Investigating the Use of Ray Tracing for Signal-level Radar Simulation in Space Monitoring Applications

---

A Comparison of Radio Propagation Models

Presented by:

Mogamat Yaaseen Martin

A thesis submitted to the Department of Electrical Engineering in fulfilment of the academic requirements for the degree of

Doctor of Philosophy

March 2023

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.

2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this thesis from the work(s) of other people has been properly attributed, cited and referenced.

3. I know the meaning of plagiarism and I declare that all work in this document, save for that which has been acknowledged, is my own. This thesis has been submitted to the Turnitin® module for checking originality, and I confirm that I have resolved all concerns that were revealed through this process.

4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own.

5. This thesis is being submitted in fulfilment of the academic requirements for the degree of Doctor of Philosophy in the Department of Electrical Engineering at the University of Cape Town. It has never been submitted for examination at any other university.

Abstract

This thesis presents the design and development of an accelerated signal-level radar simulator with an emphasis on space debris monitoring in the Low Earth Orbit. Space surveillance represents a major topic of concern to astronomers as the threat of space debris and orbital overpopulation looms – particularly due to the lack of effective mitigation techniques and the limitations of modern space-monitoring sensors. This work thus aimed to investigate and design possible tools that could be used for training, testing and research purposes, and thereby aid further study in the field.

At present, there exist no three-dimensional, ray-traced, signal-level radar simulators available for public use. As such, this thesis proposes an open-source, ray-traced radar simulator that models the interactions between spaceborne targets and terrestrial radar systems. This utilises a ray-tracing algorithm to simulate the effects of debris size, shape, orientation, and material properties when computing radar signals in a typical simulation. The generated received signals, produced at the output of the simulator, were also verified against systems theory and validated with an existing, well-established simulator.

The developed software was designed to aid astronomers and researchers in space situational awareness applications through the simulation of radar designs for orbital surveillance experiments. Due to its open-source nature, it is also expected to be used in training and research environments involving the testing of space-monitoring systems under various simulation conditions. The software offers native support for measured Two-Line Element datasets and the Simplified General Perturbations #4 orbit propagation model, enabling the accurate modelling of targets and the dynamic orbital forces acting upon

them. As a result, the software has aptly been named the Space Object Astrodynamics and Radar Simulator – or SOARS.

SOARS was built upon the foundations of a general-purpose radar simulator known as the Flexible Extensible Radar Simulator – or FERS – which provided integrated radar models for propagation loss, antenna shapes, Doppler and phase shifts, Radar Cross Section modelling, pulse waveforms, high-accuracy clock mechanisms, and interpolation algorithms. While FERS lacked various features required for space-monitoring applications, many of its implementations were used in SOARS to minimise simulation limits and maximise signal rendering accuracy by supporting an arbitrary number of transmitters, receivers, and targets. The goal was thus to have the simulator limited only by the end-user's system, and to specialise the operation of the software towards space surveillance by integrating additional features – such as built-in models for environmental and system noise, multiscatter effects, and target modelling using meshes comprised of triangular primitives.

After completing the software's development, the ray-traced simulator was compared against a more streamlined version of SOARS that made use of point-model approximations for quick-look simulations, and the trade-offs between both simulators (including software runtime, memory utilisation and simulation accuracy) were investigated and evaluated. This assessed the value of implementing ray tracing in a radar simulator operating primarily within space contexts, and evaluated the results of both simulators using detection processing as a demonstrated application of the system. And while the use of ray tracing resulted in significant costs in speed and memory, the investigation found that the ray-traced simulator generated more reliable results relative to the point-model version – providing various advantages in test scenarios involving shadowing and multiscatter.

The design of the SOARS software, as well as its point-model "baseline" alternative and the investigation into each simulator's advantages and disadvantages, are thus presented in this thesis. The developed programs were released as open-source tools under the GNU General Public Licence and are freely available for public use, modification and distribution.

Acknowledgements

"You can, you should, and if you're brave enough to start, you will."

– Stephen King

All praise be to Allah – the most gracious, the most merciful.

I dedicate this thesis to my beloved parents, Yusuf and Zulaiga, as well as to my brothers, Noer, Junaid, and Mughammad. Thank you for always being there for me in the most meaningful ways, and for providing with all the love, patience, and support that have been so instrumental in shaping me into who I am today. Nothing could ever repay you for being the very best family I could have asked for.

I also dedicate this thesis to my best friend and partner, Nabeela – the brightest beacon in my life, and my guiding star. I am so lucky to have been blessed with our beautiful friendship, and I shall forever treasure the loving bond into which it has grown. I cannot wait to spend the rest of our lives in eternal happiness together, Insha-Allah; thank you for always believing in me.

I extend my heartfelt appreciation to my closest friends – Qayyoom, Rishad, Saadiqah, Saleigh, Shoaib, and Tauriq. Each of you has stayed at my side through every difficulty, and I am immensely thankful to have you in my life. My gratitude is also offered to all of my extended family, friends, teachers, lecturers, and mentors throughout the years; every one of you has played a vital role in setting me on this path and helping me navigate it, and I could not be more grateful.

My deepest appreciation is extended to my supervisors, Prof Simon Winberg and Dr Yunus Abdul Gaffar, for the patience they have had with me and the guidance they have provided throughout this adventure. Thank you for lending your expertise towards this endeavour and motivating me all the way through to the end of it. Special thanks is also extended to Prof Fred Nicolls for all his efforts in setting me on the path of academia, and for providing me with so many amazing opportunities for furthering my career as an academic.

I would also like to thank my peers in the Department of Electrical Engineering at the University of Cape Town, as well as Mr David Macleod and his colleagues at the Centre for High Performance Computing, for their consistent support, inspiration, and encouragement. Finally, I extend my gratitude to the Council for Scientific and Industrial Research for affording me the opportunity to pursue this work through their bursary funding.

Thank you all for everything; you have my everlasting appreciation.

Contents

List of Figures

List of Tables

Listings

Nomenclature

3-D	Three-dimensional
ASAT	Anti-satellite
CA-CFAR	Cell Averaging Constant False Alarm Rate
CFAR	Constant False Alarm Rate
CHPC	Centre for High Performance Computing
CMB	Cosmic Microwave Background
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ECEF	Earth-Centered, Earth-Fixed
ECI	Earth-Centered Inertial
ESA	European Space Agency
FERS	Flexible Extensible Radar Simulator
FMA	Fused Multiply-Add
GEO	Geostationary Earth Orbit
GMST	Greenwich Mean Sidereal Time
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDF5	Hierarchical Data Format version 5

HPC	High-performance Computing
ISS	International Space Station
LEO	Low Earth Orbit
MEO	Medium Earth Orbit
NASA	National Aeronautics and Space Administration
NORAD	North American Aerospace Defense Command
ODPO	Orbital Debris Program Office
PAST	Phased Array System Toolbox
PPT3	Positions and Partials as functions of Time
PRD	Per Ray Data
PRF	Pulse Repetition Frequency
PRI	Pulse Repetition Interval
RCS	Radar Cross Section
ROC	Receiver Operating Characteristic
RSO	Resident Space Object
RTS	Ray Tracing Simulator
Rx	Receiver
SAR	Synthetic Aperture Radar
SEM	Size Estimation Model
SGP4	Simplified General Perturbations #4
SKA	Square Kilometre Array
SOARS	Space Object Astrodynamics and Radar Simulator
SSA	Space Situational Awareness
SSN	Space Surveillance Network
SST	Space Surveillance and Tracking

TEME	True Equator, Mean Equinox
TLE	Two-Line Element
Tx	Transmitter
USSSN	United States Space Surveillance Network
XML	Extensible Markup Language

Chapter 1

Introduction

This thesis focuses on the simulation of radar return signals from space objects in near-Earth orbits, as well as investigate the use of ray tracing to improve simulation accuracy. The work entails making use of accelerated computing methods to accommodate the intensive processing required for both ray tracing and the simulation of large numbers of "space junk" objects – such as the remains of old satellites and space vessels. These objects pose a significant threat to valuable spacecraft in orbit as well as the lives of astronauts aboard the International Space Station (ISS). The continued growth of the space debris population also threatens the future of all space activity, such as system launches and satellite capabilities.

1.1 Background

Orbital space debris are artificial objects such as spent rocket stages, inactive satellites, old space equipment and collision fragments. They are often produced due to high-speed collisions between objects as well as the intentional destruction of spacecraft in orbit [1, 2]. However, according to the National Aeronautics and Space Administration (NASA) and the Inter-Agency Space Debris Coordination Committee, the term "space debris" only encompasses man-made objects – hence asteroids and meteorites, as well as other objects originating from space, are not considered to be part of the space debris

population [3].

Space debris are often classified according to whether they had originated from fragmentation events. Typical objects may include inactive satellites, spent rocket stages, components released during launch, and lost equipment used in spacecraft – including possible personal belongings of astronauts. The classification of fragmentation debris, however, includes those objects produced by collisions between two pieces of debris, as well as impacts between debris and spacecraft or satellites. This latter category includes objects as small as specks of paint scraped off from the surface of a spacecraft [4], posing a significant threat to active spacecraft due to the high probability of collision [5].

An artist's interpretation of the space debris field around the Earth is depicted in Figure 1.1.

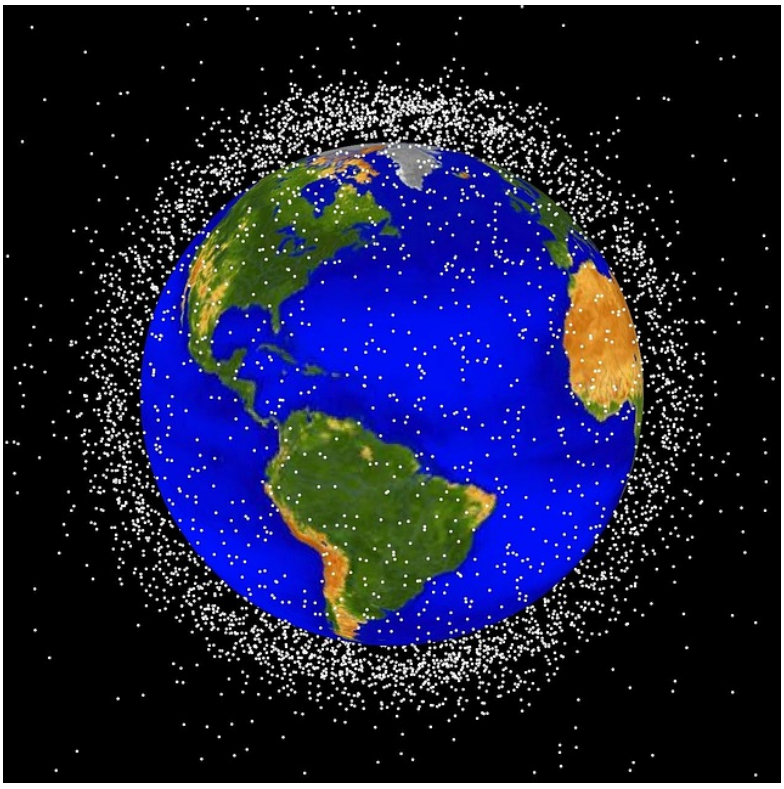


Figure 1.1: An artistic interpretation of the population of space debris around the Earth; the diagram is not to scale [6]

Despite space observation being a relevant field of study for several decades, a large portion of the population of spaceborne objects remains unclassified. These objects pose an urgent threat to spaceflight and space missions as they tend to travel at incredibly high speeds – typically around 10 km/s in the Low Earth Orbit (LEO) [5], which generally covers altitudes between 200 km and 2000 km [7]. Any potential collision between these objects and a rocket or satellite, for instance, could therefore result in significant damage and potentially put the spacecraft out of commission.

Such impacts could exponentially worsen if Resident Space Objects (RSOs) – comprised of both active spacecraft and space debris – were to collide with one another in near-Earth orbits. The result of such collisions would be the generation of many smaller debris particles, which could accumulate indefinitely unless de-orbited due to atmospheric drag. This could exponentially increase the probability of collisions between debris and spacecraft as posited by the Kessler Syndrome [8], while also posing an additional threat via re-entry through the Earth's atmosphere. The ever-growing space debris population may thus, in time, cause space activities in near-Earth regions to be fundamentally hazardous and nearly impossible.

Because of the dangers posed by space debris, asteroids, and even orbiting satellites, this project aimed to alleviate the problem by developing a complete radar simulator for use in space sensing applications. With the rising population space object numbers and the ever-increasing probability of collisions occurring, it is crucial for researchers and systems engineers to be able to accurately model potential monitoring systems in simulation and encourage their deployment on a global scale.

1.2 Problem Statement

In 2012, it was reported that the United States was able to track approximately 22,000 objects in the Earth's orbit. This figure was comprised of close to 1,000 working satellites – of which approximately 450 were in LEO – and 21,000 debris objects, each with a diameter greater than 10 cm [9]. More alarmingly, however, is the fact that debris objects with diameters smaller than 10 cm

are unable to be detected and tracked consistently by modern technologies. Estimates showed that there were potentially hundreds of millions of debris with a diameter smaller than 1 cm [10]. These pose a significant risk as there are no viable means of executing active collision avoidance manoeuvres for such tiny objects.

The uncertainties in the safety of the space environment have led to the conception of the Space Situational Awareness (SSA) programme [11] by the European Space Agency (ESA) in 2009, defined as the thorough knowledge of the space environment and incorporating the tracking of RSOs [12]. Investing in SSA is a critical issue as space systems have become increasingly valuable to developed economies, particularly for military and security purposes. However, space-based systems also serve important civilian functions, such as for commercial travel, telecommunication networks, and weather forecasting [13]. Thus, the aim of the SSA programme is to protect these invaluable functions.

This work focuses on one of the three main components of SSA, as defined by the ESA [14, 15] – namely Space Surveillance and Tracking (SST) for RSOs. SST mainly concerns the use of advanced sensing systems for the detection, imaging, and tracking of RSOs and is thus used for prediction purposes. For instance, this may include assessing the risks that space debris pose to human life and valuable spacecraft, as well as the potential threat to Earth-based property caused by RSOs re-entering the Earth's atmosphere.

For SSA to be successfully practised through the detection and tracking of space objects, advanced instrumentation and sensors are required. These typically include terrestrial or space-based radar systems, such as [16], as well as optical and laser instruments [17, 18]. However, radar offers many advantages over optical or laser systems for space monitoring, namely being able to operate during the day or at night under various environmental conditions, operating consistently even at very large ranges, and improving orbital element accuracy and orbit determination for space targets [19]. Furthermore, radar is well-suited for observing objects in LEO – which this work focuses upon – while electro-optical systems are more appropriate for monitoring objects in higher orbital regimes [4].

Some ground-based space surveillance radars also make use of radio telescopes as receivers [20, 21], providing large apertures, low noise temperatures and high gain. These factors enable them to act as highly sensitive receivers when used in bistatic or multistatic configurations. Such a multistatic radar system was proposed by Agaba [13] in 2017, which assessed the feasibility of using a single high-power transmitter (Tx) with the MeerKAT radio telescope [22] as its receiver (Rx). The latter is comprised of 64 antenna elements and was developed as a precursor to the Square Kilometre Array (SKA) telescope [23], and a portion of the deployed system is depicted in Figure 1.2.

Figure 1.2: Photograph of several antennas of the MeerKAT radio telescope as deployed in the African Karoo region [24]

This served as a strong starting point for the work, particularly due to the lack of advanced space-oriented sensors in the Southern Hemisphere [13] – which severely limits the overall coverage of the international Space Surveillance Network (SSN). This highlighted the need for additional sensors to be constructed on the African continent to improve the global coverage of space monitoring. However, before new systems are constructed, extensive design and testing procedures are required – and simulations play a major role in supporting these phases of development.

This emphasises the importance of being able to accurately simulate potential

radar designs under realistic conditions, and this project aims to improve the modelling of such simulations. Currently, there exist no three-dimensional (3-D), ray-traced, pulsed, signal-level simulators for public use in an open-source model – particularly one that is specialised towards the context of space monitoring. And while Agaba [13] made use of a low-fidelity, point-model simulator [25] to design the MeerKAT radar, the simulator itself lacked various space-oriented features such as orbital modelling and environmental noise considerations. As such, this work aimed to develop a feature-complete, ray-traced radar and space simulator for generating raw signal returns. In this way, the complete simulator would improve modelling capabilities through an all-in-one solution that accounts for various facets of the simulation, providing more accurate results when designing and testing systems.

Related research was published in [26], but this emphasised the simulation of thermal models for characterising space debris using infrared signals. Another closely-linked publication [27] developed a radar simulator focused on tracking space debris, but this specifically used tracklets (sequences of radar tracking measurements) to gauge debris population sizes. Relative to these works, this thesis focused on researching signal-level radar simulation and, through this, generating the received signals observed at the output.

This project thus aimed to improve the modelling of space surveillance systems by developing a full, ray-traced, pulsed radar simulator specialised for space monitoring using terrestrial radars. This would account for various additional features, including orbit propagation, mesh-based target simulation with translational and rotational motion, noise considerations, Radar Cross Section (RCS) approximation, and more – while also implementing ray tracing for improved target and propagation modelling.

The implementation of ray tracing is expected to further improve the simulator's capabilities, but this requires an accelerated algorithm to model propagating signals as sets of rays – each of which are traced independently and interact with target objects in the simulated environment. This provides an optics-based solution for computing reflection, refraction, power attenuation, and more [28] as opposed to directly mapping a signal from a transmitter to a target.

One major benefit of ray tracing is the fact that each ray is independent – meaning that the path traced by each simulated ray can be computed individually. As such, ray processing is highly parallelisable and well suited for acceleration on GPUs using NVIDIA®'s Compute Unified Device Architecture (CUDA) [29]. CUDA facilitates communication between a Central Processing Unit (CPU) – referred to as the "host" – and a GPU – referred to as the "device". In this project, CUDA will be used to parallelise various computations as part of the simulation algorithms.

This work thus aims to investigate the use of ray tracing within the space context through an algorithm such as the NVIDIA® OptiX™ engine [30] – a programmable ray-tracing framework that is freely available for use with NVIDIA® Graphics Processing Units (GPUs). Engines such as OptiX™ are easily accelerable and can be tuned for various use cases, making them highly flexible. Alternative standalone models, such as the Bounding Volume Hierarchy (BVH) method [31] or the shooting-and-bouncing ray model [32], were also considered and are discussed later in the thesis.

Notable works on the use of ray tracing for radar simulation include [32–43], where each work demonstrated the use of ray tracing for an array of applications such as RCS approximation and electric field modelling. However, most of these works emphasise the electromagnetic behaviours of radar systems, but not *signal-level* simulation – particularly in the form of open-source software.

A basic diagram depicting the operation of a signal-level simulator is shown in Figure 1.3.

As indicated, a signal-level simulator emphasises the use of full radar system modelling techniques in a 3-D scene involving an arbitrary combination of systems and targets. The output of the simulator is then computed as a raw return signal observed at any receivers operating in the scene, allowing for users to configure a variety of experiments when designing systems and thus appreciate the differences that arise when varying critical parameters.

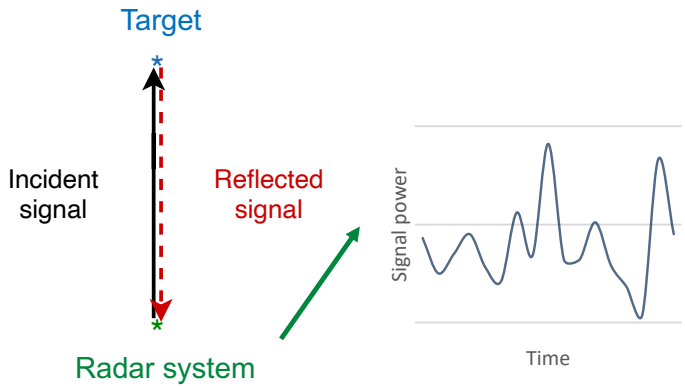


Figure 1.3: A basic signal-level experiment being simulated to produce a received signal at the output

1.3 Aims and Hypothesis

This project aims to develop a complete, ray-traced radar simulator that can model populations of space objects with various characteristics. This entails developing standalone, open-source software specialised towards space monitoring, which has aptly been named the Space Object Astrodynamics and Radar Simulator (SOARS) and is to be based on the template of an existing general-purpose radar simulator named the Flexible Extensible Radar Simulator (FERS) – a signal-level simulator initially developed by Brooker [25] in 2008.

SOARS is intended to be used for training and research purposes, as well as in the design of radar systems for space surveillance applications – such as the MeerKAT radar design proposed by Agaba [13] in 2017. Due to the inherent parallelism of ray tracing, and to mitigate the limitations of simulation complexity due to commodity computing hardware, GPU acceleration will be utilised via CUDA. This is to allow large quantities of RSOs and high-resolution radar rays to be simulated. For the testing portion of this work, all processing needs were accommodated using hardware provided by the Centre for High Performance Computing (CHPC) [44] in South Africa.

However, the overhead from setting up CUDA kernels and ray-traced propagation should not be ignored. Ray tracing is an inherently demanding process

[45] that can result in significant memory usage in large-scale applications (such as the gaming industry [46]). Thus, while the use of ray tracing should theoretically improve modelling accuracy and sample complex objects even at great distances (such as RSOs), there may also be significant trade-offs in the form of much longer program runtimes and greater memory requirements.

This work thus aims to investigate this relationship and assess the advantages and disadvantages of employing ray tracing in such a context. The hypothesis of this investigation is that both a "baseline" version of SOARS (without ray tracing) as well as a ray-traced version could *both* be useful in different scenarios, where the latter builds upon the foundations of the former by exchanging the existing propagation algorithm for an optics-based alternative. In this way, it is expected that the baseline program would run significantly faster at the cost of accuracy, as point-model approximations would be used and geometric optics theory would be ignored.

It is also hypothesised that the ray-traced version of SOARS would have minimal impact on the "accuracy" of the output signal compared to the baseline program, but it may be able to handle specific scenarios more realistically. The goal of this investigation is thus to develop both programs and compare their results, then assess the trade-offs through benchmarking, testing, and further comparisons to draw conclusions based on the hypothesis.

As such, for the main aims of this project to be achieved, the following objectives need to be met:

- Survey the literature associated with space object astrodynamics, radar systems theory and simulation, ray-tracing algorithms employed in scientific contexts, and High-performance Computing (HPC) methods; through this, identify areas in the existing research upon which this thesis could be built – and how the advancements of this work could be used to further related research in the field.
- Design and develop the baseline radar simulation program for space applications with an emphasis on software usability, simulation accuracy, and structural modularity. Thereafter, develop and implement a modular Ray Tracing Simulator (RTS) for radar signal simulation using

ray tracing. This ray tracer should also be able to serve as a standalone, open-source program that could be retrofitted onto other existing radar simulators.

- Optimise the simulators and benchmark their relative performance, and verify their operation under various scenarios. Comparisons should then be made between both programs and their advantages and disadvantages should be investigated and discussed.
- Draw conclusions based on the results achieved and the original hypothesis, and make recommendations for future work that could be done to further the work.

Chapter 3 provides further information on the approach followed to establish the requirements and focus for this work, as well as the consequential research methodology that was utilised in this project.

1.4 Scope and Limitations

The scope of this project focuses on the development of two full radar simulators for space monitoring – one with an emphasis on speed, and the other focused on simulation accuracy using ray-traced radio propagation and complex target modelling. This work aims to consider the drawbacks and advantages of both approaches and release both programs as open-source projects for further development by the community. And while the experiments in this thesis focused on the use of Agaba's MeerKAT radar [13] as the main radar system being tested, other designs were also used when convenient – such as a basic monostatic system used to test various features of the developed software.

The acceleration of the simulation code is another important aspect of this project's scope, as astronomy problems of this nature are typically too large to properly accommodate on a typical CPU. SOARS is therefore to be designed with CUDA in mind, where the program will only be able to run on an appropriate GPU. As no restrictions have been placed on the maximum number of simulated objects or simulation times, the main limitations of the

developed software are two-fold:

1. **Program runtime**: some ray-traced simulations may run for extended periods of time based on the simulation size and user limitations put in place; this is expected to change based on the number of targets, radar systems, signal rays, and more (as decided by the end-user); the baseline simulator is generally expected to offer closer to real-time performance relative to the ray-traced simulator
2. **System memory**: large volumes of host and/or device memory may be required to accommodate large-scale simulations; simulations will fail if any memory limitations are reached, and as such, this needs to be taken into account by the end-user when designing simulations

Another limitation of the SOARS software may be its computational uncertainties, some of which would naturally result from the use of floating-point representations for very large numbers. It is also worth noting that the use of different C++ compilers and/or different platform architectures may also result in calculation differences. This is particularly apparent when comparing floating-point calculation results between a CPU and GPU. Further information on such precision differences is presented in [47].

Some degree of error is also unavoidable when using Two Line Element (TLE) datasets, which represent measured data for various space objects as recorded by organisations such as the North American Aerospace Defense Command (NORAD). Additional uncertainty may result from the use of the Simplified General Perturbations #4 (SGP4) orbit propagation model, which introduces a degree of position error when propagating the orbits of space objects based on their TLE properties. However, such prediction errors may be mitigated by using the most recent version of a TLE dataset.

Further information on these uncertainties is provided in Chapters 2 and 3.

1.5 Contributions and Publications

The candidate believes that the following aspects of this project will constitute original contributions to the field:

1. Designing a generalised ray-tracing algorithm for modelling radio wave propagation in signal-level radar simulations
2. Developing a complete ray-traced, signal-level radar simulator for use in the design of terrestrial space-monitoring sensors
3. Evaluating the advantages and disadvantages of using of ray tracing for signal-level radar simulation in various scenarios, and drawing conclusions on the algorithm's suitability for space applications
4. Representing radar targets using meshes comprised of triangular primitives with size, shape, and material properties, and the effects of these parameters on ray-traced signals

Additionally, at the time of writing, the following research works have resulted from the contributions in this thesis:

- M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "The Design and Development of a GPU-accelerated Radar Simulator for Space Debris Monitoring," in *2021 5th High Performance Computing and Cluster Technologies Conference (HPCCT)*, p. 27–40, ACM, 2021. [48]
- M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "The Design and Implementation of a Ray-tracing Algorithm for Signal-level Pulsed Radar Simulation Using the NVIDIA® OptiX™ Engine," *Journal of Communications*, vol. 17, no. 9, pp. 761–768, 2022. [49]
- M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "Investigating the Use of Ray Tracing for Signal-level Radar Simulation in Space Surveillance," in *2022 5th International Conference on Information Communication and Signal Processing (ICICSP)*. IEEE, 2022, pp. 684–690. [50]
- M. Y. Martin, "Designing an Open-source Software Implementation of Ray Tracing for Signal-level Pulsed Radar Simulation Using NVIDIA® OptiX™," in *2022 Open-source Tools and Adaption for Signal Stream Processing Symposium (OSTASSP)*, Cape Town, South Africa, Oct. 2022 [Presentation only].

1.6 Thesis Outline

This section discusses the layout of the remainder of the thesis and summarises the contents of each chapter.

Chapter 2

Chapter 2 surveys the available literature relating to the major themes of this dissertation. This literature review covers the looming threat of space debris and typical properties associated with space objects, the need for SSA to be represented globally, and a summary of the dangers of uncontrolled space object population growth. The use of classical Keplerian elements and TLE sets for describing RSO orbits is also presented along with a discussion on orbital propagation models and coordinate reference frames. An overview of radar simulation frameworks is then provided, followed by a summary of some of the main aspects of ray tracing, GPU acceleration, CUDA, and the role of accelerated computing in this work.

In addition to the background theory, this chapter also serves to survey areas in the existing literature upon which this thesis could build. As a result, this chapter helps to direct this research (in furthering related work in the literature) and to identify its novel contributions to the field. Conclusions were thus drawn from the literature survey to guide the flow and development of the dissertation in the remaining chapters of this thesis.

Chapter 3

Chapter 3 discusses the overall project methodology and the evolution of the SOARS software from FERS, including its updated software structure and the integration of SGP4, TLEs, noise models, and CUDA into the source code. The chapter presents the main design considerations in developing a baseline version of the point-model SOARS software, including a discussion on the choice of programming language and external dependencies.

Additionally, this chapter explores many of the technical aspects relating to SOARS – such as the introduction of a new input file parser and a complete

list of user software requirements. The chapter also includes a discussion on the overall project methodology as well as the approach used to embed CUDA into specific software modules. The logic behind these developments is also presented such that the design ideas – and their changes from FERS – are demonstrated.

Chapter 4

Chapter 4 presents the implementation of the ray-tracing simulator and its integration into the main SOARS program. This allows for targets to be modelled by physical size, shape and material properties and accounts for the reflection and refraction of rays intersecting these objects.

Additionally, this chapter presents the logic and software structure of the ray tracer used in SOARS – as based on the NVIDIA® OptiX™ ray-tracing engine – as well as the changes made to this program for signal-level radar simulation to be made possible. This includes a discussion of the propagation, target, multiscatter and RCS models, and how these techniques were implemented in the ray tracer. A structural overview of the completed SOARS simulator is also provided to bridge the gap between chapters.

Chapter 5

Chapter 5 presents verification tests used to evaluate the operation of SOARS, with all uncertainties being acknowledged and fully discussed. Additionally, comparisons are made between the ray-traced and baseline versions of SOARS to assess their usage and trade-offs under different scenarios. Several performance benchmarks are also presented and the operation of the fully-developed simulator is compared against a list of predefined user software requirements. The chapter is then concluded with final discussions about the program and what the presented results mean for its usage.

Chapter 6

Chapter 6 presents the main conclusions derived from this thesis, summarising its most pertinent work. This is followed by an overview of some potential

future work that could be considered as extensions to the project in the long term.

The thesis is then concluded with a list of references.

Chapter 2

Literature Review

2.1 Overview

This chapter serves as a literature survey on the topics of space debris, radar system simulation for space surveillance, and ray tracing. The origins and evolution of the space debris population are first discussed, followed by an overview of SSA and its growing importance in astronomy. This leads to a discussion of Keplerian mechanics and the propagation of RSO orbits, followed by a discussion on radar simulation and then an overview of ray tracing methods. Additionally, this chapter explores various works from the literature to identify areas upon which this research could build.

The chapter concludes with a summary of the most salient points from the literature survey, offering a strong theoretical background to the work and summarising the most critical aspects of how this research aims to complement the existing literature.

2.2 Orbital Space Debris

This section summarises some of the key ideas from the literature surrounding the field of space debris and its history, properties of space junk objects, and the growth of the debris population over time.

2.2.1 History of Space Debris

The accumulation of space debris objects began with *Sputnik-1* – the first artificial satellite to be placed into orbit – which was launched on 4 October 1957 [17]. This brought about the beginning of the Space Age [51], throughout which thousands of satellites and spacecraft have been launched. Since then, the effect of collisions between space objects has become one of the main contributors to the growth of the space debris population. The first recorded collision between two RSOs occurred in July 1996, involving the French military satellite *Cerise* and a debris fragment from the *Ariane-1* rocket [52].

Another infamous debris generation event is the Chinese anti-satellite (ASAT) test that was conducted in 2007 [53], where a *Fengyun* satellite was intentionally destroyed in orbit. This was reported to have produced more than 2000 space debris objects – many of which are still being tracked by the United States Space Surveillance Network (USSSN) [54]. These objects have also been observed to cross through various high-value orbits for many spacecraft, resulting in shortened lifespans for some space assets as well as additional fuel consumption due to executing collision avoidance manoeuvres [55].

The result of the Chinese ASAT test is presented through an artist's interpretation in Figure 2.1, which highlights the approximate density of the resulting debris field produced by the satellite being destroyed. This was reported as being the single worst debris-generating event in history, with much of the produced debris expected to remain in orbit for centuries [56]. This easily surpassed the previous record that was held by the explosion of the *Pegasus* rocket body in 1996 [57].

However, studies have shown that both Russia and the United States are also responsible for a considerable proportion of the current space junk population [58]. Additionally, a similar ASAT event occurred in India in March 2019 [1,2], generating at least 400 additional space junk objects in orbit [59] which were reported to have directly threatened the lives of astronauts aboard the ISS [60]. Another ASAT missile test was conducted in November 2021, producing at least 1,500 pieces of trackable debris [61] (i.e., objects with sizes large enough to be detected by modern sensors).

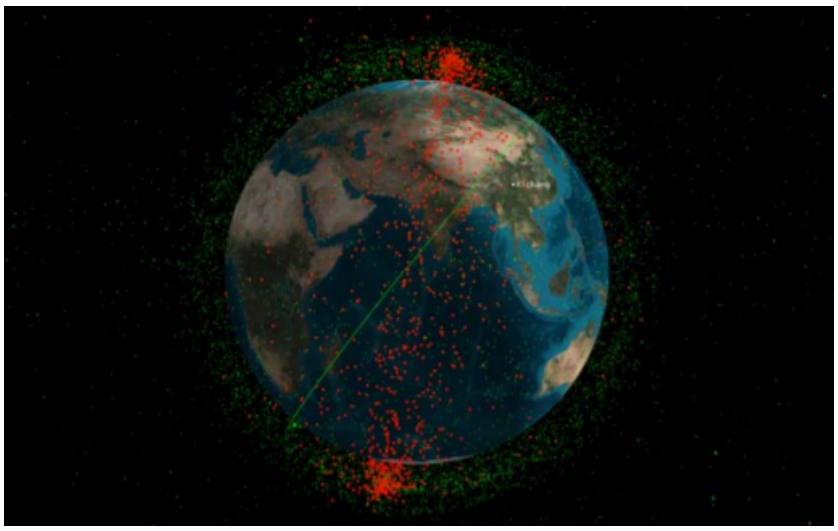


Figure 2.1: Approximation of the density of the debris cloud from the 2007 Chinese ASAT test (red) relative to the density of all other debris (green) [54]

In-orbit explosions have also generated large numbers of debris objects since the launch of *Sputnik-1*. In 1961, a portion of the *Thor-Ablestar* space rocket exploded and became the first known debris explosion, with many of the resulting RSOs still currently in orbit [62]. Since then, around 8,000 space objects have been launched into orbit, with nearly 2,000 satellites still active (as of February 2019 [63]). However, most satellites that were launched since 1957 have already undergone orbital decay, suggesting that most objects still in orbit are space junk instead of functioning assets [4].

Research has shown that out of the thousands of space missions conducted since the dawn of the Space Age, just ten of these missions are responsible for close to 33% of all catalogued satellite debris (as of July 2018) [64]. The same research also demonstrated that only four of these ten fragmentation events are attributed to discarded rocket bodies after having undergone breakage in orbit – and all four events could have been prevented if modern mitigation strategies were available [64].

2.2.2 Debris Properties

The term "orbital space debris" includes any kind of man-made space objects – or broken fragments thereof – that serve no functional purpose in orbit [65]. Most notably, defunct space assets and system fragments form part of the space debris population, but asteroids and meteorites do not. Spent stages of a rocket, as well as inoperative satellites, are some of the largest pieces of space debris, often having lengths between 10 cm and 10 m. However, objects as small as paint specks are also regarded as space debris, tending to vary in size between 2 µm and 0.2 mm [66, 67].

In fact, in June of 1983, an outer windowpane of NASA's *Shuttle Challenger* experienced an impact with a tiny space debris object. The windowpane had a thickness of around 0.625 inches (approximately 1.588 cm) and was tested to withstand immense pressures and temperatures; however, NASA eventually determined that the damage was caused by a speck of paint that was only 0.2 mm in size [68]. The impact of this paint fleck on the *Shuttle Challenger*'s outer windowpane is illustrated in Figure 2.2, highlighting the threat posed by even the smallest of debris.

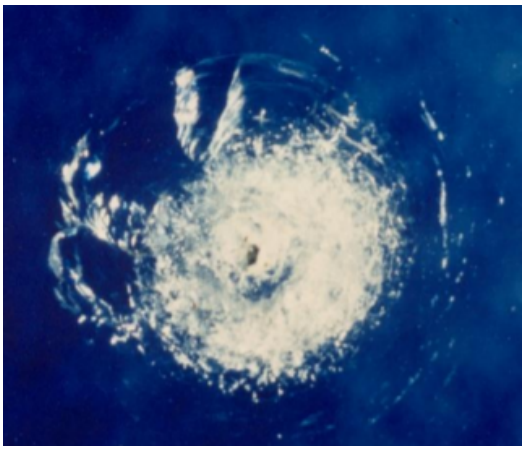


Figure 2.2: Impact of a paint fleck with a diameter of 0.2 mm against the outer windowpane of a NASA space shuttle [69]

This paint fleck was estimated to be travelling at an orbital velocity between 3 and 5 km/s [55] – a typical "hypervelocity" for an object in LEO [5]. Kennewell [17] warned that a collision between any spacecraft and a debris

object with a diameter of 10 cm, when travelling at hypervelocity speeds, would likely destroy the craft entirely. It has also been reported that, at such speeds, the impact of an aluminium sphere of 1 cm diameter "deploys the same energy as an exploding hand-grenade, with equally devastating consequences" [62].

This emphasises that, regardless of a space object's surface area or mass, it can pose a major threat to astronauts and valuable spacecraft due to its high impact velocity. This, coupled with the fact that many debris objects are metallic or ceramic, creates significant difficulty in removing space junk from the space environment. Additionally, any mechanism for salvaging or destroying these objects poses a risk of simply generating even more space debris. One such mechanism was the *solar sailing* concept [70–72], which entailed using the Sun's photon energy to propagate a physical sail through space to collect debris. The idea was ultimately abandoned due to both cost and the expected difficulty in accurately manoeuvring sails between space objects [73].

Currently, the USSSN are not able to consistently detect and track debris that are smaller than around 1 cm in size. This is a significant problem as most debris objects have sizes in the range of just a few millimetres in diameter [74], making them difficult to track even with the most sophisticated radar and electro-optical systems. The properties of a debris object can also vary significantly, such as its size, shape, velocity, area-to-mass ratio, RCS, and chemical composition – increasing the difficulty in monitoring them consistently.

Another important consideration for space debris is orbital decay, which causes an object orbiting at ±600 km or lower to eventually become de-orbited due to atmospheric drag [4]. Such an object will almost certainly burn up upon re-entry, but some larger pieces of space junk can survive this process and thus reach the Earth's surface. According to NASA's Orbital Debris Program Office (ODPO), one single catalogued piece of debris had fallen back to Earth every day for the past 50 years – however, no serious injury or significant property damage had ever been confirmed as having resulted from this phenomenon [75].

2.2.3 Kessler Syndrome

Most data involving small space debris objects are often fragmented and have large uncertainties associated with them; as such, short- and long-term mathematical models are used for predictive purposes [76]. One well-known example of this is the prediction model developed by Kessler and Cour-Palais [8] in 1978, who posited that the production of space debris would eventually lead to a chain reaction of exponential increases in the number of debris objects – particularly due to in-orbit collisions and explosions.

Kessler and Cour-Palais believed that this exponential debris generation could, at some point, create a debris "shell" in LEO and prevent any further space missions and operations from occurring in this orbital space. This theoretical chain reaction was aptly named the Kessler Syndrome [8]; this, together with the potential damage that small space junk poses to the environment, means that debris mitigation should be a high priority for astronomers worldwide.

Based on the Kessler Syndrome, Figure 2.3 shows the predicted long-term evolution of the space debris population in various orbit regimes (assuming that no mitigation measures are taken over the 100-year period). These simulations were executed by Klinkrad [62] using the ESA's MASTER 2001 software for modelling space environments.

Based on the reported results, the number of space objects in LEO is expected to increase almost exponentially throughout the rest of the century – with LEO at the greatest risk of overpopulation relative to the Medium Earth Orbit (MEO) and Geostationary Earth Orbit (GEO) regimes. This overcrowding could result in the loss of valuable daily services if any important space assets were to be damaged in orbit, including a significant impact on military, commercial and civilian uses. The loss of major artificial satellites could also affect telecommunication networks or cease navigation and weather forecasting capabilities, among others. As such, it is imperative that valuable space assets be protected through SSA.

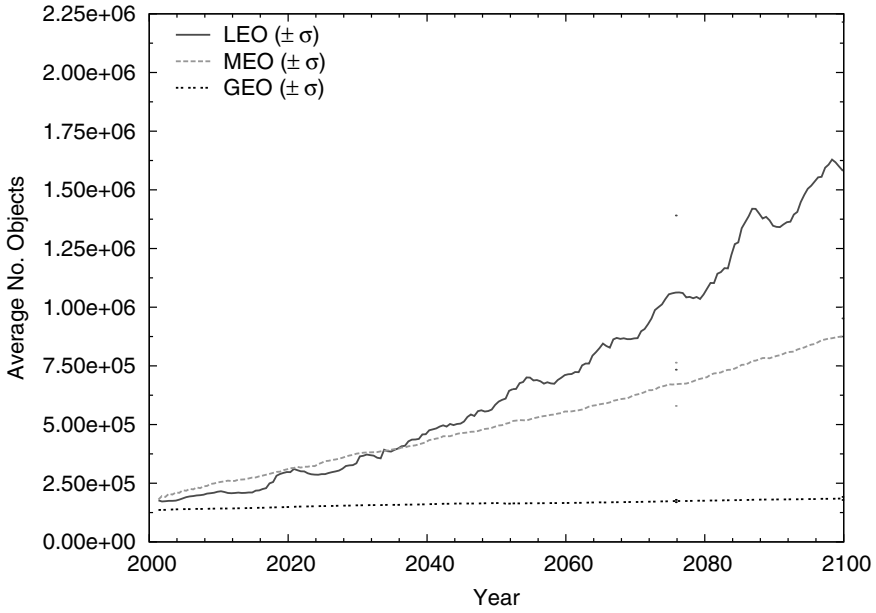


Figure 2.3: Predicted growth of the population of space debris objects with sizes greater than 1 cm as based on the Kessler Syndrome effect, shown for three orbit regimes after 10 Monte Carlo simulations (assuming no mitigation) over a 100-year time span; σ represents the standard deviation confidence intervals based on the Monte Carlo runs [62]

2.2.4 Space Junk Population

The ever-increasing number of spacecraft launches, debris collisions, and in-orbit explosions have led to the accumulation of hundreds of millions of space debris objects in orbit – most of which cannot be tracked reliably due to their small sizes and high velocities. Kennewell [17] reported that any space launch, on its own, will typically produce close to 100 individual pieces of space debris – including the launch vehicles (which remain in orbit) and the devices that are used to separate a satellite from its launch vehicle. According to the study, most fragmentation space debris objects result from explosions caused by the large number of batteries and unused fuel that remain in launch vehicles [17]. But these fragmentation events form only one of the main contributors to the debris population.

The most common of these categorisations are presented in the chart in Figure 2.4 (as adapted from [17]), which summarises the breakdown of all known space debris sources and sinks. This depicts the main catalysts that generate more space junk in orbit.

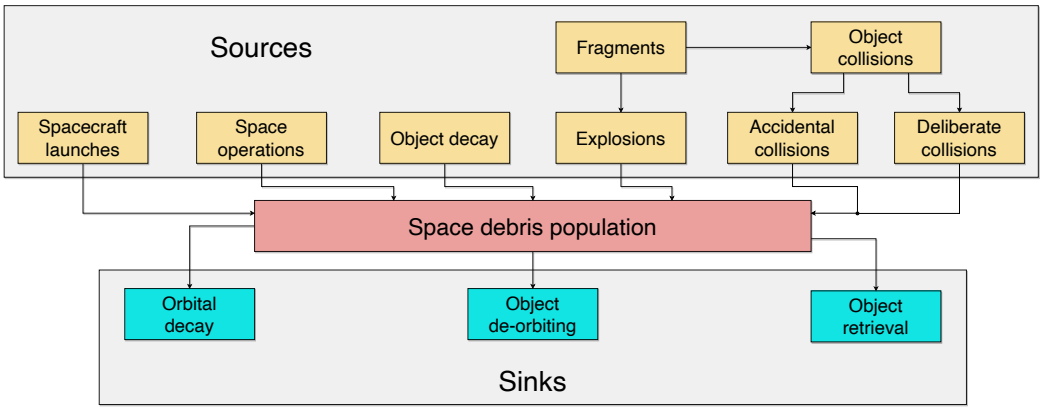


Figure 2.4: Sources and sinks of objects affecting the space debris population

Each of these sources contributes significant numbers of objects to the debris population, leading to the following ESA [77] population estimations of space junk:

- 34 000 debris objects with diameters greater than 10 cm
- 900 000 debris objects with diameters between 1 cm and 10 cm

- 128 million debris objects with diameters between 1 mm and 1 cm

Because of the alarming growth shown by space junk – particularly the large number of small, uncatalogued objects – NASA's ODPO began cataloguing detailed records of the growing population to assess both short- and long-term risks associated with space junk [78]. It has been reported that the number of debris objects generated in orbit has significantly increased on a year-to-year basis – just as mankind's knowledge of the space environment has also grown.

The recorded population of *catalogued* RSOs is illustrated in Figure 2.5, showing the changes in the number of tracked space junk objects every year since 1956 (as reported by NASA's ODPO in May 2019). This represents information documented directly by the ODPO, updated regularly via a quarterly newsletter released to the scientific community.

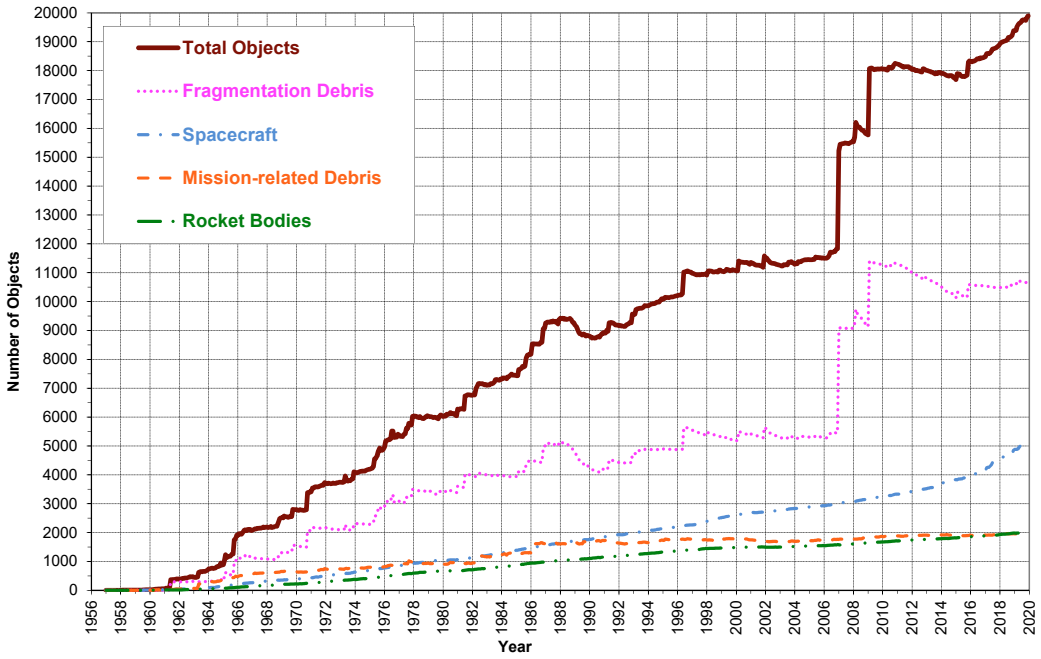


Figure 2.5: The catalogued space object population in LEO reported by the ODPO in May 2019 [79]

The "Total Objects" curve in Figure 2.5 depicts notable step increases in 2007 and 2009 – the first of which was due to the Chinese ASAT test discussed earlier in the chapter. The step increase in 2009 is attributed to the accidental

collision of an active *Iridium* satellite with an inoperative Russian satellite – resulting in the loss of a fully operational communications satellite and producing thousands of debris objects larger than 1 cm in size [80].

According to NASA, over 21,000 space debris objects have been quantified and are consistently monitored successfully even today [75]. Using a combination of both radar and optical sensors, the global SSN can detect space debris objects with diameters larger than around 5 cm in LEO. These can therefore be reliably tracked using terrestrial sensors (or even space-based systems [81]) and are being actively monitored and catalogued by the SSN.

Outside of the ODPO's space object database, there also exists the Russian Federation and ESA's joint debris catalogue – the Database and Information System Characterising Objects in Space, which is constantly being updated [82]. Unfortunately, neither this nor the SSN's database contain much information about space objects of sizes smaller than around 5 cm – mostly due to their very small RCS values, making them difficult to monitor even with advanced instrumentation. Tracking these smaller objects has thus presented significant difficulty, as millions of objects are presently uncatalogued.

2.3 Astrodynamics Aspects

This section details some of the astrodynamics aspects around which the target model (in the radar simulator) was developed. This includes the use of Keplerian elements for modelling the dynamics of RSOs, a description of NORAD's TLE datasets, and their use in describing the orbits of RSOs.

2.3.1 Keplerian Elements

RSO orbits are most often modelled by one of two common coordinate systems, namely the Keplerian and Cartesian systems. These two coordinate schemes are often used together for astrodynamics modelling and research – especially with how easily they can be alternated.

The Keplerian coordinate system is one way of describing both the position and velocity of an RSO that orbits the Earth, making use of six classical

Keplerian orbit elements – also known as Keplerians – as described in the following list:

- The semi-major axis a of the orbit ellipse (km)
- The eccentricity e of the orbit (dimensionless)
- The right-ascension of the ascending node Ω (degrees)
- The orbital inclination i (degrees)
- The argument of the perigee ω (degrees)
- The true anomaly value ν (degrees)

Together, these quantities describe the RSO's orbital size, shape, and orientation in relation to the Earth's equator, in addition to defining the object's position and movement along the orbit. The relationships between these elements are illustrated in Figure 2.6 (adapted from [83]), depicting the elliptical orbit of a typical RSO relative to the Earth.

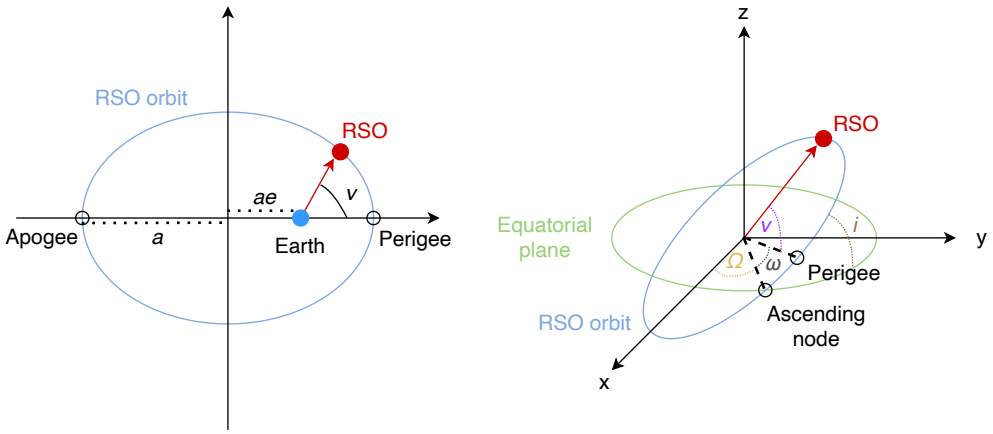


Figure 2.6: An elliptical Kepler orbit showing the classical elements

As shown in Figure 2.6, the *perigee* represents the point where the RSO is closest to the Earth, while the *apogee* represents the point where the RSO is furthest from it. The eccentricity e is also seen to be highly dependent on the size of the orbit and is related to both the *periapsis* (the distance from the Earth to the perigee) and the *apoapsis* (the distance from the Earth to the apogee).

The relationships between the eccentricity, e, and various orbit shapes (taking the form of conic sections) are presented in Table 2.1.

Table 2.1: Relationship between various orbit shapes (conic sections) and the eccentricity e

Eccentricity	**Type of Orbit Shape (Conic Section)**
$e = 0$	Circular
$0 < e < 1$	Elliptical
$e = 1$	Parabolic
$e > 1$	Hyperbolic

The illustrations in Figure 2.6 assume an elliptical orbit, but the same parameters could be derived for alternative conic sections as well. Figure 2.6 also depicts the elliptical orbit relative to a plane of reference in the Cartesian coordinate system – this is often referred to as the *equatorial plane*. The RSO's orbit crosses this plane from the south pole to the north pole at the ascending node, and the plane upon which the orbit lies is known as the *orbital plane*. The *inclination* i is then defined as the angle subtended between this orbital plane and the equatorial plane.

The right-ascension of the ascending node, Ω, describes the angle between the reference direction – namely the x-axis in Figure 2.6 – and the ascending node in the RSO's direction of movement from the ascending node. The argument of the perigee, ω, then describes the angle between the ascending node and the perigee. Thus, Ω and ω define the orbit's overall orientation. Finally, the *true anomaly* ν corresponds to the angle measured between the semi-major axis and the line drawn from the origin to the RSO of interest. As such, this parameter varies with time as the RSO propagates along its orbit around the Earth.

This completes the set of the six Keplerian elements, often denoted as the RSO's *ephemeris* and described in vector form:

$$\vec{x_k} = [a \quad e \quad i \quad \Omega \quad \omega \quad \nu] \tag{2.1}$$

An ephemeris can also be represented by a Cartesian state vector $\vec{x_c}$, defined as the concatenation of the RSO's position and velocity vectors as follows:

$$\vec{x_c} = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z}] \tag{2.2}$$

where $[x \quad y \quad z]$ represents the RSO's position vector and $[\dot{x} \quad \dot{y} \quad \dot{z}]$ represents the RSO's velocity vector in Cartesian coordinates.

2.3.2 Two Line Elements Sets

The Keplerian version of the ephemeris forms the fundamental background behind the NORAD TLE sets – measured data points of real-world RSOs that have been recorded in an extensive database for over 50 years [84]. TLE sets for unclassified RSOs are also publicly available from online sources such as Celestrak [85], providing access to a large RSO database that is frequently updated. These datasets provide a standardised method of encoding an RSO's Keplerian elements at an exact epoch, including the values of e, Ω, ω, and i; and any Keplerian information not directly recorded in the TLE set can be directly computed using basic equations from the literature [86].

Every TLE set is constructed from raw measured datasets after having undergone some minor corrections. In particular, specific periodic variations need to be removed from the data for simplicity. As a result, it is crucial that these same variations are properly reconstructed when utilising the TLE sets in practice. This is typically achieved by employing one of the well-known orbit propagation models as detailed in [87]. A common example of this is the SGP4 propagation model [88].

The format of any TLE set comprises two lines of encoded elements for an Earth-orbiting RSO at a specific epoch. This represents a standardised format used by both NORAD and NASA, allowing for all the relevant orbital data of a space object to be summarised within 70 columns in each of the two lines. Using the information encoded in the TLE dataset allows for a complete orbital trajectory to be generated for the associated RSO.

A description of each of the two lines of a TLE set is presented in Table 2.2

and Table 2.3 respectively.

Table 2.2: Format description for the first line of a TLE set

Column	Description
01	Line number of TLE set
03-07	Satellite number
08	Classification, with 'U' = Unclassified
10-11	International designator – last two digits of launch year
12-14	International designator – launch number of the year
15-17	International designator – piece of the launch
19-20	Epoch year – last two digits of year
21-32	Epoch – day of the year and fractional portion of the day
34-43	First time derivative of the mean motion
45-52	Second time derivative of mean motion; assume decimal point
54-61	BSTAR (B^*) drag term; assume decimal point
63	Ephemeris type
65-68	Element number
69	Checksum – modulo 10

This introduces multiple new quantities – including the drag term, B^*, which is computed as:

$$B^* = 0.16538 \times 10^{-3} R_e^{-1} \tag{2.3}$$

where R_e represents units of inverse Earth radii when using the World Geodetic System 1984 (WGS84) standard [89]. This term is used in the computation of the RSO's orbital decay due to atmospheric drag, which is usually accounted for in orbital propagation models such as SGP4 [90].

Table 2.3 also introduces the mean anomaly, M, and the mean motion, n, which are related to the other Keplerians as shown in Equations 2.4 and 2.5.

Table 2.3: Format description for the second line of a TLE set

Column	Description
01	Line number of TLE set
03-07	Satellite number
09-16	Inclination angle, i [degrees]
18-25	Right Ascension of the Ascending Node, Ω [degrees]
27-33	Eccentricity, e; assume decimal point
35-42	Argument of perigee, ω [degrees]
44-51	Mean anomaly, M [degrees]
53-63	Mean motion, n [revolutions per day]
64-68	Revolution number at epoch [revolutions]
69	Checksum – modulo 10

$$M = E - e \sin E \tag{2.4}$$

$$n = \frac{t_{day}}{2\pi}\sqrt{\frac{G_{grav}(M_e + m)}{(1000a)^3}} = t_{day}\sqrt{\frac{G_{grav}(M_e + m)}{4\pi^2(1000a)^3}} \tag{2.5}$$

where t_{day} represents the amount of time in a single day, G_{grav} is the gravitational constant, and M_e and m are the masses of the orbiting bodies – namely the Earth and the RSO, respectively. Note that t_{day} is typically measured in generic time units – assumed to be equivalent to that of the time unit used to measure the orbital period.

A description of each of the two lines of a TLE set is provided in [91]. All relevant TLE information can easily be transformed into a standard Keplerian ephemeris needed to define an RSO's orbit, which could then be converted into a Cartesian state vector [86]. This ephemeris serves as a vital quantity in target modelling, as it can easily be placed into the same simulated Cartesian coordinate system as other objects (such as antennas within the

radar simulator). The use of TLE sets is thus of great significance to the software developed in this project.

It should be noted that the use of TLE sets may result in semi-major axis position errors [92] that could accumulate to approximately 1 to 3 km per day, resulting in medium-fidelity target trajectories [93]. These errors are caused by deviations in the ideal orbits of the TLE sets from more realistic perturbed orbits, and they are thus unavoidable. Additionally, TLE sets also make use of averaged values derived after the removal of periodic variations using very specialised methods [87], resulting in an even larger degree of estimation. However, prediction accuracy may be improved by using the most recent version of a TLE set – as opposed to propagating an older version over longer periods [90].

An example of a typical TLE set is presented as follows:

Listing 2.1 A sample TLE set

```
1: 1 25544U 98067A   04236.56031392  .00020137  00000-0  16538-3
       0  9993
2: 2 25544  51.6335 344.7760 0007976 126.2523 325.9359
      15.70406856328906
```

This information can be decomposed into its Keplerians, its epoch, and other information using the descriptions provided in Tables 2.2 and 2.3. Quite notably, an RSO's epoch is denoted by the format **yyyy-mm-ddThh:mm:ssZ**, as used in the ISO 8601 standard [94]. The year of recording is represented by **yyyy**, while **mm** shows the month and **dd** represents the day. The date and time are then separated by **T**, with **hh** for the hour, **mm** for the minute, **ss** for the second, and **Z** to depict a time resolution of 1 s in Coordinated Universal Time.

Many existing space-monitoring radar systems thus make use of the TLE catalogue as a means of predicting the orbits of specific spaceborne objects. One application of this is to predict when an observation target would cross a radar's beam, as was done at the European Incoherent SCATter (EISCAT) facility at Troms Φ [95]. This system was used to compare measured object data against the theoretical propagation prediction from a TLE dataset, as

reflected in Figure 2.7.

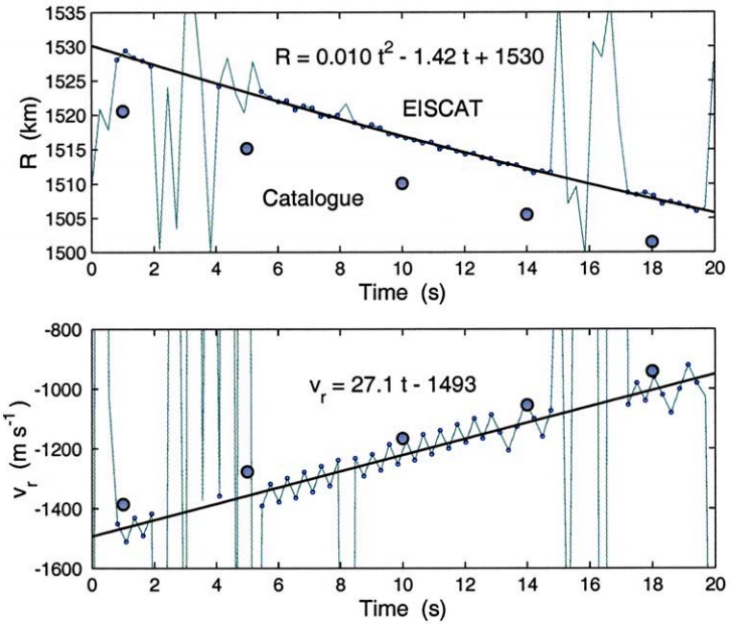


Figure 2.7: Comparison between measured data and TLE predictions for catalogued object 1994-11G on 20 February 2001 at 22:19:06; the top panel depicts the measured slant range (tiny circles), a parabolic curve fit (thick line), and the TLE prediction (filled circles); the bottom panel depicts the measured Doppler velocity (tiny circles), a linear fit (thick line), and the range rate TLE prediction (filled circles) [95]

This depicts the accuracy of TLE predictions against real-world measured data from the EISCAT radar for a catalogued object's slant range and Doppler velocity. Also presented are the parabolic and linear curve fits (respectively) for the measured data – along with their equations – to demonstrate the reliability of using TLE sets for forecasting. The use of TLE sets is thus of great significance to the software developed in this project, forming the basis for the measured datasets used to simulate RSOs and their orbital trajectories.

2.3.3 Orbit Propagation

The orbital information of space objects is often stored in the form of TLE sets or similar constructs. These datasets need to be interpreted by an orbital

propagator such that the RSO's Keplerian elements can be decoded from the data; using these elements, the object can then be propagated along its orbit around the Earth. Common examples of orbital propagation models include:

1. **Runge-Kutta**: a set of numerical integrators designed to integrate differential equations of the form $\dot{x} = f(x, t)$ [96]
2. **Nyström-Lear**: a set of numerical integrators designed to integrate differential equations of the form $\ddot{x} = f(\dot{x}, x, t)$ [96]
3. **Positions and Partials as functions of Time (PPT3)**: an analytical propagator that predicts the impact of perturbations on an RSO's near-Earth orbit due to the effects of the J_2, J_3, J_4, and J_5 zonal harmonics [97]
4. **SGP4**: an analytical propagator that predicts the effect of perturbations on a space object's orbit due to the Earth's shape, drag, and radiation, as well as due to gravitational forces from the sun and the moon [87]

The first two of these models only become classified as orbital propagators when they are combined with an environment model and a formulation of differential equations – as numerical propagators need to integrate sets of differential equations to predict an RSO's state vector at some arbitrary point in the future. Three such formulations are briefly described as follows [96, 98]:

- **Cowell's formulation**: directly integrates equations of motion in a rectangular coordinate system
- **Encke's formulation**: integrates the difference in accelerations between a perturbed orbit and an osculating orbit, then adds them to the osculating state vector to produce the perturbed state vector
- **Equinoctial elements**: a set of orbital elements used to remove singularities that are present in some of Gauss's equations when the inclination i or eccentricity e approaches zero

In a study by Shuster [96], these formulations were used with fourth-order Runge-Kutta and Nyström-Lear integrators to assess their performance relative to the standard SGP4 model. This is because most analytical planning

for satellite vulnerability in the United States is conducted using either PPT3 or SGP4 for propagation [90] – particularly SGP4, as it offers a compromise between speed and trajectory accuracy. The study aimed to compare the performance and resulting position errors between SGP4 and numerical integrators, the latter of which tend to provide reduced position uncertainty at the cost of significantly increased processing time.

As adapted from [96], Figure 2.8 compares the computation times of fourth-order Runge-Kutta and Nyström-Lear integrators relative to the SGP4 model, expressed as a function of position accuracy after 7 days. Results for the differential equation formulations (namely Cowell's, Encke's, and the use of equinoctial elements) are denoted by the colors specified in the legend. SGP4 results are represented by the green lines. The solid lines then indicate Runge-Kutta integration while the dashed lines represent Nyström-Lear integration, each of which is represented by its fourth-order variation. Results for both LEO (left) and GEO (right) are illustrated.

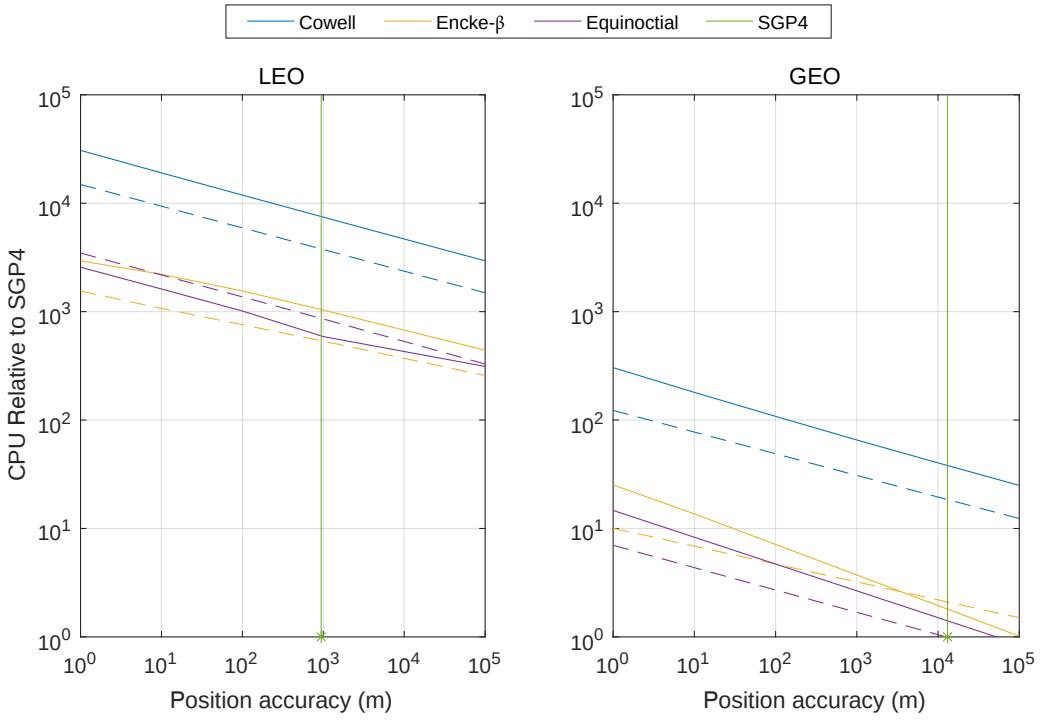


Figure 2.8: The computation times of various propagators relative to SGP4 as a function of 7-day position accuracy

The results in Figure 2.8 demonstrate the speed of SGP4 propagation over numerical integration methods. In LEO, SGP4 is shown to operate over 500 times faster than both the Runge-Kutta and Nyström-Lear integrators, while producing the same position accuracy. However, in GEO, it is shown that the use of equinoctial elements with the fourth-order Nyström-Lear integrator requires the time equivalent of only one SGP4 run to reach the highest SGP4 position accuracy. This implies that GEO propagation is best handled by Nyström-Lear numerical integration when using equinoctial elements, while SGP4 is well suited to propagating objects in LEO.

In terms of SGP4's performance against PPT3, studies have found that both algorithms produce similar degrees of error in orbital decay predictions [90]; one study also concluded that both SGP4 and PPT3 are easily capable of propagating orbits with sufficient accuracy and performance for most use cases; however, the study also showed that multiple assumptions had to be made in the calibration of the incomplete PPT3 model used in the study, leading to a greater level of uncertainty in its real-world performance [90]. This renders the SGP4 software model more complete and reliable in its accuracy than PPT3, and thus it was deemed to be better suited for this work's needs.

The reported measurements also agreed with the expected error values recorded by Vallado [99] and Schumacher [100] – namely the accumulative position error of 1-3 km per day for SGP4. From Wildt's [90] work, it was found that these propagation errors were acceptable for general planning, and Vallado [99] drew similar conclusions, stating that such errors are insignificant when rapid results are required for determining satellite visibility (using ground-based sensors).

Another important consideration for propagation is the effect of orbital decay, as was briefly mentioned earlier. This is described as the natural process of decay that an RSO experiences in orbit [101], relating to an RSO's lifetime – the time between its initial launch to the time at which it re-enters the Earth's atmosphere. This is determined almost entirely by its interactions with the atmosphere due to drag forces, which causes reductions in the velocity of an RSO and decreases its altitude until such time that orbital instability results;

at this point, the re-entry process is irreversible, and the object will either crash down onto the surface of the Earth or burn up in the atmosphere – depending on its mass and cross-sectional area [92].

It is possible for parts of de-orbited RSOs to survive the re-entry process, posing a threat to human beings, animals, and the environment. As a result, it is useful to be able to predict their time of impact. These predictions depend upon the initial orbital parameters of said object, as well as the ratio of the object's mass to its cross-sectional area. Upper atmospheric density, as well as the effects of the space environment on this density, should also be accounted for. However, even with a complete atmospheric model, it is often impossible to ascertain an exact description of an orbit's decay due to unavoidable prediction uncertainties. Studies have indicated that an RSO's predicted lifetime will typically have an error margin of approximately 10% – regardless of its actual orbital lifetime [101].

The effects of this are depicted in Figure 2.9 (adapted from [101]), which shows how the elliptical orbit of a typical RSO in LEO can change due to orbital decay over an extended period.

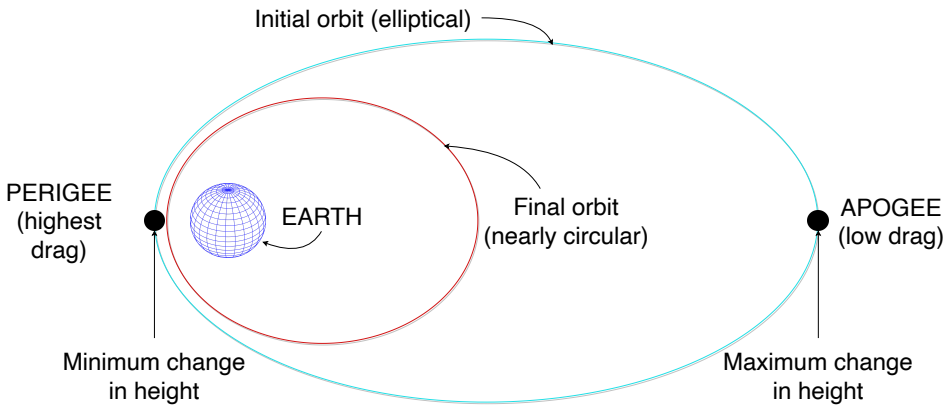


Figure 2.9: Diagrammatic view of how the elliptical orbit of an RSO in LEO changes due to orbital decay over time

As shown in Figure 2.9, the drag at the perigee causes a change in the height of the apogee. This illustrates how an elliptically-shaped orbit can become nearly circular in shape (see Table 2.1) due to the effects of atmospheric drag, which is defined most strongly at the perigee. It is thus important for orbit

propagation schemes to adequately account for the effects of orbital decay, even with the noted 10% prediction error margin.

2.3.4 Reference Frames

In astronomy, it is important to distinguish between different coordinate systems – particularly when considering both terrestrial systems and objects in space. Reference frames are thus used to make the appropriate distinctions, with each frame representing a uniquely defined coordinate system.

It is well-documented that the SGP4 propagation model generates RSO ephemerides in a reference frame called the True Equator, Mean Equinox (TEME) frame [99]. This is a special kind of Earth-Centered Inertial (ECI) frame – a geocentric equatorial coordinate system that does not rotate with the surface of the Earth as it spins about its own axis. A typical ECI frame is depicted in Figure 2.10 (adapted from [102]), showing its x-axis on the equatorial plane pointing from the Earth to the sun at the vernal equinox; the z-axis then extends through true north and the y-axis points perpendicularly to both the x- and z-axes, complying with the right-hand rule [103].

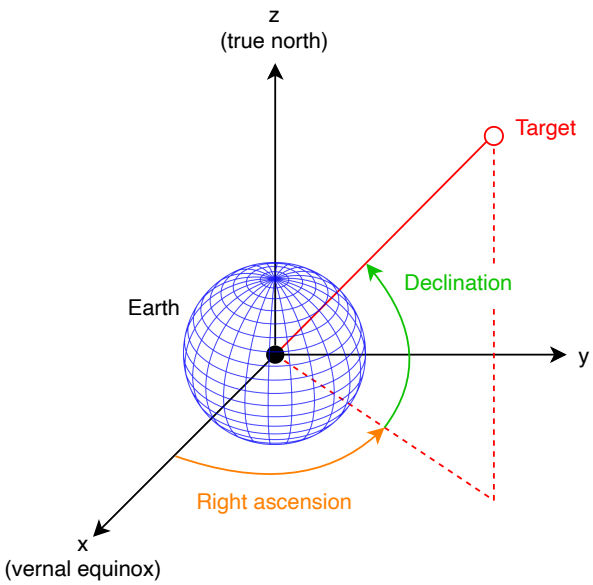


Figure 2.10: An ECI reference frame relative to a target in space

This frame (and by extension, the TEME frame) is thus useful for representing

objects in space, as the frame does not rotate with the Earth's surface and instead remains fixed. However, the ECI frame presents significant problems when also representing terrestrial objects – such as ground-based radars. This is because, in simulation schemes such as FERS, a radar's coordinates should ideally be fixed relative to other objects, such that target ranges and Doppler shifts can be reliably predicted and computed.

Terrestrial objects should thus ideally be represented in the Earth-Centered, Earth-Fixed (ECEF) frame, which remains fixed with the Earth's surface during the planet's rotation. Figure 2.11 depicts the relationship between a typical ECI frame and the ECEF frame, as adapted from [104].

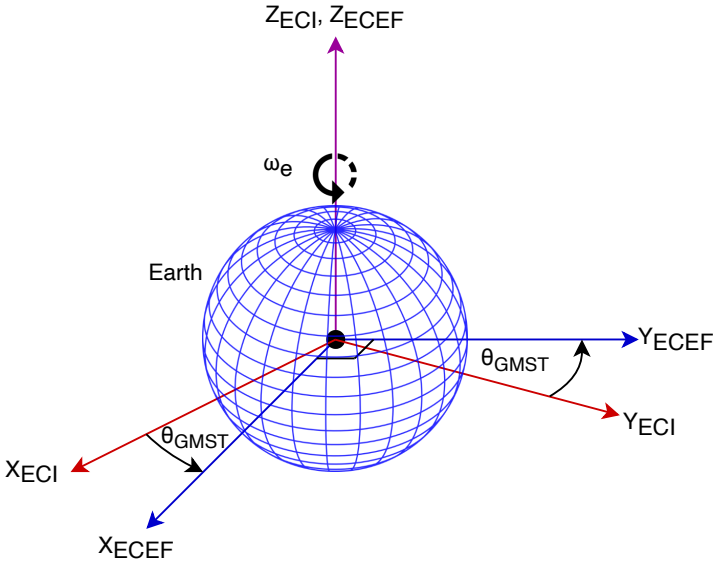


Figure 2.11: Illustration of the relationship between the ECI and ECEF reference frames

This depicts two new quantities, namely the rotation angle θ_{GMST} and the Earth's rate of rotation around its own axis ω_e. The former is defined as the Greenwich Mean Sidereal Time (GMST) angle, which is derived from the Julian date at the recorded epoch of the RSO. This thus equates to the angle through which the ECEF frame is rotated about the z-axis, and is measured from the initial x-axis direction in the ECI frame (as shown in Figure 2.11). These quantities thus play a pivotal role in the transformation from ECI to ECEF (and vice-versa).

An RSO's position vector in the ECI frame, denoted $[x \quad y \quad z]_{ECI}$, can be transformed into the ECEF reference frame using a simple 3-D rotation matrix ROT_3 through a rotation angle of θ_{GMST}. This is shown in Equation 2.6.

$$[x \quad y \quad z]_{ECI} = ROT_3(\theta_{GMST})[x \quad y \quad z]_{ECEF} \tag{2.6}$$

The ECEF frame would thus serve as the most useful reference referencing system for use in SOARS, as all TEME coordinates could be converted directly into ECEF coordinates following SGP4 processing.

2.4 Radar Simulation

This section discusses some of the most vital aspects of radar system simulation for space studies, including the importance of global surveillance, the proposed MeerKAT radar design by Agaba [13], and radar simulation tools.

2.4.1 Global Space Surveillance

No single space surveillance sensor is currently able to monitor all space objects due to inherent limitations in observation angles, field of view, and obstructions. Improved surveillance coverage therefore requires a global approach – one that is spread throughout the Northern and Southern Hemispheres. This is reflected in Figure 2.12, which depicts the distribution of various SSN sensors around the world as of 2018 – including both optical and radar systems.

As illustrated, there is a distinct lack of space-oriented sensors stationed in the Southern Hemisphere, giving rise to a need for more space-monitoring instruments to be constructed below the equator. With the in-progress development of the SKA, the proposed radar system by Agaba in [13] (designed in 2017) is one possible sensor that could alleviate this problem; in theory, this type of system could provide the desired global surveillance required to cover a larger breadth of observation angles. These types of systems could thus serve to complement the ODPO's space object database quite well and being

Figure 2.12: Worldwide SSN distribution of optical and radar sensors for a variety of use cases in space object studies [105]

able to extensively test such designs through simulation would be significantly useful.

Agaba's [13] proposed radar for space surveillance employed a bistatic design using a high-power transmitter and the MeerKAT radio telescope as its receiver, which operates at a centre operating frequency of 1.35 GHz. This "MeerKAT radar" was proposed as a coherent radar system, which would allow the system to differentiate between small changes in target velocities – and thus also small phase changes. Both improvements are invaluable in the detection of small debris objects, as such targets tend to have low RCS values and produce very small return voltages. For the sake of simulation accuracy, it is thus pertinent that changes in both the amplitude and the phase of a signal are considered by FERS/SOARS – and that both the in-phase (real) and quadrature (imaginary) components of the received signal are recorded.

It is assumed that such a system would operate in beam-park mode, whereby the transmitter and receiver(s) are stationary and their beams would be directed towards fixed points in the sky. Additionally, because the distance between any two of the 64 MeerKAT antennas is very small relative to the bistatic baseline (and the range to a spaceborne target), it can be approximated that the Rx antennas are all co-located. This means that the 64

MeerKAT antennas could be approximated as one combined receiver positioned at the nominal array centre.

It is through this approximation that the proposed MeerKAT radar can be assumed to operate in a bistatic configuration (as opposed to multistatic). This design, therefore, serves as a strong example of a typical use-case for SOARS, and how SOARS should be designed to be capable of designing and simulating similar systems in monostatic, bistatic or multistatic configurations.

It is expected that such a system would achieve excellent performance for SST purposes, namely because [4]:

- a radio telescope can operate as an interferometric array, enabling imaging of RSO trajectories with high angular resolution
- each of the radio telescope's antennas could be used to observe a particular target trajectory, yielding an improved signal-to-noise ratio at the receiver
- a receiver with low input noise temperature is invaluable in the detection of targets in long-range experiments (such as through Constant False Alarm Rate (CFAR) detectors [106])

These capabilities suggest that such a system would be amongst the most sensitive contributors to achieving global SSA. The high gain and low noise temperature of a radio telescope makes such a system well suited for the detection, tracking and imaging of space objects. Based on the lack of space-oriented sensors in the Southern Hemisphere, systems such as the proposed MeerKAT radar could prove to be a valuable step toward global space surveillance.

2.4.2 Simulation Software

For radar systems to be designed and tested accurately, the SOARS program needs to be properly defined and its use case needs to be clarified. As such, its radar simulator classification needs to be identified before development. Generally, there are four main classifications for radar simulation schemes:

1. **Result simulators** produce radar results without actually modelling the operation of a radar system; instead, they only producing the expected results from a given scenario, such as the simulation of how a given target would appear to a Synthetic Aperture Radar (SAR) system [25]. Result simulators thus tend to run extremely quickly due to the limited processing that they require. Examples of result radar simulators are detailed in [107] and [108].

2. **Electromagnetic simulators** simulate the properties of an electromagnetic field at discrete points, typically emphasising the electromagnetic behaviour of antennas and targets. Generally, processing "full-wave" electromagnetic simulations require powerful computational resources not widely available on commodity hardware; hence, large radar scenes are quite challenging to simulate electromagnetically [109]. An example of such a simulator is detailed in [110], which describes a radio wave simulation algorithm based on electromagnetic computations.

3. **Statistical simulators** generate statistical models of some signal parameters – such as coverage maps and detection probabilities – based on the performance of a radar in a particular environment. One such radar simulator was developed by Boothe [111] in 1964.

4. **Signal-level simulators** are used to simulate the raw return signal(s) that would be received by a radar system, as computed using the system's parameters and the surrounding environment of propagation. Two such simulators were developed by Golda [112] and Lengenfelder [113] at the University of Cape Town in 1997 and 1998 respectively. The most notable signal-level simulator since then is FERS [25, 114] – an open-source software that can simulate various radar configurations and scenarios. Signal-level simulators have previously been used for various applications, including pulsed systems testing [113], SAR techniques [115], modelling individual systems such as the MARSIS radar [116], and in systems design to simulate radar performance at the signal level [13]. These simulators are also useful in research and training, allowing users to explore the radar design space with minimal constraints and demonstrate the impacts of various parameters on received signals.

With these broad definitions in mind, it is desired that SOARS should be classified as a *signal-level* simulator. This is because the project aims for SOARS to simulate an unlimited number of targets and their effects on the return signal observed at a radar receiver, i.e., the output signal after passing through an analogue-to-digital converter. Based on this, FERS [25] is a viable modern candidate for use as a software base for this work and it could be extended for space monitoring purposes specifically.

This is reflected in Agaba's work [13], which used FERS as part of the design and testing process for the MeerKAT radar. The software was used to output sample received signals from the proposed radar design, and the signals were then post-processed so as to assess performance of the system both qualitatively and quantitatively. An example of this is shown in Figure 2.13, which depicts a Doppler-delay map generated from FERS output data.

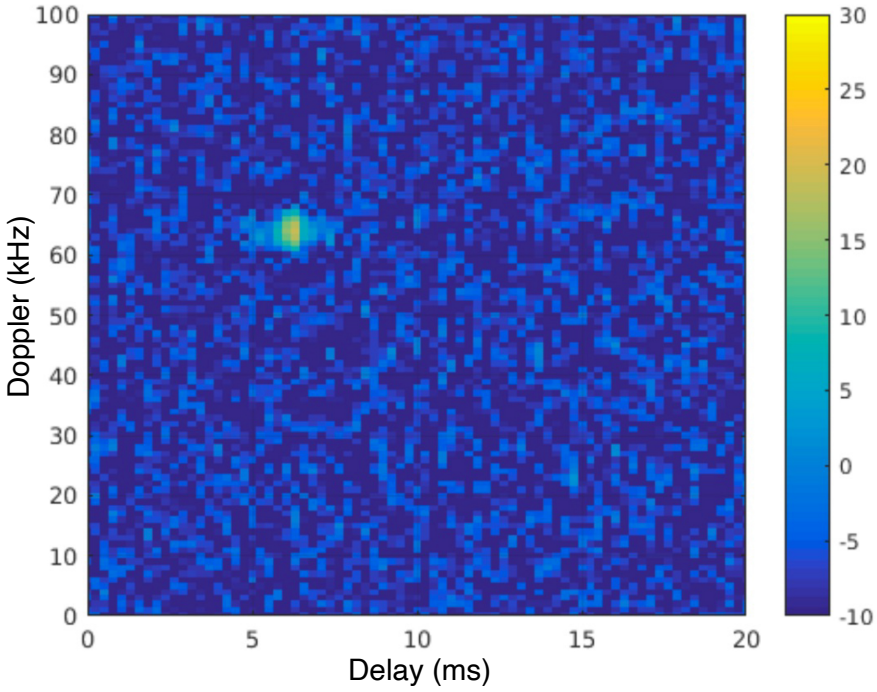


Figure 2.13: Doppler-delay map generated for a non-coherent output signal from FERS, showing the Fengyun 1-C debris object at the expected delay and Doppler frequency as observed by the MeerKAT radar

However, despite FERS being used as part of the design process, the software

had significant limitations within the context of space monitoring, and thus the idea behind this work was developed: to address the deficiencies of FERS by implementing space-oriented models and ray tracing, resulting in a more suitable tool for use in radar design for space applications.

Similar work was done by McCall [26] in 2013, which focused on the simulation of space debris objects in LEO; however, this emphasised the use of thermal modelling in characterising space debris objects via infrared signals, whereas SOARS aims to specifically consider signal-level simulations involving space debris as the targets. Other related work is documented in [117] and [27], where the former presented a radar processing back-end for real-time space debris detection; the latter presented a radar simulator for tracking space debris, but this made use of a tracklet approach to specifically gauge debris population sizes, whereas this work focused on signal-level radar simulation for generating the output signals observed at receivers.

2.4.3 Radar Computation

As part of developing SOARS as a signal-level simulator, this subsection serves to document various equations to be used as part of the radar computations performed in the program. This includes models for signal power, Doppler shifts, and more.

Received Power

In simulation, it is important to account for the loss in a signal's power as it propagates in a medium or reflects off a surface. The original FERS [25] program accounts for the basic propagation attenuation that occurs between a radar and a target as inherently computed by the multistatic radar equation for Q receivers and J transmitters. This is reflected in Equation 2.7 [118,119]:

$$P_{R_q} = \frac{P_{T_j} G_{T_j} G_{R_q} \lambda^2 \sigma_{RCS}}{(4\pi)^3 R_{T_j}^2 R_{R_q}^2}, q = 1, 2, ..., Q; j = 1, 2, ..., J \tag{2.7}$$

where P_R and P_T are the powers of the received and transmitted signals respectively (measured in W), G_R and G_T are the respective gains of the

receive and transmit antennas, σ_{RCS} is the RCS of the target (measured in m^2), λ is the wavelength of the radar, and R_R and R_T represent the respective ranges of the receive and transmit antennas from the target.

In the case of a monostatic radar, this is simplified to:

$$P_R = \frac{P_{T_j} G_T^2 \lambda^2 \sigma_{RCS}}{(4\pi)^3 R_T^4 L_T^2} \tag{2.8}$$

since the transmitter and receiver are co-located and thus $G_R = G_T$, $L_R = L_T$, and $R_R = R_T$. These equations can be used to compute the overall power of the received power after attenuation, RCS, and other factors have been accounted for.

Time and Phase Delays

Another crucial measured quantity is the time delay τ, which records the delay in receiving the observed signal after the initial transmission of the corresponding pulse. This is computed as follows:

$$\tau = \frac{R_T + R_R}{c} \tag{2.9}$$

where c is the speed of light, taken as 299,792,458 m/s. This accounts for the round-trip time measured from when a signal is transmitted to when the same signal is captured at a receiver after being re-radiated by a target. In narrowband cases, this also corresponds to the group delay and is related to the carrier phase delay through Equation 2.10 [25]:

$$\theta_d = -(2\pi\tau f_c) \bmod (2\pi) \tag{2.10}$$

where f_c is the carrier frequency.

Regarding time delays, the concept of dead time also needs to be considered; this plays an important role in monostatic radar configurations, where the single antenna will toggle between its transmitting and receiving mode pe-

riodically. After transmitting a pulse, the antenna switches to its receive mode and waits for return echoes for a specified period, and the process repeats when the next PRI begins. It is thus pertinent that the echoes are not received while the antenna is still in transmit mode.

In the case of FERS and SOARS, this is avoided using a "skip" or "dead time" period – a short interval specified as a fraction of the PRI that allows the radar to transmit a pulse and then safely switch to receive mode. In doing so, the simulator avoids immediate, potentially dangerous feedback that could be picked up at the antenna while still in transmit mode. Antennas in FERS and SOARS will therefore operate in transmit mode for the specified dead time, and thereafter receive mode will activate for a chosen "capture" period – usually taken as the remainder of the PRI.

Doppler Shifts

One final computation to consider is that of the Doppler shift, f_d. This is defined as the difference between the transmitted and received frequencies due to relative motion between the radar and the target, and is calculated as:

$$f_d = f_r - f_c \tag{2.11}$$

where f_r is the received frequency. Using the theory of special relativity, this received frequency can be predicted as [106]:

$$f_r = f_c \left(\frac{1 + \frac{V_r}{c}}{1 - \frac{V_r}{c}} \right) \tag{2.12}$$

where V_r is the radial component of the target velocity, i.e., the component of the target's velocity moving directly along the radar's initial line-of-sight vector. The denominator in Equation 2.12 can then be expanded as a binomial series such that:

$$\begin{aligned} f_r &= f_c \left(1 + \frac{V_r}{c}\right) \left(1 - \frac{V_r}{c}\right)^{-1} \\ &= f_c \left(1 + \frac{V_r}{c}\right) \left[1 + \frac{V_r}{c} + \left(\frac{V_r}{c}\right)^2 + ...\right] \\ &= f_c \left[1 + 2\left(\frac{V_r}{c}\right) + 2\left(\frac{V_r}{c}\right)^2 + ...\right] \end{aligned} \tag{2.13}$$

It is assumed that $|V_r| \ll c$, and hence most of the terms in Equation 2.13 are assumed to be very close to zero. If all second-order and higher terms in (V_r/c) are thus discarded, the following formulation results:

$$f_r = f_c \left(1 + 2\left(\frac{V_r}{c}\right)\right) \tag{2.14}$$

Combining Equations 2.11 and 2.14 then generates the well-known formulation relating V_r and f_d:

$$\begin{aligned} f_d &= f_c \left(\frac{2V_r}{c}\right) \\ &= \frac{2V_r}{\lambda} \end{aligned} \tag{2.15}$$

This equation could technically be used to compute Doppler shifts for monostatic *and* bistatic configurations – as long as the radial velocity V_r is calculated to be the same for both the transmitter-to-target and receiver-to-target lines of sight. To confirm this, it is important to understand the geometry of a bistatic radar configuration, as reflected in Figure 2.14 (adapted from [120]).

where f_d corresponds to the Doppler shift, β is the bisecting angle between the R_T and R_R vectors, L is the bistatic baseline measurement, and δ is the angle between the bistatic bisector and the target's velocity vector V. Note that only V and δ do not lie on the bistatic plane as the velocity vector can be pointed in any direction in 3-D space.

Using this geometry, the bistatic Doppler shift can be related to the velocity

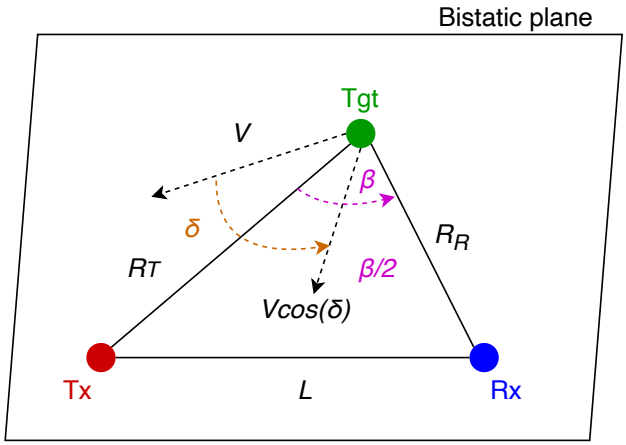


Figure 2.14: Bistatic radar geometry showing the bistatic plane, a transmitter **Tx**, receiver **Rx**, and target **Tgt**

of a target V (as projected onto the bistatic plane). Ignoring relativistic effects, this is achieved via Equation 2.16 [121].

$$\begin{aligned} f_d &= \frac{1}{\lambda} V \left(\cos\left(\delta - \frac{\beta}{2}\right) + \cos\left(\delta + \frac{\beta}{2}\right) \right) \\ &= \frac{2}{\lambda} V \cos\left(\delta\right) \cos\left(\frac{\beta}{2}\right) \end{aligned} \tag{2.16}$$

As shown, Equations 2.15 and 2.16 represent the same relationship between f_d and V_r, but in the bistatic scenario, $V_r = V\cos(\delta)\cos(\beta/2)$. This is thus also applicable for the monostatic case, where $\beta = 0°$ and thus $V_r = V\cos(\delta)$. This provides two separate approaches to computing the Doppler shift: either using a combination of Equations 2.11 and 2.12, or using Equation 2.16. Technically, the former would be more accurate since it accounts for all high-order terms in the binomial expansion and should thus be implemented in SOARS for improved accuracy.

It should be noted that, in a three-dimensional space, a target velocity vector V_{tgt} (computed as the change in target coordinates over time) must first be projected onto the bistatic plane as vector V. Only after this should V be projected onto the transmitter and receiver line-of-sight vectors. It is also worth noting that V, as well as V_r, should be used as a scalar in the above

equations, i.e., the vector's direction is not used in the computation of the Doppler frequency.

2.5 Ray Tracing

This section discusses some of the most salient points surrounding the topics of ray tracing for radio wave simulation. The concept of ray tracing and some of its possible implementations are presented along with an overview of GPU acceleration using CUDA on compatible devices.

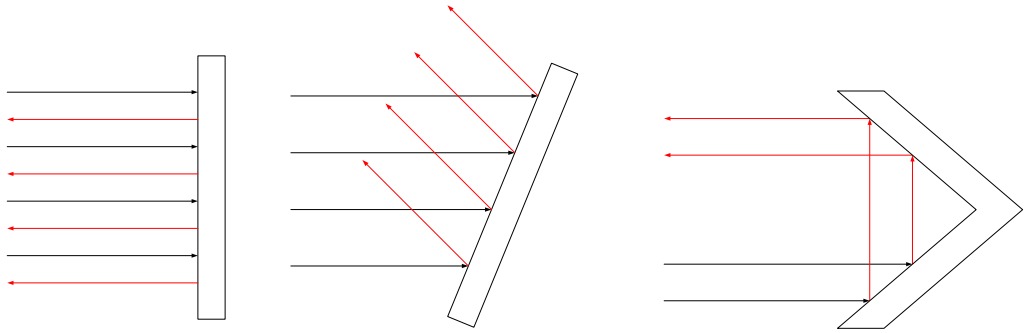
2.5.1 Geometric Optics

Ray tracing is a concept used widely in the electromagnetics [122] and computer graphics fields [123,124]. As a concept, it is based on the approximation of a wave as a set of rays – each of which is launched from a source and then traced as it travels through a simulated environment and intersects target objects under test. As a model, ray tracing is based on the idea of geometric optics solving Maxwell's equations at high frequencies.

It is defined as a propagation modelling tool for electromagnetic analysis that "provides estimates of path loss, angle of arrival/departure, and time delays" [32]. When modelling radio wave propagation using ray tracing, a radar signal is represented as a set of rays – each of which is assumed to travel in a straight line in a homogeneous medium. The rays then carry signal energy as they propagate and the ray properties are varied as they interact with objects in the simulation environment.

Most notably, when a ray collides with an object's surface, a partial reflection results as it bounces off the surface. These are known as "specular reflections" and often result in the propagating ray losing power and changing direction, as determined by the incident angle and the reflection coefficient of the surface with which the ray had collided. Various examples of such reflections are depicted in Figure 2.15.

A ray-object intersection can also result in a *refracted* ray being launched at the point of collision, where the refracted ray propagates *through* the physical



(a) 90° reflection (b) $\neq 90°$ reflection (c) 90° retro reflector

Figure 2.15: Reflected rays (red) resulting from incident rays (black) bouncing off surfaces of various orientations [125]

object with reduced power; the rest of the power is then either lost as heat energy on the surface or is transferred into the partially-reflected ray bouncing off the object. The directions of both the reflected ray and the refracted ray (assuming they both exist) can be computed using Snell's Law [126] using the geometry illustrated in Figure 2.16 (adapted from [127]).

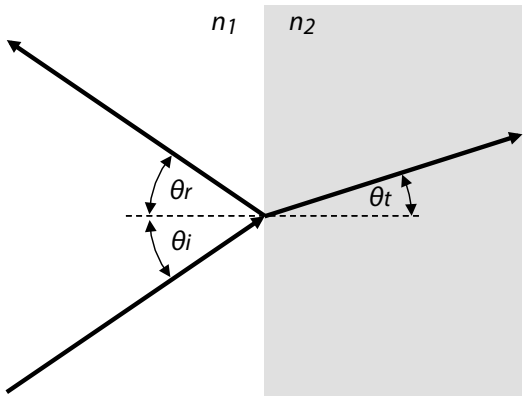


Figure 2.16: Representation of an incident and refracted wave at an interface between two mediums with refractive indices of n_1 and n_2 respectively

The illustration in Figure 2.16 depicts the situation where a ray intersects an object with a different medium to the medium of propagation, where n_2 and n_1 are the refractive indices of the two mediums respectively. In this example, θ_i represents the incident angle, θ_r represents the reflection angle, and θ_t is the transmit angle – which determines the direction of the refracted ray. The

formulation of Snell's Law is thus as follows:

$$n_1 \sin(\theta_i) = n_2 \sin(\theta_t) \tag{2.17}$$

This is useful in computing the transmit angle of a refracted ray, provided that the incidence angle and the mediums' refractive indices are known. Note that Snell's Law can also be used in computing the reflected ray's direction since $\theta_i = \theta_r$.

Other ray considerations include attenuation (or absorption), diffraction, and diffuse scattering – which results when a ray intersects a non-uniform surface and produces multiple diffuse reflections. However, of these three considerations, only attenuation is to be accounted for within this work. This accounts for the loss in a signal's power as it propagates in a medium or intersects a surface. FERS accounts for this via the basic propagation attenuation that occurs between a radar and a target (as computed using Equation 2.7) during signal traversal.

Diffraction and diffuse scattering, however, are considered to be outside the scope of this project, but they could be considered as possible future additions for the work. While it is possible for diffraction to have a significant effect in some cases, this work aimed to focus mainly on ray-traced reflections – similarly to FERS, which considered only direct reflection effects. However, refraction was also considered and implemented due to the existing built-in model in the ray-tracing engine that was selected for use (as discussed later in the thesis), while diffraction effects were chosen to be ignored based on the ray-tracing model for space debris described in [128].

2.5.2 Ray Algorithms

The main idea behind ray tracing is to compute the paths of rays from a source transmitter to a specific point – in this work, the point is representative of the location of a radar receiver. The most trivial case is when a single ray propagates directly from the source to the receiver along a straight line in free space, also known as direct transmission in a radar context. However,

this ignores the ultimate objective of introducing ray tracing into a radar simulator, which is to enable the modelling of targets in the environment; more specifically, it is desired that objects are modelled as physical objects rather than point targets (as used in FERS).

In this work, three main ray-tracing algorithms are considered:

1. **The image method**: This is a simple recursive algorithm used to account for ray reflections, but it is quite inefficient when there exist a large number of reflection surfaces. The image method works as shown in Figure 2.17 (as adapted from [32]), whereby an "image" of the receiver Rx is spawned – denoted Ri – relative to the planar reflection surface Σ. The transmitter Tx and the receiver image Ri are connected via a line segment that intersects Σ at point Q, and the reflected ray path is then determined by three points: Tx, Q, and Rx. This can easily be extended for multiple transmitters and receivers, and the current version of FERS offers support for a simple implementation of the image method for multipath propagation analysis.

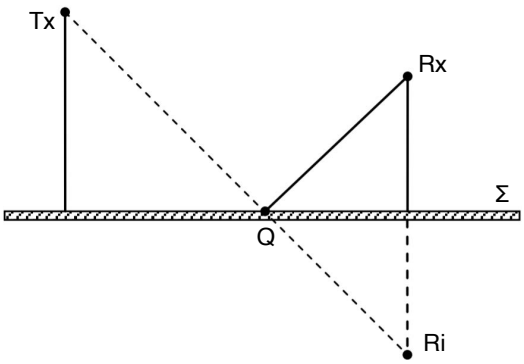


Figure 2.17: Illustration of the image method, where *Ri* is the image of *Rx* relative to the surface Σ

2. **Shooting and bouncing ray method** [32]: While it was initially used for computing the RCS of cavities in 1989 [129], this method has increased in popularity for modelling radio wave propagation. It operates on the idea of tracing rays from a transmitting source and determining *if* they arrive at a receiver. This requires that the transmit rays be uniformly distributed such that each ray carries similar power

for an isotropic source; each ray is then monitored to identify any intersections with other objects in the scene, as these intersections would spawn reflected rays. Finally, when a receiver is found to lie within a propagating ray's tube, it is concluded that the ray is received by that receiver [32].

3. **Bounding Volume Hierarchy (BVH) method** [130]: This approach makes use of a tree-based structure that stores a bounding volume at every node of the "tree". Each of these internal nodes may also have child nodes, and each leaf node may contain any number of geometric primitives, i.e., triangles that model an object. The outermost bounding box is guaranteed to enclose all other boxes, and every primitive is in exactly one "leaf". This approach is demonstrated using four triangles in Figure 2.18, as adapted from [131].

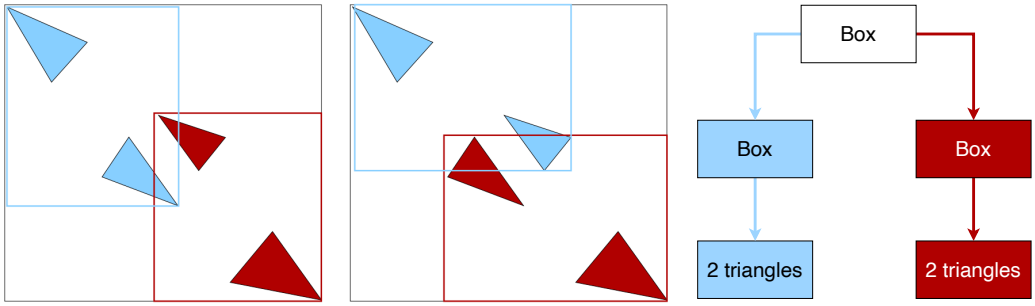


Figure 2.18: Two different BVH configurations for four triangles

The BVH method is applied in ray tracing via ray-triangle intersection testing, whereby checks are performed to determine whether a ray has hit a specific triangle or not [31]. BVH is used as part of the NVIDIA® OptiX™ ray-tracing engine [30] for testing ray intersections with a given target. OptiX™ provides a freely-available, programmable ray-tracing engine for supported NVIDIA® GPUs, and is accelerable and usable under various contexts – including radio propagation modelling [30, 34, 132].

In [37], it is demonstrated that OptiX™ provided greater accuracy and performance than another ray tracer that was being tested, known as Paray – a polarimetric parallel ray-tracing simulator developed at the

RWTH University Aachen [133]. The study concluded that OptiX™ "provides a good basis for the implementation of a very efficient polarimetric electromagnetical wave propagation simulator" [37], as it was able to find more valid ray paths (and more quickly) under tested conditions.

It is thus feasible that one of these algorithms could be used to implement ray tracing within the context of space monitoring and signal-level radar simulation. Ideally, one of these methods would be adapted and integrated into FERS to account for the effects of reflection, refraction, and absorption within the context of SOARS. However, it is also important to consider how these methods were utilised in the existing literature.

Specifically, the work in [33, 132] detailed the development of an OptiX™-based, ray-traced simulation tool for use in medical imaging involving millimeter-wave systems. Other notable works in the field of ray-traced radar simulation include [32, 34–42, 134] – all of which demonstrated the use of various ray-tracing algorithms for different purposes, such as RCS approximation, electric field modelling, indoor beam tracing, and more. However, most of these works model electromagnetic and target radar simulators, typically emphasising the electrical behaviours of objects in relation to radar simulation.

It is also possible for electromagnetic radar simulators to generate the simulated radar returns from targets [135, 136], using techniques such as ray tracing to output the baseband return signal observed at a simulated radar receiver. While these bare some resemblance to signal-level simulators, they are often denoted as "target simulators" instead, whereby the focus is on emulating the electromagnetic behaviours of observation targets so as to model aspects such as micro-Doppler signatures [43].

While the work in [135] was closely related to the research conducted in this thesis, it specifically pertained to an electromagnetic simulator (with signal rendering capabilities) as applied in the context of traffic surveying. Unlike signal-level simulators, such target simulators also tend to disregard the surrounding environment of propagation (including external interference) and ignore vital aspects of the signal rendering process, such as oversampling,

phase noise, clock drift, and delay filtering. This work also aimed to focus on applications in the space environment, with systems operating at vastly different centre frequencies and a significantly different approach to ray tracing using triangular primitives to model targets (whereas the simulator in [135] uses arrays of point scatterers).

At present, it can thus be seen that there exist no 3-D, ray-traced, signal-level radar simulators available for public use – particularly one that is oriented towards space monitoring and accounts for aspects such as noise modelling, optics-based computation, and customisable RCS and antenna patterns. This represents a gap in the literature that this thesis aims to address, and further study was conducted to assess the trade-offs between ray tracing (as a radio propagation model) and point-model approximations for radar simulation (within the context of this work).

While using ray tracing generally provides various advantages and disadvantages over alternative methods, one major benefit of the model is that every ray is independent of all others; as such, each ray path can be computed individually. This makes the model highly conducive to parallelism. Ray-tracing algorithms have thus demonstrated both high performance and throughput on GPU hardware [137] due to the hardware's specialised design for large tasks split across thousands of independent threads. This makes GPUs very well suited for ray tracing.

However, the divergence of rays due to *branching* introduces many computational complexities and an increase in software runtime. This branching effect creates some doubt in the notion that ray tracing is "embarrassingly parallel" [138], as each ray can give rise to many more (independent) rays through the effects of reflection and refraction. Accordingly, to maximise the performance of a ray tracing engine, the acceleration structure being used needs to be carefully programmed for specific use cases [139, 140], and efficient stack and thread management must be well coordinated for recursion purposes [132]. The recommendation is thus that an existing, well-documented ray-tracing algorithm should be used as part of SOARS.

2.5.3 GPU Acceleration

One major benefit of ray tracing is that the path traced by each simulated ray can be computed independently, making it well suited for GPU acceleration with CUDA [29]. This represents a classification of parallel computing through which multiple computations are carried out simultaneously to solve a single problem [141, 142]. Many specialised HPC facilities have thus been built for this purpose – particularly for large-scale astronomy applications, such as those conducted by the SKA [143]. This makes HPC methods well suited to this project's needs, especially with the use of GPUs becoming increasingly commonplace in HPC [144].

To take full advantage of GPU architectures, NVIDIA®'s CUDA model [145] was developed for use in the C, C++, and Fortran programming languages [28]. On a heterogeneous CPU-GPU (host-device) system using CUDA, the CPU can allocate memory on the GPU, transfer data between host and device memory [146], and launch parallel program segments (known as *kernels*) on the GPU – after which the outputs are copied from GPU memory to CPU memory for further processing.

Each kernel can be executed on multiple CUDA threads, which together make up a "block"; a group of blocks is then referred to as a "grid", and each block has its own shared memory that its threads can access [28]. This concept is illustrated in Figure 2.19, depicting the host and device as well as two CUDA kernels being launched.

In this work, all HPC hardware needs were accommodated by the CHPC – the largest HPC facility in Africa. The CHPC is managed by the Council for Scientific and Industrial Research, serving as one of the core pillars of national cyber-infrastructure intervention in South Africa [44].

2.6 Conclusions

In developing SOARS, it was important to understand the properties of space objects and the radar systems designed to monitor them; as a result, both of these aspects have been explicitly highlighted in this chapter. This literature

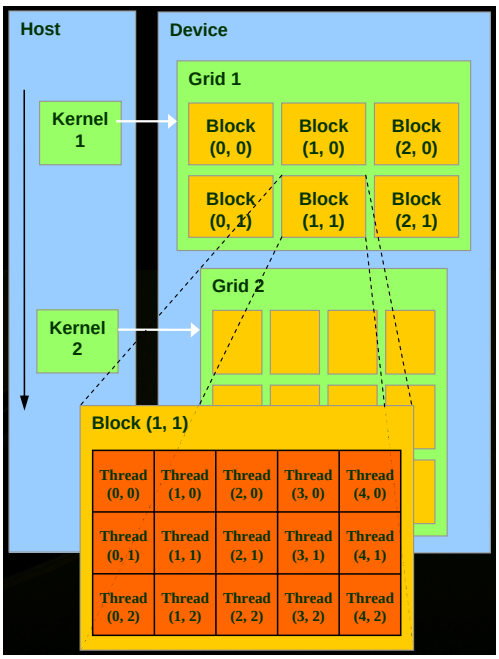


Figure 2.19: Overview of the CUDA programming model and the interactions between CPU (host) and GPU (device) [29]

review provided an overview of the history of the space debris population and its growth over time, as well as some of the defining characteristics of debris objects. The importance of SSA has also been highlighted, including a summary of the potentially disastrous impacts of debris population growth if countermeasures are not put into place. A summary of classical Keplerian elements was then presented, including a discussion on the use of TLE sets and SGP4 orbit propagation.

This was followed by a description of some of the most vital aspects to be considered in the design of a radar system for space debris monitoring, as well as an overview of radar simulation classifications. Various ray-tracing properties and algorithms were also presented along with a discussion on the use of GPU acceleration for enabling extensive ray-traced radar simulations on a signal level.

Additionally, this chapter explored recent works from the literature and identified areas upon which the rest of this thesis could build. This included critical reviews of some of the most relevant literature relating to radar space

surveillance, as well as recent developments in radar simulation using ray tracing. As a result, this literature survey concluded that the development of a signal-level, ray-traced radar simulator specialised towards space monitoring would represent a novel contribution to the field. This literature review has thus directly contributed to the main objectives of this project, with the following chapters aiming to detail the more specific design and development aspects of SOARS.

Chapter 3

Simulator Design

This chapter presents the overall project and research methodology as well as the main design criteria and considerations in developing the baseline SOARS software – which also serves as the basis for the ray-traced simulator described later in the thesis. This includes a discussion on the approach taken in carrying out the work and some of the fundamental background theory used in the software's design. The choice of programming language for SOARS is also explained, along with comparisons between the designs of SOARS and FERS. Additionally, the implementation of RCS models, CUDA, SGP4, a new file parser, and external noise sources into the existing simulator source code is explained and justified. Much of the work covered in this chapter was published in [48].

3.1 Methodology

The first step in designing the simulator was to gain an understanding of the most salient concepts surrounding the topic. This involved reviewing the relevant literature associated with space debris, as well as the theory behind radar signal processing and the use of GPUs for large-scale scientific problems; this resulted in the literature survey presented in Chapter 2. The next step was thus to develop a design plan and define a methodology through which the software design could be approached.

It is thus important to determine the desired operation of the SOARS program and how it could be used to address the problem statement described in the previous chapters. It is intended for SOARS to operate using a single simulation file as the main program input to define all relevant simulation properties; after processing, the program should then simply output the raw signal(s) computed at each simulated receiver. This is in keeping with the signal-level radar simulator classification, allowing users to post-process the output signal(s) in a variety of ways.

Based on the knowledge accumulated from the literature review, a small-scale pulse-Doppler radar simulator was initially developed in MATLAB® [147] as a proof of concept. This provided an understanding of the basic radar system concepts and signal processing methods that would be used in the final software and resulted in a simple simulation scheme capable of simulating multiple targets in 3-D space – with each target supporting both rotational and translational motion. The software could also be operated using either a monostatic or bistatic radar configuration.

Developing this simulator provided a strong background to the work, which made designing the full program significantly more convenient and straightforward; however, the limitations of using MATLAB® for this type of large-scale simulator – namely in terms of speed and scale – gave rise to the need for a more computationally-capable simulator. This is when alternative, more complete radar simulators were considered – namely independent pieces of software that could be appropriately modified for the specialised purpose of rapid target processing for multitudes of space objects.

This eventually led to the final, fully-formed idea for the C++-based [148, 149] SOARS – the design of which will be expanded upon in the following sections of the thesis. Ultimately, it was decided that the software would be designed around the core idea of being a pulse-Doppler, signal-level radar simulator specialised for space debris monitoring.

A high-level overview of the intended structure of SOARS is presented in the block diagram in Figure 3.1.

This depicts the expected basic operation of the software, whereby a simulation

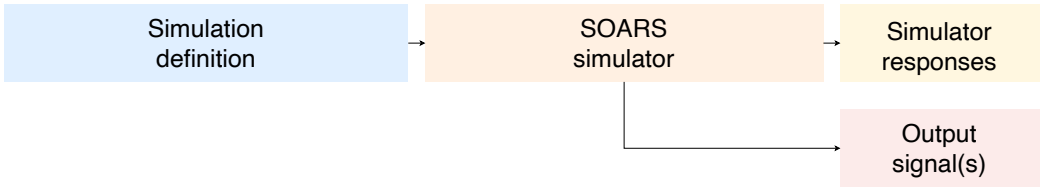


Figure 3.1: High-level block diagram of the intended SOARS software structure

definition file would serve as the single input to the simulator and summarise all the required system parameters, target properties, and more. This file would then be read into the simulator, which would process the simulation and generate the output results (including all receiver response information such as Doppler shifts and time delays) as well as the raw received signal yielded at each receiver.

It was intended for this approach to be simplified further by developing a simple Graphical User Interface (GUI) to be utilised alongside the SOARS software. This was done to improve the overall usability of the simulator and allow for a more user-friendly introduction to SOARS and its capabilities. An early rendition of this program was developed as part of the SOARS software and they are both expected to be released alongside each other as open-source programs.

Because of the limitations of running a large-scale software like SOARS on commodity hardware, GPU-based parallelism was introduced into parts of the source code through CUDA and NVIDIA®'s OptiX™ ray-tracing engine. This was expected to increase the simulator's computational potential, improve simulation modelling accuracy, and allow for larger numbers of complex space objects to be represented in a ray-traced radar scene simultaneously. The code was thus optimised, parallelised and benchmarked, and verification testing processes were performed to fully validate the simulator's implementation.

Ultimately, the project methodology was broken down into the following three phases:

- **Phase 1**: Investigate existing radar simulation schemes and orbit propagation techniques and use this knowledge to develop a signal-level radar simulator that is specialised for space object monitoring. This

"baseline" simulator – using point-model representations to simplify radar-target interactions – should represent a complete software that can be used to generate theoretical received signals for radar systems used to detect, track and/or image spaceborne objects such as debris and satellites. The program should account for various facets relating to environmental and internal system noise, radar theory, the orbital dynamics of objects under test, and antenna properties. The software should also be verified against established theory to ensure its accuracy.

- **Phase 2**: Research the concept of ray tracing and its application within the context of Phase 1, then consider possible implementations of such a model to improve simulator accuracy. The ray-tracing engine could be developed as a standalone module to be used for extending existing programs such as FERS, and simultaneously, the engine could be directly integrated into a second iteration of the baseline SOARS program developed in this work.
- **Phase 3**: Perform comparisons between the point-model and ray-traced simulators to assess their trade-offs and evaluate the usefulness of ray tracing in the context of space surveillance. Theoretically, the ray-traced software would act as the final version of the signal-level simulator and would enable more accurate representations of wave propagation and target models by accounting for geometric optics as well as object size, shape and material parameters. The program should also be appropriately tested and validated before being released as open-source software.

After Phase 3, the project would be complete. More specifically, however, the methodology could be separated into the following steps:

1. Explore and review the relevant literature associated with space junk, orbital modelling, radar simulation and HPC to better understand the space debris problem and its challenges. This was covered in Chapter 2.
2. Design the simulator through the selection of an existing radar simulation scheme to serve as a foundation; programming language(s) to use, hardware models to be used (and the associated properties and

simulation parameters), and noise models to be employed. The design of the software will primarily be progressed incrementally using the spiral development model as detailed in [150]. These aspects are covered later in the chapter.

3. Develop the core simulation software in code, combining elements of an existing radar simulator base, the orbit propagation scheme, noise considerations, and CUDA integration for GPU acceleration purposes. Each of these elements will need to be verified through independent unit tests against established theory, as well as by comparing the overall system's outputs to expected results. This is also covered later in the thesis.

4. Introduce a ray-tracing engine into the software to improve modelling accuracy and allow for targets and radar systems to be simulated with physical parameters such as volume and material properties. The developed ray-tracing module could serve as both a standalone software as well as a second iteration of SOARS that directly integrates the module into its programming. This is covered in significant detail in Chapter 4.

5. Validate the output results of the software against theoretical expectations as well as an existing, well-established radar simulator. While FERS [25] was originally validated against measured data, further verification tests need to be performed to cover the new additions to the overall software, as will be detailed later in this thesis. These aspects are covered in Chapter 5 as part of the final testing for the program, including the validation methodology in Subsection 5.3.1.

6. Benchmark the performance costs introduced by the integration of CUDA-based parallelism and ray tracing into the simulator. This also includes benchmarking any changes in computational speed due to the inclusion of a ray-tracing model, which should improve modelling accuracy at the cost of runtime. Tests should also be conducted to compare the baseline version of SOARS against the ray-traced version of the software. CUDA benchmarks (as well as ray-tracing benchmarks)

are covered in Chapter 5 as part of the final results for the completed simulator.

7. Evaluate the completed software against the functional and non-functional user requirements through valid evaluation criteria. Thereafter, draw conclusions based on the outputs of this project and the results of the developed software, and make recommendations for future work and possible directions for expansion of the simulator. This is covered in Chapter 6.

This thesis thus serves to document the development and testing of the completed SOARS software, as well as to identify the capabilities and limitations of the final build. The project will thus be concluded once these steps have been completed.

3.2 Software Requirements

As mentioned in the previous section, the developed SOARS program needs to be evaluated against a set of functional and non-functional user requirements before it could be released as open-source software. This section discusses these user requirements based on the expected use cases for the simulator.

3.2.1 Non-functional User Requirements

The desired non-functional requirements for the software are described as follows:

1. **Portability**: This work entails developing a ray-traced, signal-level radar simulator that is specialised for the purpose of monitoring simulated space objects – while also being unrestricted in terms of user compilation, editing, and execution. The simulator should therefore be accessible on a variety of operating systems and platform architectures. Additionally, if possible, an open-source programming language should be chosen to ensure that the software can be installed without any cost to the end-user. This should maximise the size of the program's user base and encourage further development and future improvement.

2. **Speed**: The developed simulator should be able to run efficiently on various hardware platforms. This entails developing the code to take advantage of the user's computing system such that simulations can be completed quickly and the system's throughput can be maximised, allowing for large-scale deployment of the simulator.

3. **Accuracy**: The simulated software should produce results which are reliable and accurate, making use of well-established models that are properly implemented, verified, and referenced. This will ensure that the user is aware of the models' uses and any associated uncertainties. Similarly, all required input parameters must be adequately detailed such that the user is aware of how to correctly interface with the software, maximising the accuracy of their inputs. The software outputs must also be extensively tested through verification processes to ensure that the results are realistic (relative to real-world expectations) and useful in practice.

4. **Usability**: The software should be effective in achieving its aims, and thus it needs to be as user-friendly as possible for both experienced users and newcomers (such as students or engineers in training). The software should make use of a clear interface that produces predictable outputs, and it should be well-documented to accommodate new users (see [151, 152]). In these ways, users would be more encouraged to use the program and results would be generated more reliably.

5. **Modularity**: SOARS should be easily modifiable for various use cases and should be developed in such a way that a user could easily implement additional features, introduce improvements, or make small adjustments to the code base. One example of this is the addition of ray-tracing which is discussed later in the thesis, which acts as an additional development module that was added to the core program and integrated accordingly. Additionally, the GUI should be easily customisable if a user wishes to make modifications or additions of their own. These facets should improve the future capabilities of the program and increase the likelihood of program expansion and improvement.

With these, the non-functional requirements of the program have been fully described.

3.2.2 Functional User Requirements

It is also crucial that the software is defined in terms of its functional requirements such that the main technical aspects of the work can be verified. The main functional requirements of the software are thus defined as follows:

1. **Measured datasets**: The software should be able to natively support measured datasets (from established sources) as inputs for target representations.
2. **Astrodynamics aspects**: The software should consider physical perturbations during orbit propagation, accounting for such effects as orbital decay and the Earth's shape and gravitational forces. Additionally, reference frames must be carefully selected and specified for the software such that transformations can be appropriately made between Keplerian and rectangular coordinate systems.
3. **Radar systems theory**: The software should account for all basic models required for radar simulation, including that of RCS, propagation loss, antenna modelling, system noise, pulsed signal properties, high-accuracy clock timings, and Doppler shifts.
4. **Noise sources**: The simulator should account for the effects of internal thermal system noise, galactic and sky noise, and other external sources of noise from natural or man-made phenomena in the simulation environment.
5. **Ray tracing**: The software should accurately model objects as targets specified by size, shape, and material properties – and model the effects of these aspects on a ray-traced return signal. Additionally, the propagation of simulated signals should be modelled using geometric optics as applied to travelling rays that interact with their environment and adapt to the properties of objects under test – thereby modelling radio wave propagation more realistically.

With these requirements, the main functional aims of the program have been defined. This covers the main technical aspects investigated as part of the literature survey in Chapter 2 and provides a fully formed idea of what the software should account for in its simulations.

3.3 Development Tools

This section details some of the core design possibilities and development tools to consider in developing the baseline SOARS software. This includes information on the choice of programming language, simulation base, external package requirements, and more.

3.3.1 Programming Languages

Before a radar simulation scheme can be developed, it is useful to consider various software tools and programming languages that would enable the development of such a specialised piece of software. This needs to account for various aspects discussed in Chapter 2, such as the decision to develop a signal-level radar simulator. A critical review of some of the more recent signal-level simulators was presented in Subsection 2.4.2, along with a description of how SOARS aims to differentiate itself from existing programs.

Based on the software requirements specified earlier in the chapter as well as the literature survey, three main programming languages were considered for use in this work, namely MATLAB®, Python [153], and C++ – primarily due to their extensive use in the radar field. Additionally, there are a plethora of relevant software libraries available in these languages which could greatly ease the development of SOARS and improve future maintenance. Specifically, however, each of these languages were considered for the reasons as follows:

- **MATLAB®**: offers an accessible user interface, widespread community support, and an easy-to-use programming language. The MATLAB® platform is also heavily used in radar signal processing and analysis, offering access to a built-in, point-model radar simulator known as the Phased Array System Toolbox (PAST) [154] — which could be

used for comparison testing. Despite this, however, MATLAB® is not available as open-source software, and the development of a GUI for a MATLAB® program is very restrictive due to the dependence upon its App Designer [155] software – two problems which defy the tenets upon which SOARS is to be designed.

- **Python**: satisfies the portability criterion for this project due to its open-source and cross-platform compatibilities; also a very accessible programming language. As of May 2020, Python also holds a programming market share of approximately 9.12% [156], easily making it one of the fastest-growing languages in the industry. The Python programming language also offers support for CUDA, and boasts widespread support from a large user base for external libraries and packages – including the well-known scientific libraries SciPy and NumPy [157].
- **C++**: a compiled programming language supported on a wide variety of operating systems (including Linux™, Microsoft® Windows® and macOS®) and computational architectures. C++ is also standardised, meaning that its code could be used in future projects without requiring specific compiler versions. Additionally, since C++ is a compiled language, it offers fast performance for most intensive computational tasks. It also offers CUDA support [158] and has a set of well-documented, efficient data structures and standard, built-in algorithms [159, 160].

This narrowed down the choice to either Python – a programming language that is interpreted from the source code at runtime and does not require compilation – or C++, which is a compiled programming language providing significant performance improvements over interpreted languages. Ultimately, both languages present certain advantages and disadvantages, and the choice was eventually made based on the radar simulator selected for use as part of the baseline program.

3.3.2 FERS Software

Based on the previous discussions on software support tools – as well as the extensive research that was done in investigating existing radar simulators –

the FERS software [25] was eventually selected as the simulator base to be used for SOARS. This decision was motivated by the following reasons:

1. FERS is widely used at the University of Cape Town for generic radar simulations, making it well suited for use in SOARS due to the extensive local support for the software.
2. Its source code was written in standardised C++ – one of the two programming languages selected for possible use in the development of SOARS. This also means that FERS would be natively compatible with CUDA.
3. The program makes partial use of the Python programming language for optional extensions, providing support for additional functionality. This enables some degree of modularity in the long term, allowing for further expansion of the software through Python if the need arises.
4. It is a signal-level simulator with support for pulse-Doppler simulation, which agrees with the desired operation of SOARS as a pulsed signal-level radar simulator.
5. The software can execute simulations comprised of multiple targets and multistatic radar systems. FERS also has built-in support for propagation loss, Doppler and phase shifts, pulse waveforms, clock timings, various antenna shapes, and more.
6. The FERS software repository, accessible at [161], provides some useful code for performing post-processing in MATLAB®. This has the potential for significant development advancements in the future, whereby MATLAB® development could be conducted in tandem with FERS and/or SOARS. MATLAB® thus serves as a useful software support tool for the FERS and SOARS programs.

The block diagram of the FERS software structure is presented in Figure 3.2, as adapted from [25].

This depicts FERS as a generalised radar simulator comprised of three core software elements – namely the environment model, the signal renderer, and extension modules [25]. The software requires an input file in a customised

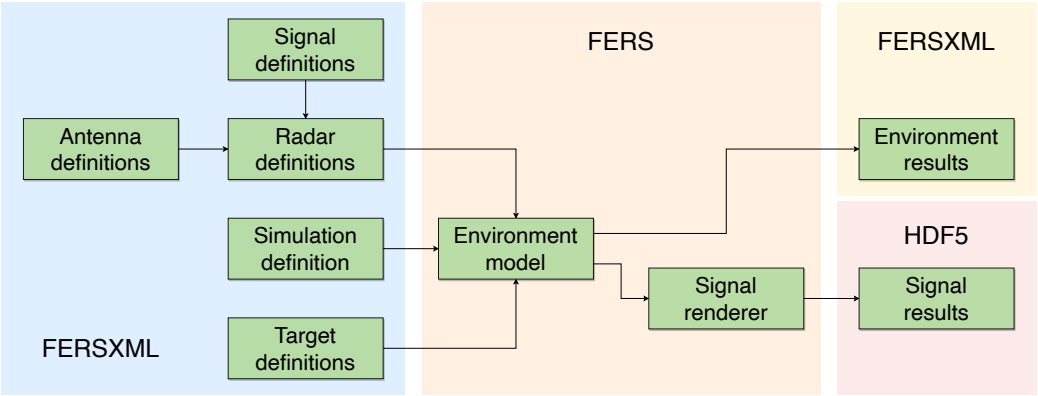


Figure 3.2: Block diagram showing the overall software structure and information flow of the FERS software

version of the Extensible Markup Language (XML) format – a *.fersxml* file – comprehensively describing the full simulation environment with all systems, signals and targets. These inputs are read into the software using an XML parser, stored in memory, and then passed to the environment model. In addition to the *.fersxml* input file, FERS requires the user to define each transmit signal in a separate file. This is typically also done in the XML format, and the signal file must contain the in-phase and quadrature data of the transmit pulse to be transmitted.

The environment model is responsible for processing the full radar simulation using all systems, targets, signals and antennas contained within an internal "world model" structure; thereafter, the signal renderer outputs the raw receiver return in the Hierarchical Data Format version 5 (HDF5) [162] file format – a highly efficient form of storage for various data types. Each HDF5 output file is generated with the raw signal samples for one simulated radar receiver, as stored in their original in-phase and quadrature form. Additional results from the environment model can also be (optionally) produced in separate *.fersxml* output files, documenting the receiver response data (such as Doppler and phase measurements) for every receiving antenna in the simulation.

In FERS, the environment model is expressed mathematically as:

$$E_{ij}(x) = P_{ij}D_{ij}(x) + \sum_{k=1}^{S} P_{ijk}D_{ijk}(x) \tag{3.1}$$

where S is the number of targets being simulated in the environment, P_{ijk} and P_{ij} are the propagation attenuations, and D_{ijk} and D_{ij} are the phase and Doppler frequency effects based on the range and relative motion of the j^{th} transmitter, the i^{th} receiver, and the k^{th} target.

In terms of signal rendering, FERS outputs the raw receiver return in the form of in-phase and quadrature samples. This can be simplified as the product of three main processes, each of which modify the transmitted signal in a specific order, as expressed by Equation 3.2.

$$y_i[n] = \sum_{n=0}^{N} H_i(E(H_t(x[n]))) \tag{3.2}$$

where i is the receiver being considered, n is the sample number, N is the signal length in samples, x is the discretised transmit signal, H_j and H_i correspond to the hardware model of the transmitter and receiver respectively, and E represents the environment model in Equation 3.1. This renders the simulation process equivalent to finding the functions H_j, E, and H_i, and then applying the effects of these functions to the transmitted signal x.

A more robust and comprehensive version of this implementation is presented in Equation 3.3.

$$y_i[n] = \sum_{j=0}^{J} \left(A_{ij}\frac{n}{f_s}d_{ij}[n] + \sum_{k=0}^{S} A_{ijk}\frac{n}{f_s}d_{ijk}[n] \right) \tag{3.3}$$

where f_s is the sampling rate, J is the number of transmitters in the simulation, and A_{ij} and d_{ij} represent the amplitude function and delayed samples for the direct path from a transmitter j to a receiver i, respectively. The quantities A_{ijk} and d_{ijk} correspond to the amplitude function and delayed samples for the path measured from j to target k to i, respectively. This provides a robust

method for computing the samples of the raw radar return signal at receiver i. Further detail on these equations is provided in Brooker's works on FERS [25, 114].

The radar configuration can be tuned as required by the user, including the system's geometry, sampling frequency, transmit power, carrier frequency, and more. A radar's transmitter and receiver can also be co-located in a monostatic configuration if the user so desires, or they can be separated by placing each antenna independently of one another at different ranges. This enables FERS to handle monostatic, bistatic, and multistatic radar scenarios seamlessly.

Post-processing can be executed on any HDF5 signal data file produced by FERS and can be used to generate useful results. However, this needs to be done independently of FERS, and all post-processing must be conducted externally using software package such as Python or MATLAB®. Additionally, the input *.fersxml* file must be defined either manually or using a software such as MATLAB®. This means that all inputs and outputs of FERS must be processed completely independently of the main program.

It should be noted that FERS lacks several features required for this project – namely, it does not natively support TLE datasets, offers no methods for orbital propagation, and does not model the effects of external noise sources. Primarily, this means that FERS uses a naïve target model for orbiting objects relative to what TLEs and SGP4 can offer, as targets can only follow linear waypoints dictated by the user and no orbital forces are modelled to act upon them. And without the addition of environmental interference, the simulator can provide unrealistic results in scenarios involving low-power returns (such as in the detection of spaceborne targets).

The effects of ray tracing are also not accounted for in FERS – which instead uses a point-model propagation scheme – and the program lacks a dedicated GUI, rendering it lacking in both user support and accessibility. However, these deficiencies are intended to be addressed as part of the development of SOARS in this work. Because FERS lacks support for ray tracing and external noise modelling, these aspects are intended to form part of the larger

additions to the software. It is envisioned that the SOARS software will thus provide a more modernised and modular version of the FERS software within the context of space monitoring.

3.3.3 External Dependencies

Before discussing the design and implementation of the simulator, it is important to highlight the external dependencies (i.e., software libraries) that will be required to compile and run the software. Many of these have been chosen due to their native use in FERS, while additional packages have been selected for implementation in SOARS for the reasons discussed as follows:

- **HDF5**: The HDF5 library [162, 163] enables the reading and writing of the highly-efficient HDF5 data storage format. Support for the package is offered across a variety of programming languages, but the C++ version is used as part of the FERS source code to store output return signals in their unaltered in-phase and quadrature forms.
- **Boost**: A widely-used, formally-reviewed library [160] providing a set of efficient C++ functions for use alongside the native libraries [164]. The Boost library also provides a form of multi-threading that is used to parallelise some FERS operations across CPU cores [25].
- **TinyXML/PugiXML**: FERS makes use of the TinyXML library [165] to read the input simulation script stored in an XML file, which describes the full radar system configurations, targets, and the environment to be simulated. This library is also used to write the XML outputs of FERS, which include the responses (such as Doppler and signal power measurements) of each simulated receiver at various time steps. However, as detailed later in Subsection 3.5.4, SOARS will replace this library with PugiXML [166] – a more streamlined and modernised equivalent to TinyXML offering significant performance gains.
- **FFTW3**: This library [167] provides access to an implementation of the discrete Fourier transform in C++ code, one which adapts efficiently to the hardware to maximise computational performance.

- **Python**: Enables the use of the Python programming language for some optional functionalities and extensions within FERS.
- **CMake**: The CMake tool [168] is an open-source, cross-platform package that enables FERS to be compiled regardless of the C++ compiler used. This is also used to link the core source code to all required external libraries as well as set specific compilation flags. Additionally, CMake is used with the NVCC compiler [169] (developed by NVIDIA®) to compile CUDA code in *.cu* (source) and *.cuh* (header) files.
- **SGP4**: An analytical orbit propagation algorithm that predicts the effect of perturbations on an RSO's orbit due to the Earth's shape, drag, and radiation, as well as other effects [87]. This library represents a standard C++ implementation of the SGP4 model as developed by Vallado *et al.* [99].
- **NVIDIA® CUDA** [29]: A model that facilitates communication between host and device, allowing for subroutines to be parallelised across a large number of GPU threads, blocks, or grids to maximise throughput for large data-parallel tasks.
- **NVIDIA® OptiX™** [30] As will be detailed in Chapter 4, this represents a freely-available, programmable ray-tracing engine that can be compiled and run on supported NVIDIA® GPUs. The program is accelerable and usable for various applications and will be used as part of SOARS to implement radar signal ray tracing.

These dependencies are all freely available as open-source software and are widely used in C++ development, and while the SOARS software was mainly tested on Linux-based systems, it should be executable on nearly any operating system due to the programming language's cross-platform compatibility. And with these dependencies defined, the development tools have been clearly described. The next section aims to discuss the actual simulation aspects of SOARS and how various elements are modelled for the baseline implementation of the simulator.

3.4 Simulation and Modelling

This section presents the software structure of SOARS based on the original design of FERS, as well as a comprehensive breakdown of various physical models – such as the transmitter and receiver hardware models – implemented in the code. Each model is described with a discussion of the input parameters required for simulation.

3.4.1 Program Structure

Based on the main requirements for the software, as well as the foundation described in Figure 3.1, the block diagram for the SOARS program structure is presented in Figure 3.3.

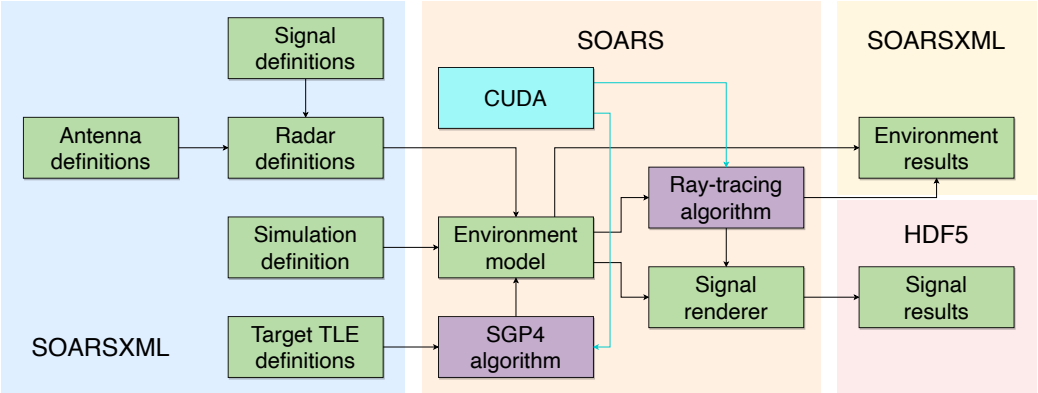


Figure 3.3: Block diagram showing the software structure and information flow of the baseline SOARS software

The block diagram shown in Figure 3.3 is largely based on the structure of FERS, as was shown in Figure 3.2. In particular, the environment model and signal render module are left largely unchanged from FERS, including their implementations shown in Equations 3.1 and 3.3. However, there are three notable differences between the two simulators' structures, namely:

1. the use of customised *.soarsxml* files in SOARS instead of *.fersxml* files, with the writing of output XML files being optional to the end-user
2. the addition of a ray-tracing algorithm to be used for signal transmission,

propagation and reception in SOARS, which directly affects the output(s) of the software

3. the addition of an SGP4 implementation for propagating target objects along their orbits in SOARS

4. the integration of CUDA-based parallelism to accelerate large, independent workloads in SOARS (where applicable)

Each of these elements contributes to the main objective of the SOARS software, and as such, they play a vital role in its development. These individual aspects will be discussed in greater detail later in the thesis.

3.4.2 Orbital Modelling

As discussed in Section 2.3, this project aims to emphasise the simulation of space debris in LEO. Based on this context and the results found in the literature, it was concluded that the orbital propagation of such RSOs would be best performed using the SGP4 model – developed specifically for decaying LEO-based RSOs based on their TLE sets [170] with adequate accuracy in most scenarios.

This is because the SPG4 model perfectly reconstructs the periodic variations that are initially removed from every TLE set [87], and also accounts for the effects of several natural impacts on a space object's orbit – such as perturbations resulting from the Earth's shape, drag, and radiation, as well as due to gravitational forces from the sun and the moon [87]. Additionally, SGP4 uses a drag model to account for the effects of orbital decay experienced by RSOs in orbit, and also factors in the Earth's oblateness [90].

As such, the choice of SGP4 should ensure that prediction accuracy is maximised when using TLE datasets. This decision was also impacted by the results shown previously in Figure 2.8, which demonstrated the performance gain achieved by SGP4 methods over numerical integration methods. In LEO, in particular, SGP4 was shown to operate over 500 times faster than both the Runge-Kutta and Nyström-Lear integrators – despite the same position accuracy being achieved across all models. It was concluded that the

Nyström-Lear numerical integrator (used with equinoctial elements) was best suited for GEO propagation, while SGP4 was best suited for propagating objects in LEO [96].

As will be discussed later in the chapter, the SGP4 model will be implemented directly into the SOARS software using a standard C++ implementation originally developed by Vallado *et al.* [171]. This makes use of the World Geodetic System 1984 standard [89] as the Earth's gravity model – as is widely used in global positioning systems [4]. Recent versions of SGP4 have also integrated the effects of Simplified Deep Space Perturbations #4 modelling [97], accounting for both near-Earth and deep space propagation [96]. Technically, SGP4 could thus be used for RSOs beyond just LEO, but this work aims to focus namely on LEO-based objects.

Both θ_{GMST} and ω_e, as introduced in Figure 2.11, are also to be accounted for in SOARS. The conversion from TEME coordinates to ECEF coordinates should thus follow seamlessly for any simulated target. This means that any radar system coordinates, as defined in the simulation inputs, should be interpreted as ECEF coordinates. Conversely, all target positions are to be derived from their TLE sets, which would then be propagated along their respective orbits in the TEME coordinate system; thereafter, the coordinates are to be transformed to the ECEF frame. Thus, when SOARS executes its simulation, all simulated elements will make use of the same ECEF coordinate system.

All systems defined in SOARS – namely the simulated antennas which are mapped to transmitters, receivers, or monostatic radars – should be added to the simulation environment with their positions specified in spherical coordinates in the ECEF reference frame. This particular coordinate system is chosen for convenience as the shape of the Earth can easily be accounted for. The spherical coordinate system used in this work is defined as illustrated in Figure 3.4.

Most notably, the angle ϕ is defined as the *elevation* and is measured from the xy-plane towards the positive z-axis along the plane defined by the angle θ. This angle θ is the *azimuth*, measured anticlockwise from the positive x-axis

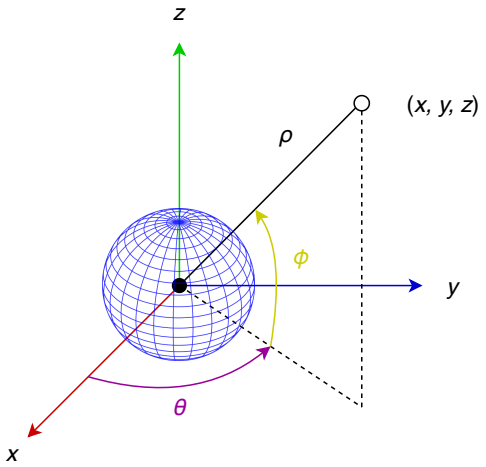


Figure 3.4: The standard definition for spherical coordinates in radar system analysis

along the xy-plane towards the target position. Finally, the Euclidean norm ρ is defined as the distance from the origin to the centre of the target under test. This represents the standard, conventional form of spherical coordinates used in radar analysis, but it is important to specify the definitions of each angle as they may vary in other contexts.

3.4.3 Radar System Modelling

Modelling the behaviours of both transmitters and receivers in SOARS requires extensive models for the radar system hardware being simulated as well as the effects that these have on the generated signals. These models also need to have strong flexibility in their designs due to the range and diversity of radar hardware available. As SOARS is based on the original design of FERS, the main aspects of their hardware models are shared and both simulators operate at baseband.

The transmitter hardware model used in SOARS is comprised of four main modules, namely the transmitter definition, the transmitter position (in spherical coordinates), the power amplifier, and the transmit antenna. This is based on the hardware design used for transmitters in FERS, which is reflected in Figure 3.5 (adapted from [25]).

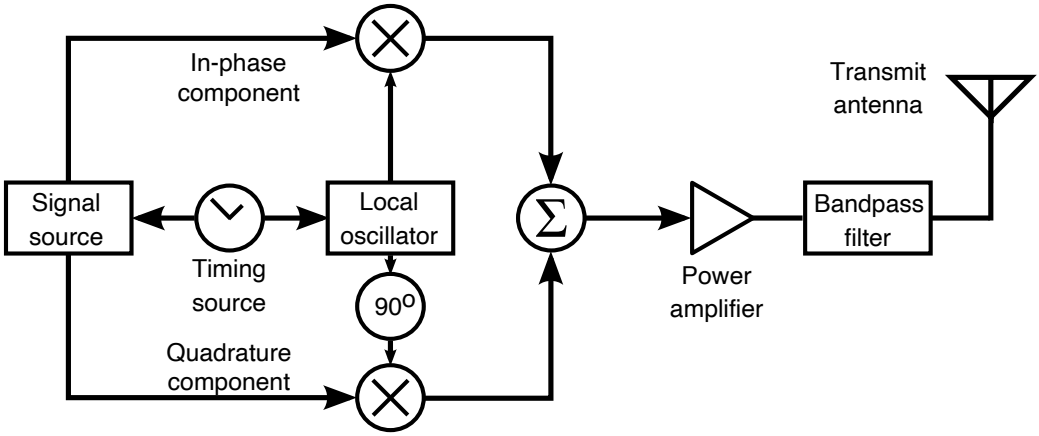


Figure 3.5: Block diagram for the transmitter hardware model used in FERS and SOARS

As shown in Figure 3.5, the FERS transmitter makes use of a signal source, quadrature up-mixer, timing source, power amplifier, bandpass filter, and antenna. The signal source is responsible for generating samples of the in-phase and quadrature parts of the complex envelope, which are then passed to the up-mixer for transmission at the specified transmit frequency with the specified Pulse Repetition Frequency (PRF). The timing source then controls the transmission schedule while the power amplifier is used to account for the gain of the transmitter. In FERS, the bandpass filter is not explicitly simulated, but it is included in the hardware model as an indication of the bandpass nature of the discrete-time simulation model employed in the program [25].

Finally, the transmit antenna is modelled by its azimuth and elevation angles, which together specify its boresight direction. The antenna may take the form of various shapes, including isotropic (ideal), parabolic, sinc, and Gaussian – and the required parameters to define the antenna are changed accordingly.

The transmitter hardware modules are summarised in Table 3.1, along with the associated parameters and required input units used in the simulation definition file.

All transmitters simulated in SOARS must be synchronised to a pulse signal, which is generally defined by a carrier frequency f_c, a bandwidth B, a transmit

Table 3.1: Key parameters of the transmitter model in SOARS

Module	**Parameter**	**Units**
Signal definition	Carrier frequency f_c	GHz
	Sampling frequency f_s	MHz
	PRF	kHz
	Duty cycle	–
Location	Spherical coordinate ρ	km
	Spherical coordinate θ	deg
	Spherical coordinate ϕ	deg
Power amplifier	Transmit power P_T	W
	System bandwidth B	MHz
Antenna definition	Antenna efficiency	–
	Boresight angle θ_B	deg
	Boresight angle ϕ_B	deg

power P_T, and a duty cycle. Similarly, each antenna must be defined separately with its own individual properties.

The units for the parameters shown here were specifically chosen based on the project's needs. As this work is only considering microwave frequencies, the carrier frequency f_c is specified in GHz; for convenience, f_s and B are measured in MHz while the PRF is specified in kHz. Distances are measured in km as the project is dealing with large distances between the Earth's surface and LEO. Watts are used as the standard unit for transmit power, and finally, degrees are used instead of radians for all angular definitions – as commonly used for specifying orbit orientations.

With respect to parameter limitations, SOARS (and FERS) impose no restrictions; the design goal behind both simulators was to allow the end-user freedom to experiment with the input parameters in any way they please, and thus notice the corresponding changes in the output signal. This served to complement the intended usage of SOARS in training or teaching environments, but it may also result in unrealistic results if the user is not careful. As such, in keeping with the software requirements, the software needs to be appropriately documented through a usability guide such that the user is aware of how to properly interface with the program and produce the best possible results for their purpose.

As for the receiver hardware model, it is comprised of five main modules, namely the receiver definition, its location, the low-noise amplifier, the antenna, and the receiving window (dead time parameters). This is based on the hardware design used for receivers in FERS, as reflected in Figure 3.6 (adapted from [25]).

As shown in Figure 3.6, the FERS receiver makes use of a low-noise amplifier, quadrature down-mixer, timing source, signal sink, and a receive antenna. The amplifier is used to represent the hardware found between the antenna and the down-mixer and is used to account for the receiver's gain as well as internal thermal noise generated in the system (detailed later in the chapter). The signal sink is then used to simulate the capturing of the return signal and the quantisation that occurs thereafter [25].

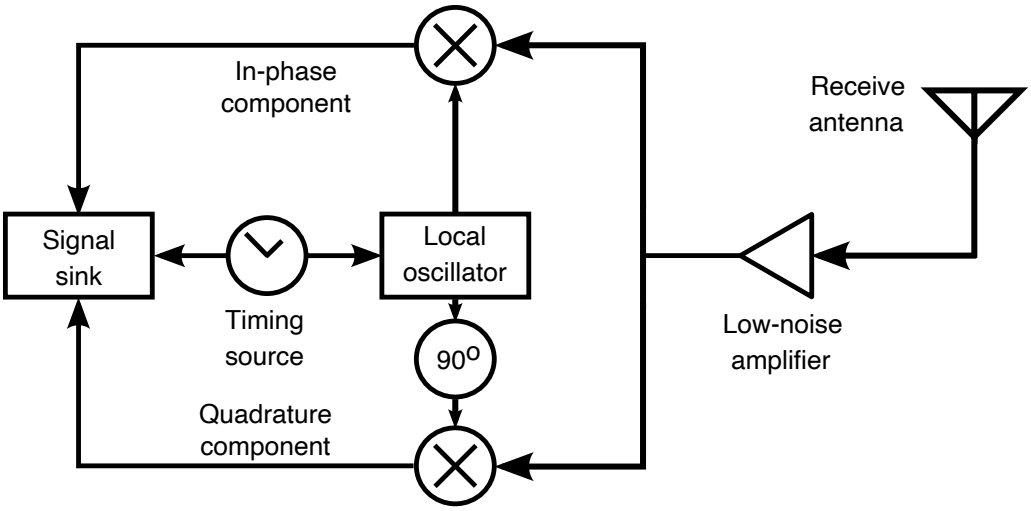


Figure 3.6: Block diagram for the receiver hardware model used in FERS and SOARS

The receiver hardware modules are summarised in Table 3.2, along with the associated parameters and required input units used in the simulation definition file.

Note that the receive window and skip lengths (specified in seconds) set the range gate in which to capture data within the Pulse Repetition Interval (PRI). This means that window length corresponds to the section of the return signal to keep within a single PRI, while the window skip accounts for the dead time just before the receiver is turned on; in this way, the simulator avoids immediate, potentially dangerous feedback that could be picked up at the receiver.

A monostatic hardware model could also be defined, but this would mainly comprise both the aforementioned transmitter and receiver hardware models – but with a single antenna being used for both transmit and receive purposes. As a result, the models described in this section fully describe the most notable physical models required by the baseline simulator.

With these models defined, the radar system model used in SOARS can be described by the class diagram shown in Figure 3.7.

This provides a simplified diagrammatic breakdown of some of the most

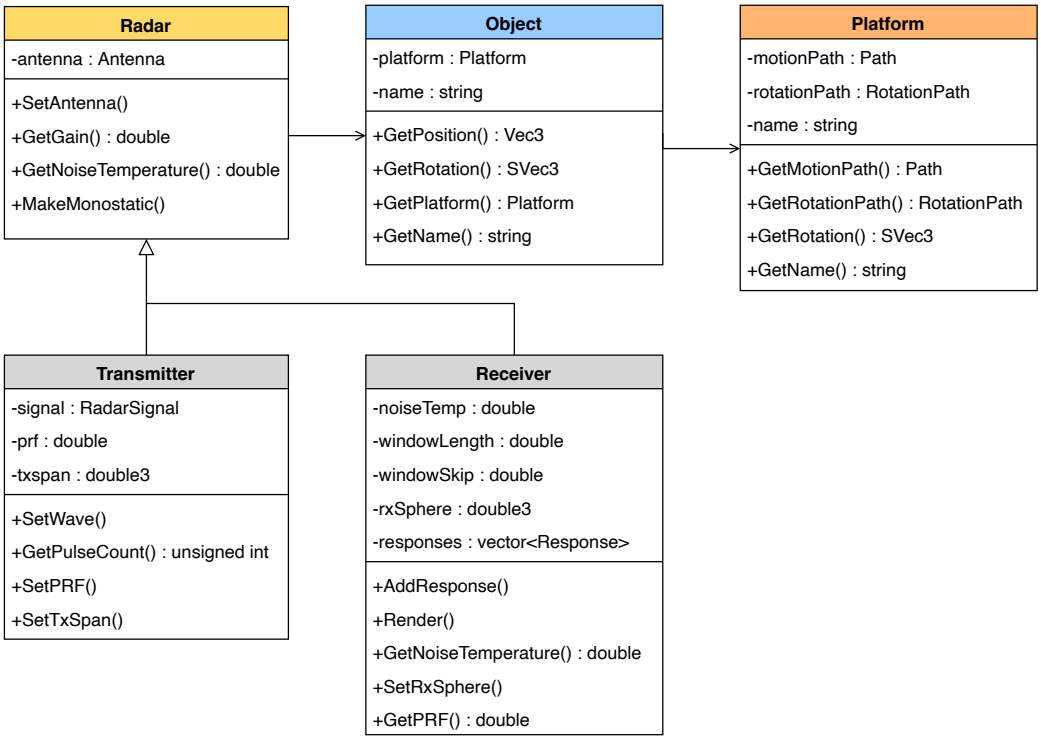


Figure 3.7: Class diagram for the radar class as used in SOARS, as well as the main associated classes and all integral methods and attributes

Table 3.2: Key parameters of the receiver model in SOARS

Module	**Parameter**	**Units**
Signal definition	PRF	kHz
Location	Spherical coordinate ρ	km
	Spherical coordinate θ	deg
	Spherical coordinate ϕ	deg
Low-noise amplifier	System noise temperature T_s	K
	System bandwidth B	MHz
Antenna definition	Antenna efficiency	–
	Boresight angle θ_B	deg
	Boresight angle ϕ_B	deg
Receive window length	Receive window length	s
	Receive window skipping length	s

notable classes associated with the radar class in SOARS – particularly the transmitter and receiver classes – as well as many of its methods, attributes, and relationships with other classes. This diagram thus depicts an overview of how the radar system model was implemented in the software. However, some classes (including RadarSignal, Antenna, and Response) were deliberately ignored in this diagram to focus on the most crucial elements – as well as due to space constraints.

Many of the methods and attributes shown in Figure 3.7 are self-explanatory, such as the functions used to set system parameters (such as the PRF) or compute variables (such as antenna gain). However, this class diagram also serves the purpose of highlighting the links between these closely related classes as well as the return type of their various functions, showing how multiple classes in the software are interconnected for maximum efficiency. Further information on each hardware model's implementation is available in [25].

It is worth noting that no size restrictions were placed on any of these classes in the software's design, and the user is able to define as many radar systems, targets, objects (each of which represents a target or system), and platforms (each comprised of objects that moving together) as desired; this is in keeping with the user requirements for the software as set out in Section 3.2.

3.5 Baseline Development

This section presents the main additions and changes made to the original FERS software as part of the development of baseline SOARS. This includes a discussion on the implementation of external noise sources, the implementation of RCS and SGP4, the use of a new XML parser, and the integration of CUDA for larger-scale processing.

3.5.1 Noise Implementation

As discussed previously, each receiver in SOARS simulates a low-noise amplifier to model the hardware between the antenna and the down-mixer [25]. This generates thermal noise which is picked up in the received signal as a result of the resistance of the amplifier. The noise power generated from this – assuming a resistance of R_n – is represented by a series voltage source with magnitude V_n, as described by Equation 3.4.

$$V_n = \sqrt{4kT_sR_nB} \tag{3.4}$$

where k is Boltzmann's constant and T_s is the system's noise temperature measured in Kelvin. In FERS and SOARS, the resistance R_n is assumed to be 1Ω, and T_s is used to account for the receiver, antenna and environmental noise temperatures.

This equation can be transformed into a system noise power expressed as:

$$P_n = kT_sB \tag{3.5}$$

In SOARS, complex Gaussian noise is randomly generated from a normal distribution with a standard deviation of $\sqrt{P_n/2}$. This appropriately scales the standard deviation of the samples in the in-phase and quadrature channels, resulting in noise samples that can be directly added to the received signal samples to produce the overall return at a receiver.

In addition to internal thermal noise, SOARS also needs to account for external noise sources from the surrounding environment. This typically includes sources such as sky noise, interference measured from the ground itself (typically taken as 300 K), and the Cosmic Microwave Background (CMB) – said to be relic radiation from the Big Bang that is often estimated as 2.7 K [172]. Noise contributions from other environmental sources have also been estimated as non-linear functions of frequency, including noise from atmospheric gases, the Earth's surface, and the Sun. These sources are often defined by a brightness temperature, which relates to the actual effective antenna temperature, T_a, through the convolution of the antenna pattern and the brightness temperature of the sky and ground [173].

Figure 3.8 depicts the frequency-dependent antenna noise temperature curves for carrier wave frequencies between 100 MHz and 100 GHz, as estimated from [173]. These curves form a relationship between f_c and T_a, where a reference temperature of 290 K was used in their development.

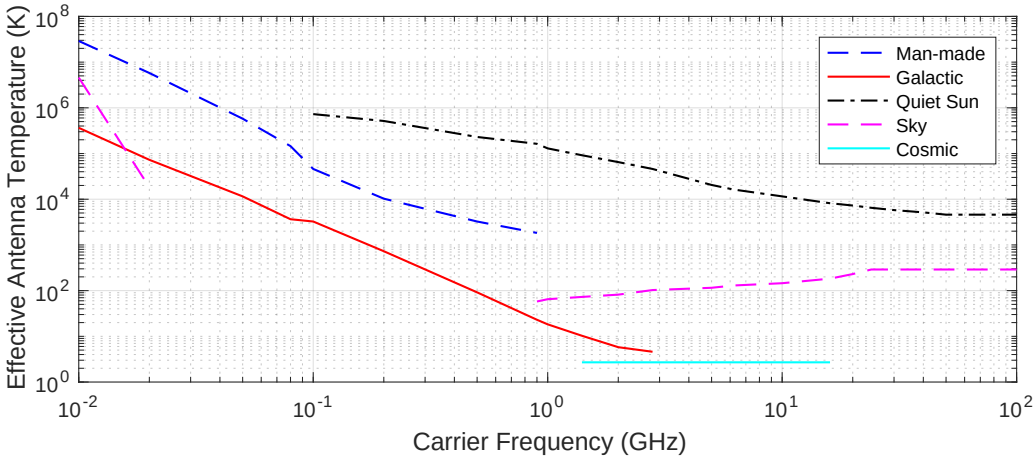


Figure 3.8: Estimated antenna noise temperatures due to external sources for common microwave frequencies

It should be noted that the beam of a horizontally-oriented receiving antenna, as denoted by the "5° elevation" curve in Figure 3.9 (adapted from [174]), will always radiate through more of the Earth's lossy atmosphere than the same vertically-oriented antenna beam (depicted by the "90° elevation" curve) [175]. As such, the latter results in a higher overall sky noise temperature than the former case.

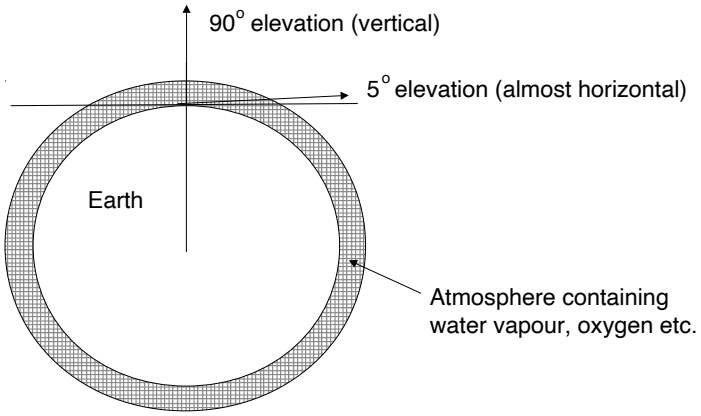


Figure 3.9: Two possible antenna elevations relative to the Earth and its atmosphere

However, when designing radio links, it is often more useful to account for the worst-case scenario in terms of received noise and interference. As a result of this, SOARS is designed to only account for the worst-case atmospheric sky noise given by the 5° curve in Figure 3.9, and this is represented by the dashed violet curve in Figure 3.8. The remainder of the external noise contributions come from any nearby man-made sources (estimated in Figure 3.8 as the median noise for a typical city area), as well as the worst-case galactic noise, variable noise emitted due to quiet Sun conditions [173], and black body radiation from the CMB.

These noise curves were selected for implementation for two main reasons, namely:

1. SOARS will only accept input centre frequencies between 10 MHz and 100 GHz (i.e., 10^7 Hz to 10^{11} Hz, as shown in Figure 3.8).

2. SOARS will only consider the worst-case noise temperature curve for

each source denoted by multiple curves in [173]. This is in keeping with the principle of prudence when designing radio communication systems.

The external noise additions are to be implemented in SOARS using linear interpolation between approximated points along each temperature curve; the user's input carrier frequency is then to be used to find the exact interpolated noise temperatures at the specified carrier frequency. Additionally, the user should be allowed to disable any (or all) of these external noise sources, allowing for flexibility if any sources are irrelevant to their simulated environment.

Once all the external noise temperatures have been calculated, they must be summed with the internal noise temperature of the receiver system to produce the total noise temperature T_s. This is then used in Equation 3.5 to define the noise power at the receiver, which is then added to each sample of the received signal before it is written to an HDF5 file at the simulator output.

3.5.2 RCS Modelling

In SOARS, a target's RCS σ can vary with both the angles of arrival and departure as well as due to statistical fluctuations [176]. An apparent RCS value can also be approximated using approximation methods such as the NASA Size Estimation Model (SEM) [177]. As a result, SOARS makes use of a combination of RCS modelling techniques – some of which have been adapted from FERS [25] while others have been added specifically to suit the use case of SOARS.

Apparent RCS Estimation

If measured RCS data is available, it can be loaded into SOARS through input datasets (representing the RCS at various azimuths and elevations) and then interpolated using a half-angle approximation based on the angles of arrival and departure. This represents the most accurate means of modelling a target's RCS in practice and ensures simulation accuracy.

However, SOARS also provides flexibility for cases where measured RCS

patterns are not available. A target's RCS σ can thus be computed through various methods as selected by the end-user; specifically, three RCS models have been made available in SOARS:

1. **File-based directional model**: as discussed, this approach uses a half-angle approximation to interpolate measured data of the RCS at various azimuths and elevations based on the angles of arrival and departure. This makes use of the RTS functionality specifically as it estimates the half-angle between an incoming signalå and an outgoing one.

2. **Isotropic**: uses a mean RCS value to model the object as a point target. This is useful for situations where the target is not complex enough in its geometry to warrant high-accuracy RCS estimation, such as a sphere. This is used in baseline SOARS as part of its point-model approach.

3. **NASA Size Estimation Model** [177]: computes a target's RCS based on the observing radar system's properties as well as the target's physical properties.

NASA's SEM was developed through a commissioned study that sought to map an object's RCS to its physical "size" based on its diameter d and the radar's wavelength λ. This is achieved using a normalised diameter (size) parameter, denoted x_s, and a normalised RCS parameter, denoted z_s:

$$x_s = \frac{d}{\lambda} \tag{3.6}$$

$$z_s = \frac{\sigma}{\lambda^2} \tag{3.7}$$

This gives rise to three operating regions; if λ is much smaller than the target size, the radar is said to be operating in the "Optical" region. If λ is larger than the object, the radar operates in the "Rayleigh" region. And finally, anything between these two extremes is said to be the "Mie Resonance" region. As a piecewise expression, the RCS is thus defined as shown in Equation 3.8.

$$\sigma = \begin{cases} (\pi d^2)/4 & x_s > \sqrt{20/\pi} \\ (9\pi^5 d^6)/(4\lambda^4) & x_s < \sqrt[6]{0.12/(9\pi^5)} \\ g^{-1}(x_s)\lambda^2 & \text{otherwise} \end{cases} \tag{3.8}$$

where $g(z_s)$ is a piecewise continuous approximation function (on a linear scale) that maps z_s to x_s, and $g^{-1}(x_s)$ represents this function's inverse.

With this piecewise expression defined, the RCS can be computed for any target using only its diameter (estimated as the longest distance between any two vertices in the object mesh) as well as the system's wavelength. Using this, the normalised RCS can be plotted against the normalised diameter as shown in Figure 3.10, verifying its implementation against [177]

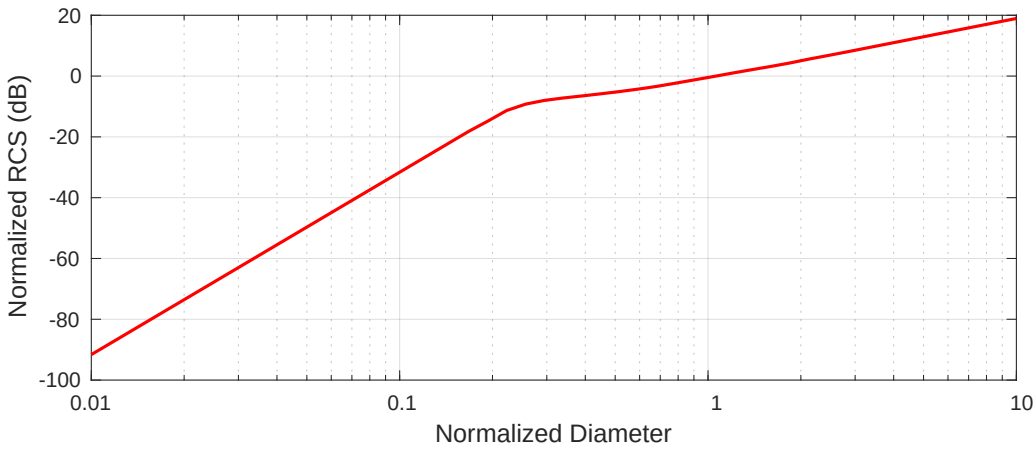


Figure 3.10: Normalised RCS as a function of an object's normalised diameter based on NASA's SEM model

This depicts how the estimated target RCS fluctuates based on the normalised diameter, illustrating the relationship between an object's physical size and its RCS as predicted by SEM. This has been directly integrated into SOARS and the RTS using the three operating regions described in Equation 3.8. This allows users to specify SEM as the RCS model for any given target along with its approximate diameter, and the mean RCS value is then determined.

Statistical Fluctuations

A target's RCS will often diverge from the apparent value due to probabilistic variations over time. These variations are accounted for via statistical fluctuation models such as the five Swerling cases [178–180], but the original FERS simulator made use of the chi-square model [181] due to its simple implementation and wide range of use cases [25].

This model has the following probability density function [181]:

$$p(\sigma, \bar{\sigma}) = \frac{k}{\Gamma(k)\bar{\sigma}} \left(\frac{k}{\bar{\sigma}}\right)^{k-1} \exp\left(\frac{-k\sigma}{\bar{\sigma}}\right) u(\sigma) \tag{3.9}$$

where $\bar{\sigma}$ is the mean RCS, $\Gamma()$ represents the gamma function, $u()$ represents the unit step function, and k is a parameter relating the chi-square model to various Swerling cases. This relationship is depicted in Table 3.3.

Table 3.3: Relationship between the chi-square model's k parameter and the Swerling models

Swerling Model	**Chi-square *k***
I and II	1
III and IV	2
V (constant RCS)	∞

While SOARS only accounts for the chi-square implementation for RCS modelling, alternative approaches could also be used based on the intended use cases. For example, models such as the Rician [182] and non-central Gamma [183] distributions may provide better experimental fits to specific contexts and measured data.

3.5.3 SGP4 Implementation

Vallado's [171] SGP4 software has already been published in multiple languages, including C++, and is well documented by the work done during the development of SGP4 theory [88]. However, given that the SGP4 program

spans a variety of use cases and is comprised of a large number of interdependent files and functions, it is important to first determine exactly what is required from the software within the context of SOARS.

Based on previous discussions in this dissertation, the following design requirements can be defined for its implementation of orbital propagation:

- The software algorithm needs to natively support TLE sets as inputs and be able to parse them
- RSOs represented by TLE sets need to be interpreted as radar targets within the context of SOARS
- Target positions must abide by the standard representation in FERS/-SOARS, i.e., a set of Cartesian coordinates (x, y, z) at a time t
- The positions of all RSOs must be defined in the ECEF reference frame

From these requirements, the desired functionality of the SGP4 library is apparent: namely, the software needs to convert TLE sets into ECEF vectors of the form (x, y, z) at various time instances t. These coordinate vectors must then be saved as the positions of target objects as they propagate along their individual paths within SOARS. Examining the C++ software developed by Vallado [171], it is evident that the necessary functions already exist to accomplish these goals – however, they require some degree of modification.

The logic involved in the SOARS implementation of the SGP4 algorithm is presented in the flowchart in Figure 3.11.

As depicted in Figure 3.11, the SGP4 routine is implemented in SOARS via three main functions adapted from the SGP4 source code: the TLE conversion function (called *twoline2rv*), the SGP4 propagation function (called *sgp4*), and the ECEF conversion function (called *teme2ecef*). The process is detailed as follows:

1. A TLE set is parsed from the *.soarsxml* input file, which is then processed by *twoline2rv*. This uses the TLE set input to produce a structure within C++ called *satrec*, which stores all the astrodynamics properties that are read in from the TLE dataset. The parsing of the TLE set also

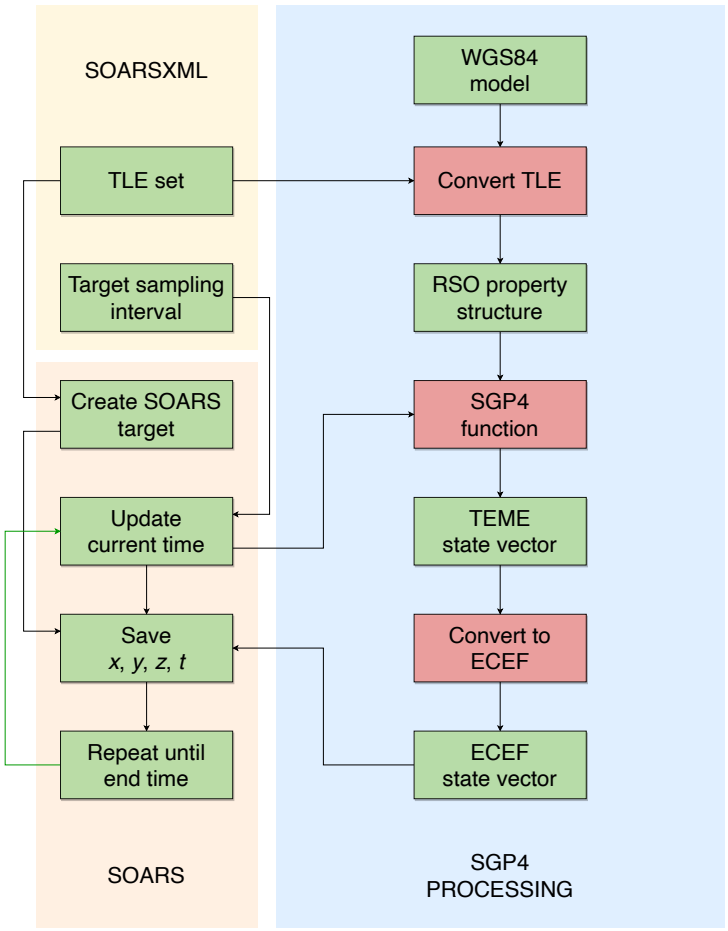


Figure 3.11: Flowchart illustrating the logic applied in integrating SGP4 functionality into SOARS

creates a new target in the simulation environment.

2. The *satrec* structure is passed into the *sgp4* function along with the current time t. This is updated iteratively based on the *target sampling interval* t_i, a simulation parameter which specifies how frequently the positions of every simulated RSO should be sampled. This function propagates the RSO (represented by *satrec*) from its initial epoch for a period defined by t (measured in minutes). The result is an output state vector in Cartesian form, but this ephemeris is generated in the TEME frame.

3. The TEME state vector is passed to the *teme2ecef* function, which applies a rotational matrix transformation to rotate the axes of the reference frame such that an ECEF state vector is produced (see Figure 2.11).

4. The ECEF state vector is passed back to SOARS, which records the RSO's position (x, y, z) at time t.

5. The full process is then repeated until t reaches the simulation end-time, at which point the target is propagated one final time and its position is recorded. If another TLE set is parsed, the full process repeats.

This approach allows for all of the software's SGP4 requirements to be met; TLE sets can be natively used as part of the input *.soarsxml* file, and each TLE set becomes defined as its own target within SOARS; the targets' positions are also all defined within the ECEF reference frame and take the form of (x, y, z) at various times, abiding by the standard used in FERS. This concludes the discussion on the algorithm and its integration into SOARS.

3.5.4 File Processing

Some of the significant changes made to FERS are the alterations made to its file parsing capabilities. These can be divided into three main categories of changes:

1. Modifying the structure of the *.fersxml* input file (renamed as *.soarsxml* for SOARS)

2. The replacement of the XML file parsing library used to read and write data in FERS
3. Processing changes made to accommodate limitations of the HDF5 library

Input Changes

The switch to the *.soarsxml* format also brought about additional changes to the input file structure and the XML parsing implementation, as some of the parameters and formatting of the original *.fersxml* file had to be adjusted for the SOARS context. For one, SOARS makes use of "system" nodes for radar representations and "target" nodes for RSOs in SOARS, whereas FERS uses "platform" nodes globally for all simulation objects [25].

Relative to the *.fersxml* file, the most notable change is that of the structure of the target input – which now requires the user to specify the RSO's TLE set. Previously, the target was simply defined as a set of coordinates (x, y, z) through which it must propagate at various time steps t, whereas in SOARS, the target's position vector is computed directly using the TLE set and SGP4 theory. However, a user is also able to use the previous implementation through manually-defined position waypoints, if preferred:

Listing 3.1 Target position waypoint demonstration in a *.soarsxml* file

```
<positionwaypoint>
    <x>0</x>
    <y>0</y>
    <z>100</z>
    <time>0.00</time>
</positionwaypoint>
```

An example of using a TLE to define a target's trajectory in the XML input file is given by:

Listing 3.2 Target TLE demonstration in a *.soarsxml* file

```
<tle>
    <line1>1 25544U 98067A   22132.35300373  .00005928
        00000-0  11162-3 0  9992</line1>
```

```
3:     <line2>2 25544  51.6429 154.0493 0006869  88.6298
          34.7251 15.50035239339623</line2>
4:     <timeskip>30</timeskip>
5: </tle>
```

To accommodate this, two additional input parameters are required from the user – the target sampling interval, t_i (a global parameter), and the target skip period (a target-specific parameter) – given by "timeskip" in the XML. The former entails propagating every target every t_i seconds and storing its new position into an array for later use when the radar equation is applied; the latter entails propagating the target by the specified skip period *before* the simulation starts, allowing some out-of-sight targets to be brought into a radar's beam.

XML Usage

During initial testing and deployment of the SOARS software, it was found that the greatest runtime bottlenecks resulted from the following processes:

- Reading in a large *.soarsxml* and setting up a simulation with a large number of targets or systems
- Writing each receiver's responses data to a separate *.soarsxml* file at the output of the software (optional)

Each of these bottlenecks was attributed to the lack of modernised subroutines inherent in the TinyXML source code, which severely handicapped the software in terms of its raw input/output capabilities. FERS was originally developed to use the TinyXML package for reading and writing XML files, but modern implementations of XML parsing mechanisms have offered significant speed-ups over this library. It was for this reason that alternative XML parsing solutions were considered, which eventually led to a comparison between potential XML libraries as illustrated in Figure 3.12.

This depicts the differences in parsing times of various C++-based XML parser packages relative to the PugiXML [166] package on 64-bit computing hardware. The parsing time is defined as the period measured in parsing

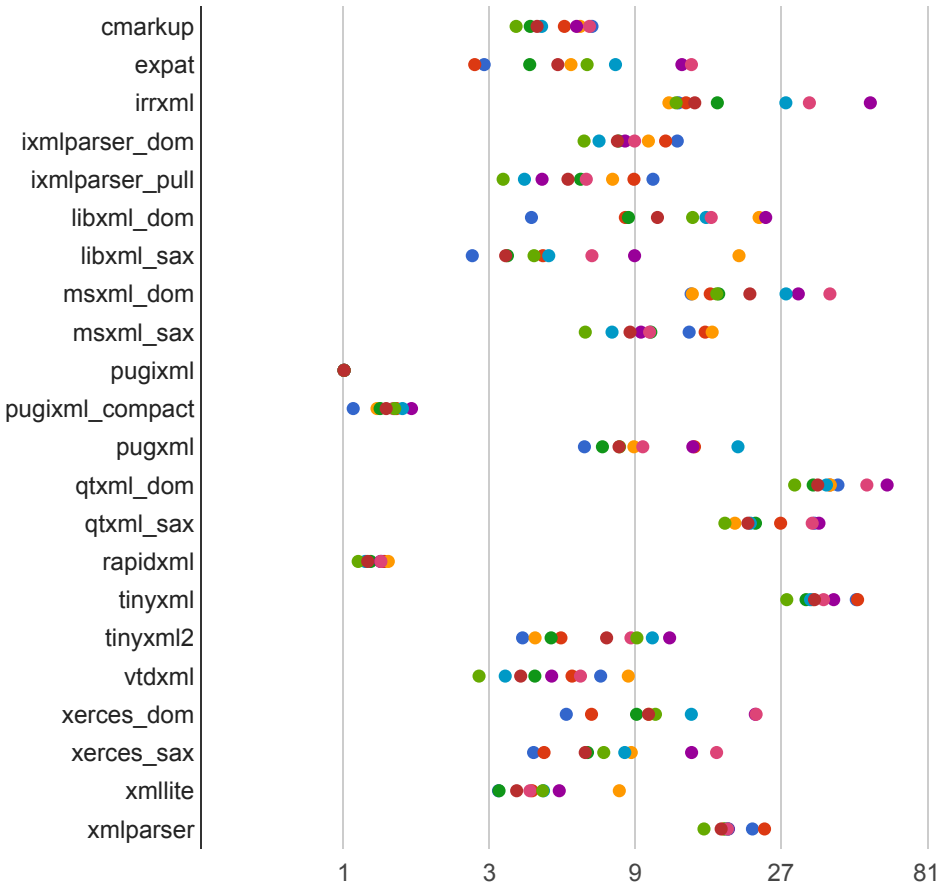


Figure 3.12: Parsing times of various C++ XML handlers relative to PugiXML for nine different (colour-coded) test files on 64-bit hardware; horizontal axis is logarithmic [184]

the document from memory on an Intel Core i7 processor clocked at 2.67 GHz. Benchmarks were performed on nine different XML files, with all timing results normalised against the PugiXML results as a baseline. The recorded timings, therefore, represent the ratio between parsing times of one parser relative to PugiXML on a logarithmic scale [184].

The results clearly illustrate the significant performance improvements provided through the PugiXML package – particularly compared to the TinyXML package used by FERS, which has already been deprecated and succeeded by TinyXML-2 from the original developer [185]. Even RapidXML – which was developed in 2006 as the de facto "fastest XML parser possible" [186] – was abandoned in 2013 and has since been overtaken by PugiXML in terms of raw performance, as depicted in Figure 3.12.

Based on these results, PugiXML was selected to replace TinyXML as the sole XML parser in SOARS. This required altering the makefile for the original FERS software, linking to the new PugiXML library, and modifying the XML parsing functions within the FERS source code to use PugiXML. This required significant rewrites to portions of the source code, and all traces of the TinyXML package were discarded. Further testing on this is documented in Chapter 5.

HDF5 Threading

The final major change for file parsing comes in the form of the HDF5 package [163]; in FERS, this package is used to render the raw received signal to an HDF5 file, maintaining the signal's in-phase and quadrature form. However, when *multiple* receivers are simulated, FERS makes use of a threaded file renderer through which multiple HDF5 files are meant to be written in parallel using CPU threads; during testing, this approach resulted in various threading errors as the HDF5 library does not natively support multi-threading [187] via Boost [160] – creating issues with some versions of the library when attempting multi-threaded file writing.

Additionally, the high-level C++ version of the HDF5 library does not support thread safety, as the package's function calls are not meant to be accessed from

different CPU threads simultaneously. A "thread-safe" version of HDF5 exists for the C programming language, but provides only the first level of thread-safety; in essence, this serialises the interface for usage in multi-threaded software, but it does not provide any actual parallelism [188].

As a result, the threaded HDF5 rendering approach in FERS was deemed to be erroneous and untested, and was thus removed from SOARS; instead, HDF5 file writing in SOARS was serialised such that all output signals are written sequentially to avoid CPU threading errors with this library.

3.5.5 CUDA Integration

As described previously, NVIDIA®'s Compute Unified Device Architecture – or CUDA – is a model that facilitates communication between a host CPU and a GPU device. CUDA makes use of kernel functions that run on the device across multiple threads for maximum throughput in general-purpose use cases. Currently, it is only supported on NVIDIA® GPUs, whereas alternative models such as OpenCL [189, 190] are less restrictive in terms of hardware. However, studies have indicated that CUDA often provides better performance than OpenCL, both in terms of transferring data to/from the device and in terms of kernel execution times [191]. CUDA is also known to be better supported in terms of development updates and community support, particularly with new CUDA-capable GPUs being released by NVIDIA® every year.

As was illustrated previously in Figure 3.3, CUDA is to be used for two express purposes in SOARS, namely:

- implementing and accelerating the ray-tracing algorithm
- parallelising the SGP4 algorithm when propagating target RSOs along their orbits

The first purpose involves using the GPU to maximise throughput of the ray-tracing mechanism. Since ray-tracing is an inherently parallel technique, it is well suited to acceleration methods across thousands of CUDA threads – with each thread computing the path of a single ray. The implementation of

the ray-tracing algorithm will be detailed in Chapter 4.

The second use for CUDA in this work is related to the parallelisation of the SGP4 model discussed earlier in the chapter. This entailed an update to the program's orbit propagation logic (from Figure 3.11 as shown in Figure 3.13.

As depicted in Figure 3.13, the CUDA version of the SGP4 algorithm works very differently from the serial implementation that was shown in Figure 3.11. The parallelised process is described as follows:

1. As before, a TLE set is parsed from the *.soarsxml* file and is processed by *twoline2rv*; this produces the *satrec* structure and also creates a new target in the simulation environment.

2. The property structure *satrec* is *not* passed into the *sgp4* function but is instead stored away in an array on the host's side. This is repeated for every TLE set serially, and a fully populated *satrec* array is generated.

3. A time array is produced on the host, which contains the time value t at various time instances. This accounts for every time sample between the simulation's start and end in increments of t_i.

4. Once both arrays are fully populated, empty position arrays are generated for every target's (x, y, z) position at every time instance t. Memory is then allocated on the device side for the total size (calculated in bits) of all five arrays, and the arrays are copied to device memory using built-in CUDA functions.

5. The CUDA kernel is called, spawned across many parallel GPU threads equal to the product of the number of targets and the number of time steps (i.e., the product of the sizes of the *satrec* and time arrays). Each thread is thus responsible for computing the SGP4-propagated path of a single target's *satrec* at a single time instance t, as determined by the thread index and array indices.

6. The *sgp4* function propagates the RSO in the TEME frame, which then undergoes conversion to the ECEF frame. The ultimate output of each thread is thus a position vector of an RSO in the ECEF frame, of which

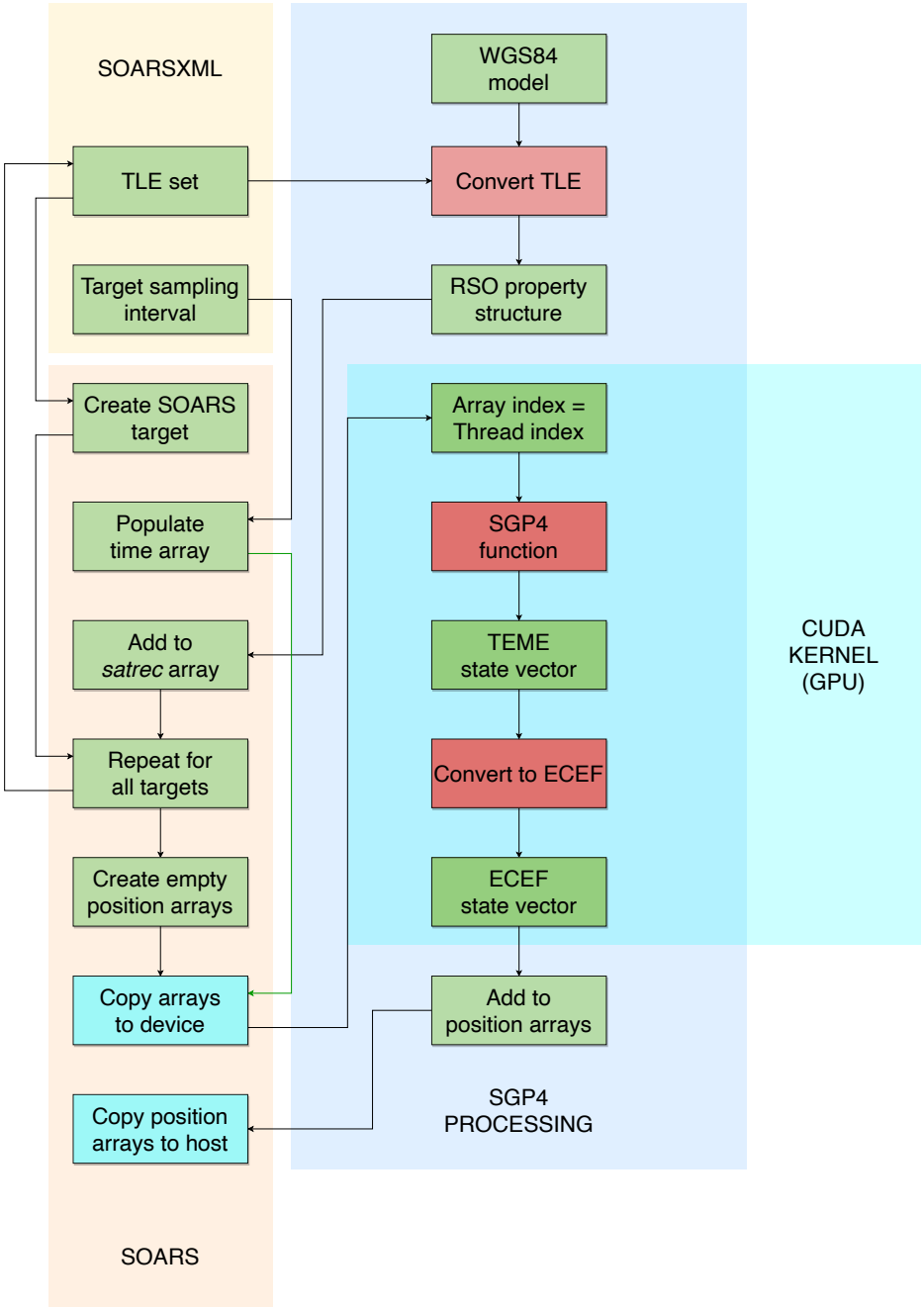


Figure 3.13: Flowchart illustrating the integration of CUDA into the SGP4 algorithm within SOARS

the (x, y, z) values are then written to the position arrays as indexed by the thread.

7. The position arrays are copied back to host memory for processing.

Through this data-parallel algorithm, all targets at every time instance can be propagated simultaneously using SGP4 theory and CUDA. However, there are drawbacks to this approach; the most notable drawbacks are the additional memory requirements for the arrays on the host side (which could become quite enormous for large numbers of targets and/or long simulation times), as well as the additional overhead introduced by using CUDA. The latter results from having to initialise the GPU, making use of the CUDA compiler NVCC [169], and – most importantly – the process of copying data to (and from) the device.

Another consideration is the difference in output values generated by the serial and CUDA versions of SGP4, as there are slight discrepancies in the computed values of the target position coordinates produced by each method. These discrepancies are attributed to the unavoidable precision differences resulting from floating-point calculations, which are often computed in slightly different ways on CPUs and GPUs. This is particularly apparent with CUDA, where functions that are compiled for the GPU make use of NVIDIA®'s *CUDA math* library, while functions compiled for the CPU use the host compiler's math library [47]. These libraries will often produce slightly different output values due to the different implementations of built-in mathematics-based functions, such as cosine, sine, arctangent, and modulus.

Additional precision discrepancies may also result due to different platform architectures, different operating systems, or different compilers. In the case of CUDA (as used in SOARS), the NVCC compiler is used to compile *.cu* source files and *.cuh* header files; on the other hand, the serial parts of SOARS makes use of standard *.cpp* and *.h* header files, which are compiled by standard CPU-bound compilers such as GCC [192]. The CPU and GPU compilers may thus generate slightly different floating-point results, and depending on an end-user's system and software configuration, some computations may be calculated differently between users.

Some of these precision errors may be alleviated by forcing the GPU to ignore the use of fused multiply-add (FMA) operations. FMA is used to perform calculations with only one rounding step, processing multiplication and addition operations in the form $round(X \times Y + Z)$. However, by disabling FMA, the calculation would instead be computed as $round(round(X \times Y) + Z)$, which requires two rounding steps – one for the multiplication operation and one for the addition [47]. Using FMA will thus produce a more accurate result as rounding only occurs once, and for this reason, FMA was added to the IEEE 754 standard in 2008 [193].

However, disabling FMA operations may actually align CPU and GPU results more closely in value, but neither would necessarily be more accurate than the other. By default, NVCC compiles the source code with FMA enabled, but it may be disabled by the user at compile time by setting the *fmad* flag to false in the CMake makefile for SOARS. This is ultimately left up to the user to decide based on their hardware and architecture, but it is recommended that the flag simply be set to its default, i.e., with FMA enabled.

With this, the main features and additions of the baseline SOARS program have been detailed. Initial results for the baseline simulator are shown in Chapter 5, while Chapter 4 focuses on the development of the ray tracer and its integration into baseline SOARS.

3.6 Conclusions

This chapter has presented the design and development of a baseline version of SOARS – an open-source, signal-level, point-model radar simulator intended to be used in the design and testing of space surveillance radar systems to improve global coverage of space objects. It is anticipated that SOARS could be used by engineers and researchers for conceptual investigations and systems design – as well as in teaching or training environments. The developed program serves as a combination of the existing FERS application, a software model for noise sources relevant to radio astronomy, various methods for RCS estimation, the established SGP4 propagation model, and the NVIDIA® CUDA model for GPU-based parallelism.

Additionally, the chapter has explored many of the technical aspects relating to the SOARS software, such as its noise implementation, the integration of RCS modelling, CUDA and SGP4 into the simulator, and the introduction of a more modern XML file parser. Complete lists of both functional and non-functional software requirements have also been presented along with a detailed description of the various hardware models used in the software. The discussions in this chapter have thus highlighted the design decisions made in developing SOARS, the basic structure used by the software, and many of the actual implementations of new features into the source code already provided by FERS.

An extensible baseline version of SOARS has thus been established through this chapter, which was planned around facilitating further expansion of the application. Chapter 4 aims to build upon this foundation further and explore the implementation of ray tracing for signal-level radar computation. However, benchmarking tests will need to be conducted to assess the trade-offs between the theoretical improvements in accuracy and the changes in computational efficiency brought about by NVIDIA®'s OptiX™ ray-tracing engine; these results, as well as comparisons between the baseline and ray-traced simulators, are detailed and analysed in Chapter 5.

Chapter 4

Ray Tracer Development

This chapter details the design and development of the ray-tracing program module and its approach to radar computation, and the implementation of this module into the baseline SOARS software. The algorithmic logic and software structure of the simulator are also presented. This chapter thus forms part of the development objective defined in the methodology in Section 3.1 and aims to conclude discussions on the design and implementation of the complete simulator package.

Additionally, this chapter acts as the second iteration of the system's overall design in keeping with the spiral development model [150]. This includes detailing some background theory and literature as well as various software design aspects relating to ray tracing. The result of this is the final evolution of the SOARS software, with a diagrammatic view of the full SOARS program provided at the end of the chapter. Results from both this program and the original baseline simulator are presented in Chapter 5 to compare the accuracy and performance of the two simulators in generating signal data.

4.1 Algorithm Design

As a concept, ray tracing has been widely used in electromagnetics for a variety of use cases, including medical imaging [33], signal strength and fading prediction [194, 195], and radar-based applications as applied to traffic

scenarios [43, 135] and electromagnetic simulation [42, 134]. However, none of these works has been found to investigate ray tracing as a valid mechanism in *signal-level* radar simulation specifically.

In 2013, Williams [132] developed an OptiX™-based [30] simulation tool for millimeter-wave systems, comparing the use of ray-tracing with the Method of Moments [196] for full-wave electromagnetic modelling. In this work, it was found that OptiX™ provided significant performance gains relative to the conventional approach, but this was specifically focused on electromagnetic modelling for medical applications.

Other related work was conducted in [32], which presented an overview of how various ray-tracing algorithms could be used for radio propagation modelling, including the image method and the shooting-and-bouncing ray method. The work concluded that ray tracing, in future, would be greatly useful "for tackling complicated propagation environments in high-frequency regimes" [32]; however, the overall work was limited to the modelling of electromagnetic quantities as governed by Maxwell's equations.

Presently, there are no 3-D, ray-traced, signal-level radar simulators freely available for public use – particularly one that is specialised towards space monitoring. In this section, the design of an open-source ray-tracing module based on OptiX™ is presented; this is to be used for signal-level radar simulation in generating raw return signals at a receiving antenna. The development of this module constitutes the main original contribution of the work with respect to the established literature, and it is expected to improve radar simulation accuracy through realistic models for both signal propagation and physical target features. An overview of how this is expected to work is shown in Figure 4.1.

This presents a high-level conceptual diagram of how ray tracing works in a radar context: rays are first transmitted from a transmit antenna and are traced through the environment; upon intersecting a target, rays reflect and/or refract (as computed by geometric methods) and may intersect further targets, a receiver, or just "miss" the rest of the environment and propagate towards nothing. Any received rays can then be processed at the receiver and

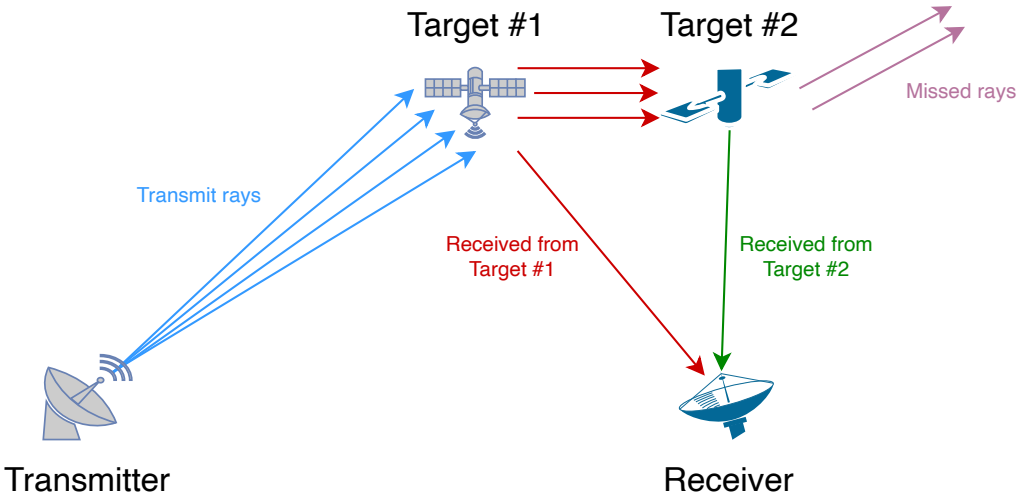


Figure 4.1: Overview of ray tracing in the context of radar signal propagation, depicting the rays being traced from a transmitter to two targets and then to a receiver; missed rays are also presented

a signal can be rendered.

To simulate a combination of targets and transmitter-receiver pairs using ray tracing, representations of the objects and systems must be defined within the ray-tracing framework; additionally, models are required to describe ray behaviours for ray-object interactions and ray capturing at a receiver. This section thus presents an overview of the OptiX™ engine and framework as well as some of the design decisions for implementing these components within the RTS.

4.1.1 OptiX™ Framework and Engine

OptiX™ is a ray-tracing tool that acts as middleware for the implementation of ray tracing on supported NVIDIA® GPUs. The first iteration of the software was initially released in 2009 and the tool remains in continued development today [197].

As a result of the discussions in Section 2.5, it was decided that an existing ray-tracing engine would be used in this work such that many of the required back-end features would already be built into the program's design. As opposed to

developing a complete engine from beginning to end, this approach represented a more practical method of introducing ray tracing into the existing software. In particular, it was decided that NVIDIA®'s OptiX™ would be a viable ray-tracing framework for this work, namely because it:

- offers free, general-purpose use and receives frequent updates by the developers
- accounts for all significant considerations in the design of a ray-tracing algorithm as previously mentioned in this section
- presents a highly-optimised strategy for implementing ray tracing in programs running on NVIDIA® GPUs
- allows for efficient interoperability with CUDA, which can also be used to accelerate other aspects of SOARS
- hides many of the optimisations, execution decisions, and stack management protocols that would otherwise need to be handled by the user and/or developer [138, 198]
- has been proven highly effective in a wide variety of fields and applications, including film animation [199], volume estimation, collision detection, and sound and radio wave propagation [132]
- already has an existing code base from the application developed by Williams [132], which could be adapted to suit this work's needs

The decision to use OptiX™ also introduces one of the main differences between SOARS and an existing radar simulator by Kastinen *et al.* [27]. Conceptually, both applications overlap in their use of SGP4 for orbit propagation and in their simulation of radar interaction with space objects. However, both pieces of software have radically different approaches and implementations; Kastinen's software is specifically a tracking simulator that makes use of tracklets to gauge debris population sizes, whereas SOARS is intended to be a signal-level radar simulator for designing and testing space-monitoring radar systems.

It is also expected that the OptiX™ engine will improve the target and

propagation modelling accuracy of simulations run in the ray-traced SOARS (relative to the point-model simulator) – particularly with targets being approximated with material properties and physical volumes and shapes. An alternative approach to this is presented in [135], whereby targets are represented in polygonal form and each polygon is replaced by a point scatterer at its centre, but this thesis aimed to specifically model objects as 3-D meshes to realistically model phenomena such as optics-based reflection and refraction. The implementation of ray tracing, as documented in this chapter, is also inherently different than the approaches used in [135] and [43].

In general, OptiX^{TM} can be described by two main components, namely the:

1. **framework component**: used to specify the rules and set-up of the simulated environment (also referred to as the ray-tracing "scene"), provide program functions and data types, and establish the structure of user-editable programs

2. **engine component**: controls the actual ray simulation events and operations, including calls to internal programs and the execution of simulation behaviours

Together, these two components enable OptiX^{TM} to be directly integrated into the baseline SOARS software developed in the previous chapter.

The OptiX^{TM} code used in this work was built upon the foundations of a simulator tool developed for medical imaging in [132]. This included adapting the program from being an electromagnetic simulator (computing electric field amplitudes at discrete points along a single cylindrical antenna) to a signal-level simulator that generates received signals based on highly-customisable and near-unlimited scene geometry and system parameters. Based on this, the ray-tracing implementation in SOARS is achieved through an RTS module (and program function) that makes use of a combination of the C and C++ programming languages for host code, and CUDA C++ for device code.

A high-level operational view of the RTS is depicted in Figure 4.2.

The diagram in Figure 4.2 depicts a basic operational overview of the RTS, with the host code being responsible for initialising variables, communicating

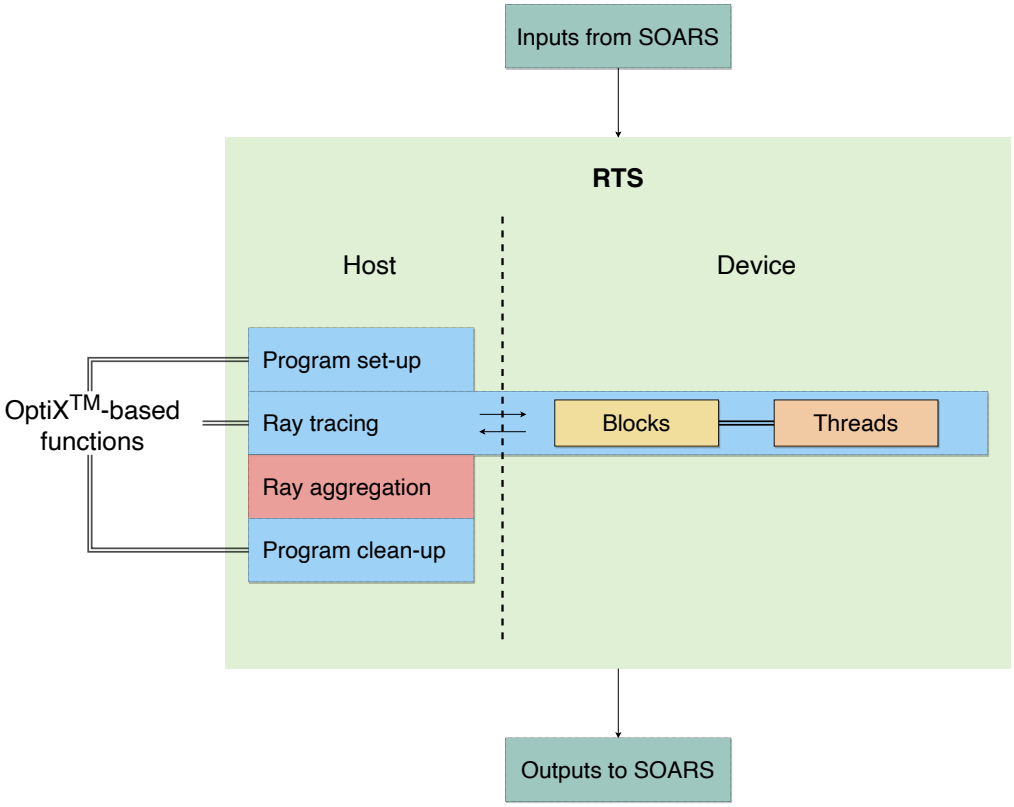


Figure 4.2: Block diagram of the OptiX™-based ray tracing in SOARS, depicting the implementation of code on the host (CPU) and device (GPU)

data between CPU and GPU, and setting up the scene within the OptiX™ framework. This is referred to as the program set-up. The device code, which is launched via a kernel, is responsible for executing the ray-tracing algorithm; this includes launching rays from transmitters, propagating and tracing them through the environment, testing them for intersections with targets, and then capturing and aggregating them at receivers.

All of this is achieved through eight editable programs provided by the OptiX™ framework, which together provide a massively parallel ray-tracing implementation. These programs are used to describe ray properties – as well as the scene itself – during program set-up, when the RTS initialises the user-defined scene and all required parameters on the host side. Ray tracing then occurs via the OptiX™ kernel, which is launched on the device and invokes many of the user-editable programs. After the rays have been traced and aggregated (or discarded if they are determined to be invalid), control is returned to the host and program clean-up occurs to free device memory. The RTS is then exited and its outputs are forwarded to SOARS.

Each of the eight user-editable programs (included with OptiX™) is a customisable module for generic use of the software and can be tuned for various functionalities. In the RTS, some of these programs are used to compute ray-target intersections, ray capture information, and more. These programs make up the foundation of the RTS, and as such, it is critical to know what each of the programs does and how they integrate into the system.

Five main OptiX™ programs are used as part of the RTS:

1. **Ray generation**: launches rays into a simulated scene. Each spawned ray initiates a BVH traversal to determine interactions with the scene's geometry, i.e., target objects. Through this program, the user can spawn rays through a point-source transmitter.

2. **Intersection**: describes the relevant computations for an intersection occurring between a ray and object geometry in the 3-D scene. In the RTS, this is used to compute ray-triangle intersections as well as interpolated surface normals at the points of intersection.

3. **Closest hit**: invoked when a valid ray-object intersection is found and computes the result of such an intersection. The RTS uses this to find the directions of reflected and refracted rays and recursively emit new rays along those directions. Additionally, many ray properties are updated in this program, such as the ray's power and path length.

4. **Miss**: describes what happens to a ray if it does not intersect any geometry and/or has no impact on objects in the scene. In the RTS, this is used to compute ray-receiver intersections, as the receivers are not modelled as physical geometries in the same way as targets. Rays that hit the Earth's surface (modelled as a sphere in the RTS) can also be identified in this program and thus ignored for the rest of the simulation.

5. **Bounding box**: used as part of the BVH process to determine primitive geometry bounds and thus speed up ray-object intersection testing.

Together, these programs are integrated to run asynchronously and create a structure with minimal dependency between traced rays, providing high-performance computation with limited memory usage. The implementation of these programs into the developed software is further discussed later in the chapter. OptiX^{TM} also provides access to another three programs, namely the *exception*, *any-hit* and *selector-visit* programs, but these are unnecessary in the context of the RTS and are thus not used as part of the software.

4.1.2 Launching Rays

As previously discussed, every SOARS simulation consists of at least one transmitter, receiver and target. The most crucial part of adding a ray-tracing algorithm into this configuration is computing the direction of rays, which heavily impacts the resulting calculations that take place at the receiver. However, it is also important that rays are *spawned* in a variety of directions such that every ray path is not computed more than once. In addition to this, the rays should be distributed uniformly across a specified span – such as in the direction of a target or along a transmitter's boresight – to accurately model the dispersion of a transmitted wave.

One approach to this is to launch rays isotropically in every direction and sample every direction evenly, but this could be viewed as a waste of resources; in a scene with only one object being simulated, for example, the majority of rays would be wasted and launched into unoccupied space with no geometry to intersect. This also means that a very small number of rays would actually hit the single target – if any at all – and thus a low-resolution result would be observed at the receiver.

An alternative approach is to spawn rays that are targeted directly towards the object(s) of interest and then spread them across a short distance surrounding the objects-under-test. This is illustrated in Figure 4.3.

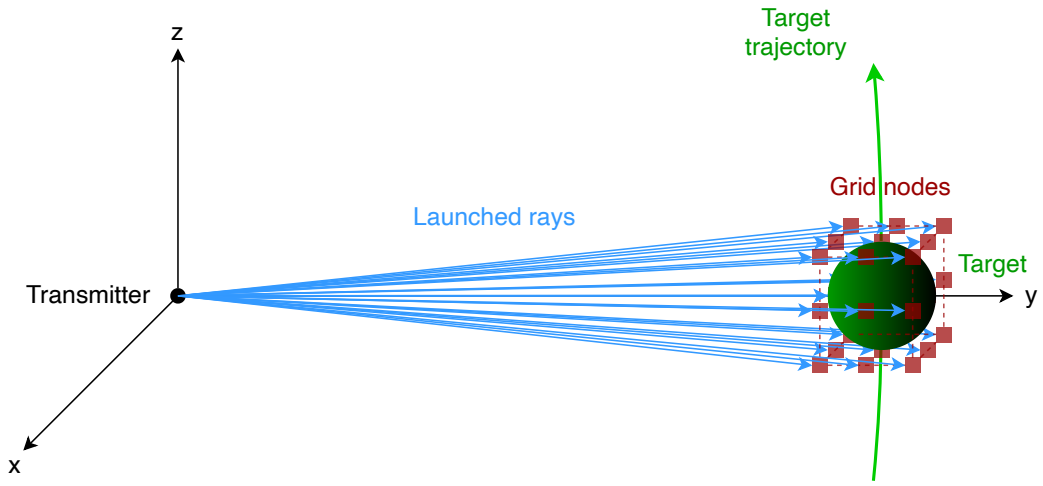


Figure 4.3: A point source transmitter launching rays towards a target with directions specified by a grid of virtual nodes spaced equally along spans in (x, y, z); the nodes do not form part of the physical scene

In the scenario depicted in Figure 4.3, a transmitter is modelled by a point source while the target is modelled as a sphere. The direction of propagation for each ray is computed by the vector between the transmitting source and a grid of nodes spread around the target – as depicted by the red squares. This virtual grid does not form part of the physical scene being simulated in the RTS but serves only to specify the direction vector for each ray as it launches from the transmitter and traverses towards a uniquely-defined grid node. Each node thus corresponds to a single ray such that the number of grid nodes is equal to the number of rays launched into the scene. In this

way, potentially millions of near-parallel rays are traced from the transmitter to accurately sample the scene target(s).

Realistically, however, this approach is inaccurate; it bears many resemblances to the model used in FERS [114], where the radar computations make use of the vectors directly from the transmitter to the target (and from the target to the receiver). The primary aim of the RTS is to introduce more realistic representations of antenna wave propagation and target models through shape, size and material parameters – as opposed to the point target model used in FERS [114]. As such, directly mapping the rays to each target is too much of a simplification. Additionally, the FERS approach does not account for a transmitter's beamwidth and ignores the impact of targets overlapping and/or shadowing one another in 3-D space.

Another approach is to direct the rays around the transmitter's boresight direction. This is deemed as the most plausible as it allows the user to specify a beamwidth in azimuth and elevation angles, which could then be used to direct the rays around the boresight vector at its centre. This represents a modification of the previous concept and is shown in Figure 4.4:

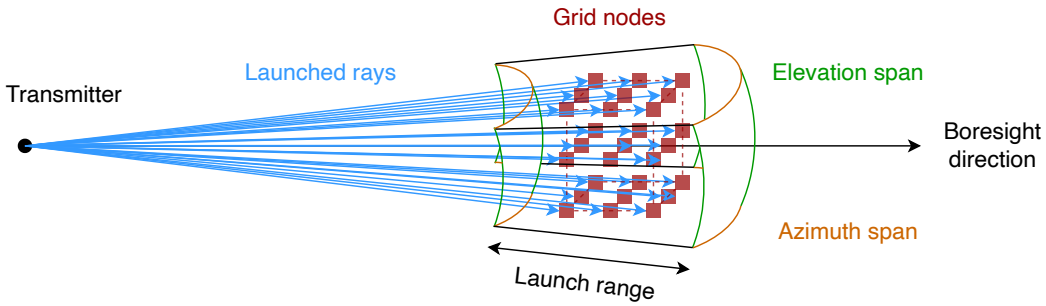


Figure 4.4: A point source transmitter launching rays towards its boresight vector with directions specified by a grid of virtual nodes spaced equally along spans in azimuth and elevation; these nodes do not form part of the physical scene

In the RTS, this approach is implemented using a simple geometry algorithm within the ray-generation program. This program acts similarly to the *main* function in C and C++ and is launched in parallel across many threads – one for each ray. On each active GPU thread, the ray-generation program then

calculates the direction vector of the incident ray based on the transmitter boresight and the ray index in (x, y, z).

The direction vector of an incident ray is thus computed by initialising a grid of nodes around the transmitter boresight vector, then creating a vector between the transmitter and each node. These nodes, depicted in Figure 4.4, are created by initialising a unit vector with a minimum "beam starting point" (set by subtracting half the beamwidth in azimuth and elevation from the vector $(1, 0, 0)$) and a maximum "beam ending point" – set by adding half the beamwidth in azimuth and elevation from the vector $(1, 0, 0)$.

After calculating these minimum and maximum corners of the node grid, nodes can be automatically placed by simply incrementing the (x, y, z) coordinates based on the launch index of the current ray in each dimension. In the case of the x-dimension, this incrementation is also extended to a user-specified launching range to reduce the number of rays with the same direction.

The ray spawn directions are then finalised by rotating them first in azimuth and then in elevation using rotation matrices. The first step in this is simply achieved by first applying the azimuth rotation matrix:

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

Applying the elevation rotation, however, requires an axis-and-angle rotation [200] about the y-axis after the axis itself was rotated through the same azimuth rotation matrix given in Equation 4.1. This redefines the y-axis as an arbitrary axis (u_x, u_y, u_z) around which an elevation rotation can be easily applied using the matrix:

$$R_\phi = \begin{bmatrix} c + u_x^2(1-c) & u_x u_y(1-c) + u_z s & u_x u_z(1-c) - u_y s \\ u_y u_x(1-c) - u_z s & c + u_y^2(1-c) & u_y u_z(1-c) + u_x s \\ u_z u_x(1-c) + u_y s & u_z u_y(1-c) - u_x s & c + u_z^2(1-c) \end{bmatrix} \tag{4.2}$$

where $c = \cos\phi$ and $s = \sin\phi$. Note that this deviates slightly from the equation in [200] to reverse the elevation direction such that it agrees with the rotation convention used in the RTS.

This provides a means for computing ray directions using the transmitter's boresight direction, angular spans (or beamwidths) in both azimuth and elevation, and a launching range – which can typically be set to 1 just to introduce more unique directions into the ray spawning process. These three new parameters thus serve as additional transmitter parameters relative to those in Table 3.1.

This implementation will thus create a set of nodes spread around the transmitter boresight direction by varying (x, y, z) using the azimuth and elevation beamwidth spans. The launch ranges are also varied to ensure that each ray has a unique direction even along the x-, y-, or z-axis – aside from a few rays that may traverse directly along the exact boresight direction. Additionally, if only one ray is being spawned, it will be set to launch directly along the boresight vector.

4.1.3 Intersection Testing

The target objects in SOARS can easily be described by procedural geometries or polygon meshes within the RTS; in particular, each target in the simulation can be modelled as a tessellated mesh of triangles to simplify ray-object interactions into (less complex) ray-triangle intersection tests. A target's mesh data is thus described by a matrix containing the coordinates of the three vertices of every triangle. These coordinates are stored in Cartesian form as (x, y, z), and each set of coordinates is associated with a single triangle belonging to the mesh.

However, this implementation brings about several considerations that need to be accounted for when computing ray-object intersections. This subsection presents some detail on a few of these considerations and how the RTS is designed to address them.

Reflection and Refraction

When a triangle is intersected and a valid ray-object intersection is found, the effects of reflection and refraction must be computed using built-in functions in OptiX^{TM}; contributions from diffraction (and other optical phenomena) are not considered as they (1) are not within the scope of this work, and (2) would likely reduce the efficiency of the algorithm due to a lack of optimisation and built-in structures. Furthermore, while diffraction is considered to be part of the future recommendations for this work, it has been previously ignored as part of the ray-tracing framework involving space debris as proposed in [128].

To properly model the reflection and refraction of rays at object intersections, it is important to understand how ray-mesh interactions are computed. When a ray intersects the surface of a target RSO, the direction of the reflected ray needs to be calculated – along with any potential changes to the ray's properties. Most notably, the ray should experience a decrease in power based on the reflection coefficient, Γ, of the object under test [25, 132]. If the maximum reflection depth (i.e., the maximum allowable number of reflections) has not yet been exceeded, the ray reflects and loses some of its power as it traverses in a new direction.

If the maximum *refraction* depth has also not been reached, each intersection will result in a second, *refracted* ray to be generated at the point of intersection. The portion of the ray's power that was not reflected is thus transformed into refractive ray power, where a scaling value of $1 - |\Gamma|$ is assumed while the reflected ray propagates with a power scaled by Γ. This is illustrated in Figure 4.5.

The refracted ray is computed using the built-in *refract* function in OptiX^{TM}, while the reflection is similarly computed using the *reflect* function. However, the *refract* function also requires knowledge of the previous and current

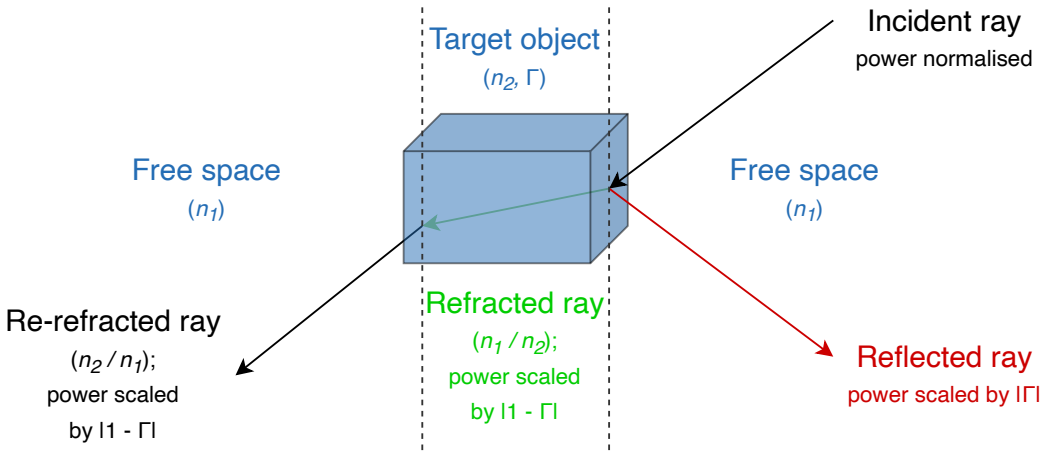


Figure 4.5: An intersection between a ray and a target that produces both reflected and refracted rays; a second refracted ray is also produced when the first refracted ray intersects the inside surface of the object

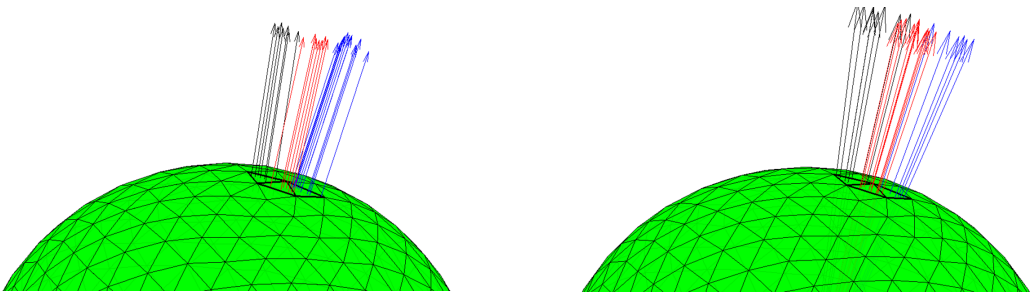
propagation mediums of the ray to employ Snell's Law via Equation 2.17. It is thus crucial that the user specifies each target's refractive index, as SOARS otherwise assumes that the propagation medium is free space.

Vertex Normals

Before new directions can be determined, the point of intersection between a ray and an object must be known. Triangular meshes were thus selected for use in SOARS, allowing for ray-object intersections to be simplified into ray-triangle intersections through barycentric coordinates [201]. If a ray-object intersection is detected (using built-in OptiX^{TM} structures), the directions of the reflected and refracted rays are both computed using Snell's law, as shown in Equation 2.17. Discretising an object into a mesh has one notable disadvantage though: each triangle becomes "locally flat", meaning that objects with curvature (such as a sphere) can yield inaccurate results.

This is because the normal of each triangle's face, computed as the cross product of any two sides of the triangle, is always directed outward and is constant at any point along the triangle's face. For a curved object such as a sphere, reflected rays will thus be computed using these constant normals, resulting in lower accuracy when larger triangles are used or when the object's

radius of curvature is decreased. This can result in position errors for curved objects, and the severity of these errors only increases as the rays propagate over larger distances. This is demonstrated in Figure 4.6, showing three sample triangles and their surface normals at different points along their faces.



(a) (b)

Figure 4.6: The surface normals computed for three triangles using (a) the cross product of two sides of the locally-flat triangles and (b) a surface normal interpolation algorithm to better account for the curvature of the object [132]

The first result in Figure 4.6 shows how all the surface normals of a single triangle have the same direction since each of the triangles is locally flat. This implies that any rays that intersect a particular triangle will have the same reflected direction, causing them to propagate in disparate groups – each of which is directed by the same vector. However, the second result in Figure 4.6 illustrates surface normals that more accurately model the curvature of the sphere. This is achieved through an interpolation method whereby a vertex's normal is calculated as the average of the locally-flat normals of all triangles that share that particular vertex; then, when a ray impacts a triangle, the vertex normals are interpolated to compute a normal more representative of the object's curvature.

To account for the possibility of the mesh containing triangles of varying sizes, this method also makes use of a weighted-average implementation to use the contribution of triangle area to the computation of the vertex normals. The end-user of the software is free to enable or disable this "smoothing" operation for situations where objects of interest are less curved, or where preserving object edges and flat surfaces is of high importance.

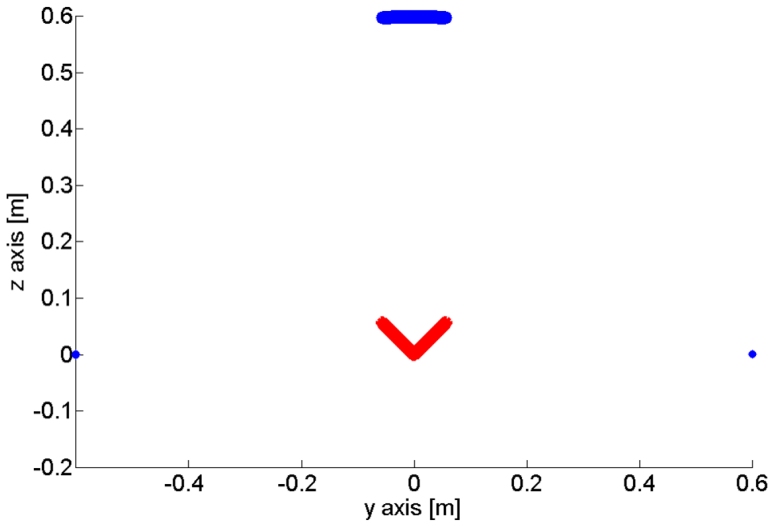
Scene Epsilons

Another intersection consideration is that of precision limitations inherent in floating-point computation [202]; when a ray intersects an object, the coordinates of the computed intersection may be slightly above (or slightly below) the actual coordinates of the surface due to precision losses. The reflected/refracted ray would then be calculated as normal, but if the computed intersection point is marginally above or below the actual surface, the new ray may unintentionally intersect the same surface from within. This phenomenon is sometimes referred to as "self-shadowing".

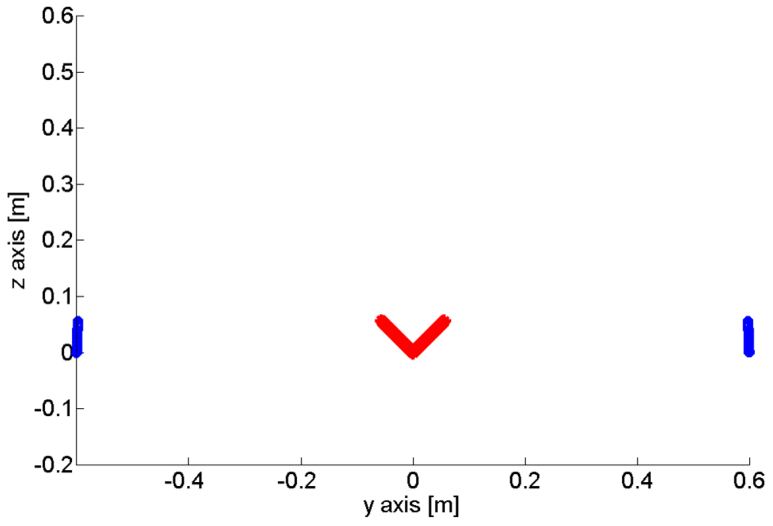
OptiX™ thus makes use of a compensation parameter called the "scene epsilon" to help guard against this problem, as is often done for ray-tracing algorithms [123]. This parameter is set as a minimum distance value, and if a ray then hits an object and the ray's length is found to be smaller than this minimum, the engine is told to ignore that specific intersection and treat it as a self-intersection; the ray is then simply allowed to continue to propagate until another intersection is reached. In this way, the scene epsilon is used to combat unintentional self-intersections.

The two scene epsilons (one value for incident rays, ϵ_i, and one value for reflected rays, ϵ_r) should thus be set by the user, who should be aware that these parameters are highly dependent upon the geometry of the scene being simulated. If a scene epsilon is too small, self-intersections may occur; but if a scene epsilon is too large, valid ray-object intersections may be ignored as they could be assumed to be self-intersections. This concept is demonstrated through an experiment conducted in [132] – the results of which are shown in Figure 4.7.

The illustrations in Figure 4.7 show two scenarios, each with a different value of scene epsilon being used. Both scenarios feature a dihedral (red) that was illuminated with thousands of rays which terminated on a cylindrical receiver surrounding the object, as shown by the points highlighted in blue. In this simulated experiment, the rays were allowed to reflect up to two times in the scene, and thereafter they were traced to the receiver. The anticipated result was that rays would be spawned by the transmitter (above the dihedral),



(a)



(b)

Figure 4.7: Demonstration of how the scene epsilon value influences a ray-tracing scene comprised of a red dihedral and a cylindrical receiver surrounding the object, with the intersection points denoted by the blue dots; in scenario (a), the scene epsilon was set such that the rays would correctly reflect from a dihedral, and in scenario (b), the scene epsilon was set to a value too large for the scene and thus the second reflections were ignored, causing the rays to intersect incorrect positions on the cylinder [132]

intersect one side of the object, reflect off the intersection point and hit the opposite side, and then reflect again at the second intersection point and propagate back towards the ray source.

In scenario (a), the scene epsilon was set to a value large enough to avoid self-intersections while also being small enough to not have the engine ignore valid ray paths. Because the scene epsilon was set correctly, the ray-tracing experiment yielded the expected results and *most* of the rays returned to the source. While the use of a smaller scene epsilon enabled smaller ray lengths to still be considered valid, there were also several incorrect ray-cylinder intersections as shown along the $z = 0$ axis. These resulted from the rays that reflected off the dihedral very close to the point $(y = 0, z = 0)$, as the reflected rays at this point had very short path lengths and were thus ignored.

In scenario (b), the scene epsilon was intentionally set too large for the scene. This led to the rays hitting one side of the dihedral, and because the scene epsilon was so small, the first reflected rays were too short in path length to be deemed valid rays – and thus they have no second intersection with the dihedral and instead continue to propagate on the y axis. These incorrect reflections gave rise to the blue dots to the left and right of the dihedral.

These two experiments have thus illustrated the importance of setting the scene epsilon correctly, while also stressing the limitations of ray-tracing engines in resolving ray reflections close to infinitesimally small corners. The variables should thus be carefully selected based on the end-user's desired use-case and should account for aspects such as target range and size, as well as shape complexity.

Computational Expense

One final consideration when conducting ray-object intersection tests is computational expense in both runtime and system memory; even though ray tracing is inherently parallelisable, its use can be very computationally expensive if every single ray is tested against every single triangle for possible intersection [123]. However, it is also desired that the number of rays should not be decreased to compensate for this. As such, the more desirable method

of reducing computation time is to use acceleration structures within the core ray-tracing algorithm, such as the BVH method (discussed in Subsection 2.5.2), which arranges mesh geometry data into a binary tree data structure.

Accordingly, when a ray is spawned, a ray-intersection test is conducted against the binary tree – as opposed to testing for intersections with every triangle. In this program, the BVH binary tree is made up of bounding boxes that enclose every individual triangle in the mesh. These boxes are then arranged into a tree structure so that when a ray is spawned, an intersection test is conducted between the ray and the parent bounding box; if a valid intersection occurs, ray intersections with child nodes are also tested until reaching a bounding box containing one – or only a few – triangles. Thereafter, the ray-triangle intersection tests are performed as normal. As such, instead of executing intersection testing that is linear in the number of triangles n_t, the BVH structure enables execution on the order of $\log(n_t)$ [138].

4.1.4 Target Modelling

Based on the discussions in the previous subsection, the SOARS target model can be defined as shown in Table 4.1, along with the associated parameters and required input units.

This introduces several new facets to the target definition, namely the addition of yaw (α), pitch (β) and roll (γ) for rotational motion, as well as the use of shape and material properties to better describe the target's mesh in the RTS scene. Additional RCS parameters are also mentioned and are discussed later in the chapter in Section 4.3.

The concepts of yaw, pitch and roll [203] represent widely-used parameterisations that give rise to rotation matrices for use in 3-D scenarios. Figure 4.8 provides a graphical definition for this set of Euler angles.

As shown in Figure 4.8, the yaw corresponds to rotations about the z-axis, pitch corresponds to rotations around the y-axis, and roll corresponds to rotations about the x-axis. The initial value and rate of change for each of these angles can therefore be set by the user for every target, allowing the objects to undergo rotational transformations around any combination of

Table 4.1: Key parameters of the target model in SOARS

Module	**Parameter**	**Units**
Target definition	TLE set	–
	RCS model	–
	Mean RCS (constant)	m^2
	Target diameter (RCS estimation)	m
	Chi-square RCS k-parameter	–
Target orientation	Starting yaw	deg
	Starting pitch	deg
	Starting roll	deg
	Yaw rotation rate	deg/s
	Pitch rotation rate	deg/s
	Roll rotation rate	deg/s
Mesh shape	Shape type	–
	Shape sizes	m
	Sphere subdivisions	–
	File definitions	–
Geometry material	Reflection coefficient	–
	Refractive index	–

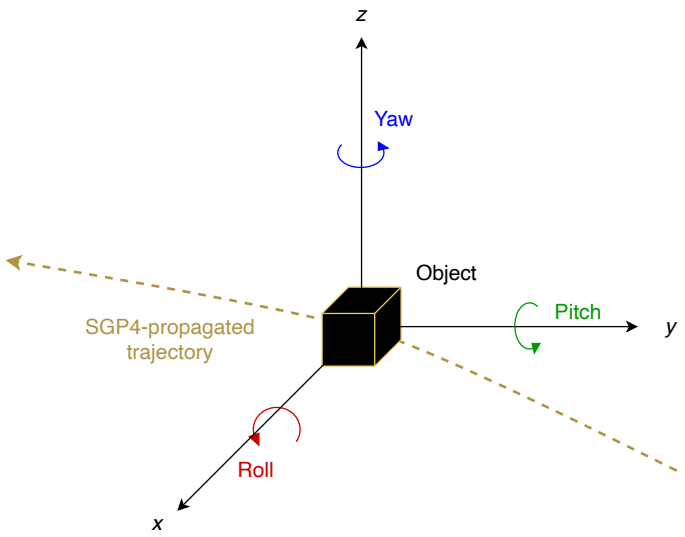


Figure 4.8: Illustration of the yaw, pitch and roll angles for an object rotating in 3-D space

axes. The yaw-pitch-roll method of rotation thus provides greater freedom of movement for targets compared to the conventional azimuth-elevation implementation used for grounded radars. The combination of this and the SGP4 model have thus served to completely model the movement of targets in SOARS.

Table 4.1 also mentions a target's mesh shape parameters. Currently, the RTS supports three main 3-D meshes, namely a cubic (or rectangular) object, a spherical object, and a file-based custom mesh (i.e., defined by user input files). The parameters for each of these mesh models are described in Table 4.2.

As depicted, the rectangular mesh requires three input parameters – the object's width, height, and depth. The spherical mesh requires a radius and a specified number of subdivisions – i.e., the number of times to recursively subdivide each of the sphere's triangular faces into smaller triangles; a greater number of subdivisions will thus generate more vertices in the mesh and a yield smoother, more spherical shape. Finally, the file mesh represents a custom, user-defined mesh shape defined by two file inputs: one that lists all the vertices of the mesh – with all the vertices of one triangle occupying

Table 4.2: Input parameters and units for three target mesh models in the RTS

Mesh Shape	**Inputs**	**Units**
Cubic/rectangular	Width	m
	Height	m
	Depth	m
Spherical	Radius	m
	Number of subdivisions	–
File (user-defined)	Vertices file input	–
	Vertex normals file input	–

one row in the file – and another file that lists the mesh's vertex normals in the same format. This allows the user to define advanced target meshes and model realistic objects in the RTS.

Both the rectangular and spherical meshes are implemented in the RTS using a predetermined set of vertex coordinates and triangle matrices that model a unit cube or unit sphere. For a rectangular mesh, these vertices are then expanded to accommodate the object's size parameters, and are then rotated based on the yaw, pitch and roll rates assigned to the object. The resulting vertex coordinates are then used to determine the face normals, which are used in place of vertex normals for this mesh shape. This is because the rectangular mesh has no curvature and hence requires no surface normal interpolation.

For the sphere mesh, the initial vertex and triangle matrices are used as part of the subdivision process, producing a mesh with the specified vertex resolution. From there, the vertices (which also represent the vertex normals in the case of a sphere) are rotated using a rotation matrix based on the yaw, pitch and roll rates. Finally, the vertex coordinates are scaled based on the input radius.

The process for a file mesh is much more streamlined; the initial vertices and

vertex normals are read into the program directly from the input files – where each line corresponds to the three vertices of a single triangle. The resulting matrices are then simply rotated in the same way as for the rectangular and spherical meshes, and the results are saved for each time instance (assuming time-varying rotations occur).

Irrespective of a target's shape, its rotated vertices are computed using a simple rotation matrix R as shown in Equations 4.3 and 4.4.

$$\begin{bmatrix} x_{rot} \\ y_{rot} \\ z_{rot} \end{bmatrix} = R \begin{bmatrix} x_{orig} \\ y_{orig} \\ z_{orig} \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} x_{orig} \\ y_{orig} \\ z_{orig} \end{bmatrix} \tag{4.3}$$

$$R = R_z R_y R_x = \begin{bmatrix} c_1 & -s_1 & 0 \\ s_1 & c_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_2 & 0 & -s_2 \\ 0 & 1 & 0 \\ s_2 & 0 & c_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_3 & -s_3 \\ 0 & s_3 & c_3 \end{bmatrix} \tag{4.4}$$

where $c_1 = \cos(\alpha)$, $c_2 = \cos(\beta)$, $c_3 = \cos(\gamma)$, $s_1 = \sin(\alpha)$, $s_2 = \sin(\beta)$, and $s_3 = \sin(\gamma)$.

Table 4.1 also lists parameters for a target's material properties. These parameters include the reflection coefficient Γ and the refractive index of the target, each of which is used in the closest-hit program to determine the properties of the reflected and refracted rays (if they exist). As will be detailed later in the chapter, Γ affects the power of both reflected and refracted rays; the refractive index of each target, however, affects the *direction* of a refracted ray that transits through (or out of) an object or medium.

With these aspects defined, the target model used in SOARS can be described by the class diagram provided in Figure 4.9. This depicts a simplified diagrammatic breakdown of various classes associated with the target class in SOARS, as well as many of their methods and attributes and the relationships between these classes.

The diagram in Figure 4.9 thus provides a functional overview of how the

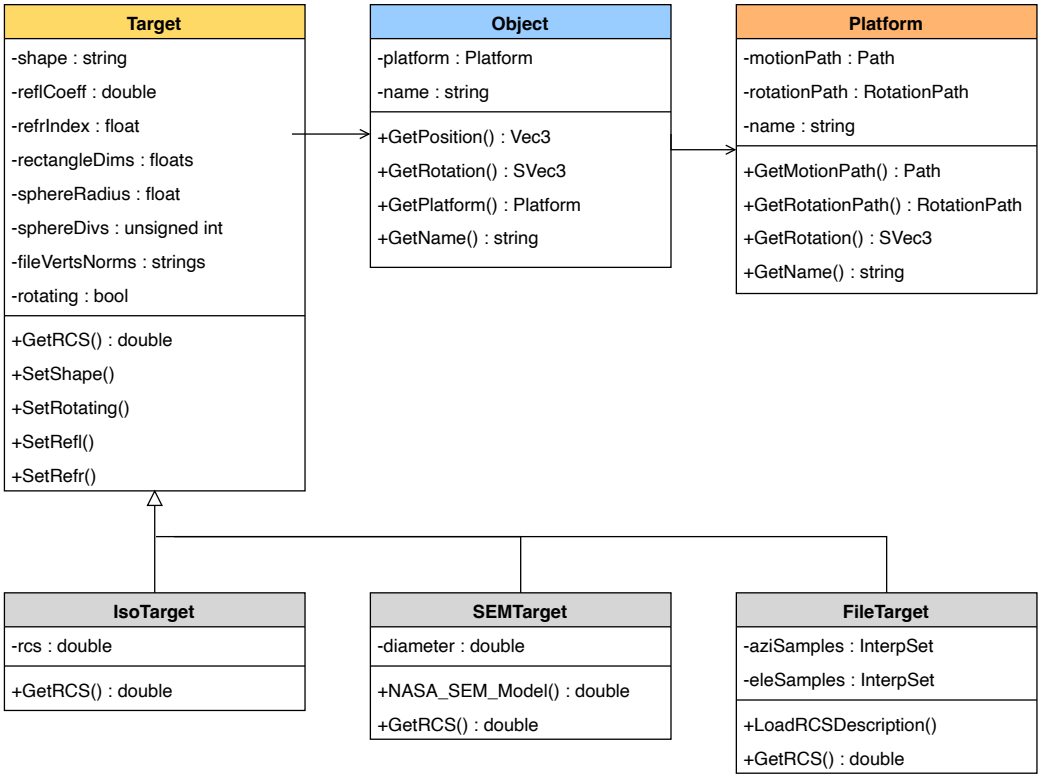


Figure 4.9: Class diagram for the target class as used in SOARS, as well as the main associated classes and all integral methods and attributes

target model – with all its complexities – was implemented in the software, and highlights the links between these classes as well as the return type of their functions. This also serves to demonstrate how multiple aspects of the target model are utilised in SOARS through the RTS, including the mesh shape parameters described in Table 4.2, rotational motion variables, object material properties, and RCS modelling aspects (as detailed in Subsection 3.5.2).

It is also worth noting that no size restrictions are placed on any of these classes, and the user is able to define as many targets (as well as transmitters and receivers) as desired; this is in keeping with the user requirements as set out in Section 3.2.

4.1.5 Ray Capture

Whenever a ray intersects a triangle, its length and the number of intersections it had undergone (i.e., its intersection depth) are updated; the ray then continues to propagate until a specified end criterion is met. In the RTS, this end criterion is when (a) the ray exceeds the maximum allowable number of intersections (measured as the reflection depth), (b) the ray hits the surface of the Earth, or (c) the ray hits a receiver's antenna. If a ray hits the Earth or exceeds the maximum depth, it is forced to miss any further reflections or refractions and is thus deemed invalid. As a result, the RTS is mostly concerned with rays that directly intersect the receiving antenna at some point during runtime. If this happens, the end coordinates of the ray are computed.

The RTS models receive antennas as spherical surface patches using segments of a sphere, defined through user-specified azimuth and elevation ranges. This is illustrated in Figure 4.10.

In this configuration, the RTS model of the receiver is defined by four main variables: the radius of the sphere used to model the antenna as a surface patch, the centre coordinates of the receiver sphere, and the angular span of the surface patch in both the azimuth and elevation dimensions. These spans are used to define the angular sizes of the surface patch relative to the inverse

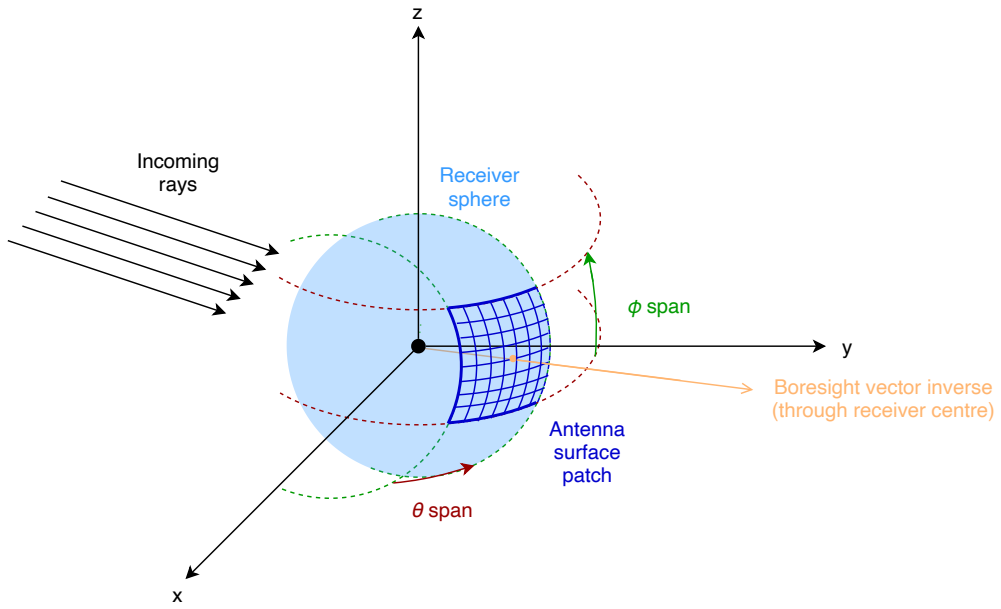


Figure 4.10: Illustration of a 3-D receiving antenna modelled in the RTS by a spherical surface patch

of the receiver's boresight vector. These angular spans, therefore, contribute to modelling the receiving antenna's size, and any rays intersecting the receiver sphere in the enclosed angular region will be captured and aggregated. The sphere radius and the two angular spans thus serve as further receiver model parameters in addition to those originally presented in Table 3.2.

However, it is also important to consider the exact coordinates of the ray's intersection with the surface patch of interest. In general, ray-sphere intersections are computed using basic geometry. In the RTS, this is achieved using a ray vector $\vec{R}$ and the equation for a sphere of radius r centred at (c_x, c_y, c_z):

$$\vec{R} = \vec{o} + t\vec{d} \tag{4.5}$$

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0 \tag{4.6}$$

Substituting $\vec{R} = (R_x, R_y, R_z)$ as (x, y, z) into Equation 4.6 leads to the quadratic equation $At^2 + Bt + C = 0$, with the coefficients defined as:

$$A = d_x^2 + d_y^2 + d_z^2 \tag{4.7}$$

$$B = 2(d_x(o_x - c_x) + d_y(o_y - c_y) + d_z(o_z - c_z)) \tag{4.8}$$

$$C = o_x^2 + o_y^2 + o_z^2 + c_x^2 + c_y^2 + c_z^2 - 2(c_x o_x + c_y o_y + c_z o_z) - r^2 \tag{4.9}$$

The value of t can then be easily computed as the quadratic roots given by:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A} \tag{4.10}$$

A valid intersection is deemed to have occurred if either value of $t \geq 0$. If both roots are valid, the ray is assumed to intersect the sphere *twice* and thus either one (or both) could have hit the receiver; as such, both roots need to be considered in the capture process.

Additionally, as shown in Figure 4.10, the inverse vector of the receiver boresight plays an integral role in these computations; logically, it is desired that the receive antenna opens outward towards potential rays directed towards it. As such, the actual boresight of the receiver is directed in the opposite direction of the inverse vector, while the centre coordinates (indicated in orange in the illustration) represent the exact position of the receiver system – as communicated to the RTS from the original SOARS inputs.

The surface patch is thus created from this point in spherical coordinates, where the minimum and maximum θ and ϕ values of the antenna are found by simply adding and subtracting half of the respective angular span. The RTS will then automatically orientate the surface patch so that it faces outward along its boresight direction. Any intersections between a ray and this patch are then processed directly in the miss program, and the relevant ray parameters (such as the receive status and ray power) are updated accordingly.

This approach is, theoretically, the most realistic in terms of physical modelling as the signal propagation model is largely based on ray optics and reflection

via Snell's Law. However, in many circumstances, this method of ray capture could also result in zero received rays depending on the scene geometry, object orientations, and transmitter properties; some scenarios may thus produce no received signal at all if no rays are reflected towards (and captured) at the receiver.

The alternative would be to automatically capture *any* rays that propagate through both a transmitter's beam and a receiver's beam, but this approach would pose several questions:

1. At which point is a ray's end criterion reached while travelling within the beams?
2. What happens to a ray that exits the beam overlap and then re-enters it at a later stage in the simulation?
3. How would the RTS account for a ray that travels within the Tx-Rx beam overlap but is shadowed by a target that breaks its line of sight with the receiver?
4. What would the range R_R be computed as (considering that there may not be a direct target-to-receiver path along which the ray travels)?
5. What physical principles would govern the ray's capture if Snell's Law is not being applied between targets and receivers?

Without any valid solutions to these problems, it was decided that the ray capture approach would remain as discussed earlier. In this way, ray directions can easily be governed by optics principles and can be definitively captured or missed by receivers. Once all rays have reached their end criterion, all the rays' properties are written to the output buffer for post-processing when control is returned to the host.

4.2 Ray Modelling

This section details the design and development of the core RTS algorithm on an application level. This includes overviews of its inputs and outputs, its implementation using the OptiX$^{\text{TM}}$ engine, the integration of its models into

the existing SOARS package, and the flow of information between the host and device as part of the complete simulator program.

4.2.1 Inputs and Outputs

The software structure of the RTS-enabled SOARS was originally shown in Figure 4.2. This depicted a high-level overview of how variables are transferred from SOARS to the RTS function on the host, which then initialises the OptiX™ engine and all associated variables and launches the ray-tracing kernel. Thereafter, outputs of the kernel are passed back to the host through an output buffer and the results are post-processed inside the host-based RTS function; control is then returned to the SOARS algorithm for processing.

By itself, the RTS requires multiple inputs at runtime – each of which is either read into SOARS at program initiation or computed directly within the software during execution. The main inputs are as follows:

- All simulation parameters from SOARS, such as the simulation time and target sampling interval.
- The world model, created in SOARS, which provides information on every object to be simulated.
- The number of rays to be launched in the x, y and z dimensions.
- The maximum number of reflections and refractions allowed for each ray – also known as the maximum reflection and refraction "depths".
- The scene epsilons for incident and reflected rays. These are both used as 5×10^{-3} by default and are set in the RTS header file.

After kernel processing on the device, the main outputs of the RTS are as follows:

- The power of each ray
- Each ray's observed Doppler frequency
- The propagation time delay experienced by each ray
- The total noise temperature of each receiver

- The ray-target intersection path traced by each ray, i.e., the order in which each ray intersected targets

Once the RTS function concludes, many of these outputs are written into the response file for each receiver as part of the usual baseline SOARS algorithm. These RTS outputs are then also used internally to compute the output signal and write the outputs to an output file for each receiver. This bridges the gap from the RTS back to the baseline SOARS program.

On its own, the RTS module is a singular function that makes use of several files and external dependencies, all of which are compiled together via CMake, NVCC and the C++ compiler *g++*. Ultimately, however, the RTS is independent of the SOARS-specific context. In essence, the RTS could be seen as a powerful, portable, independent ray tracer that could be integrated into various generic radar simulators. With a few adjustments, the program could even be adapted to work within the context of FERS – the designer of which had already previously cited ray tracing as a worthwhile future endeavour [25]. In this way, the RTS could be seen as a separate program relative to SOARS, despite being part of the same overall software package presented in this work.

4.2.2 Program Set-up

As part of the program set-up shown in Figure 4.2, the RTS requires that a node graph is created. As a framework, OptiX™ enables the user to assemble the user-editable programs in a hierarchical node graph. The following main nodes are provided by the software: context, geometry, geometry group, geometry instance, material, acceleration, and transform.

The RTS makes use of several of these nodes, namely the context, geometry group, geometry instance, geometry, and material nodes as depicted in the node graph in Figure 4.11 (adapted from [132]). This corresponds to the host-side code for the program set-up as shown in Figure 4.2.

As shown in Figure 4.11, the RTS makes use of one instance of each of the ray-generation, miss, intersection, bounding-box, and closest-hit programs. Additionally, the RTS utilises an acceleration structure to decompose scene

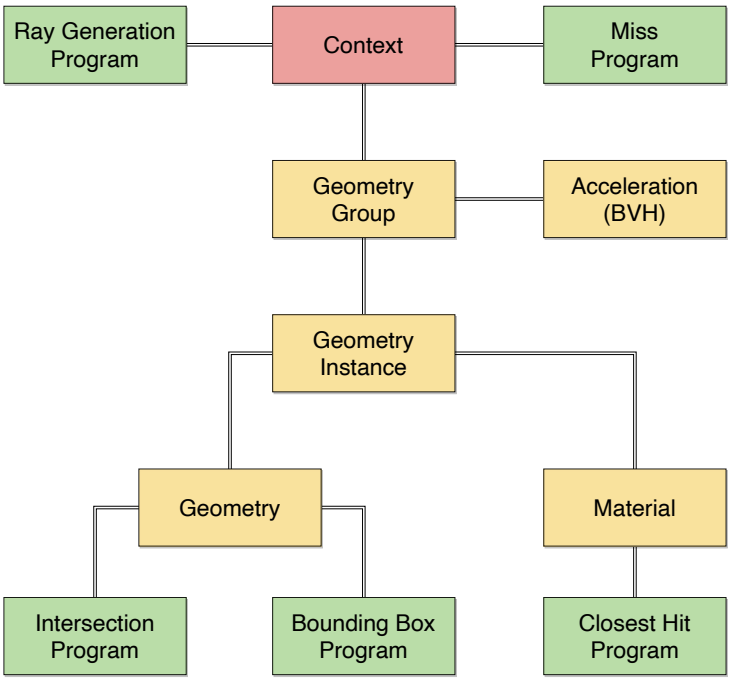


Figure 4.11: Node graph assembled for the RTS, where the yellow boxes represent the nodes, the green boxes represent user-editable programs, and the single red box represents the context node

geometries through BVH methods and thus accelerate the search for ray-geometry intersections [204]. If desired, the RTS node graph could easily be adapted for other applications through the inclusion (or exclusion) of nodes, or even by using multiple instances of some of the user-editable programs. This could theoretically be used to develop more complex systems, but may also run the risk of reducing overall performance.

From the node graph, it is evident that the context node is the most vital. This is because OptiX™ makes use of a "context" structure as the container for all information and operations relating to the simulated scene, and this node is thus responsible for managing object creation and destruction, loading programs, and managing acceleration structures and OptiX™ resources [138]. The geometry instance and material nodes are also crucial; the former is comprised of polygons or triangle meshes, while the latter determines the material properties associated with the corresponding geometry instance and thus affects ray-geometry interactions [204].

In OptiX™, a context object is declared with the data type *RTcontext* using the following C-based code:

Listing 4.1 C++ code to create a context node in OptiX™

```
1: RTcontext context;           // Declare context node
2: rtContextCreate (&context); // Create the context node
```

Other graph nodes can be declared in a similar way using the appropriate OptiX™ data types. As shown in Figure 4.11, each of the main nodes (context, geometry and material) has at least one program associated with it, such as the ray-generation program or the miss program. These programs are declared with the *RTprogram* data type, but they also need to be bound to their corresponding nodes; for instance, the ray-generation program is bound to the OptiX™ context by the function *rtContextSetRayGenerationProgram* as follows:

Listing 4.2 C++ code to create the ray generation program in OptiX™

```
1: // Create ray-generation program
2: RTprogram rtprog_ray_gen;
3:
```

```
4: // Bind ray-generation program to its graph node
5: rtContextSetRayGenerationProgram(context, 0, rtprog_ray_gen);
```

where the required inputs include the context and program names as well as the entry point index. The entry point parameter represents the desired index of the ray-generation program, which is typically set to zero as most OptiX™ simulations only use one ray source. However, the option exists for multiple instances of every program to be executed to allow for different ray behaviours to be simulated [205].

In the RTS, multiple transmitters and/or receivers can be considered in a single simulation as inherent in the design of SOARS; however, a design decision was made to not parallelise the processing of multiple *transmitters*. This is namely because each transmitter corresponds to an output buffer holding the ray information for millions of rays, and each additional transmitter would require its own output buffer if processed in parallel. This would lead to a significant waste of both host and device memory and could lead to a program crash if the system memory is exceeded during runtime. The concept of output buffers and ray information is covered in further detail in Subsection 4.2.3.

4.2.3 Data Transfer

One of the most vital aspects of the RTS is the data transfer between various processes running on the host or the device. As such, a structure needs to be put in place to optimise these transfers. By focusing on a single data structure to facilitate host-device transfer, software performance can be improved. At runtime, this is achieved by having OptiX™ compute, store and track all relevant information corresponding to each ray, which will then need to be post-processed after ray traversal has finished.

In particular, this approach makes use of a data structure defined as the "Per Ray Data" (PRD) struct – comprised of multiple runtime variables – which is initialised in one of the C++ header files in the RTS. The OptiX™ kernel stores the information of each ray in its own unique PRD, such that every ray's propagation information can be accessed as it traverses the simulated

scene. The PRD for each ray is then updated whenever a valid intersection is found – including an intersection with a receiver or the Earth.

It is worth noting that the PRDs need to be initialised before the ray tracing takes place; as such, the host needs to know the total number of rays that will be traced, and thus the number of PRDs to initialise as part of an output buffer for each transmitter. In general, this depends on the number of refractions allowed, as each refracted ray is treated as a new instance and requires its own PRD – and thus more memory. If refractions are allowed after a ray's initial reflection from its first-intersected target, the number of generated refracted rays can increase exponentially as demonstrated by Figure 4.12.

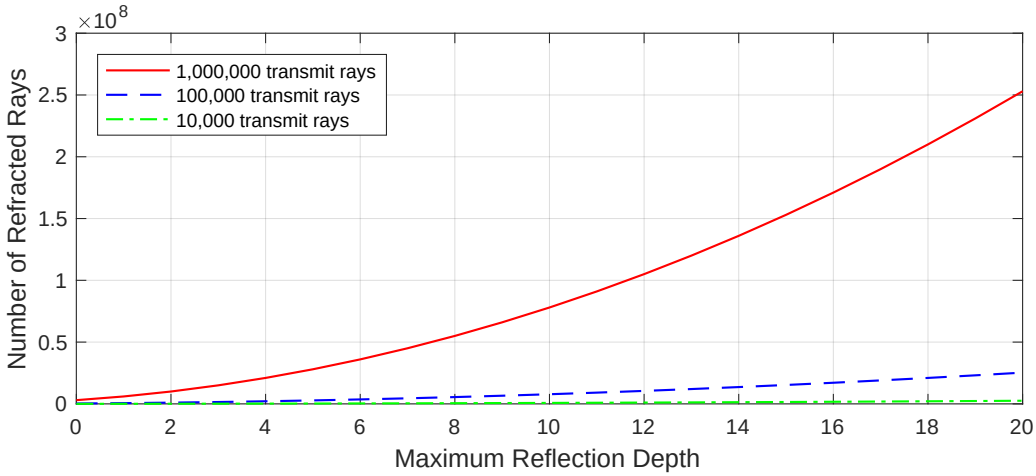


Figure 4.12: Number of refracted rays that need to be spawned (in addition to the number of original transmit rays) based on the maximum reflection depth selected by the user

With this model, a higher maximum reflection depth results in an exponentially-increasing number of refractions, each of which requires its own PRD and thus its own space in memory. Because of this, it was decided that the maximum *refraction* depth would be limited to two – resulting in only one ray refracting into an object, and then multiple rays refracting out of it based on the maximum *reflection* depth. This means that only the first ray-target intersections contribute refracted rays to the simulation; in this way, the initial dispersion of rays is modelled to contribute all of the refractive power

in the received signals, as any subsequent refractions of already-refracted rays would become increasingly negligible. This approach greatly reduces both the computation time and memory usage required for refractive modelling.

All of this leads to a large output buffer that holds the ray information for each transmitter; as such, each transmitter is processed serially to avoid the possibility of fully occupying the device's memory and crashing the program. This gives rise to an important performance consideration in the RTS, where the size of the PRD struct (in bytes) should be made as small as possible.

This is because the PRD information must be stored in *device* memory, which should be spared wherever possible, but also because the data must be communicated back to the host after ray traversal. As such, a smaller PRD size leads to better memory management and improved performance. Additionally, a greater PRD size implies a greater limitation on the number of rays that can be spawned and traced simultaneously, which restricts the sampling capabilities of the simulator.

The variables tracked in the PRD are defined as follows:

- **Ray length**: the ray's total path length. This is used to compute the final time delay and phase delay of each ray.
- **Refraction indices**: the refraction indices of the previous and current ray propagation mediums. This is used to compute the refraction ratio n_2/n_1 and thus determine if (and how) refraction occurs at an object interface.
- **Reflection depth**: the number of successful intersections between a ray and all simulated targets. Note that this is incremented even if a reflection does not occur and is used to keep track of any ray intersections.
- **Refraction depth**: the number of successful refractions that a ray has undergone; only increments when a refraction occurs.
- **Maximum ray index**: the most recent ray index that was used for a refracted ray; this is updated each time a refraction occurs and it

is used to write refracted ray properties to the correct indices of the output buffer.

- **Ray direction**: stores the latest ray direction vector of each ray as a double3 structure; provides more precision than the standard float3 used in OptiX™.
- **First hit-point**: coordinates of a ray's first hit-point (intersection coordinates); used to calculate G_T as part of the radar equation. This is discussed further later in the chapter.
- **Previous hit-point**: tracks the coordinates of a ray's previous hit-point; used to calculate target ranges as part of the radar equation. This also doubles up as the ray origin initially as it is first set to the coordinates of the transmitter under consideration; at a ray-object interface, this is updated to the coordinates of the intersection as the next ray will use this as its origin (or source coordinates).
- **Power**: the total received power of a ray; discussed later in the chapter.
- **Doppler**: the Doppler frequency observed by a ray; discussed later in the chapter.
- **Receive status**: an integer that tracks whether a ray has been received and, if so, which receiver captured it; also used to discard rays that are never captured at any receiver.
- **Termination status**: a Boolean that tracks the termination status of a ray; used to discard rays that intersect the Earth so that they cannot transit through the "Earth object" and then be captured at a receiver.

Together, these variables make up a PRD size of 129 bytes, comprised mainly of double-precision floats and unsigned integers. Depending on the application, further PRD variables may be removed or modified to improve performance (when transferring the information from the device to the host) and/or to better control memory utilisation on the device. The output buffer that tracks all PRDs will generally be the largest data structure at any point during ray tracing, with a total memory cost of (129 bytes $\times$ Total number of rays being traced) – including all refracted rays. If an RTS simulation ever fails due

to memory limitations, the simplest solution would likely be to reduce the number of rays spawned at the transmitter(s), or to simply disable refraction.

Aside from the PRD structure, the RTS also needs to retrieve many host-defined variables from the main SOARS algorithm and transfer them to the device for use within the OptiX™ engine. Multiple steps are required for a variable to be transferred to OptiX™ in this way; first and foremost, the *RTvariable* type must be used to define an OptiX™ variable. A device variable must then be created within the scope of a graph node to accommodate the inheritance of the variable. In this way, a variable may actually be defined in a global scope (such as the context) such that it will be visible to all other nodes within the context. After all this, the variable can finally be assigned a value.

This process must be followed for every variable required by the OptiX™ kernel – except for arrays, which are handled through buffers. Buffers store data in chunks of memory while an input/output device is also busy transferring that data [132]. Buffers are used in the RTS for transferring several input arrays to the device, such as the triangle vertex coordinates and vertex normals, receiver parameters, and more. One output buffer is then used to transfer the PRD results back to the host.

An example of declaring a buffer is shown in the following code, demonstrating the implementation of an input buffer for transferring all triangle vertex data to the device:

Listing 4.3 C++ code to declare a vertex buffer in OptiX™

```
// Declare the vertex buffer
RTbuffer rtbuf_vert;
rtBufferCreate (context, RT_BUFFER_INPUT, &rtbuf_vert);
rtBufferSetFormat (rtbuf_vert, RT_FORMAT_USER);
rtBufferSetElementSize (rtbuf_vert, sizeof(double3));
rtBufferSetSize1D (rtbuf_vert, verts.size());

// Declare RTS variable
RTvariable rtvar_vert;
rtContextDeclareVariable(context, "dbuf_vert", &rtvar_vert);
rtVariableSetObject (rtvar_vert, rtbuf_vert);
```

In this code sample, a buffer object with the data type *RTbuffer* and the variable name *rtbuf_vert*. The buffer object is then created within the context through the *rtBufferCreate* function, which also sets the object to be an input buffer via the *RT_BUFFER_INPUT* flag. Additionally, this particular buffer is set to be a one-dimensional array (through the *RT_FORMAT_USER* flag) that comprises variables of the type double3 – which holds three double-precision floating-point variables, corresponding to the (x, y, z) coordinates of each triangle vertex. The total size of the buffer is then set to three times the total number of triangles in the scene, as specified by the size of a host-defined vector *verts*.

As with the single variable example before, a device variable is then declared in *dbuf_vert* and it is bound to the context. Finally, the function *rtVariableSetObject* is used to assign the device buffer to the values of the triangle vertex coordinates.

To populate a buffer with values, mapping functions need to be utilised. These enable reading from (and writing to) device buffers, and can be used as shown in the example code:

Listing 4.4 C++ code to populate a vertex buffer in OptiX™

```
1: // Map buffer
2: float* hbuf_triVertices;
3: rtBufferMap (rtbuf_triVertices, (void*)&hbuf_triVertices);
4:
5: // Assign values in a for loop
6: for (int i = 0; i < verts.size(); i++)
7:     hbuf_triVertices[i].x = verts[i][0]; // Host-side array
8:
9: // Unmap buffer
10: rtBufferUnmap (rtbuf_triVertices);
```

where *verts* is a host-side array that holds all the triangle vertex coordinates for a given target mesh. This shows how values are assigned to the buffer at different indices, allowing for the same values to be accessed and used on the device. The same process can be repeated for other input buffers, and a similar approach can be taken to read from an output buffer after passing it

from the device to the host.

4.2.4 Ray Tracing Kernel

Before the OptiX™ kernel is launched from the host, the engine performs a validation test and a just-in-time compilation. The OptiX™ kernel is then launched through multiple instances of the ray-generation program, with the number of instances being computed as the product of the number of rays in width (x), height (y), and depth (z). This is achieved using the code:

Listing 4.5 C++ code to launch the OptiX™ kernel

```
1: // Launching the kernel
2: rtContextValidate (context);
3: rtContextCompile (context);
4: rtContextLaunch3D (context, 0, h_Nrays, h_Nrays, h_Nrays);
```

where the second argument of the *rtContextLaunch3D* function specifies the entry point index. In the RTS, the number of rays in width, height, and depth are set to be equal such that the ray distribution is even on all Cartesian axes. All three values are thus represented by the variable *h_Nrays*.

Launching the OptiX™ kernel begins the execution of the ray generation, miss, bounding-box, intersection, and closest-hit programs. Any variables required by these programs should have been declared and assigned values by the host for use on the device; all device variables should thus be correctly paired with their corresponding host variables at the time of launch. Several variables are also managed internally as part of the OptiX™ framework, including the launch index of the execution thread – which can thus be used as part of some computations on the device. All variables declared on the device are also directly accessible by the OptiX™ programs that use them.

The relationships between the user-editable programs are illustrated in Figure 4.13 (adapted from [30]), presenting the overall OptiX™ ray-tracing pipeline. This is comprised of both the aforementioned programs and the hard-coded internal algorithms within the engine.

The diagram shown in Figure 4.13 represents the overall OptiX™ kernel

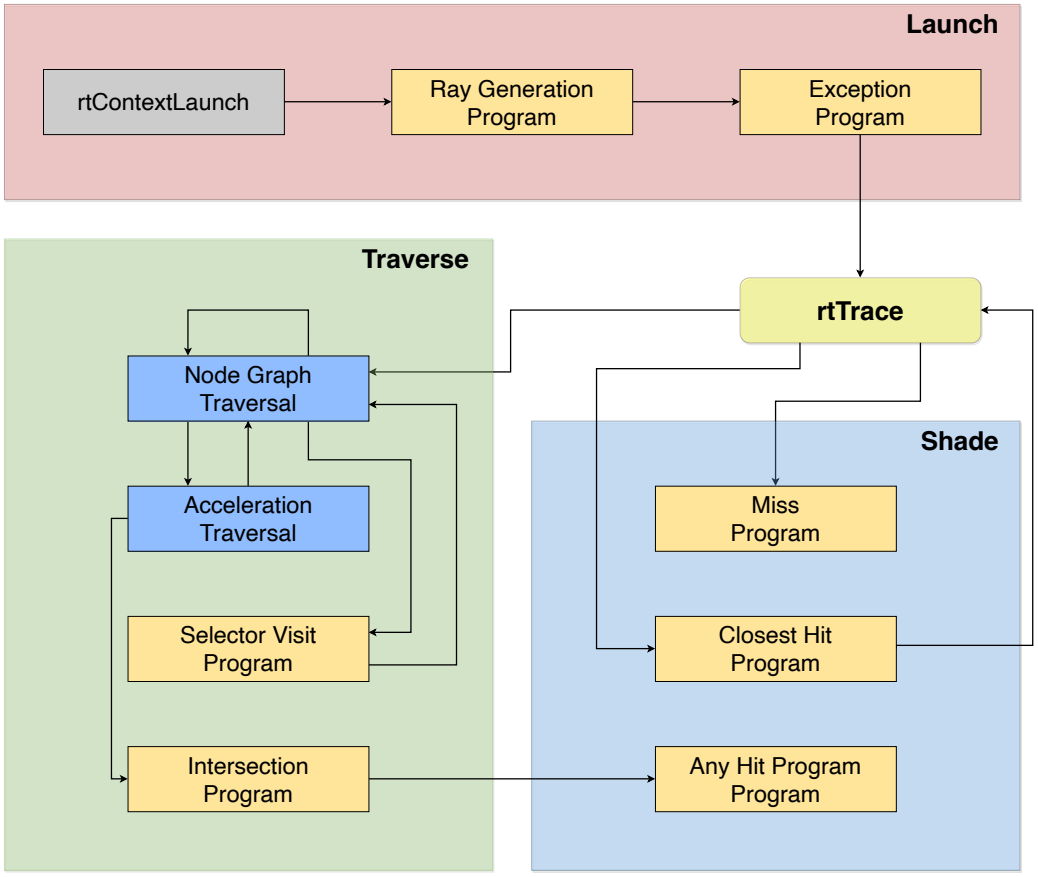


Figure 4.13: Flowchart showing the control flow through the ray-tracing pipeline in OptiX^{TM}, where the yellow boxes represent user-editable algorithms while the blue boxes represent internal programs

execution in the form of a call graph [206]. This is centred around the core operation denoted by *rtTrace*, a user-inaccessible function that spawns and directs rays (ray launching), locates object intersections (ray traversal), and responds to them (shading, in the context of graphics processing) [30]. This execution is initiated by the built-in *rtContextLaunch* function, which is used on the host to launch many instances of all the required programs.

The *rtTrace* function is used by the ray-generation program to launch rays into the simulation environment and begin the ray-tracing process. Once the *rtTrace* function has completed execution, the outputs of the process are saved as part of the end of the ray-generation program. In the RTS specifically, the *rtTrace* function is also invoked inside the closest-hit program, through which rays are recursively generated due to reflections and/or refractions occurring at the point of intersection.

The ray-generation program can thus be thought of as being equivalent to the *main* function in serial C or C++ applications – as discussed previously – as most of the processing is initiated by this program. However, this is differentiated from a typical C or C++ application through device-based parallelism, as the ray-generation program is run in parallel on multiple GPU threads [205]. Through the OptiX™ engine, all RTS code (from both the developer and internal sources) is compiled into a "megakernel" that is launched on sufficient GPU threads to fully occupy the GPU [198, 207]. If additional threads are then required for execution, they are put into a queue. In this way, OptiX™ can manage data coherency as well as efficient thread execution [198, 208].

The Ray Generation Program

The most notable OptiX™ program is the ray-generation program, which is launched on parallel device threads – one for each ray that is spawned. This program is responsible for computing the incident ray's direction vector, spawning a ray with this direction vector, initialising the PRD and the program's output buffer, and then calling the *rtTrace* function to begin ray tracing.

A PRD instance is then declared and its contents are initialised to zero by default, along with all the contents of the output buffer. After this, ray propagation will begin once the following code is invoked with the current ray, PRD and group node as its inputs:

Listing 4.6 C++ code to trace a ray in OptiX™

```
1: // Begin ray propagation
2: rtTrace(d_groupNode, ray, prd);
```

After a thread has completed its ray propagation simulation, the engine returns to the ray-generation program to store the contents of the PRD struct in the output buffer. Once all threads have completed their execution, the output buffer is transferred back to the host.

The Bounding-box Program

This program is invoked before the intersection program and is declared as follows:

Listing 4.7 C++ code to declare the bounding-box program in OptiX™

```
1: // Declare bounding-box program
2: RT_PROGRAM void bound(int prim_index, float result[6]);
```

where *prim_index* is the index of a mesh triangle that is automatically received by the program, while *result* is used as part of OptiX™'s "axis-aligned bounding box" to compute bounding boxes.

This program makes use of the triangle vertex coordinates specified for the relevant target being intersected. The coordinates for all three vertices of the triangle corresponding to *prim_index* are then computed by simply indexing the triangles buffer that is passed from the host to the device:

Listing 4.8 C++ code to retrieve vertices in the bounding-box program

```
1: // Search triangles buffer for associated vertex numbers
2: unsigned int v_idx0 = dbuf_triangles[prim_index].x;
3: unsigned int v_idx1 = dbuf_triangles[prim_index].y;
4: unsigned int v_idx2 = dbuf_triangles[prim_index].z;
5:
```

```
6: // Get the vertex coordinates
7: double3 p0 = dbuf_triVertices[v_idx0];
8: double3 p1 = dbuf_triVertices[v_idx1];
9: double3 p2 = dbuf_triVertices[v_idx2];
```

The vertices are then used in the internal OptiX™ function *aabb* – or the "axis-aligned bounding box" function – to compute a bounding box over every triangle; together, these bounding boxes are used in the construction of the BVH. This program is thus used in combination with the BVH acceleration structure to limit the possible values of *prim_index* and thus heavily reduce software runtime.

The Intersection Program

This program is responsible for performing ray-triangle intersection testing. Before this is invoked, OptiX™ uses the bounding-box program (as well as the BVH acceleration structure) to improve program runtime.

The purpose of the intersection program is to compute ray-triangle intersections via an internal OptiX™ function. This is achieved using the intersected triangle's vertices and vertex normals, which are passed to the device through an OptiX™ input buffer. The following set of equations are used to determine a few key values needed for the intersection testing algorithm:

$$\vec{e_0} = \vec{p_1} - \vec{p_0} \tag{4.11}$$

$$\vec{e_1} = \vec{p_0} - \vec{p_2} \tag{4.12}$$

$$\vec{n} = \vec{e_1} \times \vec{e_0} \tag{4.13}$$

where $\vec{n}$ is the normal, $(\vec{p_0}, \vec{p_1}, \vec{p_2})$ are the vector points of the triangle under test (each comprised of (x, y, z) vertices), and $\vec{e_0}$ and $\vec{e_1}$ are vectors. These are used in the pair of equations:

$$\vec{e_2} = \frac{1}{(\vec{n} \cdot \vec{R})}(\vec{p_0} - \vec{p_{prevHit}}) \tag{4.14}$$

$$\vec{i_n} = \vec{R} \times \vec{e_2} \tag{4.15}$$

where $\vec{e_2}$ is another vector used in the next set of equations, and $\vec{p_{prevHit}}$ is the last recorded intersection or hit-point. Finally, these are used in computing the values of β, γ, and t_n – three parameters used to determine if an intersection has occurred in OptiX™. These are computed as:

$$\beta = \vec{i_n} \cdot \vec{e_1} \tag{4.16}$$

$$\gamma = \vec{i_n} \cdot \vec{e_0} \tag{4.17}$$

$$t_n = \vec{n} \cdot \vec{e_2} \tag{4.18}$$

With these values calculated, a comparison check is done to confirm that:

$$\beta \geq 0, \gamma \geq 0, (\beta + \gamma) \leq 1 \tag{4.19}$$

Additionally, a basic check is performed to ensure that the intersecting ray's length falls within the allowed values, namely the scene epsilon (ϵ_i or ϵ_r) and the maximum allowed ray length. If all of these criteria are met, the *rtPotentialIntersection* function is called with t_n as its argument, and if this final check is passed, the closest-hit program is invoked and the (closest) intersection is deemed to be successful [30].

The Closest-hit Program

This program is invoked once the intersection program finds the closest intersection and is used for multiple purposes, namely to compute the coordinates of the point of intersection, update the ray's PRD (including ray length, refractive index, last hit-point, power, Doppler, and more), and spawn new

rays at the point of intersection. The entire reflection and refraction processes are therefore relegated to the closest-hit program. This program is declared in the same way as the ray-generation program.

If the ray has not yet exceeded the maximum depth of reflection, it will reflect off the intersected target in a new direction and its origin will be updated to the point of intersection. The program will then recursively invoke *rtTrace* so that the new, reflected ray can propagate independently and intersect other targets, the Earth, or the receiver. Similarly, rays that have not yet exceeded the maximum refraction depth will spawn an entirely new ray at the point of intersection – which will then propagate through the intersection interface and travel independently of the original ray via a recursive invocation of *rtTrace*.

Details on the inner workings of this program are presented in Section 4.3, which documents all the computations performed as part of the RTS. Many of these computations are reliant upon the closest-hit program since this is where most of the target-based calculations are performed.

The Miss Program

The miss program is used for rays that "miss" the scene. This is a loose definition as "missed" rays include any rays that never intersect the scene at all, but the term also includes rays that reflect/refract through the scene until they can no longer do so. This program is therefore used whenever the intersection program finds that a ray has no further intersections with the scene – or none at all. This program is declared in the same way as the ray-generation and closest-hit programs.

This program is also responsible for performing ray-sphere intersection tests between "missed rays" and receiver sphere(s); if a valid intersection is found, the PRD fields are updated with the necessary ray properties and the ray is considered to be captured at the receiver. However, unless the ray specifically hits the receive antenna's surface patch, the ray's data is ignored at the output.

Additionally, if a ray hits the surface of the Earth – also modelled as a sphere

but centred at the origin – the ray is discarded entirely and is deemed to be invalid. Similarly, any rays that meet their end conditions without hitting the receiving antenna are also treated as invalid. These invalid rays are filtered out of the results by simply not being captured by any receiver.

After the OptiX™ kernel has completed its execution, the captured rays' information needs to be processed on the host; control is thus passed back from the device to the host, at which point program clean-up occurs. Some computations need to be performed on the host's side as not all simulation information is available on the device – only specific data is copied from host to device memory to use the GPU's memory more sparingly.

As part of the RTS clean-up shown in Figure 4.2, the contents of the PRD structures are thus copied to the host via an output buffer, which can then be further processed. Some of these computations are described in Section 4.3 and are used to determine each receiver's response (and the observed signal) based on the rays that it had captured. Thereafter, the RTS finally makes use of several destruction functions to destroy OptiX™ structures, buffers and programs and thus free allocated resources.

4.3 Ray Computation

This subsection serves as an extension of Subsection 2.4.3, documenting the changes introduced into various radar equations to accommodate the ray-tracing model. Additionally, details are presented on the computation of ray directions after undergoing reflection and/or refraction. These mathematical models are used to directly compute the parameters of the output signals as well as the observed responses at each receiver in the simulation.

4.3.1 Multiscatter Power

Equation 2.7 modelled the received power for a multistatic radar configuration and accounts for target RCS, path loss, and more. However, this equation only holds for single-scatter cases, where only one target is being considered in the configuration. For *multiscatter* cases comprised of N targets, a generalised

equation can be iteratively evolved (based on the derivation in [25]) as follows.

Consider the power that would be observed at the first target illuminated by a transmit signal:

$$P_1 = \frac{P_T G_T \sigma_1}{4\pi R_{T1}^2} \tag{4.20}$$

where P_T is the transmitted power, G_T is the transmitter gain in the direction of the first target, σ_1 is the RCS of the first target, and R_{T1} is the range between the transmitter and the first target. This is derived from the fact that the power density at a distant point away from the radar is modelled via an isotropically-radiating antenna, and thus the power decreases with respect to the squared range. The target would then re-radiate this power in a new direction, as determined by the ray-tracing model and the effects of reflection and refraction.

If another target is illuminated by this re-radiated power, the observed power at the second target would be:

$$P_2 = \frac{P_1 \sigma_2}{4\pi R_{12}^2} \tag{4.21}$$

where σ_2 is the RCS of the second illuminated target and R_{12} is the range between the first and second targets. The power would also be scaled by Γ if reflected or $1 - |\Gamma|$ if refracted. Based on the previous two equations, the observed power at an N^{th} target in a multiscatter path would be:

$$P_N = P_1 \prod_{k=2}^{N} \frac{\sigma_k}{4\pi R_{(k-1)(k)}^2} \tag{4.22}$$

With all N targets having been accounted for in Equation 4.22, the final step is to consider the power observer at the receiver:

$$P_R = P_N \frac{G_R \lambda^2}{4\pi} \frac{1}{4\pi R_{NR}^2} \tag{4.23}$$

where G_R is the receiver gain in the direction of the N^{th} target, R_{NR} is the range between the receiver and the N^{th} target, and λ is the radar wavelength. Also, note that $(G_R\lambda^2)/(4\pi)$ represents the receive antenna's effective capture area.

This leads to the complete expression for the received power along a multi-scatter path with N targets:

$$P_R = \left(\frac{1}{4\pi}\right)^{N+2} \frac{P_T G_T \sigma_1}{R_{T1}^2} \frac{G_R \lambda^2}{R_{NR}^2} \prod_{k=2}^{N} \frac{\sigma_k}{R_{(k-1)(k)}^2} \tag{4.24}$$

Note that this power also needs to be scaled by Γ for any reflection and $1 - |\Gamma|$ for refractions.

Additionally, the received power needs to be appropriately decreased to compensate for the increases experienced due to multiple rays being used. In most radar simulators [25, 48], one "ray" is mapped directly to each target and a maximum power of P_R can be received; in the RTS, however, the user decides the maximum number of rays X, leading to a maximum received power of $X \times P_R$. This would be highly unrealistic and could theoretically imply that more energy is received than what was transmitted.

To account for this, the received power needs to be scaled based on the number of rays that are traced along the same paths (i.e., interacting with the same targets in the same order); these rays thus need to be aggregated before signal rendering occurs. This is achieved via a ray aggregation method as detailed later in the section.

To account for all of this in the RTS, several steps need to be taken:

1. Initialise each ray's power to zero before launch.
2. At the first intersection of each ray with a target, compute $\sigma_1/(4\pi R_{T1}^2)$ and store the result in the ray's PRD. At the k^{th} target intersection for k 1, multiply the power stored in the PRD by $\sigma_k/(4\pi R_{(k-1)(k)}^2)$.
3. For *any* intersection, the target's index is appended to a ray path vector. This is used to track the order in which each ray intersects any targets.

4. If the ray is reflected, the power is multiplied by Γ. If the ray is refracted instead, the power is instead multiplied by $1 - |\Gamma|$ and the ray's path vector (the order in which targets are intersected by each ray) is updated to account for refracted rays spawned by collisions internal to the target.

5. If the ray intersects the receive antenna, multiply the power stored in the PRD by $1/((4\pi)^2 R_{kR}^2)$, where k is the index of the last target to be intersected before the ray was captured at the receiver.

6. After ray traversal, pass the ray path vectors from device to host, then iterate through each received ray's PRD, compute P_T, λ, G_T and G_R, and multiply the stored PRD power by $P_T G_T G_R \lambda^2$.

7. The ray aggregation kernels are then called to process ray aggregation on the GPU and scale the ray power by a factor of $1/(N_{path_i}^2)$.

8. This yields the final value of P_R, which is then recorded and saved as part of the receiver's response. Similarly, the scaled amplitudes are used to compute the actual received signal that is outputted to the HDF5 format.

This results in the final value of P_R for each ray, which can then be used to compute the return signal and printed as outputs of SOARS. However, in the case of direct transmission between a transmitter and receiver, the P_R equation is simplified to:

$$P_R = \frac{P_T G_T G_R \lambda^2}{(4\pi)^2 R_{TR}^2} \tag{4.25}$$

where R_{TR} is the range between the transmitter and receiver. If the line-of-sight between the transmitter and receiver does exist and does not intersect the Earth model, P_R is computed using Equation 4.25. This is achieved by first computing $1/(4\pi)^2 R_{TR}^2)$ at the receiver during ray traversal, and then multiplying the result by $P_T G_T G_R \lambda^2$ on the host side afterwards.

Note that all ranges are computed using the points where the ray intersected the target or receive antenna – not the centre coordinates of each object.

4.3.2 Ray Delay and Phase

Both the time delay τ and the phase delay θ_d are unaffected by the RTS in terms of implementation, except for one change: the ranges R_T and R_R are computed using the exact ray hit-points as opposed to the centres of the receiver and intersected targets. This entails saving the ray length in the PRD and updating its value each time the ray intersects a target.

If the ray hits a receiving antenna, its total length at that point of intersection is used to compute τ:

$$\tau = \frac{R_{RL}}{c} \tag{4.26}$$

where R_{RL} is the total ray length summed at the receiver intersection point.

From this, the phase delay θ_d is simply computed using Equation 2.10.

4.3.3 Multiscatter Doppler

Based on the discussions in [209], the measured Doppler frequency for a multiscatter case can be derived as follows.

Consider a generic multiscatter case with N targets, a transmitter and a receiver. The transmitter launches a single incident ray with unit direction $\hat{k_0}$ and every j^{th} target moves with a velocity v_j. Upon impact, the incident ray's direction changes to $\hat{k_j}$ based on either reflection or refraction principles. If this second ray intersects another target, it scatters again in a new direction, and this process can repeat indefinitely until the ray eventually intersects the receive antenna (assuming it does so at all). This is depicted in Figure 4.14 for a simple multiscatter configuration with two targets.

The diagram in Figure 4.14 illustrates a transmitter (**Tx**) spawning a ray in the direction of a receiver, but this does not affect the observed Doppler since there is no relative motion between the transmitter and receiver. Another ray is spawned towards a target (**Tgt1**), and it is then re-radiated in the direction of a second target (**Tgt2**); thereafter it is redirected towards the receiver. Each of these ray-target intersections results in a scattering event that produces

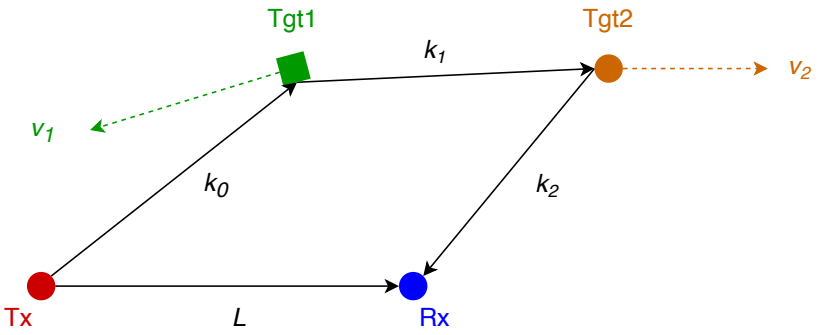


Figure 4.14: Theoretical depiction of ray propagation for a multiscatter case with two targets, a transmitter, and a receiver

a Doppler shift dependent upon the target's velocity component along the direction $\hat{k_j} - \hat{k_{j-1}}$. Summing these contributions results in the observed Doppler frequency as computed by Equation 4.27. Further information is available in [209].

$$f_d = \frac{1}{\lambda}\left(\sum_{j=1}^{N} \vec{v_j} \cdot (\hat{k_j} - \hat{k_{j-1}})\right) \tag{4.27}$$

This can be seen as an extension of Equation 2.15, where the summation in Equation 4.27 is equivalent to $2V_r$. A more accurate calculation would thus be to compute V_r from this equation, then insert it directly into Equation 2.12 to get the received frequency, and thereafter f_d can be computed using Equation 2.11 for each ray.

This represents a modified version of the equation derived in [209], as the approach in Equation 4.27 assumes the convention that a receding target corresponds to a negative Doppler frequency. This equation also holds for every combination of single-scatter or multiscatter cases and monostatic or bistatic radar configurations. For instance, in a single-scatter monostatic approximation, Equation 4.27 reduces to:

$$\begin{aligned} f_d &= \frac{1}{\lambda}\vec{v_1} \cdot (-\hat{k_0} - \hat{k_0}) = \frac{1}{\lambda}\vec{v_1} \cdot (-2\hat{k_0}) \\ &= \frac{-2}{\lambda}|\vec{v_1}||\hat{k_0}|cos(\theta_{dp}) = \frac{-2}{\lambda}|\vec{v_1}|cos(\theta_{dp}) \end{aligned} \tag{4.28}$$

where θ_{dp} is the dot-product angle between the vectors $\vec{v_1}$ and $\hat{k_0}$ (when both vectors start at a common point). This is derived from the fact that no $\hat{k_1}$ vector exists, so the $\hat{k_j}$ term instead corresponds to the direction vector of the ray returning from the target towards the radar, i.e., $-\hat{k_0}$. If the target recedes from the radar, the angle of the dot product would be smaller than $\pi/2$ rad and thus the overall f_d would be a negative value. Similarly, an approaching target would yield a θ_{dp} larger than $\pi/2$ rad, and so the Doppler frequency would be positive. Equation 4.28 thus agrees with the standard Doppler frequency equation for single-scatter cases as was shown in Equation 2.15.

This is implemented in the RTS through the algorithm that follows:

1. Before ray traversal, calculate the positions of each target at every time sample under consideration in the simulation. At every time sample, except the final one, the j^{th} target's velocity vector is determined by:

$$\vec{v_j} = f_s((x_1, y_1, z_1) - (x_0, y_0, z_0)) \tag{4.29}$$

   where f_s is the sampling frequency, (x_1, y_1, z_1) is the target's position at the next sample, and (x_0, y_0, z_0) is the target position at the current sample. These velocities are then copied from the host to the device via a buffer, ready for use in the GPU-bound closest-hit program.

2. Initialise each ray's Doppler to zero before launch.

3. When the ray intersects target j, add $\vec{v_j} \cdot (\hat{k_j} - \hat{k_{j-1}})$ to the Doppler stored in the PRD – where $\vec{v_j}$ is the target's velocity, k_{j-1} is the ray's current direction, and k_j is the newly-computed ray direction resulting from refraction or reflection. The PRD Doppler value is then updated.

4. After ray traversal, iterate through each received ray's PRD on the host and divide the stored PRD Doppler by 2 to determine V_r. This is then inserted into Equation 2.12 and f_d can finally be computed using Equation 2.11. The value of f_d is then recorded for that ray and saved as part of the receiver's response.

This process results in the final calculated value of f_d for each ray, which can then be printed as an output of SOARS through its XML responses file(s).

4.3.4 Re-radiation

One of the most critical aspects of ray tracing is determining the *directions* of re-radiated rays after intersecting an object in the scene. Additionally, it is also important to track the order of targets intersected by each ray such that the amplitude of rays with common paths can later be averaged. This is achieved by updating an output buffer with the index of the target that was intersected – whenever an intersection occurs.

As for refracted ray directions, these are determined through an implementation of the pseudocode:

Listing 4.9 Pseudocode to spawn and trace refracted rays in OptiX™

```
prd_refr = prd                    // Copy current ray's PRD
prd_refr.n1 = prd_refr.n2         // Update with previous n

if ray.refractionDepth < maxRefractionDepth then
    if prd_refr.n1 is 1 then
        prd_refr.n2 = targetRefractiveIndex
    else then
        prd_refr.n2 = 1

    newDir = refract(ray, n2/n1)   // Refract direction
    prd_refr.power *= (RCS*(1 - |reflCoeff|))
    prd_refr.refractionDepth++
    k1 = normalise(newDir)         // Next direction
    k0 = normalise(RayDir)         // Current direction
    prd_refr.doppler += dot(V_target, (k1 - k0))

    // Call rtTrace
    rtTrace(prd_refr)
    outputBuffer[rayIndex] = prd_refr

prd.n2 = prd_refr.n1    // Update n2
prd.n1 = prd_refr.n1    // Update n1
```

This entails creating a copy of the PRD of the original intersecting ray; this PRD copy then acts as the PRD for the newly-spawned refracted ray, which can propagate and intersect objects independently. Both refractive indices n_1 and n_2 are initially set to 1 (free space), but if a new value is set for n_2 at some point of intersection, the value of n_1 will be updated to that value at the next intersection. Thus, at the ray's first intersection with a target, the refracted ray will adopt an n_2 equal to the refractive index of the target, as this ray will propagate *through* the object. If it then intersects another side of the object, its n_1 is set to n_2 and its n_2 is set to the index for free space.

The internal *refract* function is then called to determine the direction of the refracted ray launched at the point of intersection. This, its previous direction vector, and the target velocity are then used to determine the Doppler observed for the intersected target. Both the Doppler and ray power are thus updated in the new PRD. Finally, *rtTrace* is called to launch the new ray and recursively propagate it through the scene; when this is completed, the ray's PRD data is recorded into an output buffer for host processing.

After the refraction process, the PRD for the original ray is updated based on the previous refractive index. This is because the original ray will be reflected in the medium *before* the intersection, so if the ray was previously propagating through space, the reflected ray will do the same; if the ray was propagating through a target, the reflected ray will still do the same. Computations for the reflected ray are described as follows:

Listing 4.10 Pseudocode to spawn and trace reflected rays in OptiX™

```
 1: prd.reflDepth++              // Increment depth
 2:
 3: if ray.reflectionDepth < maxReflectionDepth then
 4:
 5:     newDir = reflect(ray)    // Reflect direction
 6:     prd.power *= (RCS*reflCoeff)
 7:     k1 = normalise(newDir)   // Next direction
 8:     k0 = normalise(RayDir)   // Current direction
 9:     prd.doppler += dot(V_target, (k1 - k0))
10:
11:     // Call rtTrace
12:     rtTrace(prd)
```

Similarly to refraction, the reflection process makes use of an internal *reflect* function to determine the direction of the reflected ray at the point of intersection. The observed Doppler and power are then computed for this ray and updated in the original ray's PRD. Thereafter, *rtTrace* is called to re-launch the original in a new direction and recursively propagate it through the scene. After traversal, the ray's PRD data is then recorded into the output buffer; however, for reflected rays, this occurs at the end of the ray-generation program instead of the closest-hit program, as that is where *rtTrace* was initially called for all transmitted rays.

4.3.5 Ray Aggregation

As previously discussed, any ray's received power needs to be aggregated to compensate for the increase in power captured at the receiver based on the number of rays being traced. In essence, this is achieved by scaling each ray's voltage by a factor of $1/N_{path_i}$, where N_{path_i} is the number of rays that followed the same intersection path as ray i. However, this concept can be extended to *all* ray quantities (as presented earlier) by averaging all rays that follow the same path – including the rays' time and phase delays as well as their Doppler shifts.

An example of a typical multi-path scenario where this algorithm is employed is depicted in Figure 4.15, where three rays are launched from a transmitter and independently reflect off two targets in the environment before being captured at a lone receiver.

In this theoretical scenario, Rays #1 and #2 follow the same target intersection path as they both hit the same targets in the same order. As a result, in the final ray computations, all of these two rays' quantities each need to be averaged, i.e., scaled by a factor of $1/(1+1) = 1/2$, as part of the aggregation process. Ray #3, meanwhile, is the only ray following its unique path, so the properties of this ray are unaffected by aggregation, i.e., its properties are scaled by a factor of $1/1 = 1$.

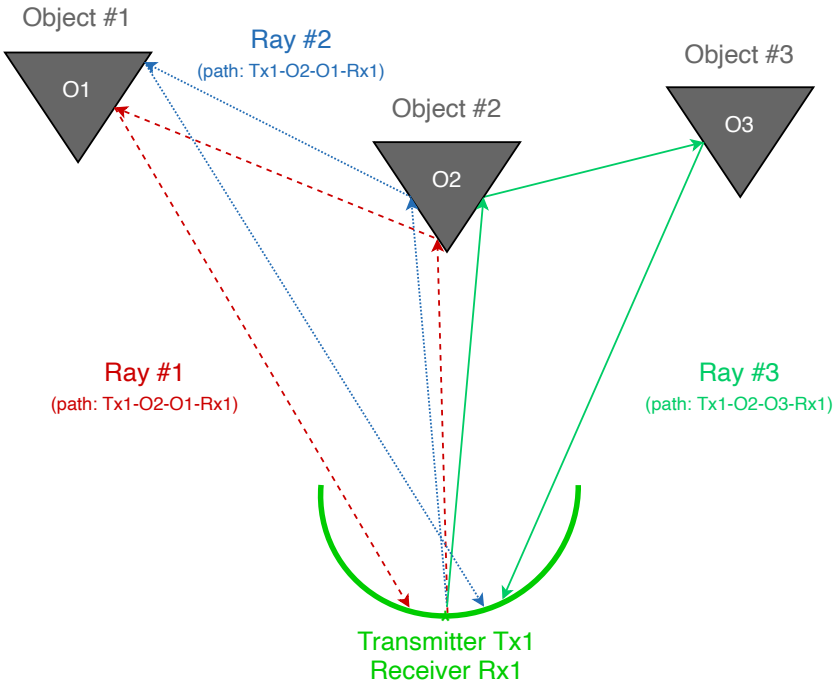


Figure 4.15: Side view of a theoretical multi-target scenario in the RTS showing two duplicate ray paths

Ray aggregation in the RTS is implemented across two kernels that run on the device via CUDA. This is done *after* ray tracing has finished, as summarised by the processing method shown in Figure 4.16. This flowchart also depicts the procedure involved in averaging each ray's quantities by N_{path_i}, finding each unique ray traversal path, and then rendering the output signal using responses generated for those paths.

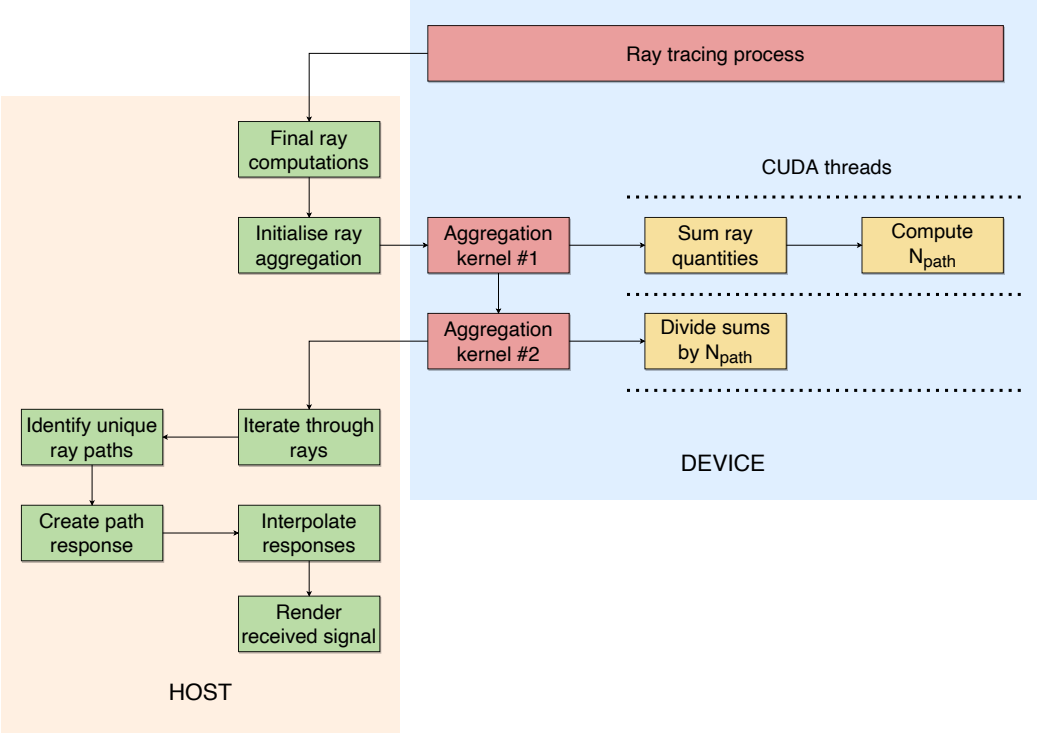


Figure 4.16: Flowchart showing the ray power computation process used in the RTS, including the path tracing algorithm

Specifically, after ray tracing is completed, two CUDA-based aggregation kernels are used to iterate through all rays in parallel and identify common paths. The first kernel is thus responsible for computing the value of N_{path_i} for each ray i as well as summing the power, Doppler, and time and phase delays across all rays with common paths. A second kernel is thereafter used to divide these summations by N_{path_i} and complete the averaging process. This is done via a separate kernel after the first has completed its processing to synchronise all threads and prevent cross-dependence when computation matrices are updated in the summation process.

Aggregating all common-path rays (and identifying unique paths among them) in this way requires updating the path order for each ray across thousands of threads, leading to an extensive process with significant overhead. However, this is necessary to ensure high accuracy in the signal-level implementation and to correctly "average" rays that trace the same path. This aggregation mechanism thus represents a novel technique in ray-traced computations on a signal level, constituting one of the contributions of this work.

With these computations defined, the RTS can be fully integrated into SOARS. These can also directly replace the previous radar computation functions that were used in both FERS and the baseline SOARS.

4.4 Software Integration

This section describes the impact of the RTS on the original SOARS program and details the operational logic and flow of data in the developed simulator. Additionally, brief details on the execution of the software and its requirements are presented.

4.4.1 System Operation

With the additions introduced by the RTS, the SOARS block diagram in Figure 3.3 was adapted as shown in Figure 4.17.

This depicts the addition of the RTS module into the SOARS program, where various parameters and simulation data are passed to the RTS function to set up the OptiX™ kernel; thereafter, the kernel invokes multiple instances of the ray generation program across thousands (or millions) of GPU threads. This leads to the bound, intersect and closest-hit programs being invoked to compute intersections between rays and targets. Finally, the miss program is called when an intersection is detected between a ray and the Earth or a receiver. As was previously shown in Figure 4.2, program clean-up then occurs and control is passed back to SOARS on the host.

This shows how the main program inputs are still transformed into two main outputs by SOARS: the HDF5 file holding the in-phase and quadrature

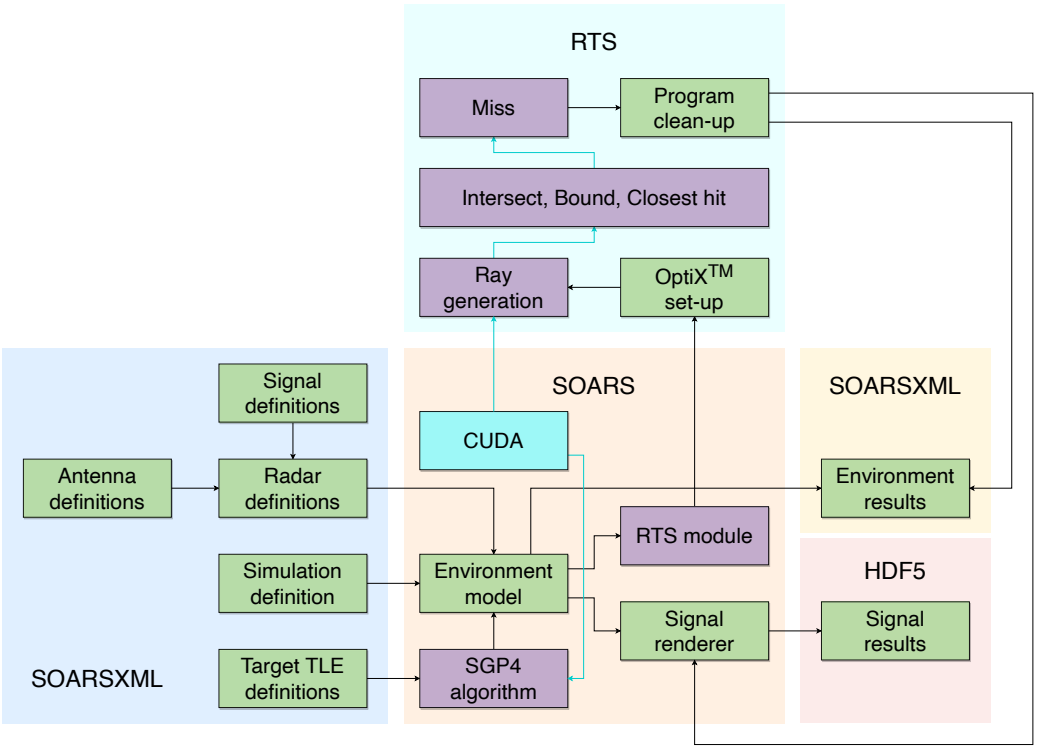


Figure 4.17: Block diagram showing the software structure and information flow of the ray-traced SOARS software

components of the received signal, and a *.soarsxml* file containing all the receiver's responses defined in terms of their pulse receive time, Doppler, power, and other properties. However, the difference between the outputs of the baseline SOARS and RTS-enabled SOARS is that the responses are defined differently; most notably, the former computes a response for every target at every relevant time sample, whereas the latter computes a response for every *received ray* at every relevant time sample.

The overall software structure can be further represented by Figure 4.18, which presents a system-level use-case diagram for the RTS-enabled SOARS program and its data flow. This illustrates the complete simulation software, depicting each of the constituent models that together form the SOARS program – including the RTS module. The flow of data between each model is presented along with the complete process executed by SOARS, transforming user inputs into the output received signals and observed responses.

With this, the design of the RTS-enabled SOARS has been fully defined.

4.4.2 Program Execution

SOARS can be run on any workstation with a supported NVIDIA® GPU; however, the dependencies listed in Subsection 3.3.3 are required as well as a valid installation of a C++ compiler (such as GCC [192]). When setting up these programs and packages, it is important that the end-user ensures that all installations are compatible with one another by confirming with the release notes – particularly CUDA and OptiX™, which have specific operating system and platform limitations.

The software developed in this work was extensively tested using NVIDIA® OptiX™ 5.1.1 and CUDA 10.0. These were used as they represented the latest software versions that were directly compatible with the hardware available – a workstation node equipped with a V100 GPU [210] and running CentOS 7 [211]. Theoretically, however, the simulator should be executable on a wide variety of operating systems and platform architecture, with the greatest limiting factor being CUDA compatibility.

SOARS is designed to be compiled using CMake [168], which links all the

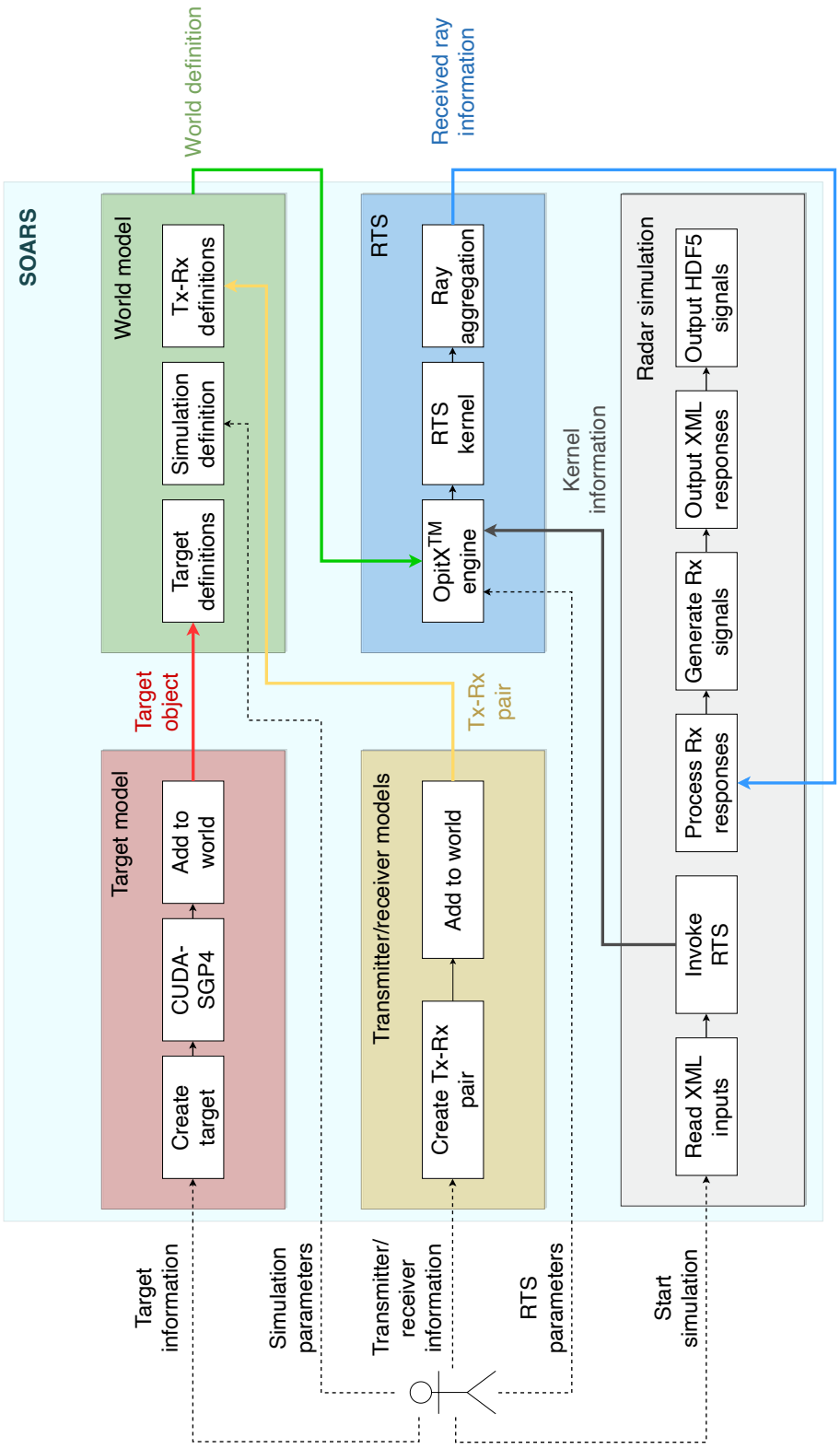


Figure 4.18: A system-level use-case diagram showing the operation of the SOARS software and the flow of data within the full program

core source code to the required external libraries and output a single binary called "soars". This can be installed as an application and sourced from any directory in a terminal, or it can be directly accessed from its installation "build" directory using:

Listing 4.11 Terminal commands used to compile SOARS and run the application on a "simulation" *.soarsxml* input file

```
1: cmake3 ../
2: make
3: ./soars simulation.soarsxml
```

CMake can also be used to set various compilation flags, allowing for additional options to be parsed to the application and enabling functionalities; these include verbose outputs, enabling or disabling FMA, specifying the versions of NVCC [169] and the C++ compiler to be used, and more. A user guide has thus been included with SOARS to simplify the process of compiling, installing, and running the program effectively.

4.5 Conclusions

This chapter has highlighted the design and implementation of the Ray Tracing Simulator – or RTS – used in SOARS. This enables target objects to be modelled by physical parameters such as size, shape and material properties, and allows simulations to account for the effects of reflection and refraction during signal propagation. The chapter has also presented the programmatic logic and software structure of the RTS, providing justifications for various design decisions and the implementation methods used to perform radar-based ray computations. Specifically, the integration of NVIDIA®'s OptiX™ ray-tracing software into the core SOARS program was described, along with details on adapting this model to fit within the radar context, i.e., performing Doppler and power calculations for multiscatter cases. Lists of the RTS inputs and outputs, and how they relate to the baseline SOARS program, were also provided.

This chapter has thus emphasised the design of the RTS and its integration

into SOARS, as well as the final software structure for the full SOARS simulator. Chapter 5 aims to verify the implementation of the simulator model, benchmark the performance of the software, and draw comparisons between the baseline and ray-traced simulators such that the investigation can be concluded.

Chapter 5

Results and Testing

This chapter presents verification, validation and benchmarking tests for the accuracy and performance of SOARS, as well as a discussion on the results of this work and the hypothesis it has proposed. This is in keeping with the accuracy, speed and portability requirements for the project software. The outputs of the selected simulations are then analysed and discussed, leading directly to the conclusions and future work described in the final chapter of this dissertation.

5.1 System Configuration

Both baseline SOARS and RTS-SOARS were developed and tested using NVIDIA® OptiX™ 5.1.1 and CUDA 10.0 with Driver Version 410.48. All benchmarks and validation tests were conducted on a single high-performance node in a GPU cluster running CentOS 7 [211].

The node is equipped with 132 GB of high-speed memory, an Intel® Xeon® E5-2695 v3 CPU clocked at 2.30 GHz (sporting 28 CPU cores), and an NVIDIA® Tesla V100 GPU [210]. The V100 sports 32 GB of GPU memory and 5120 CUDA cores. This was chosen as it was the most powerful GPU node available for student access at the CHPC and offered sufficient performance for all the experiments documented in this thesis.

During these tests, the SOARS software was compiled using the GCC g++ 4.8.5 compiler, through which the SOARS program conformed to standard C++. Additionally, the CUDA compiler NVCC 10.0.130 was used to compile device code and CMake 3.14.6 facilitated the compilation of all required source files and libraries into one executable program. In this way, the SOARS source is merged with the RTS and all other external dependencies to create a single binary, which can then be run as an ordinary program from the terminal – with the simulation definition file as the only required input parameter.

5.2 Baseline Results

The first test results from baseline SOARS were obtained by emulating the space-monitoring radar designed by Agaba [13], which proposed using the MeerKAT radio telescope as a radar receiver. This was selected as it was already extensively tested in [13] and was designed specifically for use in monitoring space debris and other spaceborne objects. This design also serves as an example of the type of system that SOARS aims to accommodate, whereby users should be able to develop similar designs and perform similar testing procedures to evaluate potential radar system parameters.

This section briefly discusses a few basic experiments that were conducted using this design to verify the baseline program's implementation against established theoretical predictions and thus measure the accuracy of some aspects of the simulator's results. This verification of the baseline version of SOARS was also presented and published in [48].

5.2.1 Simulator Set-up

This design makes use of a bistatic radar and a single manually-placed target that was propagated through nine time-steps between $t = 0$ ms and $t = 80$ ms in intervals of 10 ms. The simulation parameters are detailed in Table 5.1; these were largely chosen for simplicity and to assess the core operation of the baseline simulator using a basic experiment.

The target was initially placed at coordinates that allowed it to transit close

Table 5.1: Simulation parameters for baseline SOARS for experiments involving the MeerKAT radar

Parameter	**Value**	**Units**
Number of pulses	12	–
Carrier frequency	1.35	GHz
Sampling frequency	25	MHz
Transmit power	10	kW
Bandwidth	10	MHz
Pulse width	1	ms
Pulse Repetition Interval	6.667	ms
Receive window length	4.667	ms
Receive skip period	2	ms
System noise temperature	20	K
Parabolic transmit antenna diameter	10	m
Parabolic receive antenna diameter	13.5	m
Target RCS	0.0001	m^2

enough to both the transmit and receive antennas. These coordinates, as well as those of the transmitter and receiver, are given by:

Listing 5.1 Target, transmitter and receiver coordinates for the tested baseline experiment; values are given in km

```
1: TargPos = (5208.049, 1999.182, -3485.881)
2: TxPos = (4923.738, 1821.610, -3622.037)
3: RxPos = (5103.916, 2004.714, -3257.572)
```

Note that these receiver coordinates correspond to the nominal array centre position of the 64-antenna MeerKAT configuration. Overall, these coordinates ensured that the target range would not introduce any ambiguities. All noise sources were also disabled in these initial results for simplicity – including the 20 K noise temperature of the system amplifier.

5.2.2 Theoretical Comparisons

The transmit and receive signals are visualised by plotting their measured power against time. This is demonstrated in Figure 5.1 (generated in MATLAB®).

This depicts both the full transmit signal, which agrees perfectly with theoretical expectations, and the raw receive signal, which was stored in an HDF5 output file in its in-phase and quadrature forms. Each of these two signal components is measured in Volts, and by assuming an ideal resistance of 1 Ω, the power was computed by squaring the voltage magnitude. Note that the received signal file only records the signal samples observed at the receiver when it is operating, i.e., the receive window skip period (dead time) is not inherent in the recorded HDF5 file – hence why the received signal ends before 80 ms. The received signal in Figure 5.1 thus represents the signal observed only when the receiver is in operation – hence the shortened length in time.

Additionally, the peaks at the beginning and end of each received pulse in Figure 5.1 resulted from a time-varying group delay that is applied to each sample in the original FERS program. This was achieved using a fractional delay filtering technique to account for a pulse's round-trip time not being

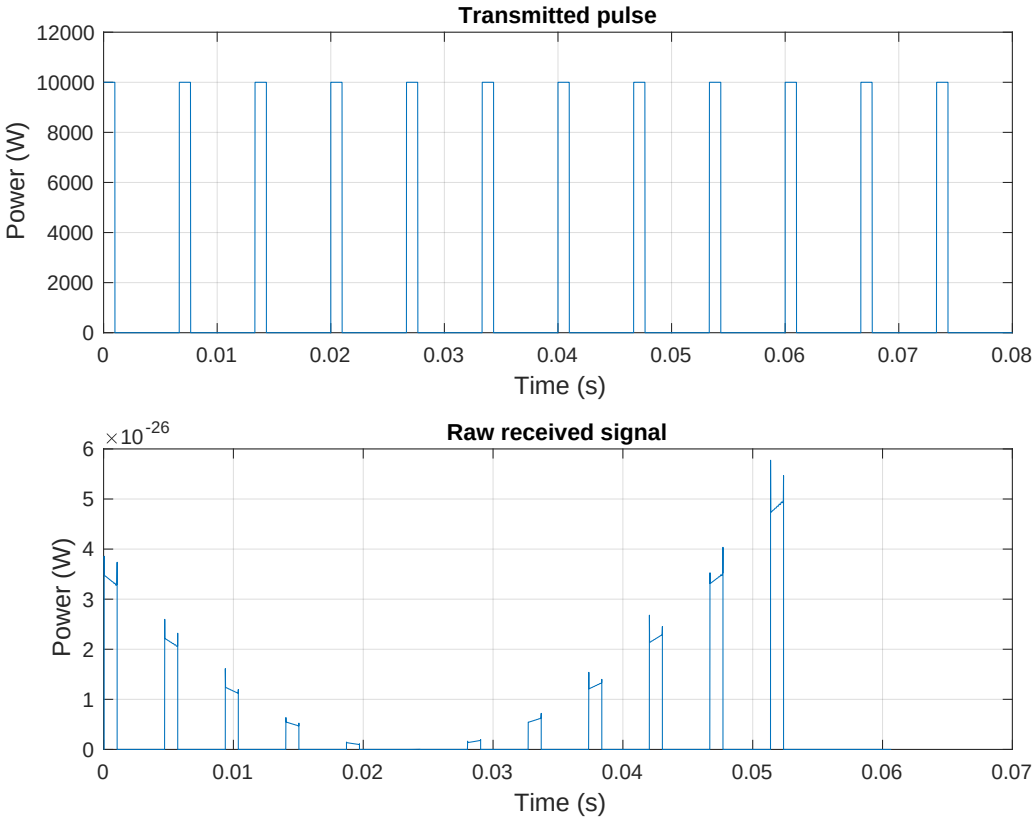


Figure 5.1: Transmit and receive signals generated by baseline SOARS using the MeerKAT radar

constant – as documented in [25] – which produces some degree of "artefacting" as seen on the sides of each received pulse.

These results were obtained by plotting the raw power signal against the sample number and then scaling the horizontal axis by the reciprocal of the sampling frequency. Following this same approach, the plots in Figure 5.2 were obtained by enlarging the first transmit and receive pulses.

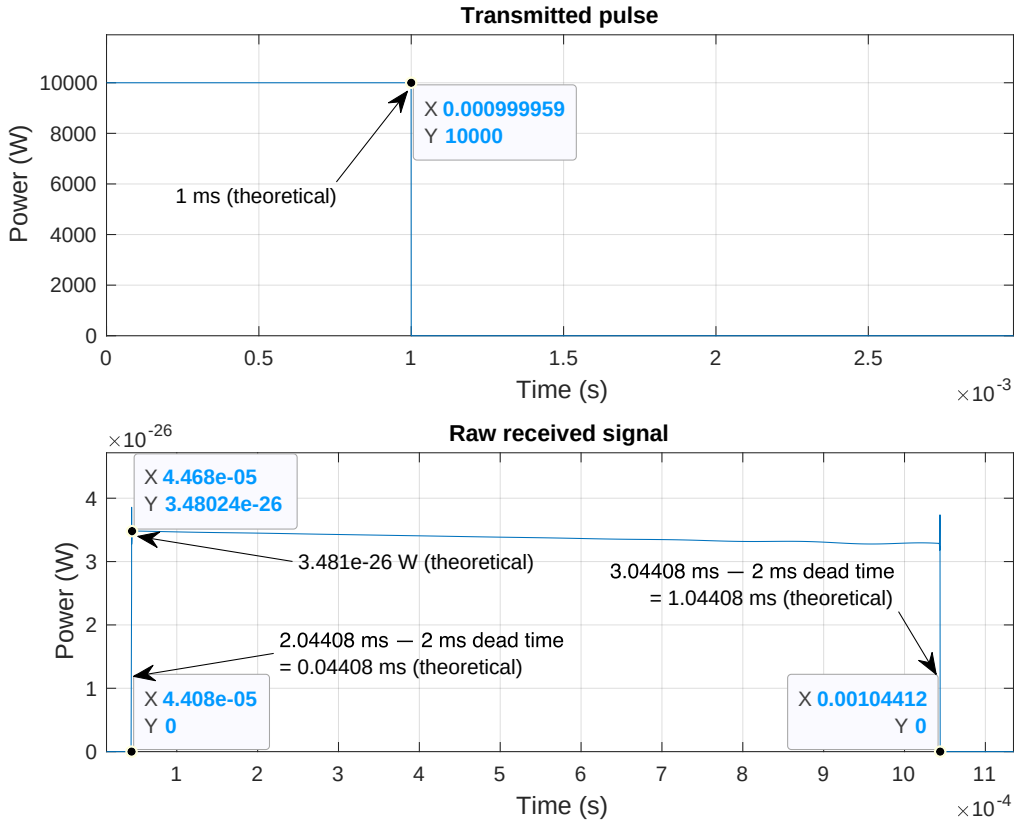


Figure 5.2: Enlargement of the first transmit and receive pulses for the MeerKAT radar experiment run in baseline SOARS

As shown in Figure 5.2, the transmit signal has a power of 10 kW and a pulse width of approximately 1 ms, as expected based on the theory. As for the first received pulse, the receive skip period (dead time) of 2 ms has clearly been accounted for – as indicated by the first return echo arriving at time $t = 0.04408$ ms. This is equivalent to the pulse arriving (in real-time) at 2.04408 ms after subtracting the dead time of 2 ms, which agrees with

the theoretical time delay (based on Equation 2.9) of 2.0441 ms using the propagation distance $R_T + R_R = 612.802$ km. This implies a time delay error of 3.2×10^{-4}%. The end-point of the pulse can also be seen to arrive 1 ms later – as expected due to the pulse width. These theoretical values have been marked on Figure 5.2 for the sake of comparison.

Additionally, the received signal's power can be verified using the formula in Equation 2.7, but this requires the transmitter and receiver gains – which are calculated using Equations 5.1 and 5.2 for a parabolic gain pattern [212]:

$$\zeta = \left(\frac{\pi d}{\lambda}\right) sin\ \theta_{OB} \tag{5.1}$$

$$G(\theta_{OB}) = G_{max}\left(\frac{2J_1(\zeta)}{\zeta}\right)^2 \tag{5.2}$$

where ζ is an interim quantity used in the calculation of the overall antenna gain G (a function of the off-boresight angle θ_{OB}), G_{max} is the maximum gain of the antenna, and J_1 is the Bessel function of the first kind of order one. The value of ζ is dependent on θ_{OB}, which corresponds to the angle measured between the vector from the antenna to a target, and the vector from the antenna to its beam centre – i.e., the antenna's boresight, along which its gain is maximised.

Thus, by using Equation 2.7 directly, the value of P_R was computed as 3.481×10^{-26} W – closely agreeing with the simulated result from SOARS and corresponding to a power error of 0.247%. This theoretical result has also been marked on Figure 5.2 for comparison with the simulated result. This confirms that both the transmit and receive signals were generated by SOARS exactly as expected.

Another quantity to be verified is that of the Doppler shift, i.e., testing the velocity computations performed in SOARS for a SPG4-propagated target. The target's velocity thus needed to be computed by considering the change in its coordinates over a specified period, and the resulting velocity vector then needed to be projected onto the bistatic plane – formed from the starting

coordinates of the transmitter (stationary), receiver (stationary), and target – denoted Tx, Rx, and Tgt_0 respectively.

The projected velocity component, denoted V in Equation 2.16, was then used to calculate the Doppler shift once the angles β and δ were determined. On the other hand, SOARS used Equations 2.11 and 2.12 to get the same result. These implementations are tested using the same simulation parameters as before, and the target was propagated over a very short 1 ms interval. This resulted in the target's initial and final coordinates, as well as its velocity in km/s, as follows:

Listing 5.2 Target's initial and final coordinates (in km) as well as its velocity (in km/s) for the tested baseline experiment

```
1: TargPos1 = (5208.049, 1999.182, -3485.881)
2: TargPos2 = (5208.045, 1999.183, -3485.875)
3: TargVel = (-4.307, 0.914, 5.639)
```

This implies a velocity magnitude of 7.155 km/s, which was then projected onto the bistatic plane as V as presented in Figure 5.3. This illustrates the bistatic bisector, every object's position, the angles δ and β, the target velocity and its projection, and the bistatic plane itself. The angle β was thus calculated as 91.553 deg and the bisector was found by normalising and adding the vectors between Tx-to-Tgt_0 and Rx-to-Tgt_0, i.e., the resultant of the range vectors from the target to the transmitter and receiver.

Determining V and δ, however, requires knowledge of the normal vector of the bistatic plane. This is determined using Tx, Rx, and Tgt_0 in the formulation shown in Equation 5.3.

$$\overrightarrow{n} = (\overrightarrow{Tgt_0} - \overrightarrow{Tx}) \times (\overrightarrow{Tgt_0} - \overrightarrow{Rx}); \tag{5.3}$$

The normal vector $\overrightarrow{n}$ is then be normalised to a unit vector $\hat{n}$, and the final coordinates of the target (denoted Tgt_1) are projected onto the bistatic plane using Equation 5.4.

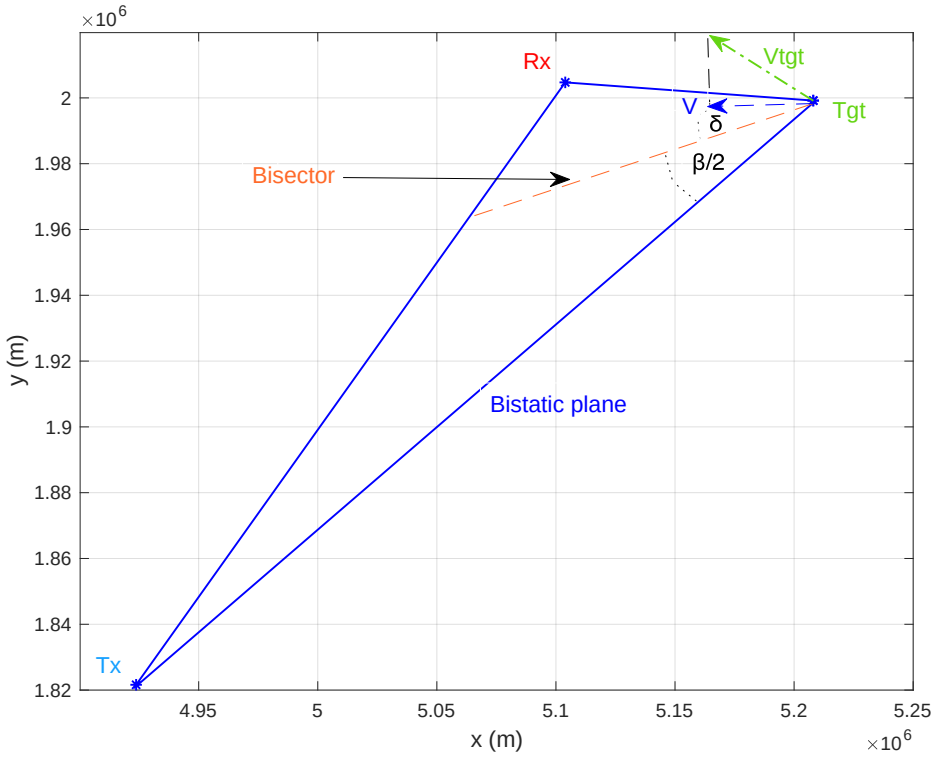


Figure 5.3: 3-D geometry of the tested bistatic configuration showing the bistatic plane and all relevant vectors and angles for Doppler computation; $Vtgt$ and δ do not lie on the bistatic place

$$\overrightarrow{Tgt_{1_{proj}}} = \overrightarrow{Tgt_1} - \hat{n}((\overrightarrow{Tgt_1} - \overrightarrow{Tgt_0}) \cdot \hat{n}) \tag{5.4}$$

Taking the vector subtraction of Tgt_0 from $Tgt_{1_{proj}}$ and dividing it by the propagation time of 1 ms thus yields the projected component V. Along with this, the angle δ can finally be computed as the angle between the bisector and V. The value of δ was manually calculated as 37.554°, leading to a calculated Doppler of +34903.930 Hz using Equation 2.16, i.e., the target moves *towards* the radar. The corresponding output from SOARS was found to be 34904.382 Hz in the output XML file, which is calculated within the simulator using Equations 2.11 and 2.12.

This leads to a Doppler error of approximately 0.00129%. Together, Equations 2.11 and 2.12 are found to be slightly more accurate than Equation 2.16. Based on the theory of special relativity, this agrees with the expected result as this method ultimately accounts for a greater number of minor Doppler contributions (as shown by the higher-order terms in Equations 2.13. For this reason, these two equations were selected for implementation in SOARS instead of Equation 2.16.

With this, the baseline simulator was successfully tested from the simulation definition files to the final output signal. These results have shown that both the simulated transmit and receive signals have met theoretical expectations, and the operation of the point-model simulator has thus been verified against the theory.

5.3 Software Validation

This section presents the results of various validation tests comparing the RTS-enabled version of SOARS against an existing radar simulator. Additionally, a discussion on system validation is provided to motivate the various methods of testing that have been used – and the reasoning behind them. Some of the results presented in this section have also been published in [49].

5.3.1 Validation Methodology

While the original FERS software was already validated against measured data in [25], further testing was done to ensure that the SOARS software also correctly simulates radar experiments – particularly with the addition of the RTS. Validation of the generated received signals was thus achieved by comparing the simulated SOARS results against a well-established, widely-trusted radar simulator – the Phased Array System Toolbox, or PAST, in MATLAB® [213, 214].

This was used to validate the outputs of SOARS while avoiding comparisons with arbitrary measured datasets from space-monitoring experiments – but also due to the inherent flexibility in PAST, allowing for various experiments to be conducted between this program and SOARS using nearly identical configurations and thus maximising the comparison accuracy. PAST makes use of fully-fledged radar simulation structures, functions and models that run directly on the MATLAB® engine; however, the toolbox lacks some features present in SOARS – such as ray tracing and fractional delay filtering – and as such, these discrepancies will need to be addressed.

PAST was previously used as part of the toolchain and workflow in the research works published in includes [215] and [216], where the former documented the design of 5G antenna arrays involving hybrid beamforming, and the latter presented the simulation of frequency-modulated, continuous-wave radar systems for road safety and collision avoidance. PAST was also used to successfully validate the development of various phased antenna arrays in [217], with the authors concluding that PAST was highly effective in the design and testing of potential antenna designs with various geometries, element orientations, and radiation patterns. As a result of these works, it was concluded that PAST was well suited for use in this work as part of the validation process.

It should be noted that it was not possible to obtain measured radar datasets (from experiments involving spaceborne objects) for this research. This was mainly due to the confidential nature of such data as well as the high level of clearance required by most organisations and governments in attaining access

to such sensitive experiments and the associated hardware. As such, the implementation of SOARS had to be validated through other, more qualitative means. Most notably, three methods have been used:

1. The design of SOARS was based on the FERS simulator, and much of the core code remains unchanged in terms of signal rendering radar processing. As FERS was originally validated [25] against measured data from multiple generic radar experiments (involving land-based objects as well as sonar experiments), it can be assumed that many modelling aspects of SOARS are still validated.

2. The baseline version of SOARS has been verified against radar theory as documented in [48], and many of the new models (SGP4, noise, and more) are based on well-established models that have already been proven against the theory and/or measured data [88, 96, 173].

3. As published in [49] (and shown later in this section), the RTS itself was validated by comparing its results against those of PAST – an existing, verified, and well-known simulator that is built into MATLAB®.

This section thus presents the results of various validation tests comparing the RTS-enabled version of SOARS against PAST, and conclusions are drawn regarding the accuracy of the results. The experiment configurations used in this section were selected to test various features of the RTS on a case-by-case basis; as such, generic radar scenarios were used in these comparisons to avoid large floating-point computations in PAST and to ensure accuracy in the simulator's basic implementations. This also served to show how the RTS is not necessarily limited to only space-based scenarios, and that it could also be used for generic radar experiments if it were to be integrated into a general-use radar simulator such as FERS. Later in the chapter, space-related scenarios are tested as per the intended use case for SOARS.

The comparisons shown in this section demonstrate that the generated results from SOARS match the MATLAB® results with a high degree of accuracy, implying that the results for other scenarios would also be quite accurate. In this way, various facets of the developed software have been further verified and validated through simple testing procedures.

5.3.2 Multi-ray Tracing

The initial simulation parameters used to verify the RTS implementation are presented in Table 5.2. This scenario made use of a monostatic radar that was positioned at the origin and set to transmit a total of 16 pulses towards a single target.

Table 5.2: Simulation parameters used in Experiment #1 for validating the RTS

Parameter	**Value**	**Units**
Number of rays	10^6	-
Number of pulses	16	-
Carrier frequency	1	GHz
Transmit power	10	kW
Receiver span (θ, ϕ)	(90, 90)	deg
Pulse width	2	µs
Pulse repetition interval	200	µs
Mean RCS	15	cm^2
Reflection coefficient	0.95	-
Refractive index	3.0	-

The target in this scenario was defined by a custom mesh of triangles modelling an airborne drone [218] (as shown later in Figure 5.6) and it was centred at $(x, y, z) = (0, 0, 100)$ with a velocity of +20 m/s on the z-axis (away from the radar). This target set-up was chosen as the mesh itself provided a smooth surface for simplified reflection testing, and the short range allowed for the RTS to be tested without concern for any floating-point errors that could occur when dealing with very large numbers.

The same scenario was set up in PAST with a single point-scatterer target positioned at the same coordinates with the same velocity. The objective of this was to run near-identical simulations in both the RTS and PAST, and to then compare the results from the two simulators to evaluate any

discrepancies. The set-ups of these simulations are illustrated in Figure 5.4 for both the RTS and PAST.

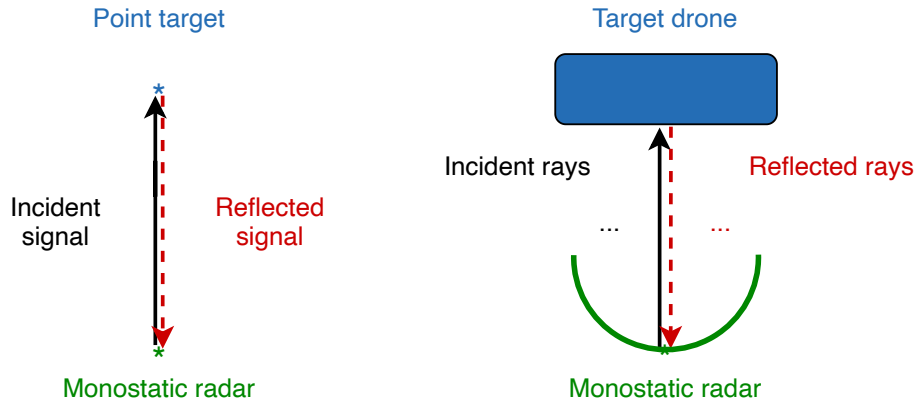


Figure 5.4: Side view of Experiment #1's multi-ray test scenario in PAST (left) and the RTS (right); the diagram is not to scale

As such, the main discrepancies between the simulations are that: (1) PAST uses a single "ray" (signal) mapped directly from the radar to a point target and then back again, whereas the RTS uses 1 million rays with angular variations; (2) PAST, as a point-model simulator like FERS and baseline SOARS, models the target and receiver as single points in space, whereas the RTS uses a mesh of triangular primitives to model the target and a spherical surface patch to model the receive antenna. The latter also allows the RTS to distinguish between material parameters and account for objects' physical shapes.

The RTS implementation can thus be verified by comparing the two simulators' received signals as depicted in Figure 5.5, which presents both the power and phase for the fourth received pulse from each simulator – where the gains in PAST were manually set to model the RTS's antenna gain and boresight in the same way.

As illustrated, the signals agree very closely, demonstrating how both rendered signals match up with a high degree of accuracy and showing that the RTS can nearly replicate PAST results. However, the RTS also makes use of more realistic target and propagation models and accounts for the effects of refraction, target and receiver shape, and object material parameters.

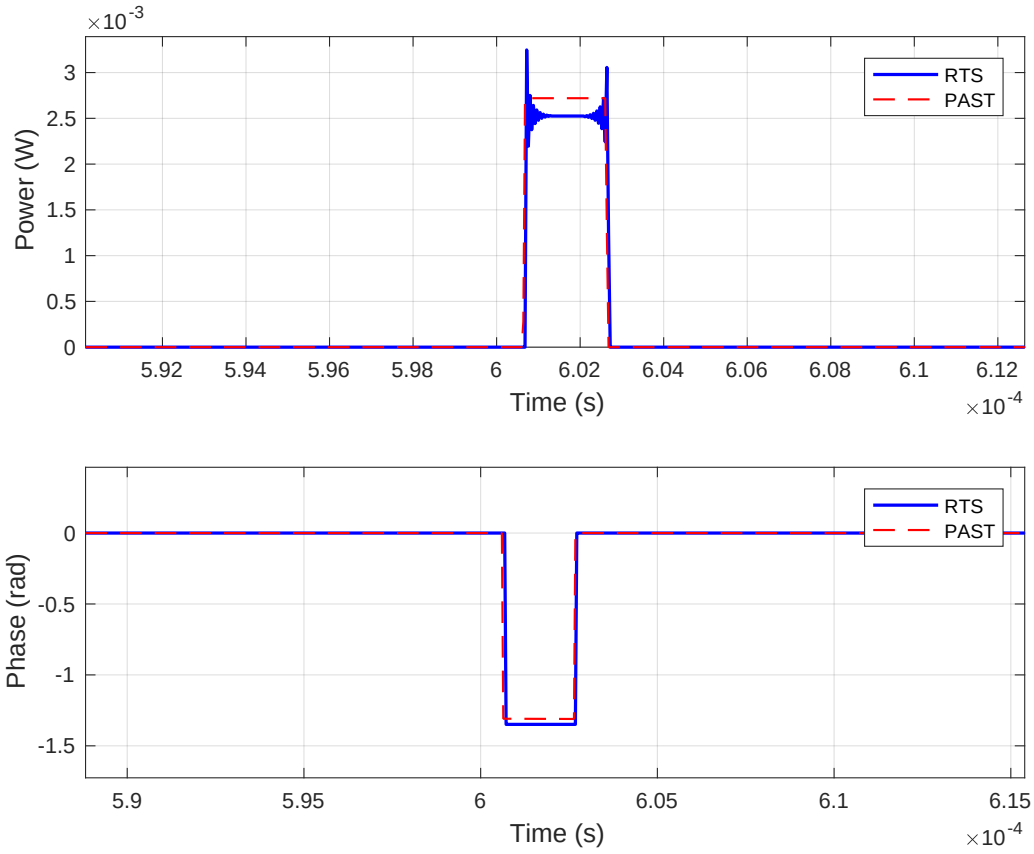


Figure 5.5: Power and phase plots for the fourth received pulses from the RTS and PAST for Experiment #1

Averaging the root mean square error (RMSE) across the in-phase and quadrature components of the signal also produced a low value of 2.19×10^{-4}, showing minimal disagreement.

There are also notable peaks present at the edges of each pulse in the RTS signals that PAST does not replicate, but these are attributed to the fractional-delay filtering technique [25] applied in the RTS. Additionally, other minor power differences are attributed to the implementation of refraction and the reflection coefficient used – which PAST does not account for – as well as minor phase differences due to time delays varying across rays.

Further results are depicted in Figure 5.6, which shows the ray intersection coordinates on the underside of the drone mesh itself.

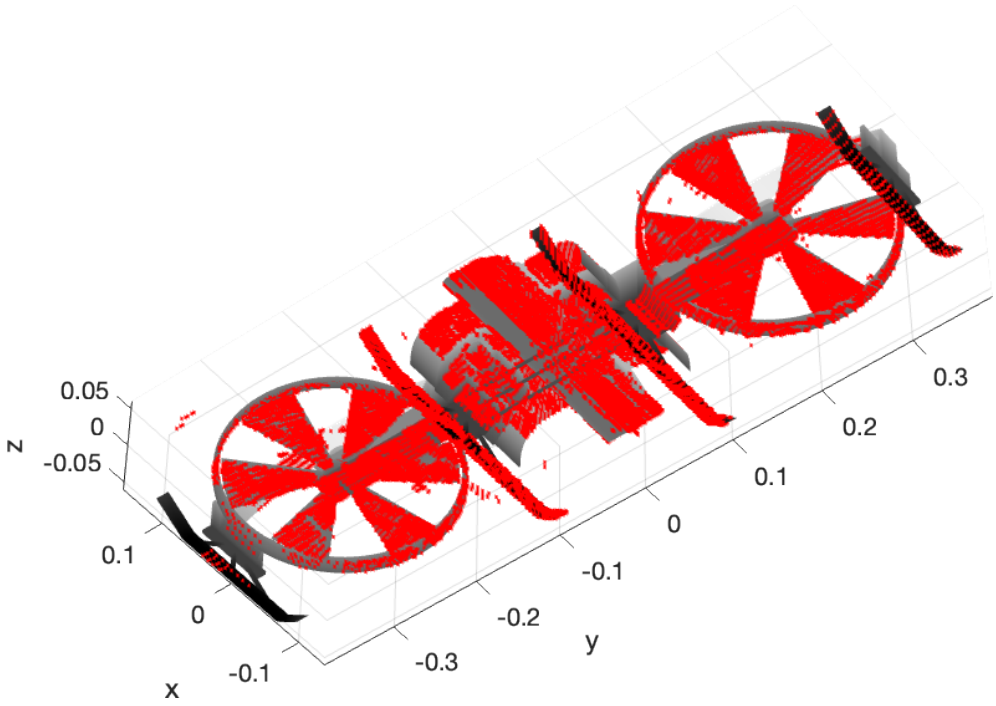


Figure 5.6: Target mesh and its ray intersection points (marked in red) for the first transmit pulse in Experiment #1 when run in the RTS

This illustrates the spread of the rays intersecting the target mesh, and thus also the spread in the received rays. This emphasises the improved accuracy in target and propagation simulation relative to PAST – which makes use of

only a single "ray" (the overall signal) that is mapped directly to the centres of both the point-model target and receiver.

5.3.3 Multi-target Simulation

This experiment considers the simulation of two targets in a monostatic radar experiment. This represents a simple test scenario with the aim of verifying the ray-traced return signal in a situation involving multiple targets, and to ensure that the RTS is able to model returns from independent objects. Many of the parameters in Table 5.2 were reused for this scenario, but for simplicity: (1) refraction was disabled, (2) only 10^3 rays were used, (3) the reflection coefficient was set to 1, and (4) the noise temperature was set to 20 K to test the RTS's noise generation capabilities.

The drone mesh from Experiment #1 was also substituted for two cuboids as their shapes allowed for more predictable results, ensuring that each target would yield an independent return at the radar; one target was centred at 5 km on the z-axis (with a velocity of 100 m/s *away* from the system), while the second was centred at 10 km on the z-axis (with a velocity of 50 m/s *towards* the system). The two targets were also slightly separated along the y-axis so that neither target was shadowed (as will be tested and discussed in Section 5.4). This is shown in the approximate geometry depicted in Figure 5.7.

Simulating the scenario in both the RTS and PAST generated the received signal voltages shown in Figure 5.8. This depicts the receiver response (from each simulator) to the twelfth transmit pulse, and both signals were separated into both their in-phase and quadrature forms. As shown, both results show strong agreement – with an RMSE of -1.55×10^{-9} as averaged across both the in-phase and quadrature components.

These results were further verified using range-Doppler maps as shown in Figure 5.9, which demonstrate how the range and velocity components agree quite strongly for both simulated targets across both programs. The results confirm that there are only minor variations in the post-processed signals in the frequency domain – primarily due to the inclusion of noise as well as the

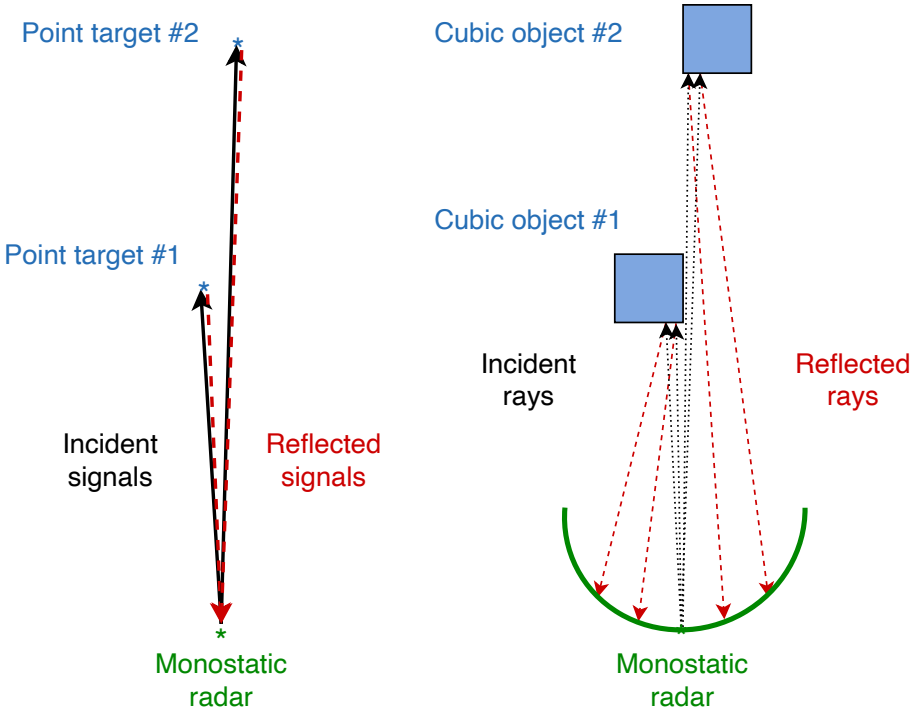


Figure 5.7: Side view of Experiment #2's multi-target test scenario in PAST (left) and the RTS (right); the diagram is not to scale

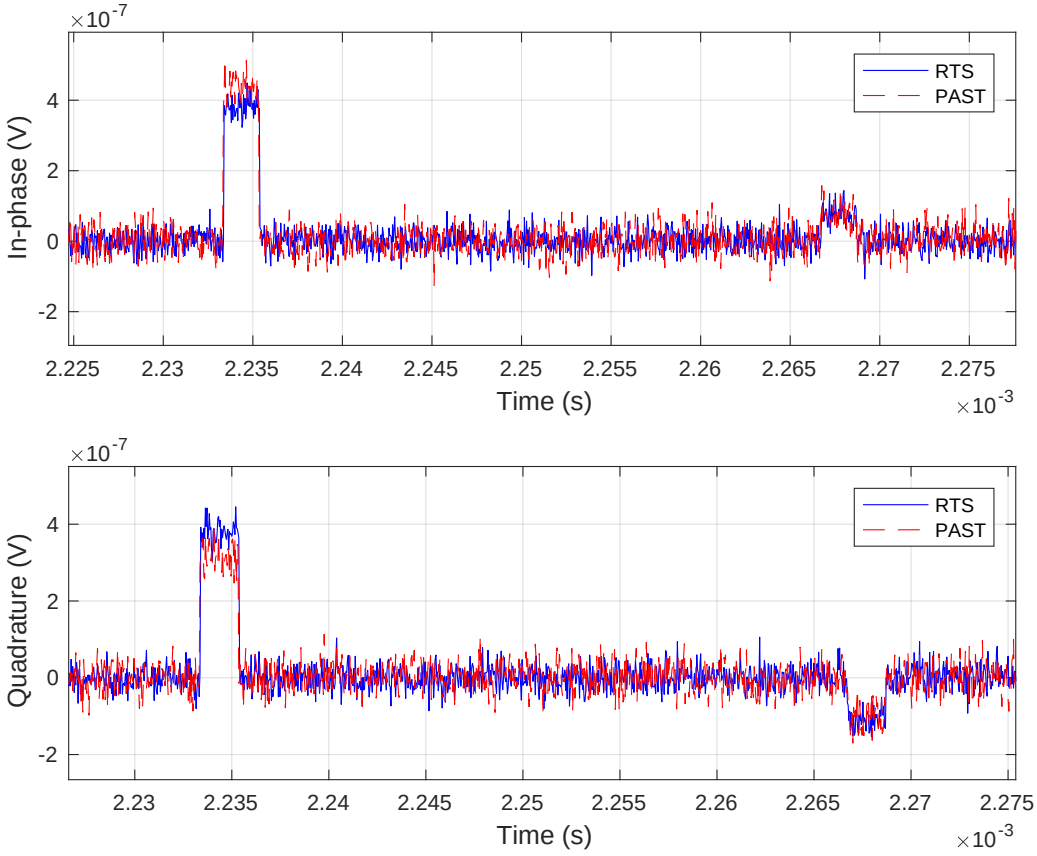


Figure 5.8: In-phase and quadrature components of the received pulses for the 12^{th} transmit pulse for Experiment #2 in the RTS and PAST

inherent design differences between the two programs.

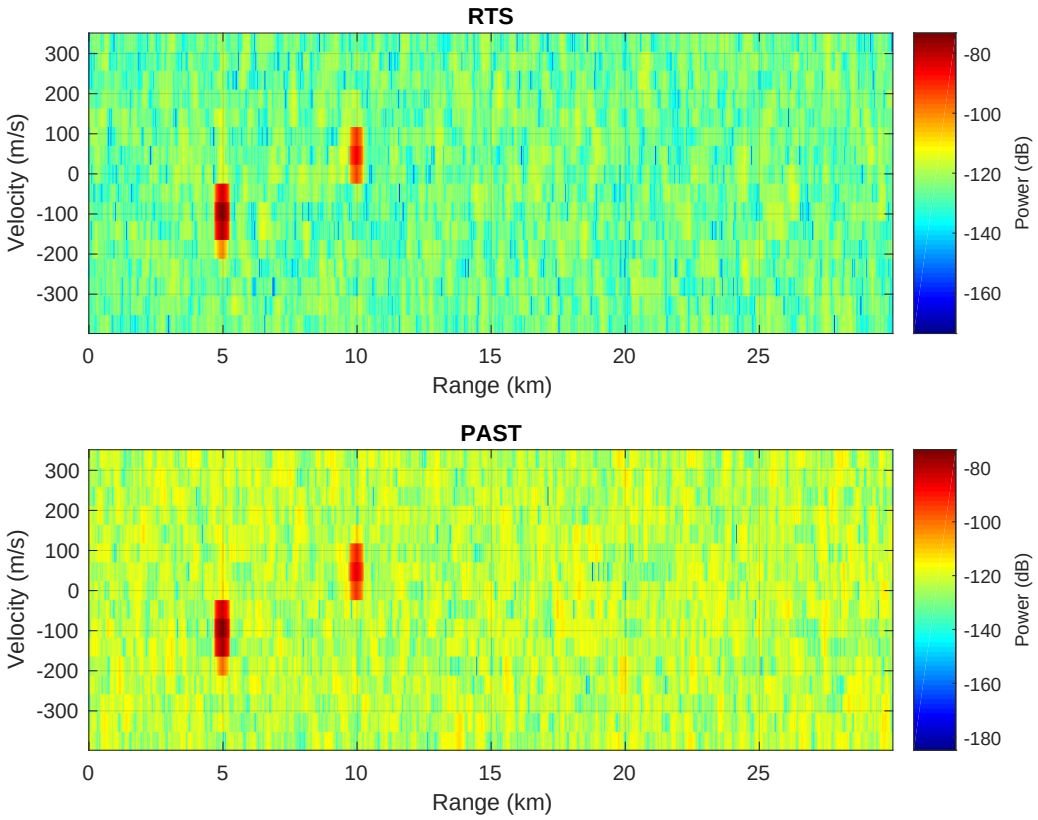


Figure 5.9: Range-Doppler maps produced from the results of Experiment #2 as run in the RTS and PAST

Overall, this section has demonstrated how the RTS: (1) reflects and refracts signals based on the theory of geometric optics, (2) models targets through physical parameters such as shape, size and material properties, (3) models the physical properties of receiving antennas (using spherical surface patches), and (4) accounts for system noise (from both internal and external sources) in the output signal. Together, these aspects allow the RTS to realistically model signal propagation and geometries with a small degree of error relative to PAST, showing strong agreement with the existing simulator despite the design differences.

It should be noted that it is possible for *no* rays to be captured in an RTS simulation – particularly when working with long-range targets such as

spaceborne objects, as some rays may miss parts of reflecting objects (or entire objects) and/or the receiving antennas. This is, in fact, quite likely given that a terrestrial antenna would be minuscule relative to the range at which space objects propagate. As such, this is one potential drawback of using the RTS-enabled SOARS as opposed to baseline SOARS, as directionality is less of a factor in the latter. However, users can mitigate this problem by increasing the number of targets and rays used in the RTS simulation, as well as by using smaller transmitter beamwidths and directing the antenna boresights directly towards the scene.

These simulations confirm that ray tracing is a viable mechanism for signal-level radar simulation, offering promising results for modelling complex radar scenes in various ways. However, it remains important to note the discrepancies between the RTS and a typical point-model simulator; while it is not always possible to replicate the same results in both programs all tested scenarios (some of which have not been listed here) demonstrated that the results in this section agreed closely. As such, this section validated that the RTS operates as expected and produces comparable results to PAST in all tested scenarios, but this raises the question of how *useful* ray tracing is for signal-level simulation. This is addressed through various tested applications of SOARS as shown in Section 5.4.

5.4 Applications and Testing

While the previous results served to verify and test SOARS for generic experiments, this section aims to compare the baseline version of SOARS against the RTS (and final) rendition of the program as well as demonstrate a few typical applications of the software in practice. In this way, key differences between the simulators could be evaluated.

Four experiments are detailed in this section – with two of them making use of NORAD TLE sets to model the simulated targets' trajectories. These TLEs are detailed in Table 5.3, which includes information on the targets' TLE identification numbers, their object classification type, Keplerian period, and estimated RCS values.

Table 5.3: TLE set and modelling information for three targets used across SOARS experiments

Target	**TLE**	**Classification**	**Period** (minutes)	**RCS** (m^2)
#1	25544	Space station	92.9	401.801
#2	35065	Debris	100.0	0.041
#3	30447	Debris	100.9	2.403

While additional targets could have been added to these experiments, the focus was placed on simulating a variety of cases with different parameters and object orientations to verify the operation of the software under specific circumstances. These include experiments relating to shadowing phenomena, multiscatter effects between consecutive targets, and the use of detection processing for detecting a large and complex space object. As such, it was deemed that two targets would be sufficient for demonstrating the necessary impacts in these trials, as any further targets would be expected to lie outside the simulated radar beamwidth or simply not contribute to the multiscatter or shadowing effects being monitored.

Computational performance benchmarks for each of the four experiments – conducted between baseline and RTS-SOARS – are provided in Section 5.5.

5.4.1 Shadowing Phenomena

In addition to mesh modelling and geometric optics, the RTS is also expected to consider situations involving shadowed targets and improve simulation accuracy in such scenarios. Specifically, this may include simulations where (1) a target is positioned directly behind another object and is thus blocked from the paths of incoming rays, and (2) multiple targets are directly overlapping and occupying the same physical space in the simulated scene, causing self-shadowing. Both scenarios are tested and investigated in this section.

Blocked Targets

Shadowing scenario (1) was tested using a similar configuration to that shown in Figure 5.7, but the two targets were positioned at $x = y = 0$ with $z = 5$ km and $z = 10$ km, respectively. Running this simulation in both the baseline and ray-traced simulators generated the received signals shown in Figure 5.10.

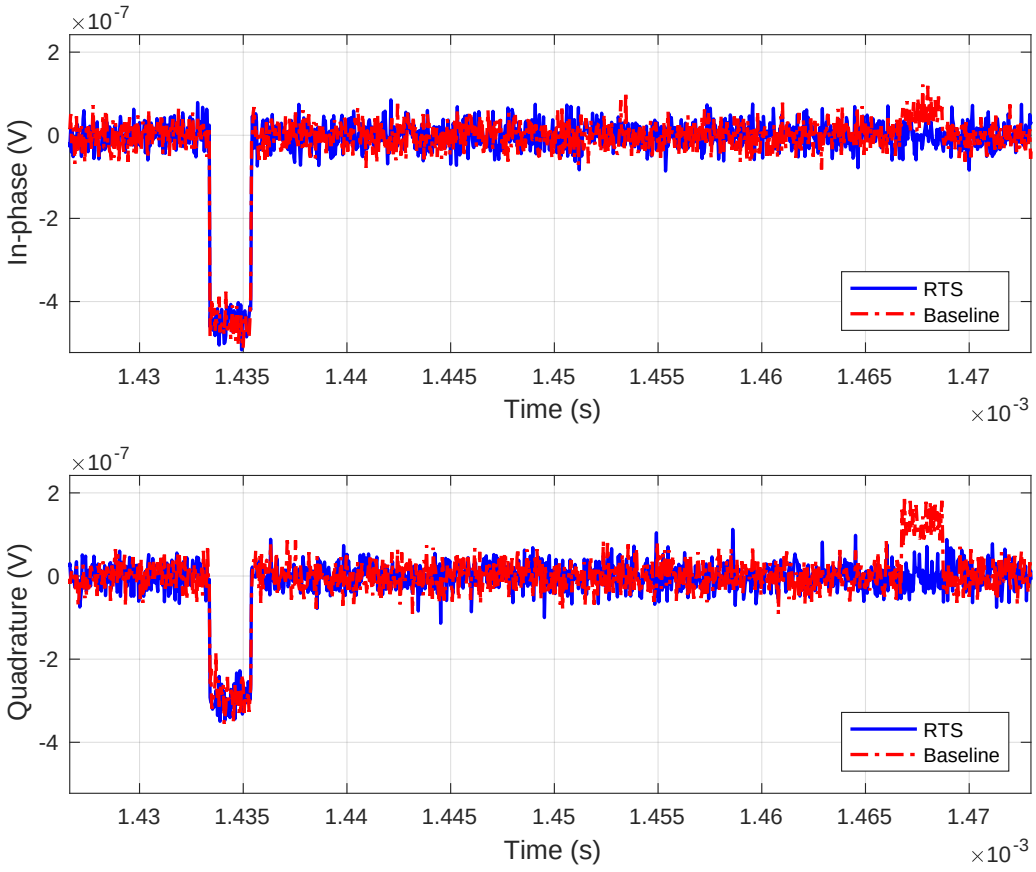


Figure 5.10: In-phase and quadrature components of the received pulses in response to the 8^{th} transmit pulse for a shadowing experiment run in both the ray-traced and baseline simulators

As expected, two pulses are received in the baseline SOARS – one from each target in the simulation as the simulator ignores the physical properties of the first (closer) target and thus allows the signal to propagate through to the second (further) target. The RTS, however, only receives a single pulse as reflected from the closest target. While the baseline simulator's

point-model approximation forces pulses to be mapped directly from the transmitter to each point target and then to the receiver, the ray-traced simulator acknowledges that the closer target lies within the propagation path and thus no pulses hit the second target.

This mirrors what would be expected in a real-world scenario, as no return would be expected from the further target since it is "shadowed" by the first. This demonstrates that the RTS accounts for such situations more realistically than baseline SOARS through its use of geometry-based reflection. This simple example represents one of many possible situations where the RTS provides more accurate results – but this comes with a performance cost as will be discussed in Section 5.5.

Overlapping Targets

For shadowing Scenario (2), self-shadowing is expected to occur when two (or more) targets are positioned in the same physical space in the RTS – as could happen if any two objects are propagated along their trajectories and happen to cross paths. In this kind of scenario, only a single return from *one* of the targets would be expected in reality. The baseline program, however, would be expected to yield approximately double the return as *both* overlapping point targets should produce an echo and thus generate a response at the receiver.

This was tested using the same radar scenario and parameters as before, but both targets were placed at exactly $z = 5$ km to occupy the same space. The output signal's in-phase and quadrature components are presented in Figure 5.11.

This demonstrates how the RTS received approximately half the return power of the baseline receiver response, as expected. This confirms that only *one* target reflected transmit pulses in the ray-traced simulator, whereas in the baseline simulator, *both* point targets yielded a response at the same time – resulting in a return pulse that is doubled in size relative to the RTS result. This is directly attributed to the naive target model used in the point-model simulator, as the physical geometries of objects are ignored and simulations

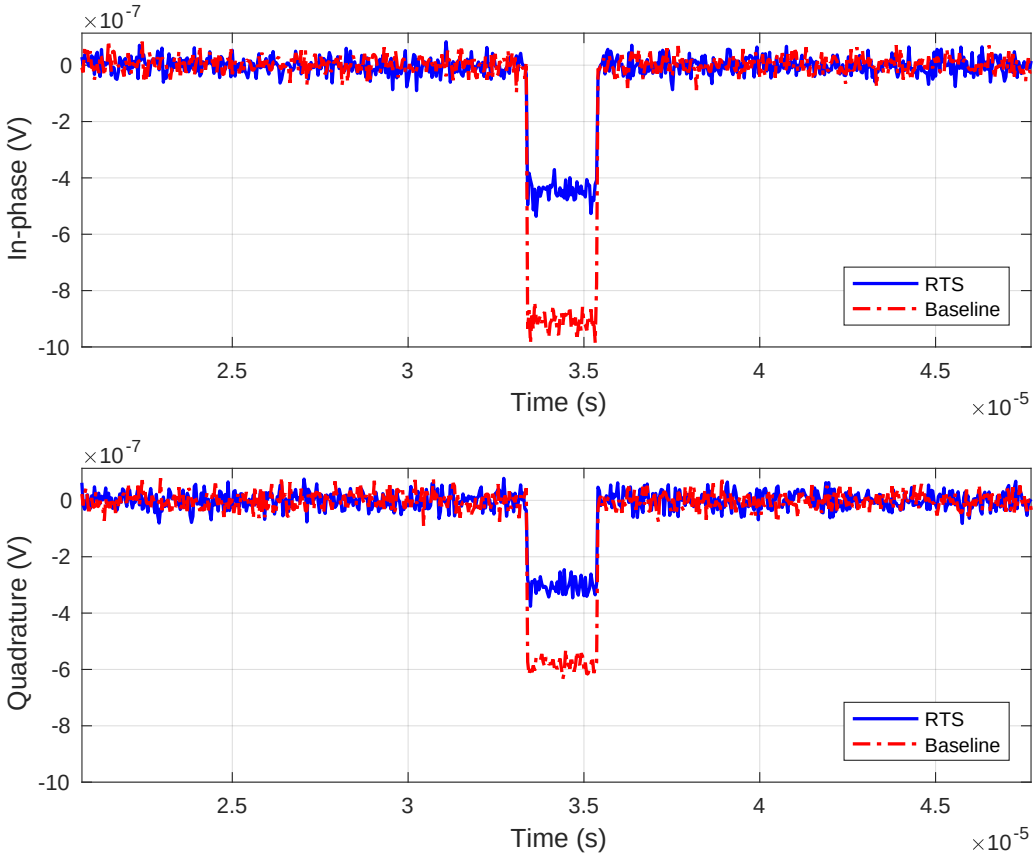


Figure 5.11: In-phase and quadrature components of the received pulses in response to the 8^{th} transmit pulse for a self-shadowing (overlapping) experiment run in both the ray-traced and baseline simulators

are physically unconstrained. These shadowing cases thus exemplify two common situations where the RTS (and, by extension, ray tracing) provides a notable boost to simulation accuracy relative to the baseline simulator.

5.4.2 Detection Processing

This subsection considers an experiment involving a bistatic radar (inspired by the MeerKAT radar design by Agaba [13]) with a Constant False Alarm Rate – or CFAR – algorithm for detection processing. This leads to the detection of an SGP4-propagated target in a noisy return signal as generated at the output of the RTS-enabled SOARS.

ISS Representation

This experiment made use of the ISS as the target for detection, as the space station has both a TLE representation as well as other specifications (such as a mesh model) that are publicly available for use. This is represented by Target #1 in Table 5.3.

The simulated scenario (as entered into SOARS) is represented in Figure 5.12, illustrating the bistatic MeerKAT radar [13] being used to monitor the ISS in orbit.

This made use of the system parameters shown in Table 5.4.

The target is based on a mesh of triangular primitives to accurately model the object in free space – particularly due to its large size. This mesh was derived as an OBJ file from a model available on NASA's official site [219] and is illustrated in 3-D space in Figure 5.13.

This demonstrates the target's size and shape, accounting for its enormous span as well as the intricacies in its physical design. Using this mesh thus allows for the ISS to be accurately modelled in simulation and provides a notable advantage over the point-model approximation used in the baseline version of SOARS. The target was also provided with an RCS value of 401.801 m^2, as taken from [220] – used as part of multiple satellite tracking sites. Additionally, the target was assigned material parameters of approximately

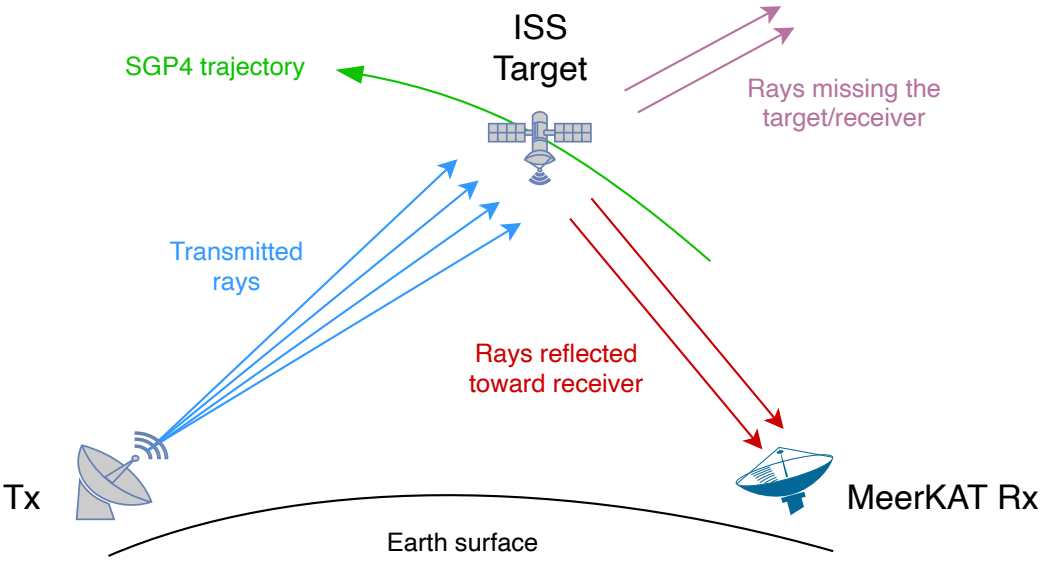


Figure 5.12: Side view of an experiment run in SOARS to simulate the MeerKAT radar with the ISS as its observation target; the diagram is not to scale

Table 5.4: Simulation parameters for the SOARS experiment involving detection of the ISS using the MeerKAT radar

Parameter	**Value**	**Units**
Carrier frequency	1.35	GHz
Sampling frequency	25	MHz
Transmit power	10	kW
Bandwidth	10	MHz
Pulse width	1	ms
Pulse Repetition Interval	6.667	ms
System noise temperature	20	K
Bistatic baseline	445.899	km
Parabolic transmit antenna diameter	10	m
Parabolic receive antenna diameter	13.5	m

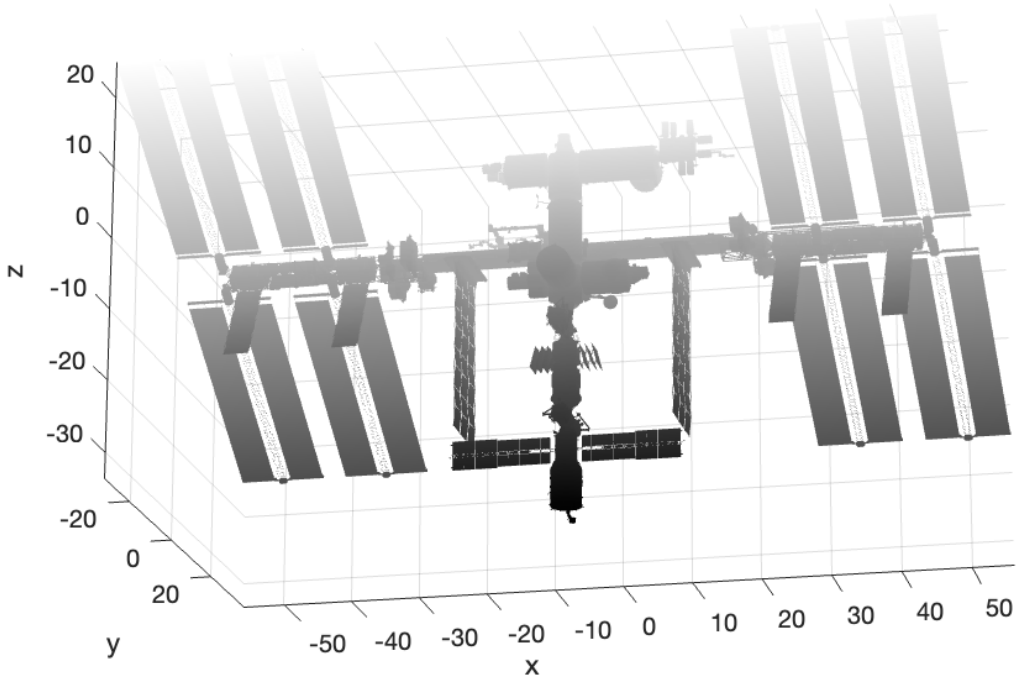


Figure 5.13: Mesh model representation of the ISS as loaded into ray-traced SOARS as a target

$\Gamma = 0.890$ and $n = 0.14054$ for reflection and refraction, respectively [221].

Using SGP4 to propagate the target's orbit over 120 minutes, the object is seen to follow the trajectory shown in Figure 5.14 relative to both the transmitter and MeerKAT receiver.

This, therefore, accounts for the object's traversal over time, and with this, a complete depiction of the target was provided to the simulator. It is also worth noting that the simulation begins when the target is at the 30-minute mark in Figure 5.14, as this brought the target to a point where lines of sight could be established with the transmitter and receiver.

CFAR Detection

CFAR algorithms operate by estimating the statistics of environmental noise and interference and then computing a detection threshold T such that a constant probability of "false alarms", denoted by P_{FA}, is achieved. The threshold is then compared against the squared magnitude of a received signal,

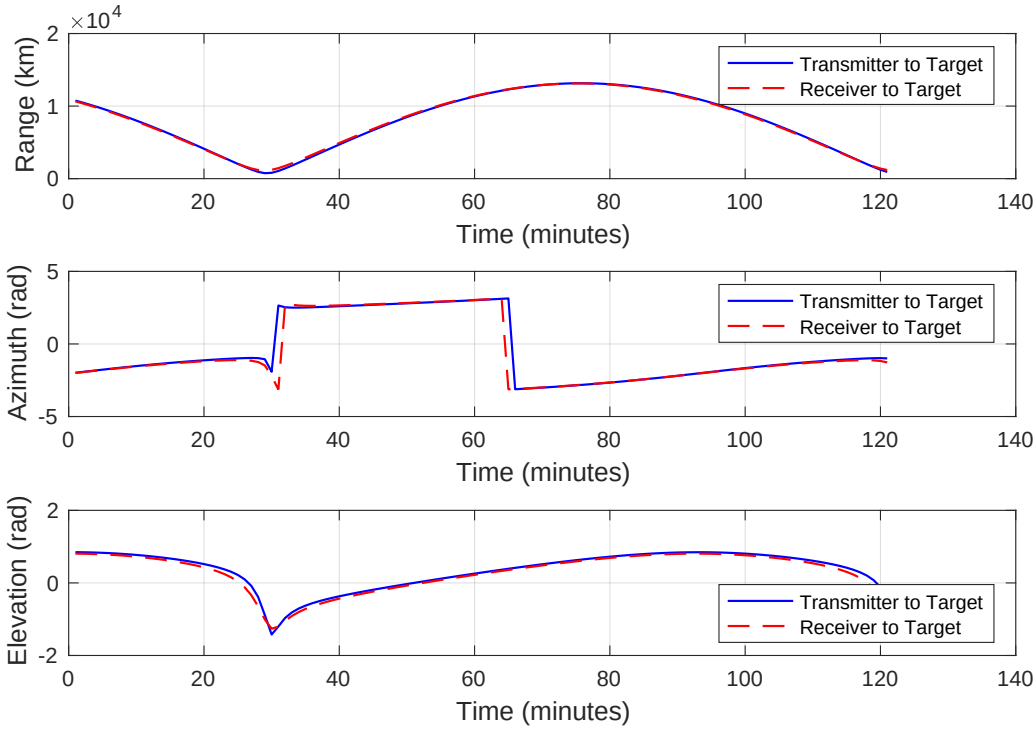


Figure 5.14: SGP4-propagated range, azimuth and elevation of the TLE-based ISS target (relative to the transmitter and receiver) over 120 minutes

and if any signal sample exceeds the threshold at that point, a target return is declared present in that sample.

The basic architecture for a CFAR detector is depicted in Figure 5.15 (adapted from [106]), making use of N reference cells to estimate the background interference statistic.

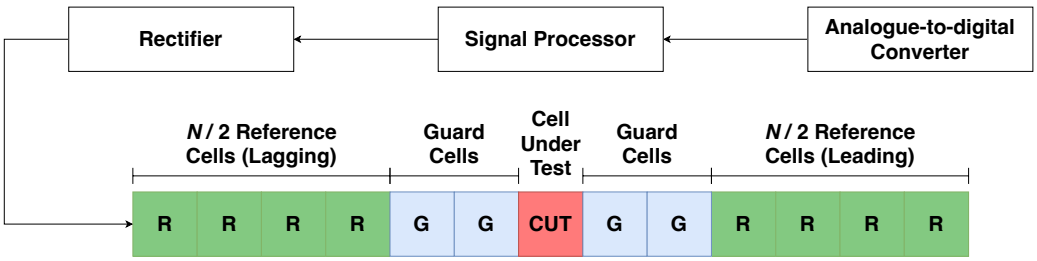


Figure 5.15: A generic CFAR detector's structure and composition

CFAR detectors thus consist of four main elements, namely:

- the cell under test (CUT)
- guard cells (denoted "G")
- N reference cells (denoted "R")
- a CFAR constant, α

The CUT is the position where the threshold is going to be applied. If a target exists in the CUT, the threshold should be exceeded. The guard cells are then used to eliminate any spillover from the target contaminating the reference cells, but the actual samples contained in the guard cells are ignored [222]. The reference cells are thus solely responsible for estimating the background interference statistic [223].

In this experiment, the Cell Averaging CFAR (CA-CFAR) algorithm was employed as it remains one of the most common techniques used in radar detection processing, and it is well suited for applications where there is only a single target in the presence of Gaussian interference [106]. For detection processing to be applied in a homogeneous environment (such as space), the CA-CFAR detector computes the background noise estimate as:

$$\hat{g}_{CA}^{'} = \frac{1}{N}\sum_{n=1}^{N} y_n \tag{5.5}$$

where y_n represents the squared magnitude of the received signal as observed in the reference cells. The CA-CFAR constant α_{CA} is then determined as:

$$\alpha_{CA} = N(P_{FA}^{-\frac{1}{N}} - 1) \tag{5.6}$$

And thus the CA-CFAR threshold T_{CA} is calculated using:

$$T_{CA} = \alpha_{CA}\hat{g}_{CA}^{'} \tag{5.7}$$

For comparison against SOARS, the implementation of the CA-CFAR algorithm was verified using the receiver operating characteristic curves in [106]. This was replicated as shown in Figure 5.16, and it was thus concluded that the implementation of CA-CFAR was correct.

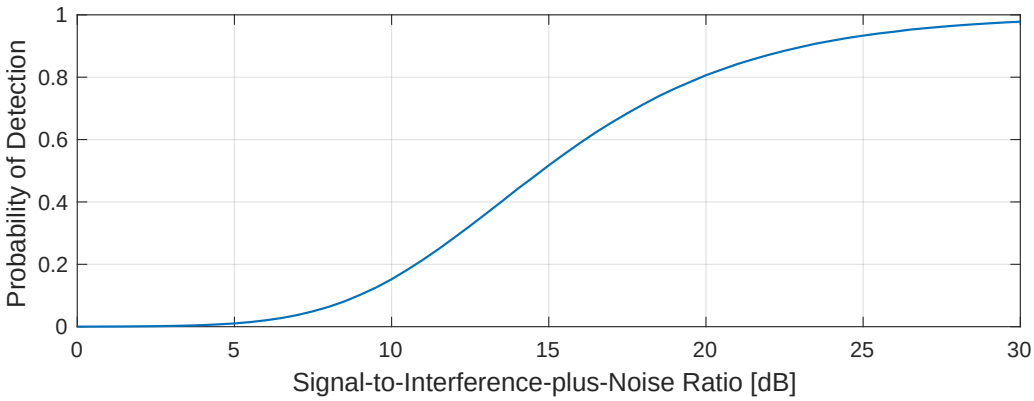


Figure 5.16: Simulated receiver operating characteristic curves used to validate the CA-CFAR implementation

With the CFAR implementation verified, the ISS experiment was executed in SOARS and the resulting output signal was passed through the CA-CFAR detector after applying a matched filter. This produced the results shown in Figures 5.17 and 5.18 for the baseline and ray-traced simulators, respectively. Note that refraction was enabled for added complexity and multiple noise

sources (internal, man-made, galactic, and cosmic) were activated to partially mask the target in interference. CFAR parameters of $N = 72000$ and $P_{FA} = 10^{-7}$ were used with $N/5 = 14400$ guard cells used on each side of the CUT as the target echo spanned thousands of sample bins.

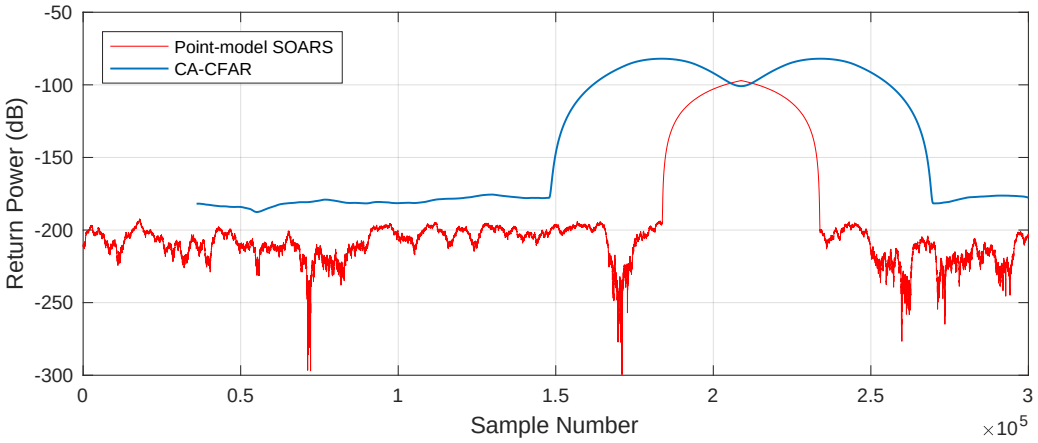


Figure 5.17: Power of the baseline simulator's first return pulse for the ISS experiment, as well as the corresponding CA-CFAR threshold

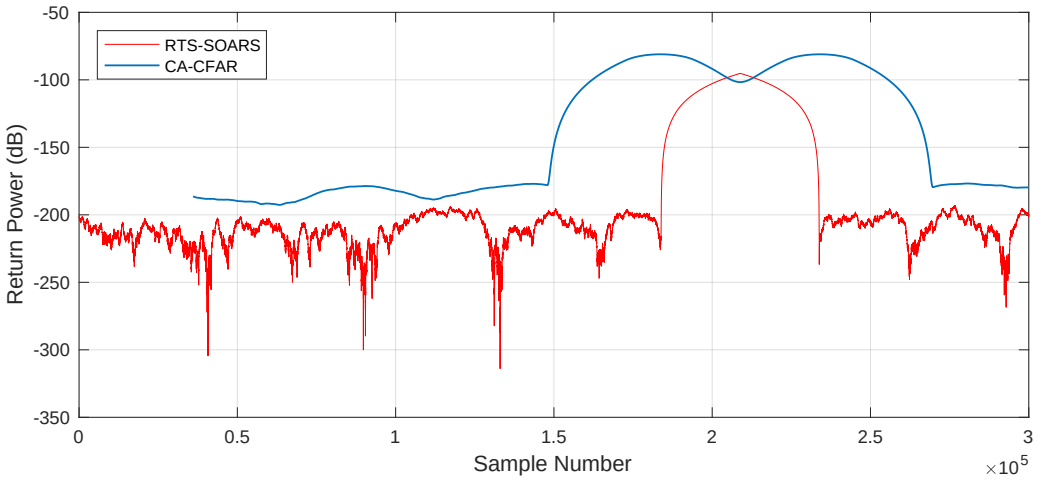


Figure 5.18: Power of the ray-traced simulator's first return pulse for the ISS experiment, as well as the corresponding CA-CFAR threshold

As shown, the CFAR thresholds dipped into the target echo and were thus exceeded by the received signal at its peak; as such, the CA-CFAR processor was successfully able to detect the ISS return in both simulators. This

highlights the target echo amidst the noisy return, indicating that the ISS was indeed detected using the resulting signals from both versions of SOARS. It is also worth noting that the RTS signal peaks approximately 4% higher than the baseline pulse, implying that the spread of the rays and the use of refraction and a mesh-based target have introduced a certain degree of improved accuracy.

Overall, these results demonstrate one possible application of the SOARS software whereby the resulting signal could be used with detection processing methods (or similar techniques) as part of a post-processing chain – thereby validating the purpose and flexibility of SOARS.

5.4.3 Multiscatter Effects

This experiment considered the same bistatic MeerKAT radar design as before – as well as the use of the SEM model and multiscatter effects – in a scenario involving two SGP4-propagated targets modelled in the space environment. This aims to evaluate the impact of ray-traced multiscatter on power and Doppler frequency results generated in SOARS.

Target Scenario

This radar scenario made use of two TLE-based space debris targets (represented by Targets #2 and #3 in Table 5.3) that are propagated using SGP4. These TLEs were propagated through a specified period before the simulation was run to position them close enough to the radar system, establish lines of sight, and allow for transmitted rays to be traced appropriately. The simulation was also set up so that rays propagated from the transmitter to the first target, then re-radiated from the first to the second target, and thereafter reflected off the second target towards the receive antenna. In this way, multiscatter properties could be evaluated.

The simulated scene is represented in Figure 5.19, illustrating the two TLE targets as well as the MeerKAT radar [13] being used to observe them. This made use of the same system parameters shown in Table 5.4, but noise and refraction were disabled for visualisation purposes.

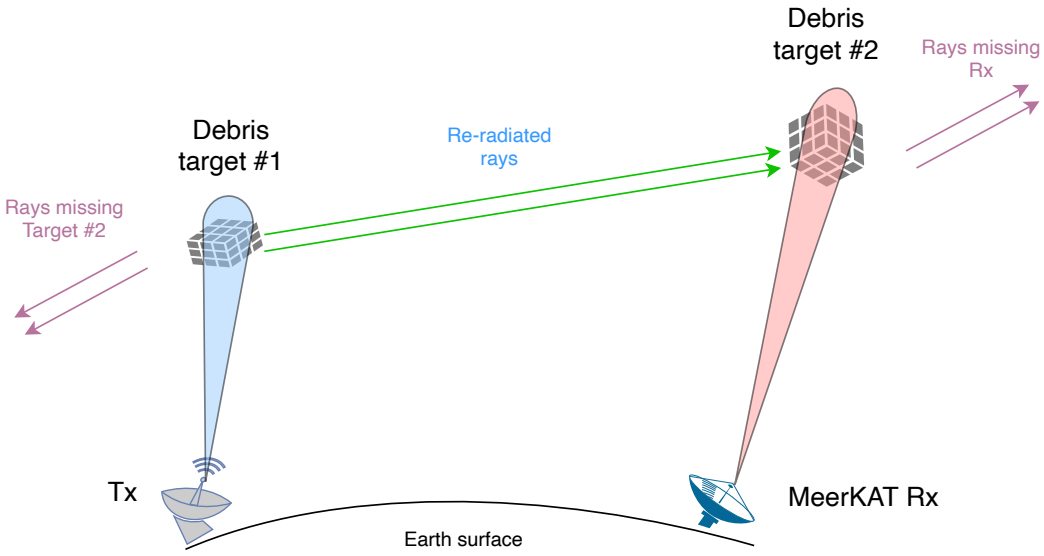


Figure 5.19: Side view of an experiment run in SOARS to simulate the MeerKAT radar with two TLEs as the targets for multiscatter testing, including the antenna beams; the diagram is not to scale

For simplicity, both targets made use of the SEM model to approximate their RCSs, and they were both assigned a Γ of 1 with a rectangular mesh to represent their shape in a physical space with simplified reflections. The first target was assigned a size of (1 x 15 x 15) cm and the second had a size of (10 x 70 x 160) cm. It is worth noting that floating-point limitations may result in changes in this experiment specifically, as the ranges from the radar to the targets are quite significant. The targets in this test are also meant to represent debris objects, meaning that they are modelled with much smaller sizes than the ISS. As a result, such large distances with tiny targets may result in floating-point errors (and thus also self-shadowing on a ray-tracing level).

In properly defining the targets' representations in simulation, their propagated orbits are presented in Figure 5.20 over a period of 120 minutes. These spherical coordinates are plotted for both targets and are taken to be relative to each of the transmitter and the receiver.

This provides a complete depiction of the targets' paths, with the SOARS simulation beginning when the first target is at the 90-minute mark and the

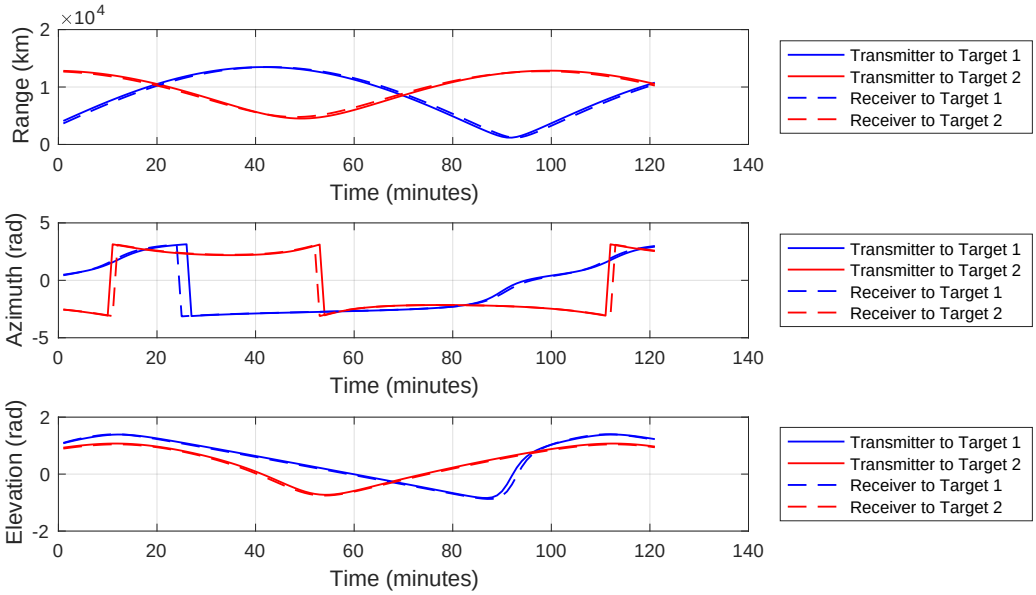


Figure 5.20: SGP4-propagated range, azimuth and elevation of the two TLE-based targets (relative to the transmitter and receiver in a multiscatter test) over 120 minutes

second target is at the 48-minute mark in Figure 5.20.

Power and Doppler Shifts

Running this simulation in the ray-traced and baseline simulators (as far as could be replicated in both programs) produced the results shown in Figure 5.21.

This demonstrates several notable points:

- The RTS output only produced one return pulse that arrived significantly later than the first baseline pulse from either target, as only the first target (in the RTS) was illuminated by the transmitter beam. This is because the signal that reflects off the first target was traced to the second target (illuminated by the receiver beam) and then re-radiated to the receive antenna, resulting in a significantly longer time delay.
- As a result of the above, only one pulse was received within the observed time window in the RTS, whereas the baseline simulator received multi-

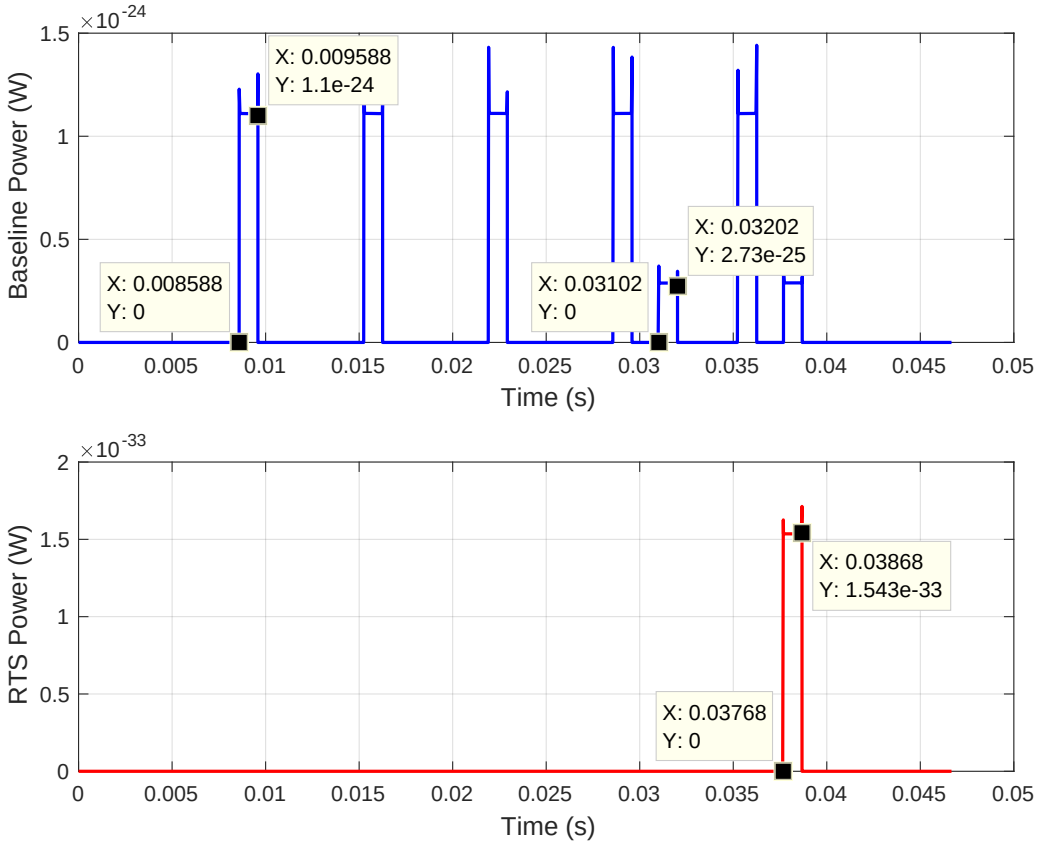


Figure 5.21: Measured signal power resulting from the multiscatter experiment after being run in both the baseline and ray-traced simulators

ple pulses from both the first and second targets due to its single-scatter approach. The point-model simulator thus incorrectly assumed that *both* targets were illuminated by the transmit beam, as every target is forced to produce a received echo regardless of its position or orientation.

- The increased time delay in the RTS also implies much longer ray lengths, meaning that the multiscatter power was significantly impacted by the extended range. The baseline simulator, however, experienced much greater return power in each pulse due to the shortened ranges.

The labelled data points in Figure 5.21 also highlight the first pulse received from each target (in baseline SOARS) as well as the first multiscatter pulse received in RTS-SOARS, showing that the received pulses are of the correct length (1 ms) and agree with the expected time delays and power values. These results also confirm that the path followed by the RTS transmit signal is significantly longer than the baseline simulator path – as expected.

The use of multiscatter also impacted the Doppler shifts that were observed by the system, as illustrated in Figure 5.22. This shows measurements of the Doppler frequency at various points during execution – including at the points of ray intersection with each of the two targets and the receiver, as well as the average Doppler shift computed during ray aggregation.

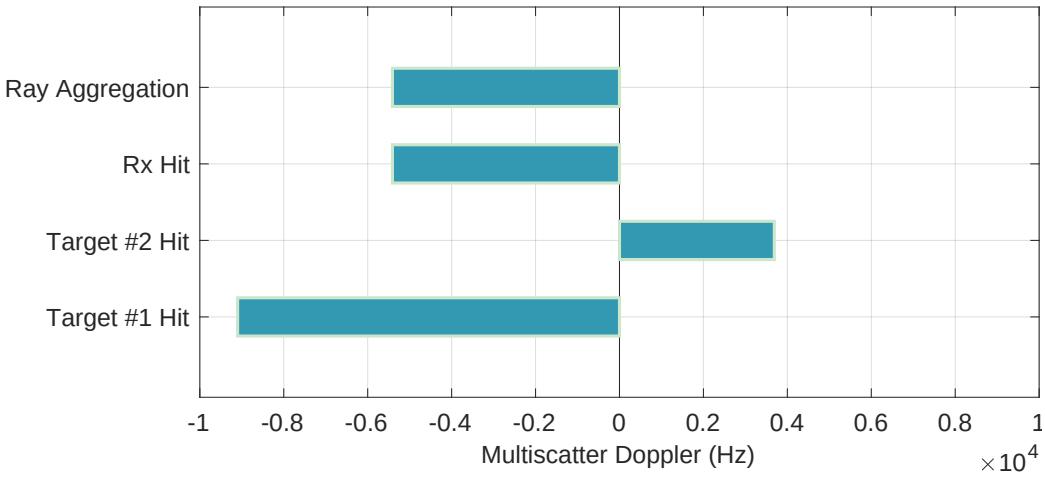


Figure 5.22: Total multiscatter Doppler shift computed at various points during the simulated experiment

For comparison, the baseline simulator yields an approximate Doppler shift of 22938 Hz for the first target and a shift of -1167 Hz for the second target. These were recorded separately as each target produced its own return due to baseline SOARS's single-scatter implementation. Comparing this against the results in Figure 5.22 further highlights the significant differences that can arise by accounting for multiscatter effects – particularly at such long ranges.

As part of this experiment, one final test was conducted purely for visualisation purposes, whereby Gaussian noise was added to both the baseline and ray-traced simulator output signals with a low noise power P_n. This was done to observe the impact of noise on each signal and to assess how CA-CFAR (known for its ineptitude in detecting multiple targets) would perform in detecting several target returns. The results of this are illustrated in Figure 5.23, where CFAR parameters of $N = 72000$, $P_{FA} = 10^{-7}$, and $N/5 = 14400$ guard cells (on each side of the CUT) were used.

As indicated, the CFAR algorithm was able to detect most of the target returns in the baseline signal (where each wide lobe corresponds to one of the target return pulses shown in Figure 5.21); however, one target return was missed as indicated in Figure 5.23. This could be attributed to CA-CFAR specifically not being optimised for multiple targets, but the results also demonstrate how the point-model simulator generated easily-detectable target echoes – namely because its original power signal was much stronger than that of the ray-traced simulator. Despite this, however, it was still unable to detect every target return present in the signal.

In the RTS result, the single expected target echo was completely buried in interference; moreover, a false alarm was also raised at one of the higher peaks of the matched-filtered signal. The RTS's more prudent output thus yielded both a false detection *and* missed the actual target return due to its low power – once again highlighting the significance of multiscatter effects. It can thus be seen that, in a real-world scenario, the ray-traced result is much more likely – whereby the target returns are drowned in noise due to the positions of the targets relative to the antenna beams.

These results illustrate how RTS-SOARS still produces a realistic output

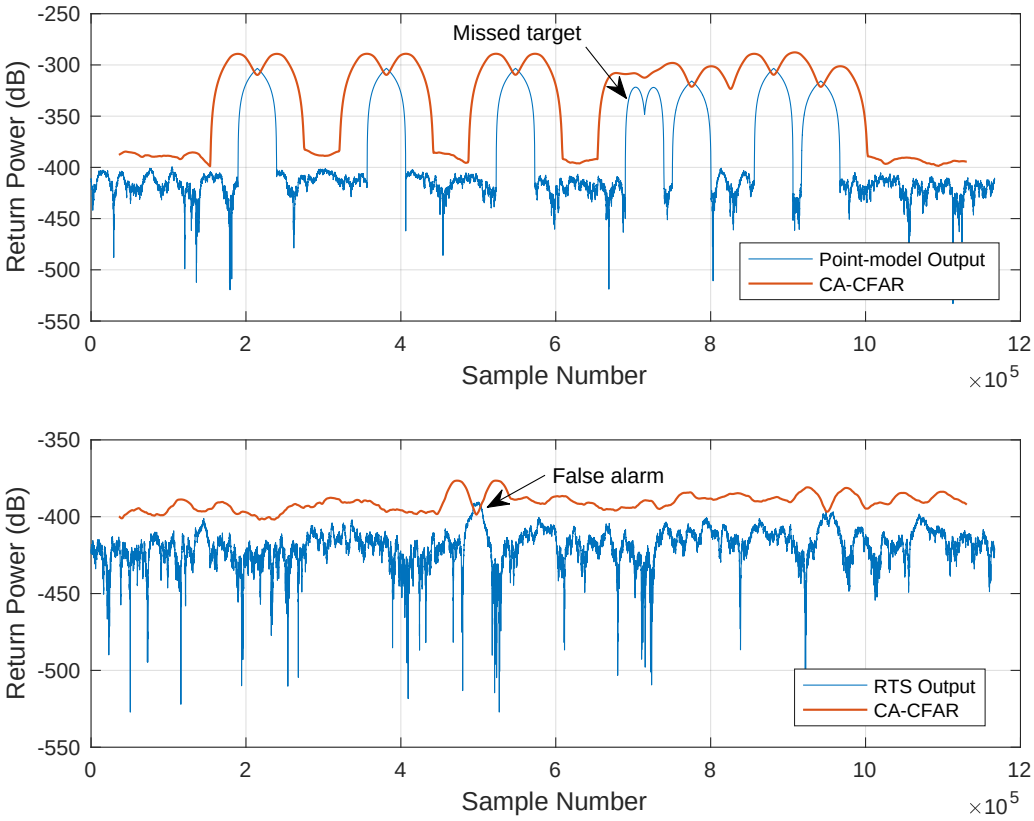


Figure 5.23: Measured signal power resulting from a noisy multiscatter experiment after being run in both the point-model and ray-traced simulators, along with their corresponding CA-CFAR thresholds

when considering small targets at large distances, and highlights how great the difference between multiscatter and single-scatter returns can be. It was shown that baseline SOARS receives echoes from *both* targets independently – despite the second target not transiting through the transmitter's beam at all. As a result, it is worth noting that the RTS return had more accurately accounted for the spread of the pulse's electromagnetic energy – with all the signal's power concentrated within the transmitter's beamwidth.

5.5 Computational Benchmarks

This section aims to compare the baseline and RTS versions of SOARS for the four experiments conducted in Section 5.4. Additionally, benchmarks are run for the file processing used at the input of the software (and optionally for the output) as well as for the use of CUDA for SGP4 propagation.

5.5.1 RTS vs Baseline

Using the RTS was always expected to result in significant performance degradation relative to the baseline simulator, and this is attributed to the set-up, execution, and processing of OptiX$^{\text{TM}}$-based ray tracing, CUDA, and all associated GPU overhead. The ray aggregation algorithm also significantly increases runtime in having to iterate through large numbers of rays and making use of two CUDA kernels.

As part of this work's investigation in comparing the two simulators, all four of the aforementioned experiments (in this section) were run in both the ray-traced and baseline simulators and their runtimes were recorded. These are reflected in Figure 5.24, showing the ratios of the RTS-SOARS runtime to that of baseline SOARS for each experiment. Measurements are shown for each program's *complete* runtime, i.e., the full time taken to read in the simulator inputs, process the simulation, and then output a received signal.

This demonstrates the speed-ups experienced by using the simplified point-model program with no ray tracing (i.e., baseline SOARS). Interestingly, the experiments involving the shadowed and overlapped targets showed the

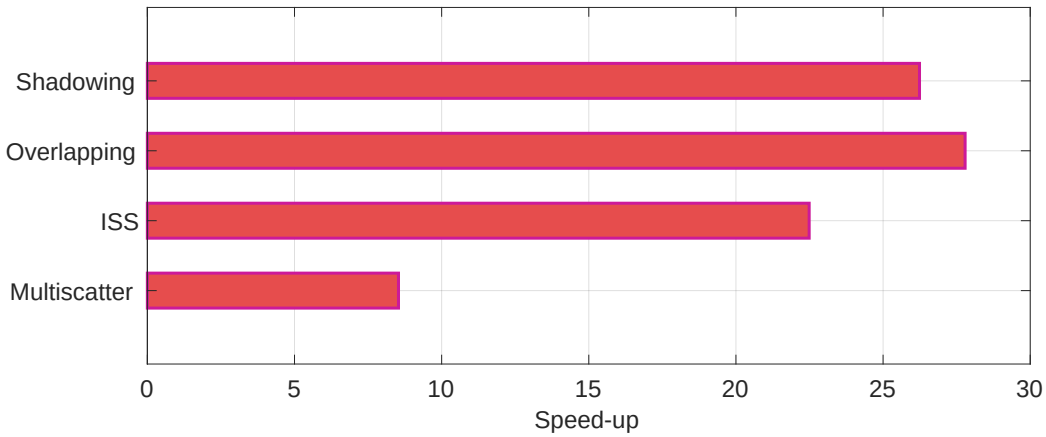


Figure 5.24: Speed-up observed in baseline SOARS relative to ray-traced SOARS for each of the four benchmarked experiments

greatest performance increases through baseline SOARS, which provided a speed-up greater than 25x. This is attributed to the longer runtime used in these experiments relative to the other two, each transmitting 16 pulses instead of 4 (ISS experiment) and 6 (multiscatter experiment). This is due to the RTS having to re-run the OptiX^{TM} engine many more times, resulting in much greater GPU overhead and thus larger slowdowns.

The benchmarks in Figure 5.24 also depict a notable difference in speed-up for the ISS and multiscatter experiments (greater than 20x and 5x, respectively), and this is attributed to the use of refraction and a file-based mesh in the former test. Each of these factors significantly increased the RTS runtime due to the intensive processing (and additional ray spawning) required for refraction and the reading in of long text files to generate the ISS mesh in the form of vertices and vertex normals. The RTS also needs to account for many additional steps and sub-processes in its implementation of propagation modelling and radar computation, such as engine initialisation and ray aggregation. These were timed for each of the four experiments as presented in Table 5.5.

As depicted, the ISS experiment had the longest runtimes across all tested processes – and by a significant margin. This is attributed to the use of a complex file-based mesh to model the ISS target, whereas each of the other

Table 5.5: Measured RTS timings (in seconds) for various sub-processes (initial set-up, OptiX^{TM} kernel execution, ray aggregation, and post-processing of results) for the first transmit pulse in each of the four benchmarked experiments

Experiment	**Set-up**	**Kernel**	**Aggregation**	**Results**
Shadowing	1.473	2.149	3.5×10^{-5}	5.6×10^{-3}
Overlapping	1.488	2.145	3.9×10^{-5}	6.6×10^{-3}
ISS	6.829	2.723	5.5×10^{-5}	3.0×10^{-1}
Multiscatter	0.173	2.174	5.0×10^{-5}	7.6×10^{-2}

experiments made use of much simpler shapes and required no additional file reading to represent the targets' vertices and vertex normals; the use of refraction also resulted in significantly more processing relative to the other three scenarios. Both the ISS and multiscatter experiments also demonstrate much longer runtimes for post-processing relative to the shadowing scenarios, as both had to process a larger number of received rays. However, it is also worth noting that the shadowing experiments have increased set-up times relative to the multiscatter experiment – likely due to the GPU overhead in setting up CUDA at the same time as the OptiX^{TM} context for simulations involving manually-positioned targets (as opposed to SGP4-propagated targets, in which case CUDA is initialised earlier in the simulation).

Memory comparisons between the baseline and ray-traced simulators were also conducted across both the CPU (main memory pool) and GPU (memory utilised by CUDA). The same four experiments were tested and their maximum memory usage were recorded as shown in Figure 5.25. As presented, running the ISS experiment in RTS-SOARS uses nearly 4x as much GPU memory relative to baseline SOARS, and up to around 1.5x the CPU memory. In general, a trend is observed where the RTS uses at least 2x the GPU memory of the baseline simulation while the CPU memory footprints are more closely aligned across the two programs.

Despite the notable disadvantages of using the RTS, it is important to consider the simulator differences resulting in these results. The worst performance

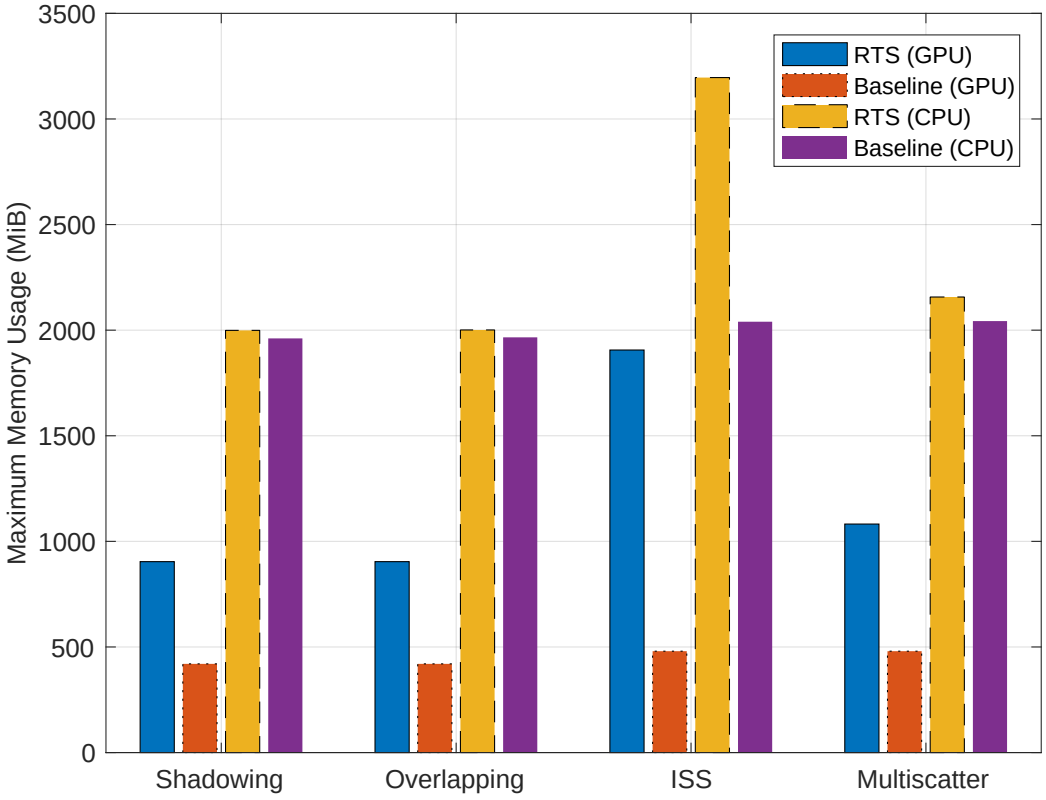


Figure 5.25: Total memory usage (across the CPU and GPU) observed when running each of the four benchmarked experiments in baseline and ray-traced SOARS

slowdown would likely be experienced when millions of rays are traced (including refracted rays) relative to just one "ray" in baseline SOARS. However, it is worth noting that these rays are traced in *parallel* through OptiX™, whereas the equivalent scenario in baseline SOARS would be to simulate an equivalent number of targets with each one mapped to a single "ray" and processed serially; this would, logically, be significantly slower than the RTS in computation.

The baseline execution runtime would thus also be worsened as it would need to post-process and aggregate the same number of ray responses as the RTS. As is, the RTS performs more computations and yields a greatly improved ray resolution relative to the slowdown. Nonetheless, this expected performance degradation is important to note for anyone using the RTS. And while the ray-tracing algorithm *does* lead to improved modelling accuracy, software runtime and memory utilisation remain important factors to consider and are notable advantages provided by the baseline program.

5.5.2 XML Processing

To verify the potential performance improvements brought about by PugiXML (as discussed in Section 3.5), both PugiXML and TinyXML were tested in practice within the context of SOARS. Timing tests were thus conducted by measuring the time taken by each of PugiXML and TinyXML to read and write specific SOARS files. It should be noted that there are slight discrepancies in the XML file structure used by each method due to the change from "platform" nodes to "system" and "target" nodes. As a result, there are slight variations in the sizes of files used by TinyXML and PugiXML, but in practice, these discrepancies are insignificant to the parser's performance.

The XML timing tests were conducted using the same simulation scenarios and executed on the same hardware running the same operating system – namely the V100 CPU-GPU node detailed earlier. The experiment made use of a monostatic radar with a single pulse, a single target, and generic parameters values offering predictable results for a typical space-oriented radar experiment. Some parameters were selected such that performance

benchmarking could be more easily conducted, but the full list of system parameters is not provided for this experiment as only the relative performance is being investigated – not the actual output values or shapes.

The target in this experiment was a TLE-based object that was propagated through nine time-steps between $t = 0$ ms and $t = 80$ ms in intervals of 10 ms. The transmit pulse was defined by a linear chirp with a bandwidth of 10 MHz and a width of 1 ms, and the target range was set as 463.405 km at $t = 0$ ms. All noise sources were disabled for simplicity.

The results of the parsing tests are demonstrated in Figure 5.26, illustrating the speed-up provided by PugiXML over TinyXML as a ratio of their performance, i.e., TinyXML's runtime to PugiXML's runtime. Each parser was tested for several full SOARS simulation files with a varying number of targets – each of which is a duplicated TLE set propagated through nine time instances. Each simulation used the same radar position and specifications, and the simulation runtime was a mere 80 ms chosen purely for the sake of being a multiple of the PRI (10 ms in this case). These recorded timings thus correspond to the time taken to fully parse a typical *.soarsxml* file into the baseline SOARS software (with the same implementation being used in RTS-SOARS).

The read speed ratios in Figure 5.26 show that PugiXML parsed the input file at up to 13x the speed of TinyXML, and the performance gap widens as the simulation inputs became larger. It also provided a speed-up of up to around 2x that of TinyXML when writing larger output files, but it is worth noting that XML outputs can be optionally disabled by the user as it is not pertinent to the operation of the simulator; only the HDF5 output is critical. Regardless, the results illustrate how the use of PugiXML provided significant performance improvements over TinyXML for every simulation tested – both in terms of reading and writing speeds.

It should be noted that there are slight variations in the output file size generated by PugiXML and TinyXML, but these discrepancies are ultimately insignificant to the parser's runtime performance. As the actual simulation scenarios are the same across both the PugiXML and TinyXML test cases, any variations in the output file sizes are attributed to the tiny differences in

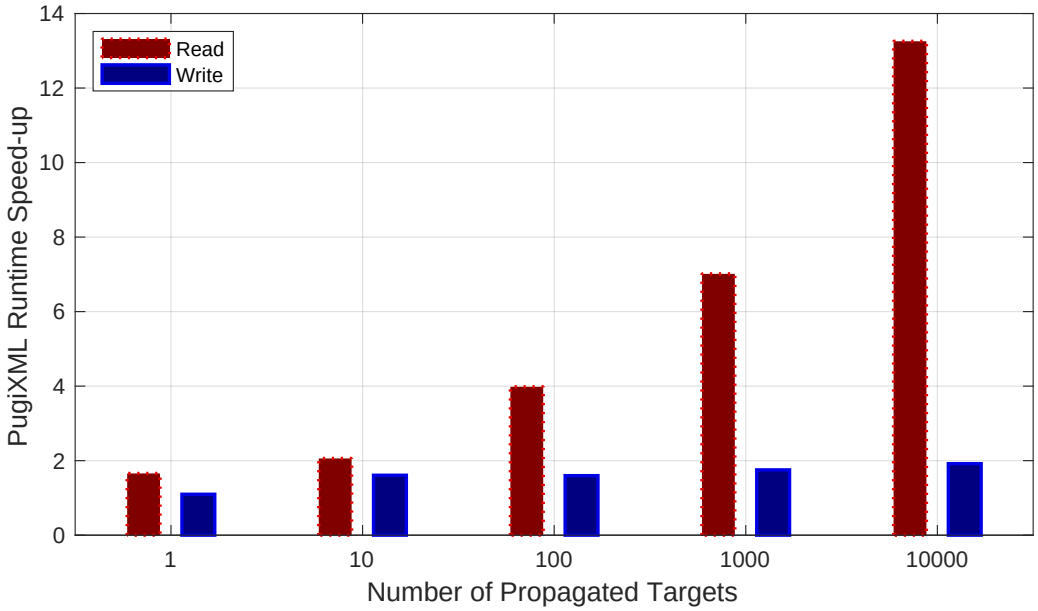


Figure 5.26: Speed-up in read/write times using PugiXML over TinyXML for *.soarsxml* simulation files with a varying number of targets, as averaged over five consecutive runs

spacing sizes and element naming conventions in the XML files; these result in slightly larger file sizes when using TinyXML.

5.5.3 Orbit Propagation

It is crucial to consider the possibility that the use of CUDA in the SGP4 algorithm (as discussed in Section 3.5) may *degrade* code performance – as opposed to accelerating it. Benchmarking tests were thus conducted for the SGP4 implementation, comparing the serial version against the CUDA alternative developed in this work. This entailed conducting several ordinary SOARS simulations with a varying number of targets (ranging from 1 to 1,000,000) – each propagated through nine time-steps – and then using a C++ timer function to measure the time taken to process all targets. This approach was followed for both the serial and CUDA implementations of the SGP4 code.

All benchmarking tests were conducted on the same heterogeneous V100

CPU-GPU hardware described previously with the same simulation set-up as the previous experiment. The results of these benchmarks are illustrated in Figure 5.27, showing a ratio of the serial software's runtime to the CUDA version's runtime for an increasing number of targets.

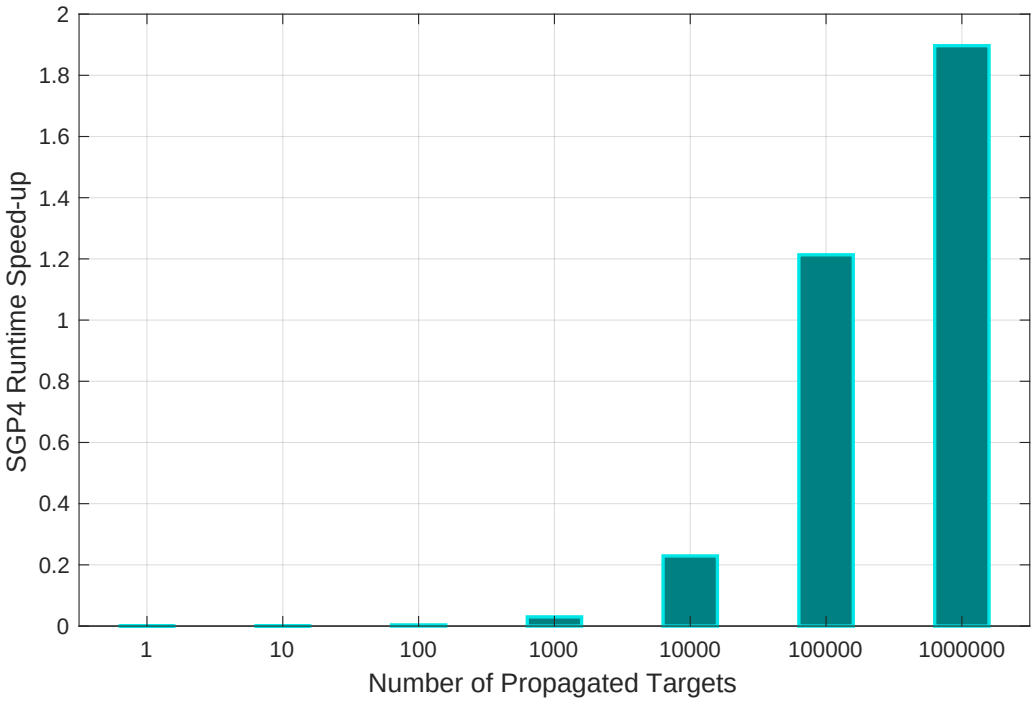


Figure 5.27: Speed-up in processing times for propagating a varying number of targets using SGP4 functions serially and with CUDA

From the results depicted in Figure 5.27, it is clear that there is indeed a significant drawback to using the CUDA version when running small-scale simulations. Based on the data, the CUDA version of SGP4 runs significantly slower than the serial version for small simulation sizes. It also requires nearly the same amount of processing time regardless of the number of targets for up to 10,000 targets. Thereafter, the results scale differently based on the size of the simulation. This is attributed to the larger array sizes that require copying to/from the device, which increases the runtime non-linearly.

For simulations with more than 100,000 targets, the CUDA version outperforms the serial SGP4 algorithm. Despite the CUDA version's slower performance for small simulations, the maximum runtime difference between

it and the serial version is only around 4 s; once the CUDA version overtakes the serial version, however, the runtime gap is significantly larger at around 13 s. This makes the CUDA version a viable candidate for large-scale simulations with large numbers of targets, and overall, the results confirm that the CUDA-based SGP4 implementation is well suited for large-scale deployment – as per the software requirements.

5.6 Evaluations and Discussion

This section presents an evaluation of the research investigation conducted in this work and an assessment of the completed software program against the user requirements in Section 3.2.

5.6.1 Analysis of Results

Overall, the following main points have been highlighted in this chapter:

- Both the baseline simulator and the RTS-enabled SOARS programs have been verified against the theory and expectations as well as validated through various means. While measured radar data was unavailable for final comparisons, tests were conducted between the RTS and PAST (an existing, well-established simulator) and the validation results were published in [49]. Additionally, comparisons were made between SOARS and its baseline version – the latter of which was verified in [48] – and the outputs agreed with expectations. Based on the comparisons between the baseline and RTS versions of SOARS, it was concluded that there are significant trade-offs to using each program, but overall, SOARS was deemed fit for deployment in real-world use cases involving the design of space-monitoring systems.
- In addition to the previous point, the flexibility of SOARS (and the RTS on its own) should allow for generic radar designs as well. The RTS could be separated into its own independent module and easily retrofitted onto an existing simulator such as FERS [25], allowing for applications that are not orientated towards space. Such results were

also demonstrated earlier in this chapter through the multi-ray and multi-target experiments, which each made use of generic position waypoints to model the targets' trajectories (as opposed to using TLEs and SGP4).

- As shown by the benchmarks, baseline SOARS boasts nearly real-time operation and its speed is held back only by CUDA's GPU overhead when running small-scale experiments. Its implementation of point models also makes it well suited for most applications – particularly when dealing with small objects such as space debris, as the point target approximation offers decent fidelity for such tiny targets at long ranges.
- RTS-SOARS offers notable advantages over baseline SOARS in object and propagation modelling, particularly due to its optics-based implementations. Despite the trade-offs in runtime speed and memory, this version provided more realistic outputs in every tested experiment and successfully (1) demonstrated target shadowing, (2) avoided returns from overlapped targets, (3) accounted for refraction, beamwidth, and mesh modelling through the ISS experiment (showing how large space objects can be accurately modelled), and (4) included multiscatter effects in the computation of power and Doppler. As a result, the RTS is observed to be notably more accurate than the more simplistic propagation and target implementation in baseline SOARS.
- Based on the two previous points, it can be concluded that both the baseline and RTS programs offer some degree of usefulness under different scenarios. The ISS experiment, for instance, only demonstrated a minor error in peak power of around 4% between the simulators but baseline SOARS is approximately 22x faster in generating the result. However, the multiscatter experiment showed a much more significant change in output; despite the baseline program's 8x speed-up, the RTS provided a more realistic result by accounting for targets transiting through the antenna beams and implementing multiscatter effects.

As a result of these discussions, it was concluded that the initial hypothesis of this investigation was indeed correct; both baseline SOARS and RTS-SOARS

were sufficiently compared under various testing conditions, and multiple potential applications were highlighted in this chapter. The comparison between these two simulators could also be further investigated through representative case studies, as well as their potential use for designing intricate radar systems or measuring prediction accuracy against real-world data.

with the main difference between the two being the implementations of radio propagation and target representation. In general, it was found that the baseline program ran significantly faster than the RTS model, but this came at the cost of simulation accuracy. While the RTS does not always guarantee drastically differing results, it was found to be able to handle specific scenarios (such as shadowing and multiscatter) more realistically. The point-model is thus ideal for near-real-time applications, while the RTS is best suited to maximising accuracy – such as for designing complex systems or measuring prediction capabilities.

With this, the hypothesis supporting this work was proven correct and the investigation was concluded.

5.6.2 User Interface

A basic GUI was developed for SOARS using C++ and the Qt library [224], acting as a complete front-end interfacing tool for SOARS. This allows users to specify all their simulation parameters, such as definitions for transmitters, receivers, antennas, pulses, and targets, through user-friendly and highly visual means.

A screenshot of the GUI is depicted in Figure 5.28, which shows the first panel of the program through which all system-related inputs can be defined. As presented, this panel also serves as a platform for defining the simulation parameters, such as the simulation name, time, sampling frequency, PRF, target sampling interval, and more.

The second panel of the SOARS GUI is shown in Figure 5.29, where all simulated targets can be defined by their TLE sets as well as additional parameters that are required for RTS processing.

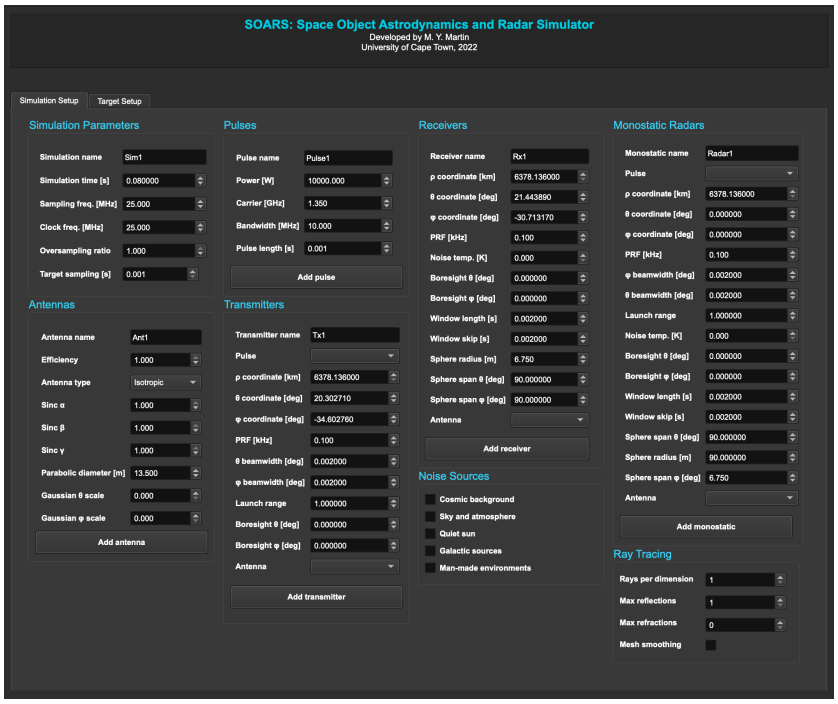


Figure 5.28: Screenshot of the first panel of the SOARS GUI

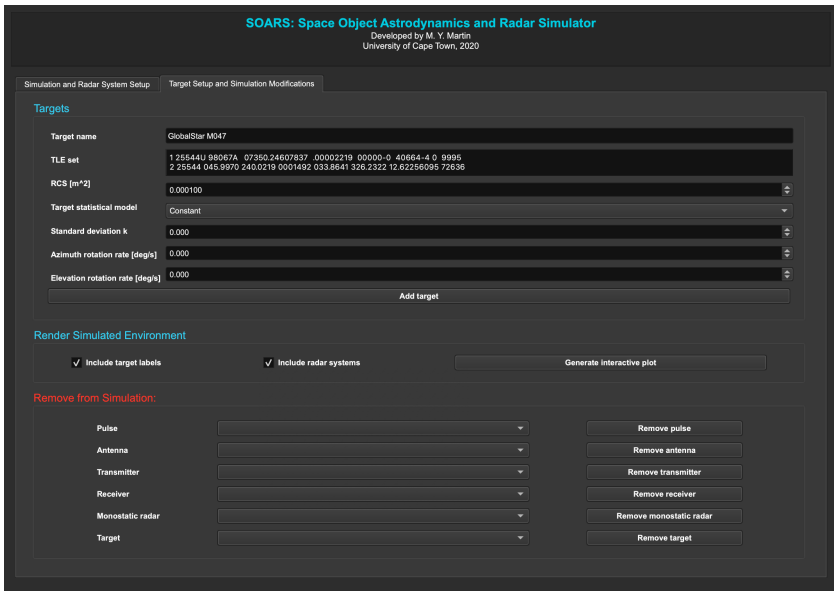


Figure 5.29: Screenshot of the second panel of the SOARS GUI

Once all the inputs are specified, the simulation can be exported into a singular *.soarsxml* file to be executed on a system supporting CUDA. This output file can also be directly loaded back into the GUI if necessary, retaining all the antenna, pulse, radar, and target information that was previously saved. This is useful for validating a target's trajectory, calculating distances between targets and radars, and more – and all of it can be done independently of the main SOARS program.

The use of this GUI offers several advantages over alternative programs:

- Setting up the core simulation is intuitive and user-friendly, enabling simpler interaction with SOARS. This is beneficial for any future work done that is on the project, and some users may not need to interact with the source code at all.
- The use of Qt (in developing the SOARS GUI) provides a robust and extensible design, as the GUI could easily be modified with additional features due to its minimalistic approach. This is easily achieved using programs such as Qt Creator [225], which provide simplified interfaces for Qt design.
- The GUI can be compiled on multiple platforms and operating systems.
- The use of the *.soarsxml* file allows simulations to be run independently of the GUI, enabling the use of higher-throughput hardware and GPU architectures. Even after processing, the *.soarsxml* can easily be loaded back into the GUI if changes are required.

As such, it is highly recommended that users utilise this interface to design and modify their SOARS simulations.

5.6.3 Requirements Evaluation

The final step of the project was to compare the developed software against its initial design goals; this was achieved by assessing various aspects of the software in relation to the specified non-functional and functional user requirements of SOARS. These evaluations, as well as how they were satisfied by the completed software, are summarised in Tables 5.6 and 5.7, respectively.

Table 5.6: Evaluation of the non-functional user requirements against the completed SOARS software

Requirement	**Evaluation**
1. Portability	This requirement is satisfied via the use of C++ as the main programming language. This represents an open-source language that is highly portable and compilable across a variety of operating systems and hardware architectures.
2. Speed	This requirement has been met through the benchmarking tests performed in this chapter and Chapter 3. These tests have highlighted the significant speed-ups achieved over the original FERS software, and how CUDA has improved the throughput of software models such as SGP4.
3. Accuracy	This requirement has been satisfied through the validation tests conducted in this chapter. These tests have confirmed the validity of the simulator's results and its accuracy against established measured data.
4. Usability	This requirement is satisfied via the developed user interface, providing ease of use and accessibility to users with various levels of experience. This is further satisfied by the publication of a software guide that has been released with SOARS.
5. Modularity	This requirement has been met through the frequent use of comments in the source code as well as the software guide that has been published alongside the simulator and the use of user-friendly programs in designing the software. This provides maximum modularity in the long term, making the software easily editable, adjustable and scalable as the user requires.

Table 5.7: Evaluation of the functional user requirements against the completed SOARS software

Requirement	**Evaluation**
1. Measured datasets	This is satisfied via the use of TLE datasets, which are based on real-world measured data of RSO trajectories. These are also updated regularly, with many being available for public use and dissemination.
2. Keplerian dynamics	This requirement has been met through the SGP4 algorithm, which accounts for orbital perturbations such as the effects of orbital decay and the Earth's shape and gravitational forces. The SGP4 model also works directly with the measured TLE datasets.
3. Radar theory	This requirement has been satisfied by using FERS as a radar simulation base, accounting for most of the standard radar hardware models and computations such as RCS, propagation loss, antennas, system noise, Doppler shifts, and more.
4. Ray tracing	This requirement is satisfied via the integration of the NVIDIA® OptiX™ ray-tracing tool into SOARS. This presents a highly-optimised model for implementing ray tracing and provides a much more comprehensive target model compared to FERS, accounting for an object's size, shape, and material parameters.
5. Noise sources	This requirement has been met through the use of the external noise temperature models presented in Chapter 3 and in [173]. These account for various noise sources in the simulation environment and directly affect the resulting received signal(s).

This addressed the final step of the project methodology as discussed in Section 3.1, and all the software design requirements were found to be satisfied by the developed simulator. As a result, the project was deemed to be completed and conclusions were drawn based on the research results presented.

5.7 Conclusions

This chapter served the purpose of presenting the main results for the SOARS software, validating its operation, and testing the developed simulator's performance under various conditions. With this, the implementation and design of SOARS were shown to satisfy the main objectives of this work as well as the software requirements presented in Section 3.2.

Results from both the baseline and ray-traced simulators were also compared and the differences were discussed. It was thus concluded that the baseline program ran significantly faster than the ray-traced version at the cost of accuracy and flexibility. While it was found that the RTS did not always generate drastically different results relative to the baseline software, it was able to handle specific scenarios more realistically and provided more reliable outputs in those experiments.

Chapter 6

Conclusions and Future Work

6.1 Summary and Contributions

This thesis detailed the design and development of the Space Object Astrodynamics and Radar Simulator, SOARS – a high-performance signal-level radar simulator to be used for simulating and predicting LEO space environments. SOARS can simulate large numbers of spaceborne targets orbiting the Earth, as well as an arbitrary number of terrestrial radar systems, and the software's portability, usability and accuracy together make it a useful contribution in the field of astrodynamics and space science – particularly for further research purposes.

Additionally, the developed program's flexibility and modularity make it ideal for use in training and development —- allowing for students, astronomers and radar engineers to explore potential radar designs with minimal constraints on the simulation, system, or target parameters. SOARS is thus also useful for training and education purposes, such as through the demonstration of the effects of various parameters on a return signal, generating received signals for post-processing activities, or enabling engineers to estimate the performance of theoretical radar systems (such as the MeerKAT design).

Because of the limitations of commodity hardware, GPU-based parallelism was used to broaden the capabilities of SOARS, and thus parts of the software

were parallelised using NVIDIA® CUDA. This was mainly to allow for large numbers of space targets and radar systems to be simulated in the baseline version of SOARS, but this also served as the basis for the implementation of the Ray Tracing Simulator – the RTS – via NVIDIA® OptiX™, which was used in the final iteration of the software. Both the baseline and RTS versions of SOARS, as well as the RTS itself, serve as the main original contributions of this work. The research documented in this thesis was also reviewed and published in [48], [49], and [50].

This thesis detailed the use of the baseline and RTS versions of SOARS in investigating a hypothesis, which posited that *both* could be useful for some scenarios and for different end-users. This thesis thus focused on the design, development, testing, and comparison of these programs for various applications, presenting the results in the form of analysis and discussion on their advantages and disadvantages. The implementations of both programs have also been presented in significant detail, highlighting how the main requirements of the software were satisfied as per Tables 5.6 and 5.7.

As a result of this discussion, it was concluded that the hypothesis of this investigation was correct and potential applications for both the baseline and RTS versions of SOARS were demonstrated under different circumstances. In particular, it was found that the baseline program is significantly faster than the RTS version (with a speed-up factor of up to 27x) and utilises less memory (with the RTS requiring up to 2x the baseline simulator's GPU memory consumption). It was also noted that the RTS provided very minor changes in one experiment, with the output signal peaking only 4% higher than the baseline output, but it can also provide drastically different results under certain circumstances. As such, it is capable of handling some scenarios (such as target shadowing and overlapping, multiscatter effects, beamwidth modelling, and others) more realistically than the point-model simulator.

As such, the decision of which software is more appropriate depends upon the end-user's needs and priorities. When modelling small targets, such as space debris with physical dimensions smaller than the range resolution of the radar system, the point target approximation used by baseline SOARS would likely be sufficient due to the range at which the target travels. However, when

modelling larger targets that span multiple range bins, such as a space station, it is more accurate to use the RTS to model the target as a complex mesh in 3-D space. The point-model simulator is thus mainly recommended for near real-time applications while the RTS should be used to maximise accuracy – such as for designing intricate radar systems or measuring prediction accuracy against real-world data.

6.2 Future Work

This section describes possible future work to be done on the SOARS simulator, including potential improvements to internal models. Additionally, further investigation and development approaches are presented for future consideration.

RTS Improvements

The RTS model could be further optimised to better distribute the computational load, and it could be very much improved in future iterations through further parallelisation of some of its methods; for instance, each transmitter in a simulation is currently processed serially in the RTS, resulting in many calls to the OptiX™ kernel. Theoretically, it may be possible to adapt this method such that the kernel is only invoked once and all host-to-device data transfer is done simultaneously. However, this may drastically increase memory usage – particularly on the device side.

Further improvements could also potentially be made by using the latest version of OptiX™. In the development of the RTS, an older version of the OptiX™ engine was used based on compatibility with the available V100 GPU as well as CentOS 7 – but users should be able to easily adapt the code for a newer version of the software if they have compatible hardware available. This may come with a host of improvements such as additional built-in optimisations and a more easily adaptable framework.

Additionally, further improvements could be made to the ray-tracing implementation by accounting for additional modelling effects – such as directional

RCS, diffraction, diffuse scattering, terrain and atmospheric effects on signal propagation, and more. However, such features would further increase the computational and algorithmic complexity, but the trade-offs may be worthwhile if they lead to an even more realistic model.

Space Debris Tracking

A major potential avenue for future work would be to research the implementation of tracking filters to the SOARS software. Based on initial research work that was done on this topic, Gauss-Newton filtering would likely be a viable technique for statistical orbit determination purposes (see [226]) – entailing the use of a tracking filter to estimate an RSO's orbit. Such a model was implemented for a similar purpose in [4], where it was used to schedule the MeerKAT sensor array for orbit determination. This could potentially be combined with SOARS to provide a more complete package for space monitoring.

In relation to this, future work could also be done in using SOARS to predict the performance of new radar systems (such as the MeerKAT radar designed in [13]) in monitoring space debris. Through this, the implementation of the developed SOARS model could be validated even further using measured data from a real-world system that was designed using the simulator. This could also be taken a step further by considering the possibility of using SOARS for space-based radar designs – an extension of its terrestrial capabilities. An example of such a system is presented in [81].

Object Database and Collision Modelling

Another potential area to explore would be to extend the SOARS simulator to act as a database of known RSOs and their orbits (such as [82]), which would allow it to be used as a complete tool for space mission planning based on the orbits of known clutter and targets. This would be especially valuable if the simulator was able to further model objects' interactions with one another, such as collisions and the scattering forces involved in these impacts.

References

[1] S. Majumdar, "Flexing Muscles in Space The ASAT success," *Vayu Aerospace and Defence Review*, no. 2, pp. 25–26, 2019.

[2] V. Akhmetov, V. Savanevych, and E. Dikov, "Analysis of the Indian ASAT test on 27 March 2019," *arXiv preprint arXiv:1905.09659*, 2019.

[3] D. H. Humes, "Large craters on the meteoroid and space debris impact experiment," *NASA Technical Reports Server*, 1992.

[4] A. R. Dhondea, "Mission Planning Tool for Space Debris Studies with the MeerKAT Radar," Master's thesis, University of Cape Town, 2018, available at: https://open.uct.ac.za/handle/11427/29617. [Accessed 21 August 2022].

[5] J.-C. Liou, "Orbital debris challenges for space operations," in *ICAO / UNOOSA Symposium in Abu Dhabi, United Arab Emirates, 15-17 March 2016*. NASA, Mar. 2016, pp. 1–12.

[6] WikiImages, "Space junk earth orbits," Pixabay, Jan. 2012, available at: https://pixabay.com/illustrations/space-junk-space-debris-earth-orbits-11645/. [Accessed 06 July 2022].

[7] T. Donath, T. Schildknecht, P. Brousse, J. Laycock, T. Michal, P. Ameline, and L. Leushacke, "Proposal for a European space surveillance system," in *4th European Conference on Space Debris*, vol. 587, 2005, pp. 31–38.

[8] D. J. Kessler and B. G. Cour-Palais, "Collision Frequency of Artificial Satellites: The Creation of a Debris Belt," *Journal of Geophysical Research: Space Physics*, vol. 83, no. A6, pp. 2637–2646, 1978.

[9] N. Adilov, P. J. Alexander, and B. M. Cunningham, "The economics of orbital debris generation, accumulation, mitigation, and remediation," *Journal of Space Safety Engineering*, vol. 7, no. 3, pp. 447–450, 2020.

[10] B. Tarran, "Prepare for impact: Space debris and statistics," *Significance*, vol. 18, no. 3, pp. 18–23, 2021.

[11] D. Mehrholz, L. Leushacke, W. Flury, R. Jehn, H. Klinkrad, and M. Landgraf, "Detecting, tracking and imaging space debris," *ESA Bulletin(0376-4265)*, no. 109, pp. 128–134, 2002.

[12] K. Becker, E. Detsis, C. Nwosa, V. Palevska, M. Rathnasabapathy, and M. Taheran, "Space situational awareness," *Space Safety and Sustainability Project Group, Space Generation Advisory Council*, 2012.

[13] D. Agaba, "System Design of the MeerKAT L-band 3D Radar for Monitoring Near Earth Objects," Ph.D. dissertation, University of Cape Town, 2017, available at: https://open.uct.ac.za/handle/11427/26890. [Accessed 21 August 2022].

[14] H. Klinkrad, T. Donath, and T. Schildknecht, "Investigations of the Feasibility of a European Space Surveillance System," in *7th US/Russian Space Surveillance Workshop*. Naval Postgraduate School, 2007.

[15] H. Krag, H. Klinkrad, T. Flohrer, E. Fletcher, and N. Bobrinsky, "The European space surveillance system–required performance and design concepts," in *Proceedings of the 8th US/Russian Space Surveillance Workshop, Space Surveillance Detecting and Tracking Innovation, Maui, Hawaii, USA*, Jan. 2010, pp. 1–18.

[16] L. Ionescu, A. Rusu-Casandra, C. Bira, A. Tatomirescu, I. Tramandan, R. Scagnoli, D. Istriteanu, and A.-E. Popa, "Development of the Romanian Radar Sensor for Space Surveillance and Tracking Activities," *Sensors*, vol. 22, no. 9, pp. 3546–3564, 2022.

[17] J. A. Kennewell and B.-N. Vo, "An overview of space situational awareness." in *FUSION*, 2013, pp. 1029–1036.

[18] O. Kalden and C. Bodemann, "Building space situational awareness capability," in *2011 5th International Conference on Recent Advances in Space Technologies (RAST)*. IEEE, 2011, pp. 650–654.

[19] I. Ritchie, "Remote control southern hemisphere SSA observatory," in *Proceedings of the 18th international workshop on laser ranging, Fujiyoshida, Japan*, 2013, pp. 1–9.

[20] J. Ender, L. Leushacke, A. Brenner, and H. Wilden, "Radar techniques for space situational awareness," in *Radar Symposium (IRS), 2011 Proceedings International*. IEEE, 2011, pp. 21–26.

[21] D. Vigilante, F. Feudo, S. Immediata, R. Petrucci, V. Simeone, A. Lanzillotti, and F. Mosconi, "On the use of long-range radars for Space Situational Awareness: an experimental test," in *Radar Conference (RadarCon), 2015 IEEE*. IEEE, 2015, pp. 1744–1749.

[22] J. L. Jonas, "MeerKAT," in *General Assembly and Scientific Symposium (URSI GASS), 2014 XXXIth URSI*. IEEE, 2014, p. 1.

[23] SKA Africa, "SKA SA Project," Science & Technology Parliamentary Portfolio Committee, May 2017, available at: http://www.ska.ac.za/wp-content/uploads/2017/06/presentation_parliament_2017.pdf. [Accessed 10 September 2020].

[24] SKA, "MeerKAT Radio Telescope: Image and Video Gallery," 2018, available at: https://www.sarao.ac.za/gallery/. [Accessed 29 June 2022].

[25] M. Brooker, "The design and implementation of a simulator for multistatic radar systems," Ph.D. dissertation, University of Cape Town, 2008, available at: https://open.uct.ac.za/handle/11427/5253. [Accessed 21 August 2022].

[26] P. D. McCall, "Modeling, simulation, and characterization of space debris in low-earth orbit," Ph.D. dissertation, Florida International University, Nov. 2013.

[27] D. Kastinen, J. Vierinen, J. Kero, S. Hesselbach, T. Grydeland, and H. Krag, "Next-generation space object radar tracking simulator: Sorts++," in *1st NEO and Debris Detection Conference*. European Space Agency, 2019, pp. 1–8.

[28] A. Zubaroglu, "GPU Accelerated Radio Wave Propagation Modeling Using Ray Tracing," Master's thesis, Middle East Technical University, 2014.

[29] D. Kirk, "NVIDIA® CUDA software and GPU parallel computing architecture," in *6th International Symposium on Memory Management*, vol. 7, 2007, pp. 103–104.

[30] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "OptiX™: A General Purpose Ray Tracing Engine," in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 66. ACM, 2010, pp. 1–13.

[31] NVIDIA® Corporation, "NVIDIA® Turing GPU Architecture," 2018, available at: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. [Accessed 31 October 2019].

[32] Z. Yun and M. F. Iskander, "Ray tracing for radio propagation modeling: Principles and applications," *IEEE Access*, vol. 3, pp. 1089–1100, 2015.

[33] K. Williams, L. Tirado, Z. Chen, B. Gonzalez-Valdes, J. A. Martinez, and C. M. Rappaport, "Ray Tracing for Simulation of Millimeter-Wave Whole Body Imaging Systems," *IEEE Trans. Antennas Propag.*, vol. 63, no. 12, pp. 5913–5918, Oct. 2015.

[34] G. Xu, C. Dong, T. Zhao, H. Yin, and X. Chen, "Acceleration of shooting and bouncing ray method based on OptiX and normal vectors correction," *Plos one*, vol. 16, no. 6, p. e0253743, 2021.

[35] D. He, B. Ai, K. Guan, L. Wang, Z. Zhong, and T. Kürner, "The design and applications of high-performance ray-tracing simulation platform for 5G and beyond wireless communications: A tutorial,"

IEEE Communications Surveys & Tutorials, vol. 21, no. 1, pp. 10–27, 2018.

[36] Y. Tao, H. Lin, and H. Bao, "GPU-based shooting and bouncing ray method for fast RCS prediction," *IEEE Trans. Antennas Propag.*, vol. 58, no. 2, pp. 494–502, Dec. 2009.

[37] 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)*. IEEE, 2012, pp. 488–492.

[38] Y. Liu, D. Shi, A. Li, and P. Zeng, "Radio Wave Propagation Prediction Based on the NVIDIA OptiX GPU Ray Tracing Engine," in *2019 IEEE 6th International Symposium on Electromagnetic Compatibility (ISEMC)*. IEEE, 2019, pp. 1–3.

[39] J. S. Lu, E. M. Vitucci, V. Degli-Esposti, F. Fuschini, M. Barbiroli, J. A. Blaha, and H. L. Bertoni, "A discrete environment-driven GPU-based ray launching algorithm," *IEEE Trans. Antennas Propag.*, vol. 67, no. 2, pp. 1180–1192, Nov. 2018.

[40] J. Tan, Z. Su, and Y. Long, "A full 3-D GPU-based beam-tracing method for complex indoor environments propagation modeling," *IEEE Trans. Antennas Propag.*, vol. 63, no. 6, pp. 2705–2718, Mar. 2015.

[41] S. Auer, R. Bamler, and P. Reinartz, "RaySAR-3D SAR simulator: Now open source," in *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. IEEE, 2016, pp. 6730–6733.

[42] C. M. Michael, T. K. Yeoman, D. M. Wright, S. E. Milan, and M. K. James, "A Ray Tracing Simulation of HF Ionospheric Radar Performance at African Equatorial Latitudes," *Radio Science*, vol. 55, no. 2, p. e2019RS006936, 2020.

[43] S. Wald, T. Dallmann, and F. Weinmann, "Ray-Tracing-Based Micro-Doppler Simulation for 77 GHz Automotive Scenarios," in *2021 18th European Radar Conference (EuRAD)*. IEEE, 2022, pp. 281–284.

[44] Centre for High Performance Computing, "About us," available at: https://www.chpc.ac.za/index.php/about-us. [Accessed 11 February 2019].

[45] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and applications*, vol. 4, no. 10, pp. 15–24, 1984.

[46] S. Panghal, D. A. Bilung, N. Gupta, and G. Kumar, "Enhancing Graphic Performance Curve using Ray Tracing," in *2020 12th International Conference on Computational Intelligence and Communication Networks (CICN)*. IEEE, 2020, pp. 55–59.

[47] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA® GPUs," *rn (A+ B)*, vol. 21, no. 1, pp. 1–25, 2011.

[48] M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "The Design and Development of a GPU-Accelerated Radar Simulator for Space Debris Monitoring," in *2021 5th High Performance Computing and Cluster Technologies Conference (HPCCT)*. ACM, 2021, pp. 27–40.

[49] M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "The Design and Implementation of a Ray-tracing Algorithm for Signal-level Pulsed Radar Simulation Using the NVIDIA® OptiX™ Engine," *Journal of Communications*, vol. 17, no. 9, pp. 761–768, 2022.

[50] M. Y. Martin, S. L. Winberg, M. Y. Abdul Gaffar, and D. Macleod, "Investigating the Use of Ray Tracing for Signal-level Radar Simulation in Space Surveillance," in *2022 5th International Conference on Information Communication and Signal Processing (ICICSP)*. IEEE, 2022, pp. 684–690.

[51] H. Gavaghan, *Something new under the Sun: Satellites and the beginning of the space age*. Springer Science & Business Media, 1997.

[52] F. Alby, E. Lansard, and T. Michal, "Collision of Cerise with space debris," in *Second European Conference on Space Debris*, vol. 393, 1997, pp. 589–596.

[53] S. Kan, "China's anti-satellite weapon test," in *The Library of Congress (April 2007)*, 2007.

[54] T. Kelso, "Analysis of the 2007 Chinese ASAT Test and the Impact of its Debris on the Space Environment," in *8th Advanced Maui Optical and Space Surveillance Technologies Conference, Maui, HI*, vol. 7, 2007.

[55] J. A. Hokett, "Flying blind at over 7 kilometers per second: A concept for improving space collision avoidance," National Defense University, Norfolk, Virginia, Tech. Rep., 2010.

[56] C. Pardini and L. Anselmo, "Evolution of the debris cloud generated by the Fengyun-1C fragmentation event," *NASA Technical Reports Server*, 2007.

[57] M. Matney, T. Settecerri, N. Johnson, and E. Stansbery, "Characterization of the breakup of the Pegasus rocket body 1994-029b," in *Second European Conference on Space Debris*, vol. 393, 1997, pp. 289–292.

[58] J. S. Imburgia, "Space debris and its threat to national security: a proposal for a binding international agreement to clean up the junk," *Vand. J. Transnat'l L.*, vol. 44, pp. 589–642, 2011.

[59] P. D. Anz-Meador, "Orbital debris quarterly news," *Orbital Debris Program Office*, 2019.

[60] M. Safi and H. Devlin, "'A terrible thing': India's destruction of satellite threatens ISS, says NASA," The Guardian, 2019, available at: https://amp.theguardian.com/science/2019/apr/02/a-terrible-thing-nasa-condemns-indias-destruction-of-satellite-and-resulting-space-junk. [Accessed 03 April 2019].

[61] J. Amos, "Russian anti-satellite missile test draws condemnation," BBC, Nov. 2021, available at: https://www.bbc.com/news/science-environment-59299101. [Accessed 19 November 2021].

[62] H. Klinkrad, *Space debris: Models and Risk Analysis*. Springer Science & Business Media, 2006.

[63] Union of Concerned Scientists, "Satellite database," available at: http://www.ucsusa.org/nuclear-weapons/space-weapons/satellite-database. [Accessed 10 January 2019].

[64] P. D. Anz-Meador, J. N. Opiela, D. Shoots, and J.-C. Liou, "History of on-orbit satellite fragmentations," *Orbital Debris Program Office*, Jul. 2018.

[65] T. Schildknecht, "Optical surveys for space debris," *The Astronomy and Astrophysics Review*, vol. 14, no. 1, pp. 41–111, 2007.

[66] H. Klinkrad, P. Beltrami, S. Hauptmann, C. Martin, H. Sdunnus, H. Stokes, R. Walker, and J. Wilkinson, "The ESA space debris mitigation handbook 2002," *Advances in Space Research*, vol. 34, no. 5, pp. 1251–1259, 2004.

[67] R. Walker, H. Klinkrad, H. Sdunnus, and H. Stokes, "Update of the ESA space debris mitigation handbook," in *Space Debris*, vol. 473, 2001, pp. 821–826.

[68] T. G. Nelson, "Regulating the void: In-orbit collisions and space debris," *J. Space L.*, vol. 40, pp. 105–130, 2015.

[69] National Aeronautics and Space Administration, "Handbook for limiting orbital debris," *NASA Handbook*, 2008.

[70] K. Tsiolkovsky, "Extension of man into outer space," in *Proceedings of the Symposium on Jet Propulsion*, vol. 2, 1921.

[71] B. Wie, "Solar sail attitude control and dynamics, part 1," *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 4, pp. 526–535, 2004.

[72] C. R. McInnes, *Solar sailing: technology, dynamics and mission applications*. Springer Science & Business Media, 2013.

[73] M. A. Afful, "Orbital lifetime predictions of low earth orbit satellites and the effect of a deorbitsail," Ph.D. dissertation, Stellenbosch University, 2013.

[74] P. Martinez, "Space Debris – An overview," Feb. 2012, presentation to SA Council for Space Affairs available at: https://www.mofa.go.jp/mofaj/gaiko/space/pdfs/kankyou_gaiyou_201212_04.pdf. [Accessed 21 August 2022].

[75] National Aeronautics and Space Administration, "Frequently asked questions," NASA Orbital Debris Program Office, 2019, available at: https://www.orbitaldebris.jsc.nasa.gov/faq.html. [Accessed 20 September 2020].

[76] C. R. Englert, J. T. Bays, K. D. Marr, C. M. Brown, A. C. Nicholas, and T. T. Finne, "Optical orbital debris spotter," *Acta Astronautica*, vol. 104, no. 1, pp. 99–105, 2014.

[77] European Space Agency, "Space debris by the numbers," available at: https://www.esa.int/Our_Activities/Space_Safety/Space_Debris/Space_debris_by_the_numbers. [Accessed 29 July 2019].

[78] P. H. Krisko, "NASA's new orbital debris engineering model, ORDEM2010," *Making Safety Matter of ESA Special Publication*, vol. 680, pp. 1–19, 2010.

[79] National Aeronautics and Space Administration, "Orbital debris quarterly news," NASA Orbital Debris Program Office, May 2019, available at: https://orbitaldebris.jsc.nasa.gov/quarterly-news/pdfs/odqnv23i1.pdf. [Accessed 23 May 2019].

[80] T. S. Kelso *et al.*, "Analysis of the Iridium 33 Cosmos 2251 collision," *Proceedings of the Advanced Maui Optical and Space Surveillance technologies Conference Technical Conference*, Sep. 2009.

[81] D. Cerutti-Maori, J. Rosebrock, I. Maouloud, L. Leushacke, and H. Krag, "Preliminary Concept of a Space-based Radar for Detecting mm-size Space Debris," in *Proc. 7th European Conf. on Space Debris, Darmstadt, Germany*, 2017.

[82] H. Klinkrad, "DISCOS-ESA's database and information system characterising objects in space," *Advances in Space Research*, vol. 11, no. 12, pp. 43–52, 1991.

[83] J. Tombasco, "Orbit estimation of geosynchronous objects via ground-based and space-based optical tracking," Ph.D. dissertation, University of Colorado, Jan. 2011.

[84] J. Picone, J. Emmert, and J. Lean, "Thermospheric densities derived from spacecraft orbits: Accurate processing of two-line element sets," *Journal of Geophysical Research: Space Physics*, vol. 110, no. A3, 2005.

[85] Celestrak, "NORAD Two-Line Element Sets Current Data," The Center for Space Standards & Innovation, available at: https://celestrak.com/NORAD/elements/. [Accessed 17 March 2019].

[86] D. A. Vallado and W. McClain, *Fundamentals of astrodynamics and applications*. Space Technology Series, McGraw-Hill, 1997.

[87] F. R. Hoots, R. L. Roehrich, and T. Kelso, "Spacetrack report no. 3," *Project Spacetrack Reports, Office of Astrodynamics, Aerospace Defense Center, ADC/DO6, Peterson AFB, CO*, vol. 80914, pp. 1–91, 1980.

[88] D. Vallado and P. Crawford, "SGP4 orbit determination," in *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, 2008, pp. 6770–6798.

[89] B. L. Decker, "World geodetic system 1984," Defense Mapping Agency Aerospace Center St Louis Afs Mo, Tech. Rep., 1986.

[90] C. F. Wildt, "Accuracy in orbital propagation: A comparison of predictive software models," Naval Postgraduate School Monterey United States, Tech. Rep., 2017.

[91] E.-I. Croitoru and G. Oancea, "Satellite tracking using NORAD two-line element set format," *Scientific Research and Education in the Air Force-AFASES*, vol. 1, pp. 423–431, 2016.

[92] M. Haneveer, "Orbital lifetime predictions: An assessment of model-based ballistic coefficient estimations and adjustment for temporal drag coefficient variations," Master's thesis, Delft University of Technology, 2017.

[93] B. Rhodes, "sgp4 1.4," Python Software Foundation, Jan. 2015, available at: https://pypi.org/project/sgp4/. [Accessed 20 May 2019].

[94] ISO International Standard, "ISO/WD 8601-1," ISO Working Draft, Feb. 2016, available at: http://www.loc.gov/standards/datetime/iso-tc154-wg5_n0038_iso_wd_8601-1_2016-02-16.pdf. [Accessed 23 May 2019].

[95] M. Landgraf, R. Jehn, and W. Flury, "Comparison of EISCAT radar data on space debris with model predictions by the master model of ESA," *Advances in Space Research*, vol. 34, no. 5, pp. 872–877, 2004.

[96] S. P. Shuster, "A Survey and Performance Analysis of Orbit Propagators for LEO, GEO, and Highly Elliptical Orbits," Master's thesis, Utah State University, May 2017.

[97] F. R. Hoots, P. W. Schumacher Jr, and R. A. Glover, "History of analytical orbit modeling in the US space surveillance system," *Journal of Guidance, Control, and Dynamics*, vol. 27, no. 2, pp. 174–185, 2004.

[98] R. H. Battin, *An Introduction to the Mathematics and Methods of Astrodynamics, revised edition*. American Institute of Aeronautics and Astronautics, 1999.

[99] D. A. Vallado, P. Crawford, R. Hujsak, and T. Kelso, "Revisiting Spacetrack report# 3: Rev 2," in *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, 2006, pp. 6753–6846.

[100] P. W. Schumacher Jr and R. A. Glover, "Analytic orbit model for US naval space surveillance: An overview," in *Proceedings of US-Russian Second Space Surveillance Workshop, 4–6 July 1996*, 1996, pp. 66–105.

[101] J. Kennewell and R. Panwar, "Satellite orbital decay calculations," *Australian Space Weather Agency*, 1999.

[102] K. Alfriend, S. R. Vadali, P. Gurfil, J. How, and L. Breger, *Spacecraft formation flying: Dynamics, control and navigation*. Elsevier, 2009, vol. 2.

[103] K. Wedeward and A. El-Osery, "EE 570: Location and Navigation, Navigation Mathematics: Coordinate Frames," Electrical Engineering Department, New Mexico Tech, Jan. 2016.

[104] M. Breivik, "Nonlinear maneuvering control of underactuated ships," *MS thesis*, 2003.

[105] G. Shepherd, "Space surveillance network," 2018, available at: http://climateviewer.org/img/gallery/ssa2shepherd.pdf. [Accessed 21 September 2020].

[106] M. Richards, J. Scheer, and W. Holm, *Principles of Modern Radar – Volume 1: Basic Principles*. SciTech Publishing, Incorporated, 2010.

[107] L. Sevgi and S. Sanal, "Surface wave HF radar simulator," *IEEE Radar 1997*, 1997.

[108] J. Meyer-Hilberg, "Pirdis: A new versatile tool for SAR/MTI systems simulation," in *Sixth European Conference on Synthetic Aperture Radar (EUSAR2006)*, 2006.

[109] K. Kulpa, P. Samczynski, M. Malanowski, W. Gwarek, B. Salski, and G. Tański, "SAR Raw Radar Simulator combining optical geometry and full-wave electromagnetic approaches," in *EUSAR 2012; 9th European Conference on Synthetic Aperture Radar*. VDE, 2012, pp. 24–27.

[110] S. Hussain, "Efficient ray-tracing algorithms for radio wave propagation in urban environments," Ph.D. dissertation, Dublin City University, 2017.

[111] R. R. Boothe, "A digital computer program for determining the performance of an acquisition radar through application of radar detection probability theory," U.S. Army Missile Command, Report No. RD-TR-64-2., Tech. Rep., 1964.

[112] P. J. Golda, "Software simulation of synthetic aperture radar," Master's thesis, University of Cape Town, 1997, available at: https://open.uct.ac.za/handle/11427/26092. [Accessed 21 August 2022].

[113] R. Lengenfelder, "The design and implementation of a radar simulator," Master's thesis, University of Cape Town, 1998, available at: https://open.uct.ac.za/handle/11427/17858. [Accessed 21 August 2022].

[114] M. Brooker and M. Inggs, "A Signal Level Simulator for Multistatic and Netted Radar Systems," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 47, no. 1, pp. 178–186, Jan. 2011.

[115] G. Franceschetti, M. Migliaccio, D. Riccio, and G. Schirinzi, "SARAS: a synthetic aperture radar (SAR) raw signal simulator," *IEEE Trans. Geosci. Remote Sens.*, vol. 30, no. 1, pp. 110–123, Jan. 1992.

[116] J.-F. Nouvel, A. Herique, W. Kofman, and A. Safaeinili, "Marsis radar signal simulation," in *IGARSS 2003. 2003 IEEE International Geoscience and Remote Sensing Symposium. Proceedings (IEEE Cat. No. 03CH37477)*, vol. 4. IEEE, 2003, pp. 2756–2758.

[117] D. Cutajar, A. Magro, J. Borg, K. Adami, G. Bianchi, G. Pupillo, A. Mattana, G. Naldi, C. Bortolotti, F. Perini *et al.*, "PyBIRALES: A radar data processing backend for the real-time detection of space debris," *JAI*, vol. 9, pp. 1–14, 2020.

[118] K. Milne, "Principles and concepts of multistatic surveillance radars," *RPI*, pp. 46–52, 1977.

[119] V. S. Chernyak, *Fundamentals of multisite radar systems: multistatic radars and multistatic radar systems.* CRC press, 1998.

[120] T. Johnsen and K. E. Olsen, "Bi-and multistatic radar," Norwegian Defence Research Establishment Kjeller, Tech. Rep., 2006.

[121] N. J. Willis, *Bistatic radar.* SciTech Publishing, 2005, vol. 2.

[122] C. A. Balanis, *Advanced Engineering Electromagnetics.* John Wiley & Sons, 2012.

[123] A. S. Glassner, *An introduction to ray tracing.* Elsevier, 1989.

[124] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation.* Morgan Kaufmann, 2016.

[125] M. Grgic, "Generic radar model for automotive applications," Master's thesis, Graz University of Technology, Sep. 2015.

[126] R. W. Scharstein, "Visualization and interpretation for the electromagnetic derivation of Snell's laws," *IEEE Trans. Edu.*, vol. 41, no. 4, pp. 286–292, Nov. 1998.

[127] Massachusetts Institute of Technology, "Refraction and Snell's Law," Electromagnetic Energy: From Motors to Lasers, 2011.

[128] J. I. Peltoniemi, O. Wilkman, M. Gritsevich, M. Poutanen, A. Raja-Halli, J. Näränen, T. Flohrer, and A. Di Mira, "Steering Reflective Space Debris Using Polarised Lasers," *Advances in Space Research*, vol. 67, no. 6, pp. 1721–1732, 2021.

[129] H. Ling, R.-C. Chou, and S.-W. Lee, "Shooting and bouncing rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Trans. Antennas Propag.*, vol. 37, no. 2, pp. 194–205, Feb. 1989.

[130] S.-K. Wong, Y.-C. Cheng, and S.-Y. Lii, "GPU ray tracing based on reduced bounding volume hierarchies," in *2012 Ninth International Conference on Computer Graphics, Imaging and Visualization*. IEEE, 2012, pp. 1–6.

[131] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 1–18, 2007.

[132] K. Williams, "Ray tracing simulation tool for portal-based millimeter-wave security systems using the NVIDIA® OptiX™ ray tracing engine," Master's thesis, Northeastern University, 2013.

[133] T. Frach, W. Fischer, and B. Rembold, "Fast parallel ray-tracing simulator," in *Millenium Conference on Antennas & Propagation (AP2000), Davos*, 2000.

[134] D. G. Muff, "Electromagnetic Ray-tracing for the Investigation of Multipath and Vibration Signatures in Radar Imagery," Ph.D. dissertation, UCL (University College London), 2018.

[135] T. Dallmann, J.-K. Mende, and S. Wald, "ATRIUM: A Radar Target Simulator for Complex Traffic Scenarios," in *2018 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*. IEEE, 2018, pp. 1–4.

[136] M. Potgieter, J. E. Cilliers, and C. Blaauw, "The Use of SigmaHat for Modelling of Electrically Large Practical Radar Problems," in *2020 IEEE International Radar Conference (RADAR)*. IEEE, 2020, pp. 186–191.

[137] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of the conference on high performance graphics 2009*, 2009, pp. 145–149.

[138] D. McAllister, "GPU Ray Tracing Using OptiX," in *2013 GPU Technology Conference, NVIDIA® Corporation*, 2013.

[139] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, 2009, pp. 7–13.

[140] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," in *Computer graphics forum*, vol. 28. Wiley Online Library, 2009, pp. 1691–1722.

[141] G. S. Almasi and A. Gottlieb, "Highly parallel computing," *U.S. Department of Energy, Office of Scientific and Technical Information*, 1988.

[142] M. J. Quinn, "Parallel Programming in C with MPI and OpenMP," McGraw Hill Higher Education, New York, NY, Tech. Rep., 2004.

[143] M. R. Cawood, "HI Lightcones for LADUMA using Gadget-3: performance profiling and application of an HPC code," Master's thesis, University of Cape Town, 2014, available at: https://open.uct.ac.za/handle/11427/13107. [Accessed 21 August 2022].

[144] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[145] M. Fatica, "CUDA toolkit and libraries," in *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE, 2008, pp. 1–22.

[146] N. Sunitha, K. Raju, and N. N. Chiplunkar, "Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead," in *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE, 2017, pp. 211–215.

[147] MathWorks®, "MATLAB," available at: https://www.mathworks.com/products/matlab.html. [Accessed 15 May 2020].

[148] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[149] International Organization for Standardization, "ISO/IEC 14882:2017 Programming languages — C++," International Organization for Standardization, Dec. 2017.

[150] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[151] A. Holzinger, "Usability engineering methods for software developers," *Communications of the ACM*, vol. 48, no. 1, pp. 71–74, 2005.

[152] D. M. Nichols and M. B. Twidale, "Usability and open source software." 2002.

[153] P. S. Foundation, "Python™," May 2019, available at: https://www.python.org/. [Accessed 10 September 2020].

[154] MathWorks®, "Phased array system toolbox," available at: https://www.mathworks.com/products/phased-array.html. [Accessed 15 May 2020].

[155] MathWorks®, "MATLAB App Designer," available at: https://www.mathworks.com/products/matlab/app-designer.html. [Accessed 15 May 2020].

[156] TIOBE, "TIOBE Index for May 2020," available at: https://www.tiobe.com/tiobe-index/. [Accessed 15 May 2020].

[157] E. Bressert, *SciPy and NumPy: an overview for developers.* " O'Reilly Media, Inc.", 2012.

[158] C. Zeller, "CUDA C/C++ Basics," *NVIDIA® Corporation*, 2011.

[159] N. M. Josuttis, *The C++ standard library: a tutorial and reference.* Addison-Wesley, 2012.

[160] S. Meyers, *Effective STL: 50 specific ways to improve your use of the standard template library.* Pearson Education, 2001.

[161] M. Brooker, "FERS – The Flexible Extensible Radar Simulator," GitHub, available at: https://github.com/stpaine/FERS. [Accessed 20 October 2021].

[162] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases.* ACM, 2011, pp. 36–47.

[163] S. Koranne, "Hierarchical data format 5: HDF5," in *Handbook of Open Source Tools.* Springer, 2011, pp. 191–200.

[164] B. Schäling, *The Boost C++ libraries.* Boris Schäling, 2011.

[165] L. Thomason, "TinyXML," Mar. 2015, available at: http://www.grinninglizard.com/tinyxml/. [Accessed 15 May 2020].

[166] A. Kapoulkine, "A light-weight C++ XML processing library," *URL: http://pugixml. org*, Sep. 2019.

[167] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[168] K. Martin and B. Hoffman, *Mastering CMake: a cross-platform build system.* Kitware, 2010.

[169] V. Grover and Y. Lin, "Compiling CUDA and other languages for GPUs," in *GPU Technology Conference (GTC)*, 2012, pp. 1–59.

[170] M. Lane and F. Hoots, "Spacetrack Report# 2: General Perturbations Theories Derived from the 1965 Lane Drag Theory," *Aerospace Defense Command, Peterson AFB, CO*, 1979.

[171] D. Vallado, "Astrodynamics software," Apr. 2018, available at: https://celestrak.com/software/vallado-sw.php. [Accessed 21 May 2020].

[172] G. Smoot, "Cosmic Microwave Background Radiation anisotropies: their discovery and utilization," *Bulletin of the American Physical Society*, vol. 52, 2007.

[173] ITU, "Recommendation ITU-R P. 372-13. Radio Noise," *ITU-R P Series, Radiowave Propagation*, 2016.

[174] A. J. Wilkinson, "Line-of-sight radio communications link," EEE3086F: Signals and Systems II Course Notes, University of Cape Town, May 2017.

[175] M. I. Skolnik, *Radar Handbook, Second Edition.* McGraw-Hill, 1970.

[176] J. D. Wilson, "Probability of detecting aircraft targets," *IEEE Trans. Aerosp. Electron. Syst.*, no. 6, pp. 757–761, Nov. 1972.

[177] J. Murray and T. Kennedy, "Haystack Ultra-Wideband Satellite Imaging Radar Measurements of the Orbital Debris Environment: 2019," *Orbital Debris Program Office Technical Publication*, 2021.

[178] P. Swerling, "Detection of fluctuating pulsed signals in the presence of noise," *IRE Transactions on Information Theory*, vol. 3, no. 3, pp. 175–178, 1957.

[179] P. Swerling, "Studies of target detection by pulsed radar," *Transactions on Information Theory, IRE (later IEEE)*, 1960.

[180] P. Swerling, "Probability of detection for fluctuating targets," *IRE Transactions on Information theory*, vol. 6, no. 2, pp. 269–308, 1960.

[181] P. Swerling, "Radar probability of detection for some additional fluctuating target cases," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 33, no. 2, pp. 698–709, Apr. 1997.

[182] J. K. Jao and M. Elbaum, "First-order statistics of a non-Rayleigh fading signal and its detection," *Proceedings of the IEEE*, vol. 66, no. 7, pp. 781–789, 1978.

[183] D. Shnidman, "Expanded Swerling target models," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 39, no. 3, pp. 1059–1069, Jul. 2003.

[184] A. Kapoulkine, "PugiXML Benchmarks," 2020, available at: https://pugixml.org/benchmark.html. [Accessed 27 June 2021].

[185] L. Thomason, "TinyXML-2," Feb. 2020, available at: http://www.grinninglizard.com/tinyxml2/. [Accessed 18 May 2020].

[186] M. Kalicinski, "RapidXml," Apr. 2013, available at: http://rapidxml.sourceforge.net/. [Accessed 18 May 2020].

[187] The HDF Group, "HDF5 FAQ – Questions About the Software," Jul. 2017, available at: https://support.hdfgroup.org/HDF5/hdf5-quest.html#mthread. [Accessed 18 May 2020].

[188] The HDF Group, "Thread-safe HDF5," Sep. 2019, available at: https://portal.hdfgroup.org/display/HDF5/Thread-Safe+HDF5. [Accessed 18 May 2020].

[189] A. Munshi, "The OpenCL specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009, pp. 1–314.

[190] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[191] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," *arXiv preprint arXiv:1005.2581*, 2010.

[192] A. Griffith, *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[193] P. Markstein, "The new IEEE-754 standard for floating point arithmetic," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008, pp. 1–3.

[194] R. P. Torres, S. Loredo, L. Valle, and M. Domingo, "An Accurate and Efficient Method Based on Ray-tracing for the Prediction of Local Flat-fading Statistics in Picocell Radio Channels," *IEEE Journal on selected areas in communications*, vol. 19, no. 2, pp. 170–178, 2001.

[195] G. E. Corazza, V. Degli-Esposti, M. Frullone, and G. Riva, "A Characterization of Indoor Space and Frequency Diversity by Ray-tracing Modeling," *IEEE Journal on selected areas in communications*, vol. 14, no. 3, pp. 411–419, 1996.

[196] R. F. Harrington, *Field Computation by Moment Methods*. Wiley-IEEE Press, 1993.

[197] NVIDIA® Corporation, "Nvidia optix™ ray tracing engine," May 2020, available at: https://developer.nvidia.com/designworks/optix/download. [Accessed 08 June 2020].

[198] A. Robison, P. Miller, and S. Parker, "GPU Ray Tracing Exposed: Under the Hood of the NVIDIA OptiX Ray Tracing Engine," in *2010 GPU Technology Conference, NVIDIA® Corporation*, 2010.

[199] J. Nahmias, "Using NVIDIA OptiX for lighting preview in a Katana-based production pipeline," in *Proc. SIGGRAPH*, 2013.

[200] C. J. Taylor and D. J. Kriegman, "Minimization on the Lie group SO (3) and related manifolds," 1994.

[201] P. Shirley and R. Morley, "Realistic ray tracing," 2003.

[202] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[203] H. A. Ardakani and T. Bridges, "Review of the 3-2-1 Euler angles: A yaw-pitch-roll sequence," *Department of Mathematics, University of Surrey, Guildford GU2 7XH UK, Tech. Rep*, 2010.

[204] A. R. Willis, M. S. Hossain, and J. Godwin, "Hardware-accelerated SAR simulation with NVIDIA-RTX technology," in *Algorithms for Synthetic Aperture Radar Imagery XXVII*, vol. 11393. International Society for Optics and Photonics, 2020, pp. 1–17.

[205] NVIDIA® Corporation, "NVIDIA OptiX 5.1 Programming Guide," 2018, available at: https://raytracing-docs.nvidia.com/optix/guide/nvidia_optix_programming_guide.180409.LTR.pdf. [Accessed 07 October 2020].

[206] F. M. Carano and J. J. Prichard, "Data Abstraction and problem solving with C++," *Walls and Mirrors*, vol. 5, 2007.

[207] D. McAllister, "Advanced OptiX Programming and Optimization," in *2013 GPU Technology Conference, NVIDIA Corporation*, 2013.

[208] D. McAllister and J. Bigler, "OptiX Out-of-Core and CPU Rendering," in *2012 GPU Technology Conference, NVIDIA Corporation*, 2012.

[209] A. Battaglia and S. Tanelli, "DOMUS: DOppler MUltiple-Scattering Simulator," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 1, pp. 442–450, Jul. 2010.

[210] NVIDIA® Corporation, "NVIDIA® Tesla® V100 GPU Accelerator," 2018, available at: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf. [Accessed 05 August 2020].

[211] O. Pelz and J. Hobson, *CentOS 7 Linux Server Cookbook*. Packt Publishing Ltd, 2016.

[212] R. M. Gagliardi, *Satellite communications*. Springer Science & Business Media, 2012.

[213] N. Tucker, "Phased Array Design Toolbox V2.3 for MATLAB," *Technical Report*, 2009.

[214] MathWorks®, "Phased Array System Toolbox," available at: https://www.mathworks.com/products/phased-array.html. [Accessed 02 September 2021].

[215] G. Thiagarajan, "Hybrid Beamforming for Massive MIMO Phased Array Systems Whitepaper," November 2020, available at: https://www.mathworks.com/content/dam/mathworks/white-paper/gated/93096v00_Beamforming_Whitepaper.pdf. [Accessed 02 September 2021].

[216] M. Roggero, J. Zhao, and G. Zucchelli, "Design of FMCW Radars for Active Safety Applications," in *Embedded World Conference*, 2015.

[217] A. A. Qasim and A. H. Sallomi, "Design and Analysis of Phased Array System by MATLAB Toolbox," *Al-Kitab Journal for Pure Sciences*, vol. 4, no. 1, 2020.

[218] D. Haupt, "Controllable Drone – Blender Game Engine 3D Model," March 2017, available at: https://free3d.com/3d-model/controllable-drone-blender-game-engine-67381.html. [Accessed 04 December 2021].

[219] National Aeronautics and Space Administration, "International Space Station 3D Model," NASA Science, Apr. 2019, available at: https://solarsystem.nasa.gov/resources/2378/international-space-station-3d-model/. [Accessed 16 May 2022].

[220] N2YO, "International Space Station, NORAD ID: 25544," N2YO, 2014, available at: https://www.n2yo.com/satellite/?s=25544. [Accessed 16 May 2022].

[221] MatWeb, "Overview of materials for Aluminum Alloy," MatWeb Material Property Data, 2016, available at:

https://www.matweb.com/search/DataSheet.aspx?MatGUID=ab8aeb2d293041c4a844e397b5cfbd4e& ckck=1. [Accessed 16 May 2022].

[222] A. M. Balakhder, "Intelligent approach to improve standard CFAR detection in non-Gaussian sea clutter," Ph.D. dissertation, The Ohio State University, 2015.

[223] P. P. Gandhi and S. A. Kassam, "Analysis of CFAR processors in nonhomogeneous background," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 24, no. 4, pp. 427–445, Jul. 1988.

[224] L. Z. Eng, *Qt5 C++ GUI Programming Cookbook*. Packt Publishing Ltd, 2016.

[225] R. Rischpater, *Application development with Qt Creator*. Packt Publishing Ltd, 2014.

[226] N. Morrison, *Tracking Filter Engineering: The Gauss-Newton and Polynomial Filters*. Institution of engineering and technology London, 2013.