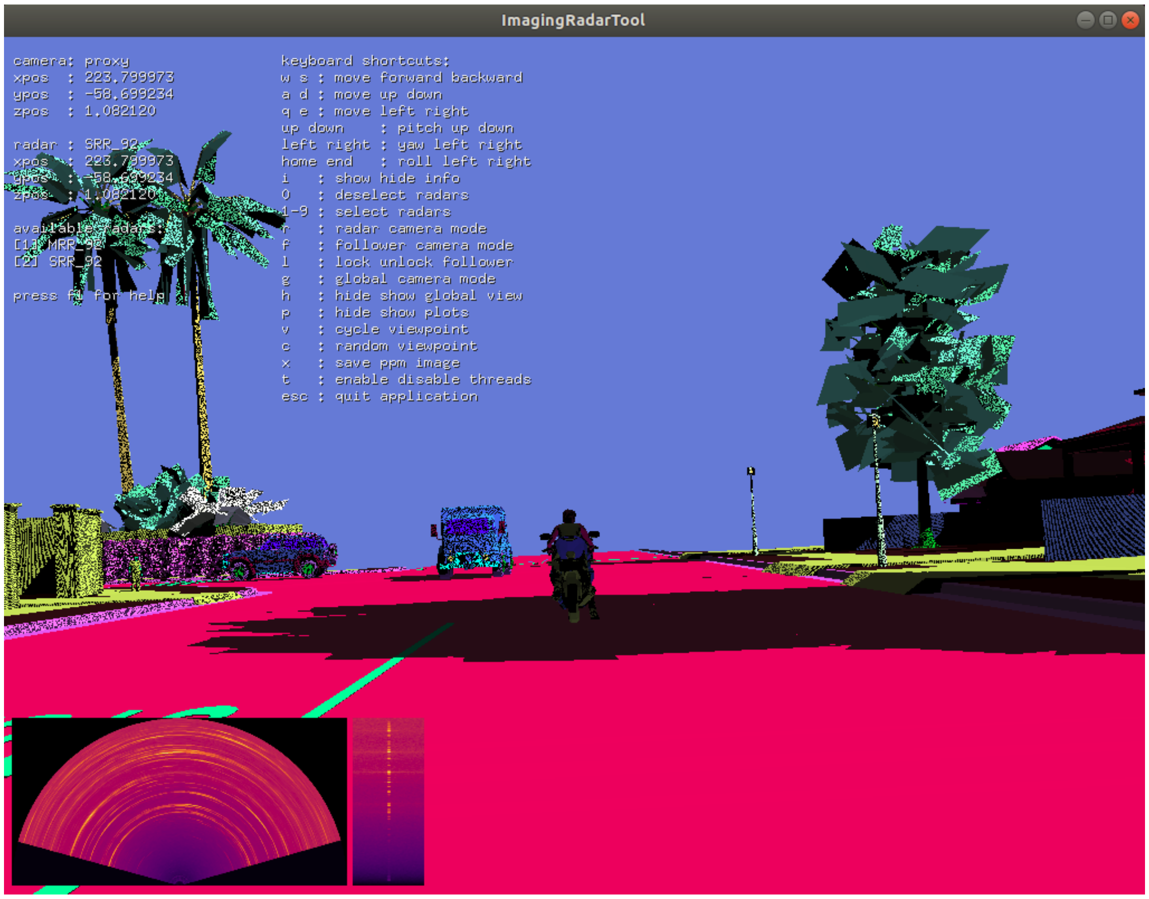
plugin with the first two options. The first one automatically opens up a GLUT-based GUI, and the second one is intended for remote operations without a GUI. If neither of these options is specified, then we have to run GameMeshViewer or ImagingRadarTool externally. The third option is a set of flags indicating: write game data to the filesystem, enable shared memory, expect processed data back to the game. For example, to run CARLA without any radar plugin, integrated or external, use `--000`. We can run with the external tool without writing game data nor expecting processed data back by using `--010`. The third flag depends on the second one, for example, `--001` would be invalid, but `--011` is valid. In Figure 4.16 we show the game inside the editor running concurrently with the external ImagingRadarTool.

In Fig.4.15, we show the GUI of the tool with a recorded scene from CARLA, and in Fig.4.16, we show it running together with the game engine editor. We implemented a spherical mesh filter for the radar context. The radius of the filter is greater than the maximum range of the radar. The physics of each radar is also decoupled.

In Fig.4.17, we show another scene and annotated the overlay plot with objects. Notice that the horizontal field of view of the proxy camera is smaller than the radar.

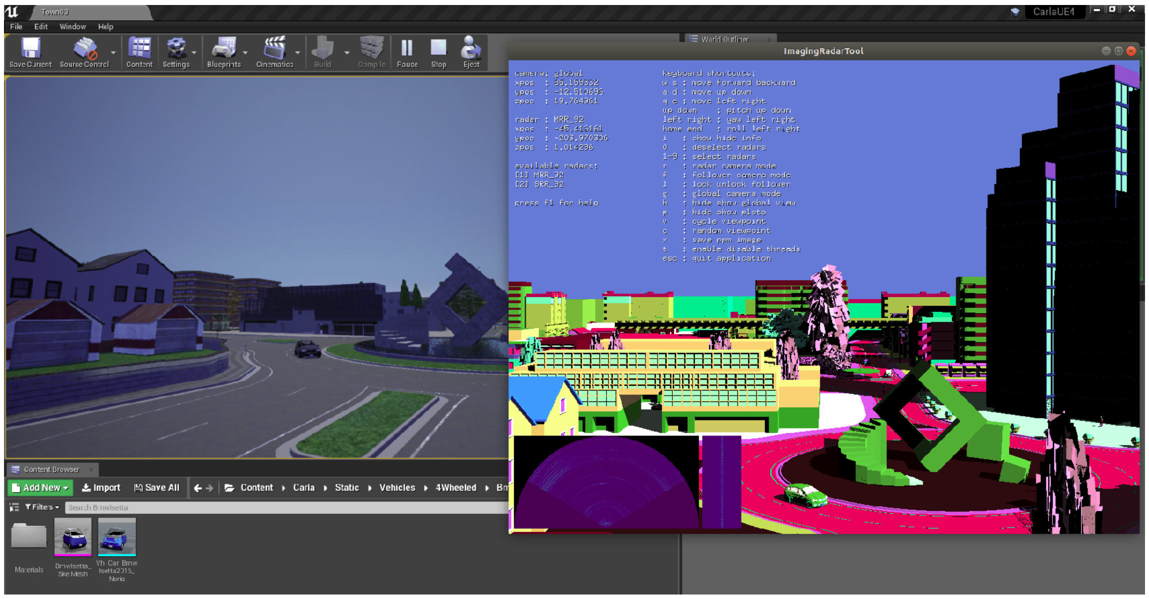
In Fig.4.18, we show the dependence of the details to the number of rays used. We generally use maximum bounce $n_b = 5$. In Fig.4.19b, we show the output from a raycast simulation ($n_b = 1$). We also normalize the power level for each distance R by multiplying with R^4 (Eqn.2.1); without this extra step, the output looks like Fig.4.19a.

Unity. In Unity, we define the plugin behavior inside RadarObserver.cs script with booleans and use external tools. In Figure 4.20, we show a simple static scene in the editor and ImagingRadarTool running concurrently.



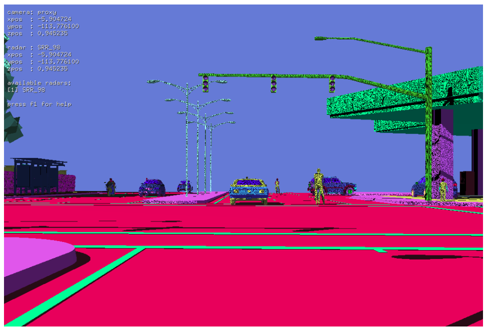
```
camera: proxy                   keyboard shortcuts:
xpos   : 223.799973             w s : move forward backward
ypos   : -58.699234             a d : move up down
zpos   : 1.082120               q e : move left right
                                up down     : pitch up down
radar  : SRR_92                 left right  : yaw left right
xpos   : 223.799973             home end    : roll left right
ypos   : -58.699234             i   : show hide info
zpos   : 1.082120               0   : deselect radars
                                1-9 : select radars
available radars:               r   : radar camera mode
[1] MRR_92                      f   : follower camera mode
[2] SRR_92                      l   : lock unlock follower
                                g   : global camera mode
press f1 for help               h   : hide show global view
                                p   : hide show plots
                                v   : cycle viewpoint
                                c   : random viewpoint
                                x   : save ppm image
                                t   : enable disable threads
                                esc : quit application
```

Figure 4.15: ImagingRadarTool



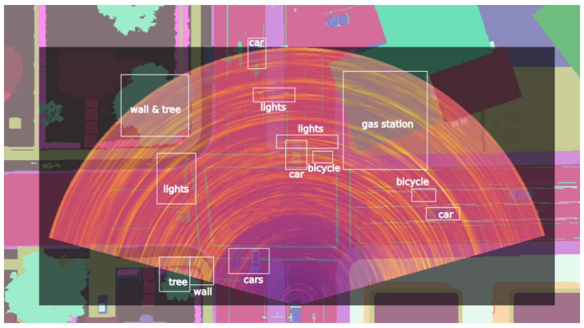
```
camera: global                  keyboard shortcuts:
xpos   : 85.159332              w s : move forward backward
ypos   : -12.510696             a d : move up down
zpos   : 19.764961              q e : move left right
                                up down     : pitch up down
radar  : MRR_92                 left right  : yaw left right
xpos   : -45.416161             home end    : roll left right
ypos   : -203.970306            i   : show hide info
zpos   : 1.014296               0   : deselect radars
                                1-9 : select radars
available radars:               r   : radar camera mode
[1] MRR_92                      f   : follower camera mode
[2] SRR_92                      l   : lock unlock follower
                                g   : global camera mode
press f1 for help               h   : hide show global view
                                p   : hide show plots
                                v   : cycle viewpoint
                                c   : random viewpoint
                                x   : save ppm image
                                t   : enable disable threads
                                esc : quit application
```

Figure 4.16: ImagingRadarTool running online with Unreal Editor CARLA game



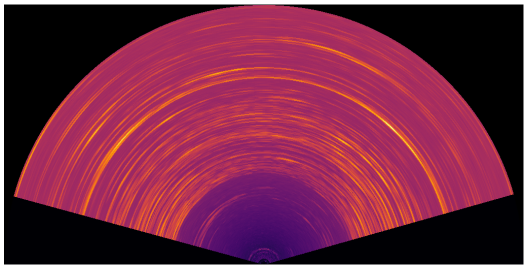
```
camera: proxy
xpos  : -5.904724
ypos  : -113.776100
zpos  : 0.945235

radar : SRR_98
xpos  : -5.904724
ypos  : -113.776100
zpos  : 0.945235

available radars:
[1] SRR_98

press f1 for help
```

(a) Proxy camera

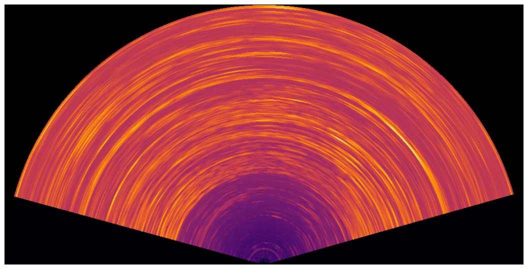


(b) Top view RA overlay

Figure 4.17: Scene near the gas station

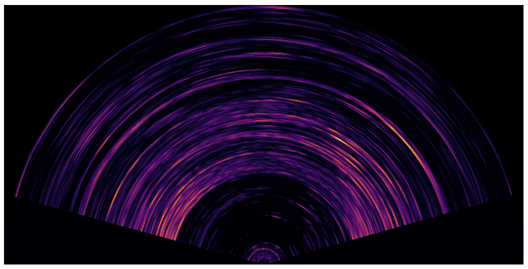


(a) $N = 500k$

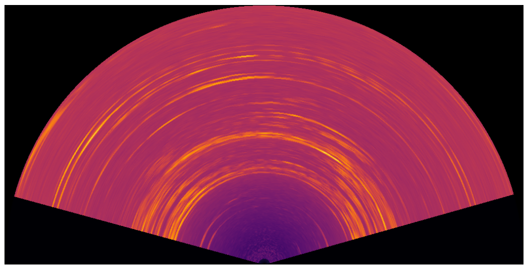


(b) $N = 5M$

Figure 4.18: Range-Angle with different number of rays, $n_b = 5$

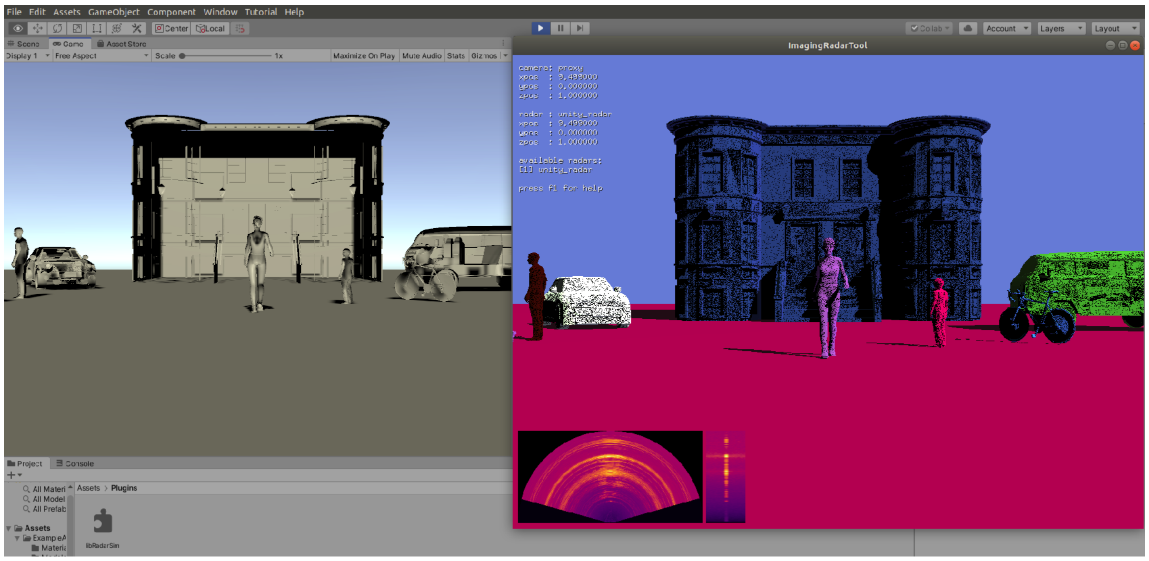


(a) Without R^4 correction



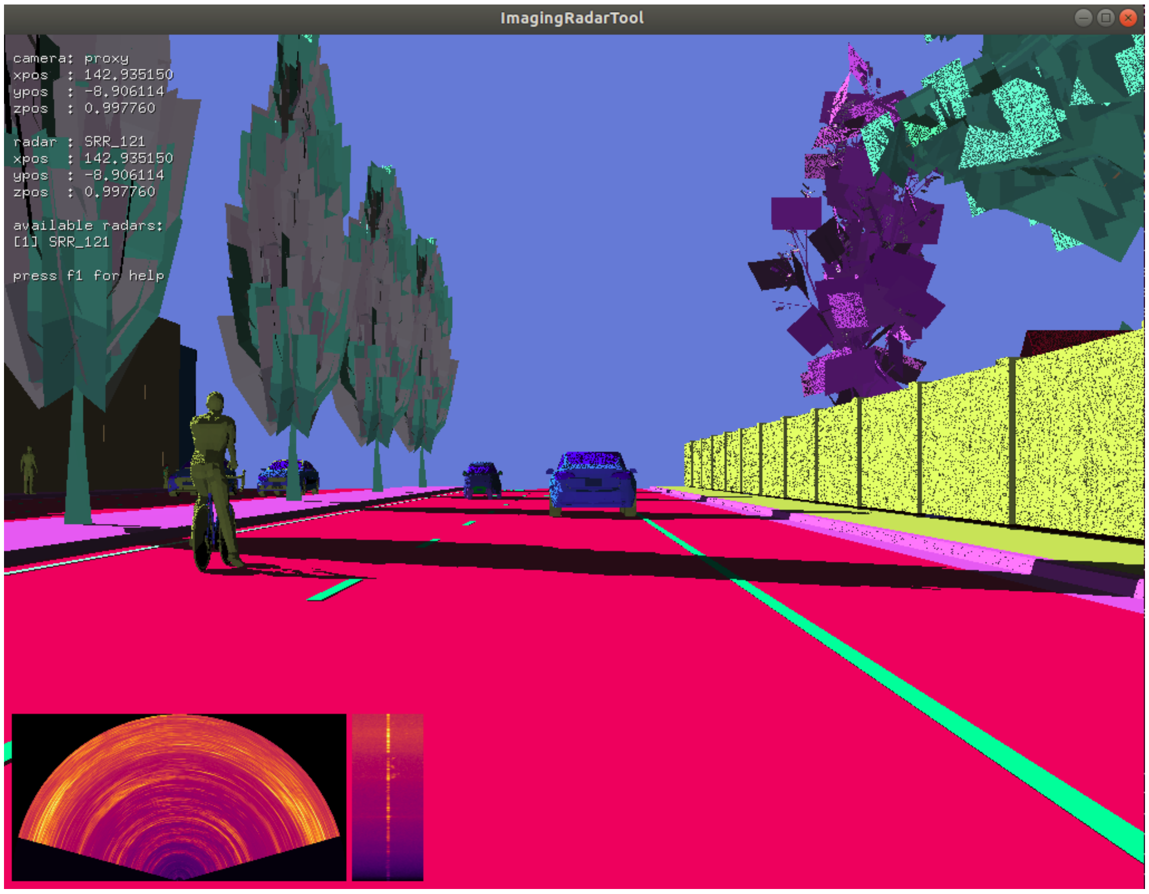
(b) Raycasting $n_b = 1$

Figure 4.19: Range-Angle plots with $N = 5M$



```
camera: proxy
xpos  : 9.499000
ypos  : 0.000000
zpos  : 1.000000

radar : unity_radar
xpos  : 9.499000
ypos  : 0.000000
zpos  : 1.000000

available radars:
[1] unity_radar

press f1 for help
```

Figure 4.20: ImagingRadarTool running online with Unity Editor



```
camera: proxy
xpos  : 142.935150
ypos  : -8.906114
zpos  : 0.997760

radar : SRR_121
xpos  : 142.935150
ypos  : -8.906114
zpos  : 0.997760

available radars:
[1] SRR_121

press f1 for help
```

Figure 4.21: Another scene from Town03 of CARLA

Chapter 5: Summary

Our main goal was to create a tool to generate synthetic radar data for autonomous driving research. We wanted our tool to work with game engines to capture multi-modal sensor data using publicly available vehicle simulators. Our approach was to replicate game data in memory and simulate electromagnetic wave propagation and scattering at the high-frequency regime using raytracing. We developed an FMCW MIMO radar simulator and integrated it via plugin interfaces to multiple vehicle simulators built with different game engines.

Below, we share some of our experiences during this project and point out possible future improvements.

5.1. Conclusions and Future Work

In general, we did not have any issues with the build-systems themselves. Using multiple Linux, game engine, and simulator versions causes some problems with compiler and other system dependencies. We experimented with Dockerized builds to solve these issues for Unreal-based simulators, which requires significant disk space. Eventually, we decided to fix our development versions.

Game data replication is costly and affects the overall performance. It requires an understanding of how each game engine organizes its data. We could only cover a subset of all possible mesh data in Unreal and found it easier to work with Unity in this respect; however, replication performance was better with Unreal.

Data homogenization enabled us to define our radar simulator as an external software component and made it portable; however, as highlighted in the first chapter, creating lower-level libraries and reusing them within engine plugins might be a better, more efficient approach. Only for practical reasons, we chose to build game plugins.

We had limited information on what types of data we could extract from the game engines. Some of the decisions on raytracing implementation got affected by it. Since our raytracing approach samples the geometry limited by solid-angle size, using detailed mesh for distant objects, will not increase accuracy. Therefore we could have selected appropriate LOD for some of the meshes in each radar scene to improve performance.

We did not have much time to delve into numerical methods for the radiation integral; instead, we focused on other software development activities. That is why we accepted the side effects of triangle aberration. We still need some carefully designed experiments to observe any issues and implement something more accurate later.

We introduced mesh filtering for radar scenes late, right after we build the global scene mesh. It caused significant performance problems than the one it intended to solve. Rather than a single-stage vertex-distance-based filter, we could have used a pre-filter using an axis-aligned bounding box for each component at SimulationDataManager or even at RadarBridge. Such changes would have resulted in slightly different responsibilities for multiple classes and would have changed the shared memory organization. We decided to leave it for future performance improvements.

Finding material properties at 77GHz was another difficulty we faced. A straightforward internet search did not result in links to easily accessible material databases. We had to search the materials individually and collate data from published materials. We assumed material properties as constants to avoid material updates, which is too ideal; for example, a thin water film may form on objects based on weather conditions, or water puddles or ice might form on the road surface. These nonidealities were beyond the scope of our thesis. Incorporating weather information into sensor state data can also be useful to mimic variations of averages in space for large scale simulations.

Since we used separate managers for each radar and already filtered their mesh, we can quickly adapt the implementation to work on multiple GPUs. Having different contexts raytraced independently limits some use cases. For example, with many radars on the road, there is a possibility that they may interfere, and without a global context, we can not simulate this effect. However, we can simulate multiple radars on the same vehicle with the same filtered mesh and process them on the same GPU.

We did not consider any waveforms other than periodic upward-ramps with linear chirp. We can apply similar semianalytic approaches for other waveforms.

We also did not carefully consider timing effects, and assumed radars start transmitting at the beginning of each tick. Since we processed data at every tick, we implicitly assumed that the game FPS was equal to the radar frame rate. We fixed game engine timestep to get deterministic delta time values; however, our implementation is slow, and when the real-time FPS goes below 10Hz, the physics engine continues updating positions. We require a velocity scaling to correct this behavior.

The samples of the IF signal, which we filled our radar cube, do not contain all the terms from the mixer. For a high-performance radar system, we have to include more terms. In that case, we also have to raytrace on a motion-blurred scene, which would increase the computational cost.

References

[1] N. Karla and S. M. Paddock., "Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?" *RAND Corporation. Santa Monica, CA*, 2016. [Online]. Available: https://www.rand.org/pubs/research_reports/RR1478.html

[2] NVIDIA, "A Path for Safe Self-Driving: NVIDIA Opens DRIVE Constellation Platform to Simulation Partners," 2018. [Online]. Available: https://blogs.nvidia.com/blog/2018/09/12/drive-constellation-open-simulation/

[3] D. Bruckner, R. Velik, and Y. Penya, "Machine Perception in Automation:A Call to Arms," *EURASIP Journal on Embedded Systems*, vol. 2011, no. 1, p. 608423, Jan 2011. [Online]. Available: https://doi.org/10.1155/2011/608423

[4] F. Rosique, P. J. Navarro, C. Fernndez, and A. Padilla, "A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research," *Sensors*, vol. 19, no. 3, 2019. [Online]. Available: http://www.mdpi.com/1424-8220/19/3/648

[5] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[6] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An Open Approach to Autonomous Vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[7] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[8] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[9] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Moeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving," 2020.

[10] M. Müller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, "Sim4CV: A Photo-Realistic Simulator for Computer Vision Applications," *Int. J. Comput. Vision*, vol. 126, no. 9, p. 902919, Sep. 2018. [Online]. Available: https://doi.org/10.1007/s11263-018-1073-7

[11] B. Fu, K. S. Shankar, and N. Michael, "RaD-VIO: Rangefinder-aided downward visual-inertial odometry," in *Proceedings of (ICRA) International Conference on Robotics and Automation*, May 2019, pp. 1841 – 1847.

[12] J. Chen, S. E. Li, and M. Tomizuka, "Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning," 2020.

[13] J. Gregory, *Game Engine Architecture, Third Edition*, 3rd ed. CRC Press, 2018.

[14] M. Pharr, W. Jakob, and G. Humphreys, “Physically Based Rendering.” Elsevier Inc, 2017, pp. 1235–1235.

[15] E. Haines and T. Akenine Mller, *Ray Tracing Gems.* Berkeley, CA: Springer, 2019. [Online]. Available: http://www.oapen.org/download/?type=document&docid=1007324

[16] D. Jenn, *Radar and Laser Cross Section Engineering, Second Edition.* American Institute of Aeronautics and Astronautics, Inc, 2005.

[17] H. H. Meinel, “Evolving automotive radar From the very beginnings into the future,” in *The 8th European Conference on Antennas and Propagation (EuCAP 2014)*, April 2014, pp. 3107–3114.

[18] S. M. Patole, M. Torlak, D. Wang, and M. Ali, “Automotive radars: A review of signal processing techniques,” *IEEE Signal Processing Magazine*, vol. 34, no. 2, pp. 22–35, March 2017.

[19] J. Dickmann, J. Klappstein, M. Hahn, N. Appenrodt, H. Bloecher, K. Werber, and A. Sailer, “Automotive radar the key technology for autonomous driving: From detection and ranging to environmental understanding,” in *2016 IEEE Radar Conference (RadarConf)*, May 2016, pp. 1–6.

[20] B. Major, D. Fontijne, A. Ansari, R. T. Sukhavasi, R. Gowaikar, M. Hamilton, S. Lee, S. Grzechnik, and S. Subramanian, “Vehicle Detection With Automotive Radar Using Deep Learning on Range-Azimuth-Doppler Tensors,” in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 924–932.

[21] K. Patel, K. Rambach, T. Visentin, D. Rusev, M. Pfeiffer, and B. Yang, "Deep Learning-based Object Classification on Automotive Radar Spectra," in *2019 IEEE Radar Conference (RadarConf)*. IEEE, 2019, pp. 1–6.

[22] A. Angelov, A. Robertson, R. Murray-Smith, and F. Fioranelli, "Practical classification of different moving targets using automotive radar and deep neural networks," *IET Radar, Sonar & Navigation*, vol. 12, no. 10, pp. 1082–1089, 2018.

[23] S. Kim, S. Lee, S. Doo, and B. Shim, "Moving Target Classification in Automotive Radar Systems Using Convolutional Recurrent Neural Networks," in *2018 26th European Signal Processing Conference (EUSIPCO)*. EURASIP, 2018, pp. 1482–1486.

[24] G. Zhang, H. Li, and F. Wenger, "Object Detection and 3D Estimation via an FMCW Radar Using a Fully Convolutional Network," *arXiv.org*, 2019. [Online]. Available: http://search.proquest.com/docview/2181668983/

[25] D. Feng, C. Haase-Schutz, L. Rosenbaum, H. Hertlein, C. Glaser, F. Timm, W. Wiesbeck, and K. Dietmayer, "Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–20, 2020.

[26] T.-Y. Lim, "Radar and Camera Early Fusion for Vehicle Detection in Advanced Driver Assistance Systems," 2019.

[27] M. Meyer and G. Kuschk, "Deep Learning Based 3D Object Detection for Automotive Radar and Camera," in *2019 16th European Radar Conference (EuRAD)*. EuMA, 2019, pp. 133–136.

[28] R. Prez, F. Schubert, R. Rasshofer, and E. Biebl, “A machine learning joint lidar and radar classification system in urban automotive scenarios,” *Advances in Radio Science*, vol. 17, pp. 129–136, 2019. [Online]. Available: https://doaj.org/article/19b2d69f98214bee8037738ee538bfb6

[29] R. Streubel and B. Yang, “Fusion of stereo camera and MIMO-FMCW radar for pedestrian tracking in indoor environments,” in *2016 19th International Conference on Information Fusion (FUSION)*. ISIF, 2016, pp. 565–572.

[30] A. Ouaknine, A. Newson, J. Rebut, F. Tupin, and P. Prez, “CARRADA Dataset: Camera and Automotive Radar with Range-Angle-Doppler Annotations,” *arXiv.org*, 2020. [Online]. Available: http://search.proquest.com/docview/2398387858/

[31] M. Meyer and G. Kuschk, “Automotive Radar Dataset for Deep Learning Based 3D Object Detection,” in *2019 16th European Radar Conference (EuRAD)*, 2019, pp. 129–132.

[32] D. Barnes, M. Gadd, P. Murcutt, P. Newman, and I. Posner, “The Oxford Radar RobotCar Dataset: A Radar Extension to the Oxford RobotCar Dataset,” 2019.

[33] M. Holder, P. Rosenberger, H. Winner, T. D’hondt, V. Makkapati, F. Maier, H. Schreiber, Z. Magosi, Z. Slavik, O. Bringmann, and W. Rosenstiel, “Measurements revealing Challenges in Radar Sensor Modeling for Virtual Validation of Autonomous Driving,” in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 11 2018, pp. 2616 – 2622.

[34] G. Hakobyan and B. Yang, “High-Performance Automotive Radar: A Review of Signal Processing Algorithms and Modulation Schemes,” *IEEE Signal Processing Magazine*, vol. 36, no. 5, pp. 32–44, 2019.

[35] J.-M. Jin, *Theory and computation of electromagnetic fields.* Oxford: Wiley-Blackwell, 2010.

[36] "https://github.com/redblight/raytramp."

[37] B. D. Rigling, "Raider tracer: a MATLAB-based electromagnetic scattering simulator," in *Algorithms for Synthetic Aperture Radar Imagery XIV*, E. G. Zelnio and F. D. Garber, Eds., vol. 6568, International Society for Optics and Photonics. SPIE, 2007, pp. 104 – 113. [Online]. Available: https://doi.org/10.1117/12.724779

[38] F. Chatzigeorgiadis, "Development of Code for a Physical Optics Radar Cross Section Prediction and Analysis Application," Tech. Rep., 2004. [Online]. Available: http://www.dtic.mil/docs/citations/ADA427120

[39] F. J. S. Moreira and A. Prata, "A self-checking predictor-corrector algorithm for efficient evaluation of reflector antenna radiation integrals," *IEEE Transactions on Antennas and Propagation*, vol. 42, no. 2, pp. 246–254, 1994.

[40] D. S. Lee, L. F. Gonzalez, J. Periaux, and K. Srinivas, "Efficient Hybrid-Game Strategies Coupled to Evolutionary Algorithms for Robust Multidisciplinary Design Optimization in Aerospace Engineering," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 133–150, 2011.

[41] D. Allison, C. Morris, J. Schetz, R. Kapania, L. Watson, and J. Deaton, "Development of a multidisciplinary design optimization framework for an efficient supersonic air vehicle," *Advances in Aircraft and Spacecraft Science*, vol. 2, no. 1, pp. 17–44, 2015. [Online]. Available: http://search.proquest.com/docview/1835664264/

[42] E. Wengrowski, M. Purri, K. Dana, and A. Huston, “Deep CNNs as a method to classify rotating objects based on monostatic RCS,” *IET Radar, Sonar & Navigation*, vol. 13, no. 7, pp. 1092–1100, 2019.

[43] H. Ling, R. . Chou, and S. . Lee, “Shooting and bouncing rays: calculating the RCS of an arbitrarily shaped cavity,” *IEEE Transactions on Antennas and Propagation*, vol. 37, no. 2, pp. 194–205, Feb 1989.

[44] K. E. Spagnoli, “An electromagnetic scattering solver utilizing shooting and bouncing rays implemented on modern graphics cards,” Master’s thesis, 2008, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2016-05-27. [Online]. Available: http://search.proquest.com.ezp-prod1.hul.harvard.edu/docview/304633536?accountid=11311

[45] H.-T. Meng, “Acceleration of Asymptotic Computational Electromagnetics Physical Optics - Shooting and Bouncing Ray (PO-SBR) Method Using CUDA,” Master’s thesis, University of Illinois at Urbana-Champaign, 2011.

[46] T. Courtney, J. E. Stone, and B. T. Kipp, “Using GPUs to Accelerate Installed Antenna Performance Simulations,” 2011.

[47] R. Brem and T. F. Eibert, “A Shooting and Bouncing Ray (SBR) Modeling Framework Involving Dielectrics and Perfect Conductors,” *IEEE Transactions on Antennas and Propagation*, vol. 63, no. 8, pp. 3599–3609, Aug 2015.

[48] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, “OptiX: A General Purpose Ray Tracing Engine,” *ACM Transactions on Graphics*, August 2010.

[49] S. G. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys, M. McGuire, and M. Stich, "GPU Ray Tracing," *Commun. ACM*, vol. 56, no. 5, pp. 93–101, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2447976.2447997

[50] R. Felbecker, L. Raschkowski, W. Keusgen, and M. Peter, "Electromagnetic wave propagation in the millimeter wave band using the NVIDIA OptiX GPU ray tracing engine," in *2012 6th European Conference on Antennas and Propagation (EUCAP)*, March 2012, pp. 488–492.

[51] C. Y. Kee and C. Wang, "Efficient GPU Implementation of the High-Frequency SBR-PO Method," *IEEE Antennas and Wireless Propagation Letters*, vol. 12, pp. 941–944, 2013.

[52] U. Chipengo, P. M. Krenz, and S. Carpenter, "From Antenna Design to High Fidelity, Full Physics Automotive Radar Sensor Corner Case Simulation," *Modelling and Simulation in Engineering*, vol. 2018, p. 19, 2018. [Online]. Available: https://doi.org/10.1155/2018/4239725

[53] T. Hanke, A. Schaermann, M. Geiger, K. Weiler, N. Hirsenkorn, A. Rauch, S. Schneider, and E. Biebl, "Generation and validation of virtual point cloud data for automated driving systems," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, Oct 2017, pp. 1–6.

[54] N. Hirsenkorn, P. Subkowski, T. Hanke, A. Schaermann, A. Rauch, R. Rasshofer, and E. Biebl, "A ray launching approach for modeling an FMCW radar system," in *2017 18th International Radar Symposium (IRS)*, June 2017, pp. 1–10.

[55] Epic Games, "Unreal Game Engine," https://www.unrealengine.com.

[56] Unity Technologies, “Unity Game Engine,” https://www.unity.com.

[57] J. Fadaie, “The State of Modeling, Simulation, and Data Utilization within Industry: An Autonomous Vehicles Perspective,” 2019.

[58] K. Toompea, “Simulations for Training Machine Learning Models for Autonomous Vehicles,” 2020.

[59] V. Gopalakrishnan Nair, “A Study of Driving Simulation Platforms for Automated Vehicles,” 11 2018.

[60] CARLA, “CARLA Simulator,” http://carla.org.

[61] Microsoft, “AirSim,” https://github.com/Microsoft/AirSim.

[62] LGSVL, “LGSVL Simulator,” https://www.lgsvlsimulator.com.

[63] W. Menzel and A. Moebius, “Antenna Concepts for Millimeter-Wave Automotive Radar Sensors,” *Proceedings of the IEEE*, vol. 100, no. 7, pp. 2372–2379, 2012.

[64] P. Clarberg, “Fast Equal-Area Mapping of the (Hemi)Sphere using SIMD,” *Journal of Graphics Tools*, vol. 13, no. 3, pp. 53–68, 2008. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/2151237X.2008.10129263

[65] . Gonzlez, “Measurement of Areas on a Sphere Using Fibonacci and Latitude-Longitude Lattices,” *Mathematical Geosciences*, vol. 42, no. 1, pp. 49–64, 2010.

[66] J. Rockway, J. Meloling, and J. C. Allen, “Interpolating Spherical Harmonics for Computing Antenna Patterns,” Tech. Rep., 2011. [Online]. Available: http://www.dtic.mil/docs/citations/ADA554243

[67] A. Alayon Glazunov, M. Gustafsson, A. Molisch, and F. Tufvesson, *Physical Modeling of MIMO Antennas and Channels by Means of the Spherical Vector Wave Expansion*, 2009, vol. TEAT-7177.

[68] H. Zhang, W. Hong, J. Xu, and Y. Zhu, "Research on automotive windshield impact on the W-band millimeter-wave transmission," in *2015 Asia-Pacific Microwave Conference (APMC)*, vol. 3, 2015, pp. 1–3.

[69] V. C. Chen, *The Micro-Doppler Effect in Radar.* Norwood: Artech House, 2019.

[70] V. Viikari, T. Varpula, and M. Kantanen, "Automotive radar technology for detecting road conditions. Backscattering properties of dry, wet, and icy asphalt," in *Proceedings.* United States: IEEE Institute of Electrical and Electronic Engineers, 2008, pp. 276– 279, 5th European Radar Conference 2008 ; Conference date: 30-10-2008 Through 31-10-2008.

[71] M.-K. Olkkonen, "Studies on characterization of dielectric composite materials using radar and other microwave sensors," pp. 165 + app. 69, 2016. [Online]. Available: http://urn.fi/URN:ISBN:978-952-60-6950-0

[72] C. M. Alabaster, "The microwave properties of tissue and other lossy dielectrics," 2004. [Online]. Available: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.403677

[73] T. Wu, T. Rappaport, and C. Collins, "The Human Body and Millimeter-Wave Wireless Communication Systems: Interactions and Implications," *arXiv.org*, 2015. [Online]. Available: http://search.proquest.com/docview/2083387453/

[74] M. Chen, C.-C. Chen, S. Y.-P. Chien, and R. Sherony, "Artificial Skin for 76-77 GHz Radar Mannequins," *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 11, pp. 5671–5679, 2014.

[75] Y. Deep, P. Held, S. S. Ram, D. Steinhauser, A. Gupta, F. Gruson, A. Koch, and A. Roy, "Radar cross-sections of pedestrians at automotive radar frequencies using ray tracing and point scatterer modelling," *IET Radar, Sonar Navigation*, vol. 14, no. 6, pp. 833–844, 2020.

[76] A. Sengupta, F. Jin, R. Zhang, and S. Cao, "mm-Pose: Real-Time Human Skeletal Posture Estimation using mmWave Radars and CNNs," *IEEE Sensors Journal*, pp. 1–1, 2020.

[77] M. Bijelic, T. Gruber, F. Mannan, F. Kraus, W. Ritter, K. Dietmayer, and F. Heide, "Seeing Through Fog Without Seeing Fog: Deep Multimodal Sensor Fusion in Unseen Adverse Weather," 2019.

[78] R. Gourova, O. Krasnov, and A. Yarovoy, "Analysis of rain clutter detections in commercial 77 GHz automotive radar," in *2017 European Radar Conference (EURAD)*, 2017, pp. 25–28.

[79] A. A. Hassen, "Indicators for the Signal degradation and Optimization of Automotive Radar Sensors under Adverse Weather Conditions," Ph.D. dissertation, Technische Universität, Darmstadt, January 2007. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/765/

[80] C. Mtzler, "MATLAB Functions for Mie Scattering and Absorption," University of Bern, Tech. Rep., 2002.

[81] CARLA, "http://carla.org/2019/12/11/release-0.9.7/." [Online]. Available: http://carla.org/2019/12/11/release-0.9.7/

[82] ——, "https://carla.readthedocs.io/en/0.9.7/dev/how_to_add_a_new_sensor/."

[83] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns :elements of reusable object-oriented software*, 37th ed., ser. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995.

[84] D. A. K, G. M. C, and W. R. M, "Almond test body," 1989.

[85] A. Heldring, J. M. Rius, and J. M. Tamayo, "Direct Mom solution of electrically large problems with N2 complexity," in *Proceedings of the Fourth European Conference on Antennas and Propagation.* IEEE, 2010, pp. 1–4.

[86] D. Garcia-Donoro, I. Martinez-Fernandez, L. E. Garcia-Castillo, Y. Zhang, and T. K. Sarkar, "RCS Computation using a Parallel in-core and out-of-core Direct Solver," *Progress In Electromagnetics Research*, vol. 118, pp. 505–525, 2011.

[87] A. Woo, H. Wang, M. Schuh, and M. Sanders, "EM programmer's notebook-benchmark radar targets for the validation of computational electromagnetics programs," *IEEE Antennas and Propagation Magazine*, vol. 35, no. 1, pp. 84–89, 1993.

[88] D. Escot-Bocanegra, D. Poyatos-MartNez, R. Fernandez-Recio, A. Jurado-Lucena, and I. Montiel-Sanchez, "New Benchmark Radar Targets for Scattering Analysis and Electromagnetic Software Validation," *Progress In Electromagnetics Research*, vol. 88, pp. 39–52, 2008.

[89] B. C. Brock and A. W. Doerry, "Radar cross section of triangular trihedral reflector with extended bottom plate," 2009.

[90] N. Yamada, Y. Tanaka, and K. Nishikawa, "Radar cross section for pedestrian in 76GHz band," in *2005 European Microwave Conference*, vol. 2. IEEE, 2005, pp. 4 pp.–1018.