GPU Implementation of SBR

by

Berk Cem Yamanlar

Engineering Project Report

Yeditepe University

Faculty of Engineering

Department of Computer Engineering

2024

GPU Implementation of SBR

APPROVED BY:

Prof.Dr. Ahmet Arif Ergin (Supervisor)	........[illegible]........
Assist.Prof.Dr. Esin Onbaşıoğlu (Supervisor)	........[illegible]........
Prof.Dr. Gürhan Küçük	........[illegible]........
Assist.Prof.Dr. İpek Baz	........[illegible]........
Assist.Prof.Dr. Onur Demir	........[illegible]........

DATE OF APPROVAL: 20/06/2024

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Prof. Dr. Ahmet Arif Ergin, for his invaluable guidance and support throughout this thesis. His willingness to answer my questions, no matter how big or small, and his unwavering patience were truly valuable.

Additionally, I extend my appreciation to Assistant Prof. Dr. Esin Onbaşıoğlu for her collaboration throughout this project. Her insightful advice and support have been instrumental in shaping the direction of this work.

Furthermore, I would also like to thank PhD(c) Merve Güvenç for her continuous support and assistance throughout the project. Her help was invaluable in navigating the development process and overcoming challenges.

ABSTRACT

GPU Implementation of SBR

In modern defense applications, accurate estimation of the Radar Cross Section (RCS) of designs plays an important role in various applications ranging from stealth characterization to target recognition in radar systems.

The Shooting and Bouncing Rays (SBR) method [1] is one technique for RCS estimation that operates in two stages. Initially, the SBR method utilizes geometrical optics (GO, ray-tracing) principles to simulate the propagation of electromagnetic-rays incident upon the object of interest. These rays are traced through the computational domain, interacting with the object's surfaces, typically defined as triangle meshes representing the geometric structure of the object. Subsequently, the method employs physical optics (PO) principles to calculate the scattered field resulting from these interactions. Integration of the equivalent currents induced on the object's surfaces by the incident rays, using Kirchhoff's diffraction formula, enables the accurate assessment of RCS values.

Traditionally, in SBR simulations, rays are traced sequentially, one after another, utilizing the processing capabilities of Central Processing Units (CPUs). However, this processing is time consuming, particularly when dealing with objects defined by many triangles and in high frequency simulations that require the consideration of millions of rays.

This thesis is based on an RCS simulation aimed at reducing computational constraints by utilizing the large number of threads available on Graphics Processing Units (GPUs), to parallelize the ray-tracing and accelerate the speed of the SBR method while maintaining the accuracy of the simulation. Furthermore, a bounding volume hierarchy (BVH) will be proposed as a future work to reduce the number of ray-triangle intersection checks for further accelerating the simulation. In addition, the simulation environment will be visually rendered and presented to the reader by using OpenGL, providing insights into the environment within the RCS simulation.

ÖZET

SBR Yönteminin GPU Uygulaması

Modern savunma uygulamalarında, tasarımların Radar Kesit Alanı'nın (RKS) doğru tahmini, görünmezlik karakterizasyonundan radar sistemlerinde hedef tanımaya kadar çeşitli uygulamalarda önemli bir rol oynamaktadır.

Shooting and Bouncing Rays (SBR) yöntemi [1], RKS tahmini için iki aşamada çalışan bir yöntemdir. İlk olarak, SBR yöntemi geometrik optik (GO, ışın izleme) prensiplerini kullanarak, ilgilenilen nesneye gelen elektromanyetik ışınların yayılımını simüle eder. Bu ışınlar, genellikle nesnenin geometrik yapısını temsil eden üçgenlerle tanımlanan yüzeylerle etkileşime girerek hesaplama alanında takip edilir. Ardından, yöntem fiziksel optik (PO) prensiplerini kullanarak bu etkileşimlerden kaynaklanan saçılmış alanı hesaplar. Gelen ışınlar tarafından nesnenin yüzeylerinde indüklenen eşdeğer akımların Kirchhoff'un kırınım formülü ile entegrasyonu, RKS değerlerinin doğru bir şekilde değerlendirilmesini sağlar.

Geleneksel olarak, SBR simülasyonlarında ışınlar, Merkezi İşlem Birimlerinin (CPU'lar) işlem yeteneklerini kullanarak birbiri ardına izlenir. Ancak, bu işlem, özellikle birçok üçgenle tanımlanan nesnelerle ve milyonlarca ışının dikkate alınmasını gerektiren yüksek frekanslı simülasyonlarda, zaman alıcıdır.

Bu tez, SBR metodunun hızlandırılmasını amaçlamakta olup, ışın-izlemeyi paralel hale getirmek ve SBR yönteminin hızını artırmak için Grafik İşlem Birimlerinde (GPU'lar) bulunan çok sayıda iş parçacığını kullanmaktadır. Ayrıca, gelecekteki çalışmalar için simülasyonu daha da hızlandırmak amacıyla ışın-üçgen kesişim kontrollerinin sayısını azaltacak bir sınır hacmi hiyerarşisi (BVH) önerilecektir. Bunun yanı sıra, simülasyon ortamı OpenGL kullanılarak görsel olarak render edilecek ve okuyucuya RKS simülasyonu ortamına dair içgörüler sunulacaktır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

σ	Radar Cross Section
$\mathbf{E^i}$	Incident Electric Field
$\mathbf{E^s}$	Backscattered Electric Field
$\mathbf{k^i}$	Incident Ray Direction
$\mathbf{k^s}$	Scattered Ray Direction
I	Amplitude of Vertical Polarization
$\bar{I}$	Amplitude of Horizontal Polarization
$\mathbf{E_1}$	First Edge of Triangle
$\mathbf{E_2}$	Second Edge of Triangle
$\mathbf{E_3}$	Third Edge of Triangle
R	Radius of Circumscribing Circle of Triangle
Δ	Distance Between Each Ray on Plane Wave
λ	Wavelength
f	Frequency
c	Speed of Light
$\mathbf{E^i}_{\boldsymbol{\theta}}$	Theta Component of Incident Electric Field
$\mathbf{E^i}_{\boldsymbol{\varphi}}$	Phi Component of Incident Electric Field
$\mathbf{R(t)}$	Ray Vector
$\mathbf{O_i}$	Incident Position of Ray
$\mathrm{T}(u, v)$	A Point on Triangle
Ai	Power of Ray
$\mathbf{E^s}_{\boldsymbol{\theta}}$	Theta Component of Scattered Electric Field
$\mathbf{E^s}_{\boldsymbol{\varphi}}$	Phi Component of Scattered Electric Field
$\hat{n}$	Surface Normal
k	Wave Number
ρ	Reflection Coefficient of Surface
Z_0	Characteristic Impedance of Medium

$\mathbf{E_{ap}}(\mathbf{r})$	Electric Field on the Exit Aperture
$\mathbf{H_{ap}}(\mathbf{r})$	Magnetic Field on the Exit Aperture
$S(\theta,\varphi)$	Shape Function
$\hat{s}$	Exit Ray Direction
$\mathrm{r_A}$	Last Bounce Point of Ray
$\mathbf{B_\theta}$	Theta Component of Magnetic Flux Density
$\mathbf{B_\varphi}$	Phi Component of Magnetic Flux Density
$\mathbf{J^s}_{\mathbf{tri}}$	Induced Current Density on Triangle
$\mathbf{H^i}$	Incident Magnetic Field
$\mathbf{H^s}$	Scattered Magnetic Field
$\mathrm{r_o}$	Radius of the Sphere of the MRMB
r_{patch}^{min}	Radius of the Smallest Enclosing Sphere

RCS	Radar Cross Section
SBR	Shooting and Bouncing Rays
GO	Geometrical Optics
PO	Physical Optics
CPU	Central Processing Unit
GPU	Graphical Processing Unit
BVH	Bounding Volume Hierarchy
RF	Radio Frequency
CELO	Complex Electrically Large Objects
FEM	Finite Element Method
FDTD	Finite-Difference Time-Domain
PTD	Physical Theory of Diffraction
GTD	Geometrical Theory of Diffraction
ALU	Arithmetic Logic Unit
SM	Streaming Multiprocessor

SIMD	Single Instruction Multiple Data
SAR	Synthetic Aperture Radar
ISAR	Inverse Synthetic Aperture Radar
MRBM	Minimum Bounding Rectangular Box
PEC	Perfect Electric Conductor
SOA	Structure of Arrays
AOS	Array of Structures

1. INTRODUCTION

Radar is a remote sensing technology employed for detecting and tracking objects, typically utilized in aviation, maritime navigation, meteorology, and military applications. Radars operate on the principle of emitting radio waves and analyzing their reflections from objects in the surrounding environment. Whereas, Radar Cross Section (RCS) refers to the measure of a target's ability to reflect those RF (Radio Frequency) waves back to the receiver antenna of the radar system. It is a crucial aspect in radar technology as it determines how easily a target can be detected by radar systems.

Hence, the ability to accurately predict the RCS of different objects is essential for designing effective stealth technologies. When designing platforms, engineers can use RCS measurements to evaluate the stealth characteristics of the design and optimize it to minimize its radar signature. Moreover, RCS simulations are essential for target recognition applications. By analyzing the radar signatures of different objects, such as aircraft, ships, and ground vehicles, RCS results can be used for identifying and classifying targets detected by real time radar systems, allowing commanders to make informed decisions based on the type and characteristics of detected targets.

There are several computational techniques to calculate the RCS of an object. One widely used technique is the SBR method, an approximate method particularly suited for complex electrically large objects (CELOs) at high frequencies. In SBR, rays representing the incident electromagnetic waves are "shot" towards the object, and their reflections or "bounces" are analyzed to compute the scattered field and the target RCS. For more comprehensive analysis, especially of smaller objects, other techniques are employed. These include full-wave solvers such as the Finite Element Method (FEM) and the Finite-Difference Time-Domain (FDTD) method. These methods discretize the target geometry and solve Maxwell's equations numerically, providing a more detailed picture of the electromagnetic field distributions and the RCS of the object. Additionally, some high-frequency techniques like Physical Theory of Diffraction (PTD) and Geometrical Theory of Diffraction (GTD) can be used in conjunction with SBR to account for edge effects and diffraction not captured by the SBR method itself. The choice of method depends on the specific needs of the analysis, considering factors like object size, complexity, and desired level of accuracy.

Several commercial and open-source software programs are widely used for RCS simulations, such as CST Studio Suite, FEKO, ANSYS HFSS, and COMSOL Multiphysics. These programs implement various computational methods, including the SBR, FEM, and FDTD methods. While these programs offer powerful tools for RCS analysis, they can be extremely slow, especially when dealing with objects defined by a large number of triangles or when performing large-scale simulations involving complex geometries or wide frequency ranges.

Hence, due to the computational complexity and time-consuming nature of RCS simulations, engineers have been developing alternative techniques to accelerate the process. In this thesis, a parallel SBR method will be presented that utilizes the large number of threads available in GPUs. This approach assigns each ray in the simulation to a dedicated GPU thread. Once a thread completes tracing its assigned ray, it calculates the ray's contribution to the scattered electric field value and is immediately assigned a new ray for processing. This method maximizes parallelism and significantly reduces overall simulation time. However, even with parallelization, tracing rays and checking intersections with the CELO (defined by numerous triangles) in each bounce can be time-consuming. To address these constrains, this thesis proposes a hybrid bounding volume hierarchy (BVH) utilizing a combination of KD-trees and octrees as a future work. These hierarchical data structures group objects within nested bounding volumes, allowing the algorithm to quickly discard irrelevant triangles during ray tracing and further increasing the speed.

1.1. Fundamentals of Radar Cross Section

The IEEE Dictionary of Electrical and Electronics Terms [2] defines RCS as a quantitative measure of how well a target reflects radar signals. It is calculated as 4π times the ratio of the scattered power per unit solid angle in a specific direction to the power density of the incident plane wave from a specified direction. Mathematically, RCS is the limiting value of this ratio as the distance between the target and the point where the scattered power is measured approaches infinity:

$$\sigma = \lim_{r\to\infty} 4\pi r^2 \left|\frac{\boldsymbol{E^s}}{\boldsymbol{E^i}}\right|^2 \tag{1.1}$$

where $\boldsymbol{E^s}$ is the scattered electric field and $\boldsymbol{E^i}$ is the field incident at the target.

RCS of a platform is influenced by several factors. First of all, the geometry of the platform plays a significant role, as the shape and size determine how radar waves interact with the object's surface. Additionally, the materials constituting the platform also have an influence on the RCS. Materials with low reflection coefficients can absorb radar energy, reducing the amount of energy reflected back to the radar receiver and thus lowering the platform's RCS. The frequency of the radar also affects RCS, as certain frequencies may be absorbed or scattered differently by the platform's surface. Finally, the look angle at which the radar observes the platform is crucial, as it affects the effective area of the platform exposed to the radar beam, consequently influencing RCS measurements.

Furthermore, the RCS can be categorized into two main types: bistatic RCS and monostatic RCS, which differ in the relative positions of the radar transmitter and receiver.

1.2. Bistatic RCS

The type of RCS displayed by a target depends on the position of the radar receiver relative to the radar transmitter. Bistatic RCS represents the measure of a target's ability to reflect radar signals when the transmitter and receiver are separated. Hence, in order to calculate bistatic RCS of a target, two separate antennas are required.

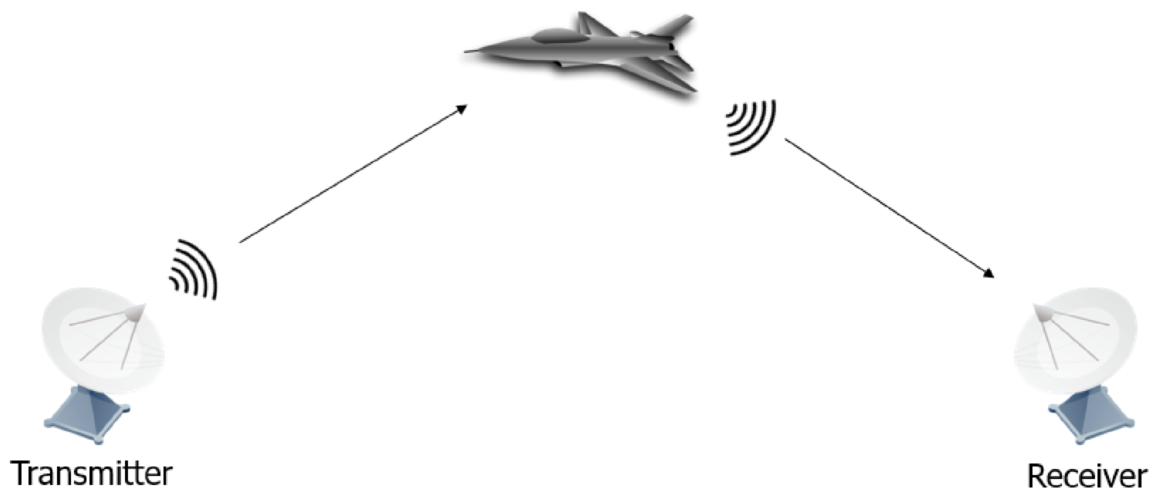


Figure 1. 1 Bistatic RCS formation

1.3. Monostatic RCS

In contrast to the bistatic case, monostatic RCS refers to the RCS measured when the transmitter and receiver antennas are integrated, meaning they are at the same location.

Monostatic measurements can provide valuable information about the target's scattering properties from a certain angle, which is commonly used for stealth characterization purposes.

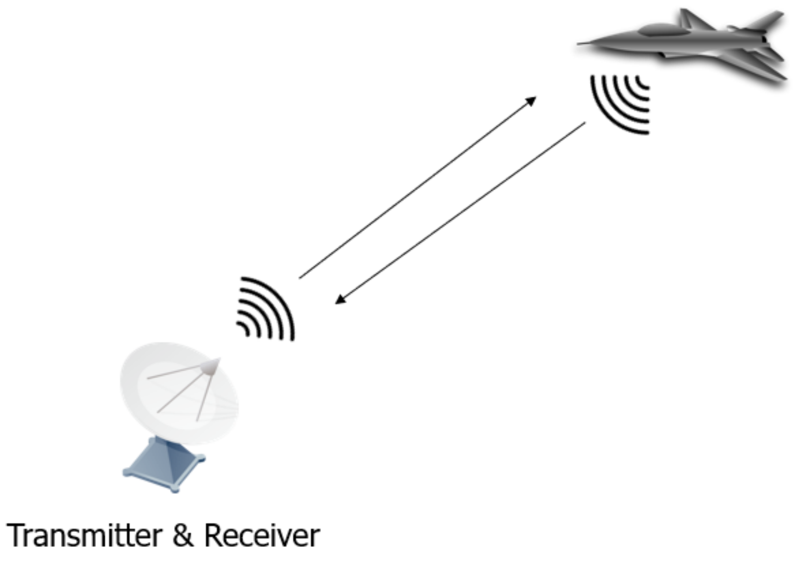


Figure 1. 2 Monostatic RCS formation

1.4. Radar Frequency Bands

Since World War II, radar systems engineers have used letter designations as a short notation for describing the frequency band of operation. The division of frequency bands was to ensure clear communication and minimize any potential conflicts or misunderstandings regarding the operational parameters of radar systems. This convention has continued over time and become a standard practice among radar engineers.

An overview of radar frequency band designations, as defined in the IEEE standards, is provided in the following table [3].

Table 1. 1 Standard radar-frequency letter band nomenclature

International Table	
HF	3 MHz to 30 MHz
VHF	30 MHz to 300 MHz
UHF	300 MHz to 1000 MHz
L	1 GHz to 2 GHz

S	2 GHz to 4 GHz
C	4 GHz to 8 GHz
X	8 GHz to 12 GHz
K_u	12 GHz to 18 GHz
K	18 GHz to 27 GHz
K_a	27 GHz to 40 GHz
V	40 GHz to 75 GHz
W	75 GHz to 110 GHz
mm	110 GHz to 300 GHz

1.5. Shooting and Bouncing Rays

The SBR method has been a fundamental technique for over three decades when predicting the high-frequency RCS of complex electrically large objects (CELOs) is required. Additionally, it has been utilized in identifying the scattering centers of CELO. For either metallic or coated CELO, a dense grid of GO rays representing an incident plane wave is "shot" into the CELO and followed as the rays bounce from the surface of the CELO and eventually escape the object or become attenuated below a specified threshold.

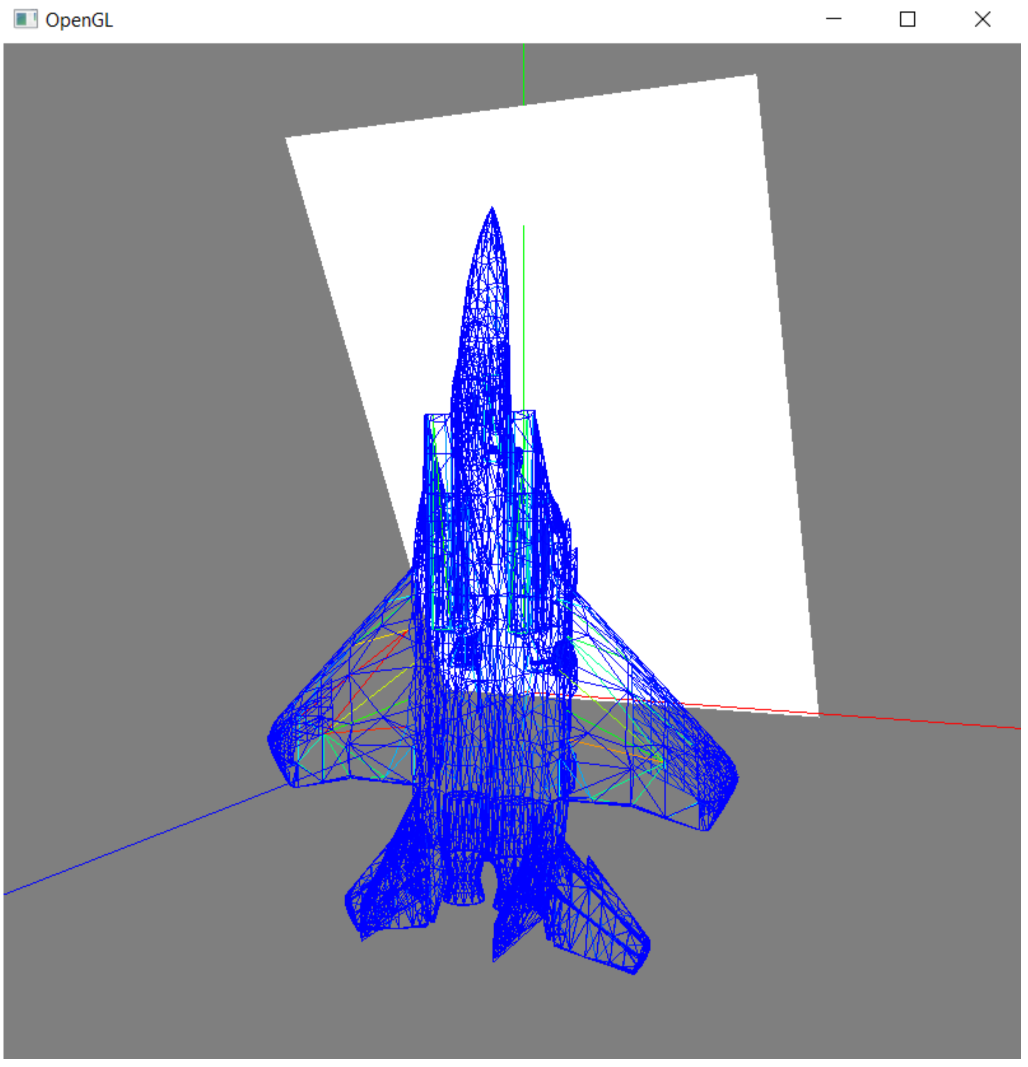


Figure 1. 3 OpenGL-based simulation environment with an F-15 model and the ray plane

Referring to Figure 1.3, consider a CELO (an F-15 fighter model) represented by a triangular mesh geometry, and the ray plane shown in white as the incident plane. Then, the incident plane wave is given by (for e^{+jwt} time convention)

$$\boldsymbol{E^i} = \left[-\hat{\varphi}^i I + \hat{\theta}^i \bar{I}\,\right] e^{j\boldsymbol{k^i}\cdot\boldsymbol{r}} \qquad (1.2)$$

where

$\boldsymbol{k^i} = k_0(\hat{x}sin\theta^i cos\phi^i + \hat{y}sin\theta^i sin\phi^i + \hat{z}cos\theta^i)$

$I\ =\ amplitude\ of\ the\ perpendicular\ (or\ vertical)\ polarization.$

$\bar{I}$ = *amplitude of the parallel (or horizontal) polarization.*

Then, the problem at hand is to determine the backscattered field $\boldsymbol{E}^s$ in the direction of $\boldsymbol{k}^i$ (monostatic RCS) or in another direction $\boldsymbol{k}^s$ (bistatic RCS) and followed by the calculating the RCS of the object.

1.6. Architectural Differences: CPU and GPU

In the world of computing, Central Processing Units (CPUs) have traditionally been at the core of processing power. These units are known for having a smaller number of powerful cores that are adapt at performing complex sequential tasks and are optimized for general-purpose computing. Meanwhile, Graphics Processing Units (GPUs) have emerged as less powerful computational processors for performing massively parallel tasks. Unlike CPUs, GPUs have thousands of smaller, more specialized cores that can work simultaneously. The following diagram, Figure 1.4 provides a side-by-side comparison of CPU and GPU architectures. On the left, the CPU's architecture with its limited number of ALUs (Arithmetic Logic Units) and larger control units, emphasizing its capability for complex sequential processing and task prioritization. On the right, the GPU’s architecture is illustrated with its extensive array of smaller ALUs designed for lightweight parallel tasks.

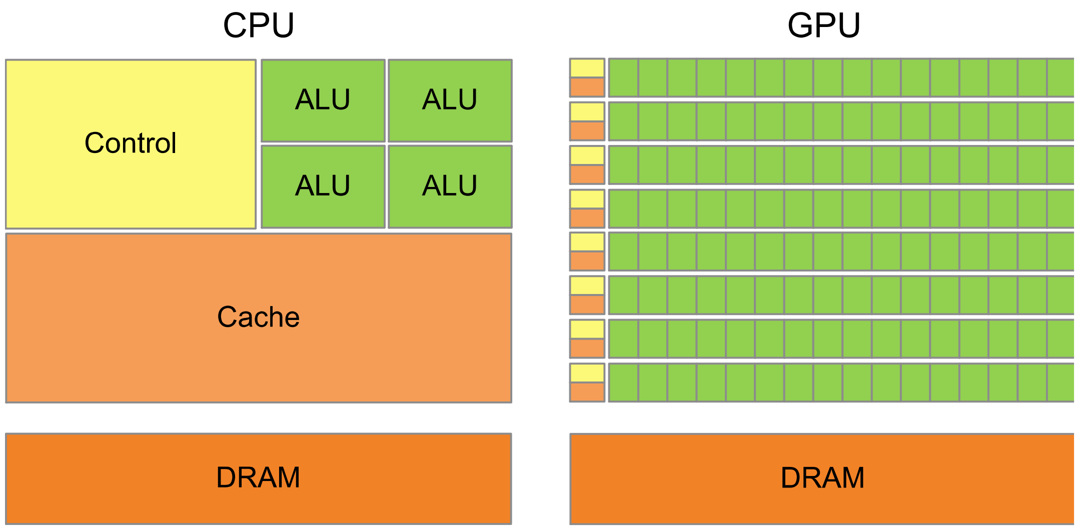


Figure 1. 4 Comparative architectures of CPU and GPU

Building upon the architectural differences, Figure 1.5 provides an abstraction of GPU memory hierarchy. Here, the GPU is called device and GPU memory likewise called device memory. GPUs are divided into core processing units called Streaming Multiprocessors (SMs) and they are responsible for parallel execution of tasks. Each SM consists of ALUs and Control Units, allowing them to perform mathematical operations in parallel. Typically, an SM contains hundreds of threads, grouped into warps. The number of threads per warp is often 32 for many GPUs. Threads within a warp execute in a synchronized fashion, meaning that each thread in the warp follows the same execution path, carrying out the same instructions at the same time on different pieces of data. This technique is known as Single Instruction Multiple Data (SIMD). Each SM is equipped with a Constant Cache, Shared Memory, and an L1 Cache. The L1 Cache and Shared Memory are fast-accessible memories and can be accessed by each thread within the same block. Constant Cache is a read-only memory that is used to quickly deliver frequently used constants to the cores. Additionally, each thread has a small private memory called Register. These small units are assigned to a thread and they are not visible to other threads. The L2 cache is shared across all SMs, so every thread in every block can access this memory. Lastly, the size of the GPU's framebuffer and the DRAM located in the GPU called Global Memory.

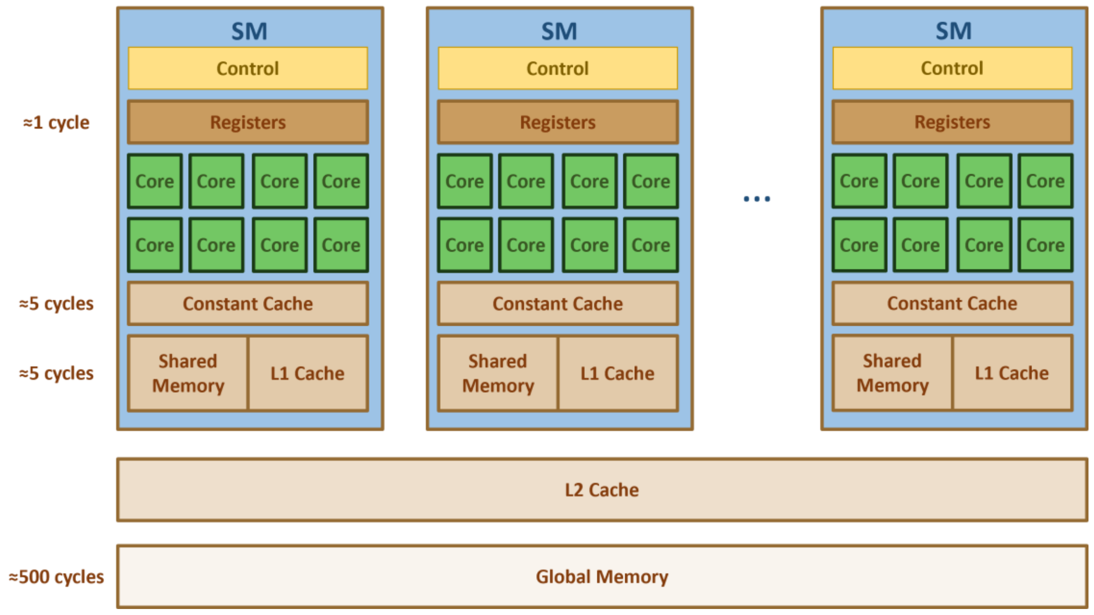


Figure 1. 5 GPU memory hierarchy

1.7. Requirements

1.7.1. Hardware and Software Requirements

To enhance performance through parallelization, a GPU unit with compatibility for CUDA or OpenCL is required. To render the simulation environment, compatibility with OpenGL is needed, with Glut facilitating the OpenGL functionality. Python, along with the Matplotlib library, must be employed for generating plots, while the Dear ImGui library is necessary to provide a user-friendly Graphical User Interface (GUI) for easy interaction.

1.7.2. Evaluation Criteria

As this project was a collaborative effort between the Electrical and Electronics Engineering and Computer Engineering departments, distinct evaluation criteria were established for each department.

In the first part of the evaluation, precision was a key metric, with the requirement that the RCS error remains within +/- 1dB. Additionally, computational efficiency was assessed, focusing on time improvements.

Building upon the foundation established in Part I, the second part of the evaluation involved further optimization of the parallelized algorithm. Additionally, the simulation environment was rendered for visualization using OpenGL. The scalability of the simulation algorithm was also evaluated.

2. BACKGROUND

The foundation of the SBR method dates back to its initial introduction in 1989 by H. Ling et al [1]. Since its first introduction, engineers have been working to improve several features of the method. For instance, improvements in the ray tracing component have drawn inspiration from advancements in game rendering. Techniques such as acceleration structures (e.g., bounding volume hierarchies, KD-trees, octrees) and parallelization methods have been applied to accelerate ray tracing processes, thereby improving the efficiency and scalability of the SBR method. In the context of this implementation, the ray tracing component draws from a serial implementation conducted by PhD(c) Merve Güvenç under the guidance of the same advisor as me. Whereas, the ray tube integration formula used for backscattered field calculation is from a study referenced as [4], while the proposed hybrid BVH data structure is in the recommendations provided by reference [5].

2.1. Previous Work

Reference [6] highlights the use of a stackless kd-tree traversal algorithm in their GPU-based SBR method. Unlike traditional kd-tree traversal algorithms that typically rely on recursive approaches involving a stack to traverse the tree structure, the stackless variant utilized in this method eliminates the need for a stack, thereby reducing memory overhead and potentially improving computational efficiency.

Reference [7] introduces the physical optics-shooting and bouncing rays method as a highly efficient approach for predicting the electromagnetic scattering of electrically large objects. Using OptiX, a GPU ray-tracing library, the authors initially optimized the PO-SBR method for single frequencies. However, for applications such as synthetic aperture radar (SAR), which necessitates analysis across numerous frequency points, further optimization was required. The paper explains the implementation details aimed at accelerating the SBR process within the PO-SBR method for multiple frequencies.

Reference [8] presents an effective implementation of the SBR method on GPUs for addressing scattering problems involving electrically large complex targets and composite models of randomly rough surfaces. The study compares results obtained through the proposed GPU-based approach with those generated by commercial software and CPU-based SBR methods.

Reference [9] proposes several implementation strategies for efficiently translating the SBR-PO method onto GPUs to expedite the solution process of scattering problems. The study employs NVIDIA's CUDA and OptiX to reduce computational time involved in modeling monostatic scattering from electrically large and arbitrarily shaped objects.

Reference [10] addresses the primary computational bottleneck of the conventional SBR method, which stems from the time-intensive ray tracing process. The study introduces an accelerated SBR method aimed at mitigating this computational burden. This acceleration strategy involves two key approaches: first, reducing the number of ray-triangle intersection tests through octree decomposition of the 3D mesh structure, and second, parallelizing the ray tracing algorithm on a GPU using MATLAB. The proposed method is applied to obtain Inverse Synthetic Aperture Radar (ISAR) images of electrically-large complex objects.

3. ANALYSIS AND DESIGN

3.1. Analysis

3.1.1. Constructing the Object

The RCS of an object we want to calculate is modeled using triangle meshes. The simulation receives the coordinates of each triangle's vertices as inputs. In a typical input file, each line represents a triangle and the (x, y, z) coordinates of the triangle's vertices are given by nine values per line. Figure 3.1 shows a trihedral object and Figure 3.2 shows the input file for this object.

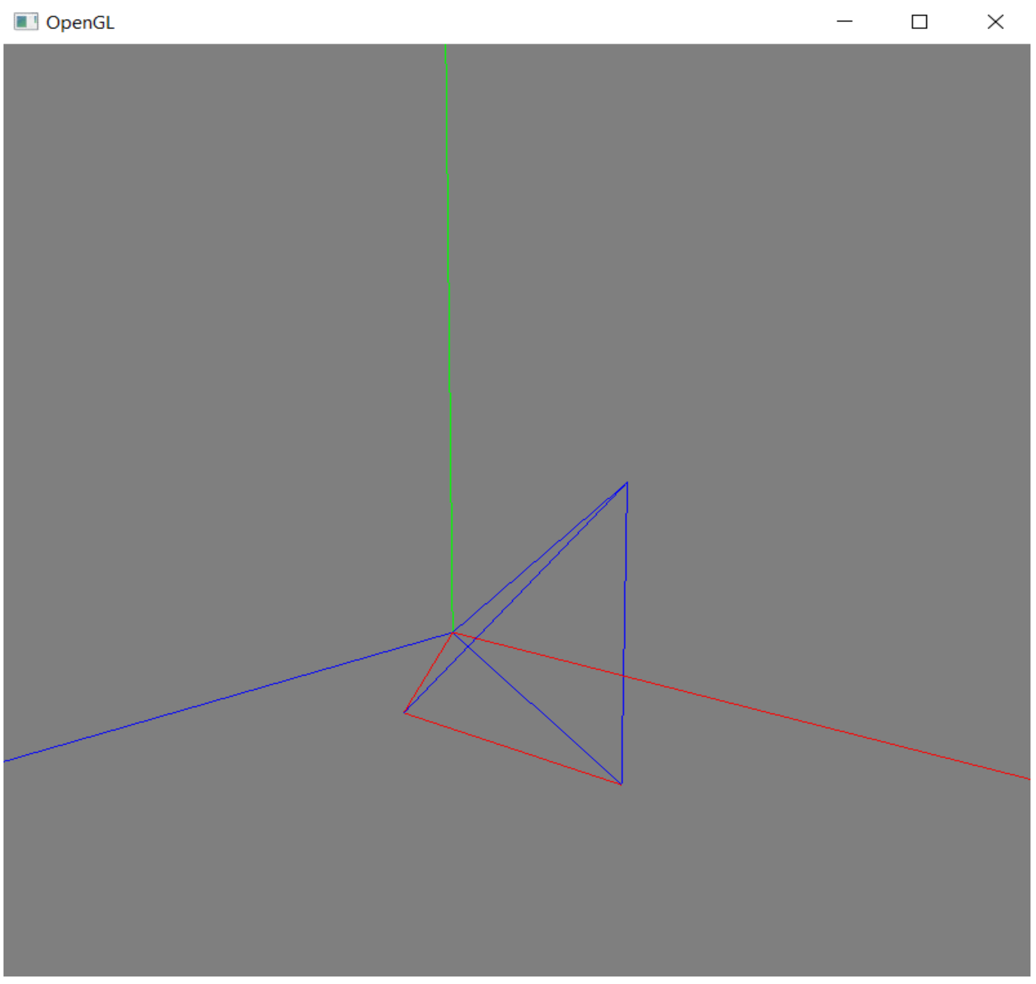
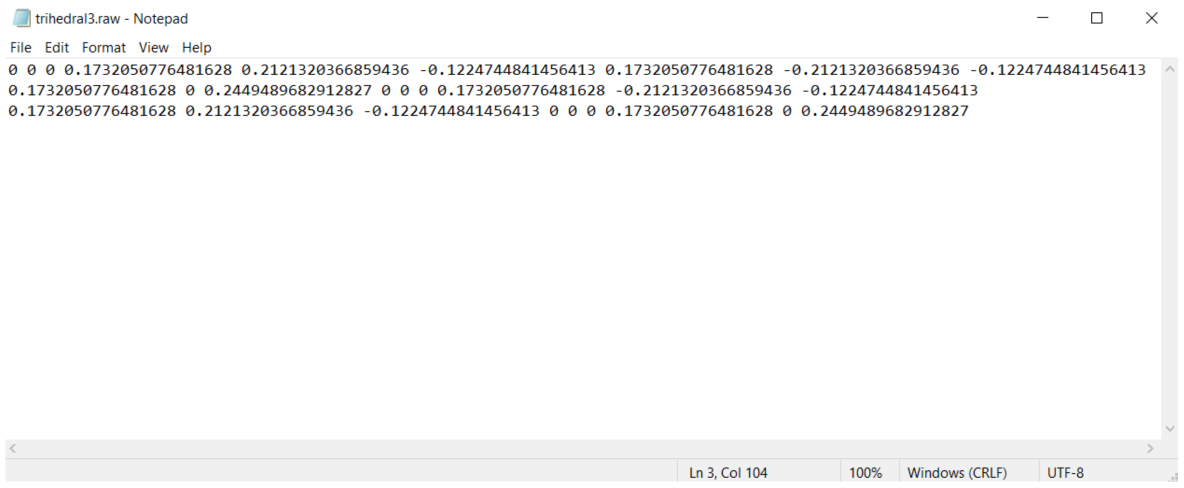


Figure 3. 1 Example input file for a trihedral object comprising three triangles



```
0 0 0 0.1732050776481628 0.2121320366859436 -0.1224744841456413 0.1732050776481628 -0.2121320366859436 -0.1224744841456413
0.1732050776481628 0 0.2449489682912827 0 0 0 0.1732050776481628 -0.2121320366859436 -0.1224744841456413
0.1732050776481628 0.2121320366859436 -0.1224744841456413 0 0 0 0.1732050776481628 0 0.2449489682912827
```

Figure 3. 2 Example input file for a trihedral object comprising three triangles

Every time a line is read, an instance of the triangle object is created and stored in an array. The edges of a triangle can be found by calculating the distance between each pair of vertices as

$$\mathbf{E_1} = \hat{\boldsymbol{x}}(v_{2,x} - v_{1,x}) + \hat{\boldsymbol{y}}(v_{2,y} - v_{1,y}) + \hat{\boldsymbol{z}}(v_{2,z} - v_{1,z}) \tag{3.1}$$

$$\mathbf{E_2} = \hat{\boldsymbol{x}}(v_{3,x} - v_{2,x}) + \hat{\boldsymbol{y}}(v_{3,y} - v_{2,y}) + \hat{\boldsymbol{z}}(v_{3,z} - v_{2,z}) \tag{3.2}$$

$$\mathbf{E_3} = \hat{\boldsymbol{x}}(v_{1,x} - v_{3,x}) + \hat{\boldsymbol{y}}(v_{1,y} - v_{3,y}) + \hat{\boldsymbol{z}}(v_{1,z} - v_{3,z}) \tag{3.3}$$

Following that, triangle area is obtained by the formula:

$$Area = \frac{1}{2}|\mathbf{E_2} \times \mathbf{E_1}| \tag{3.4}$$

Next, the normalized normal of a triangle can be found as

$$Normal = \frac{\mathbf{E_2} \times \mathbf{E_1}}{||\mathbf{E_2} \times \mathbf{E_1}||} \tag{3.5}$$

Finally, the radius of the circumscribing circle of a triangle to be used later in BVH can be calculated as

$$s = \frac{|\mathbf{E_1}| + |\mathbf{E_2}| + |\mathbf{E_3}|}{2} \tag{3.6}$$

$$R = \frac{|\mathbf{E_1}| \cdot |\mathbf{E_2}| \cdot |\mathbf{E_3}|}{\sqrt{s \cdot (s - |\mathbf{E_1}|) \cdot (s - |\mathbf{E_2}|) \cdot (s - |\mathbf{E_3}|)}} \tag{3.7}$$

The final structure of the triangle structure is shown in Figure 3.3.

3.1.2. Constructing the Ray Plane

Now that the object has been constructed, we turn the attention to the incident ray plane, from which the rays will be shot. In order to determine the size of the ray plane, we traverse the triangular object and find the coordinates of a cube that tightly covers the object. Figure 3.3 is an illustrative image demonstrating the method used to find the coordinates of the minimum bounding rectangular box (MRBM) around the object.

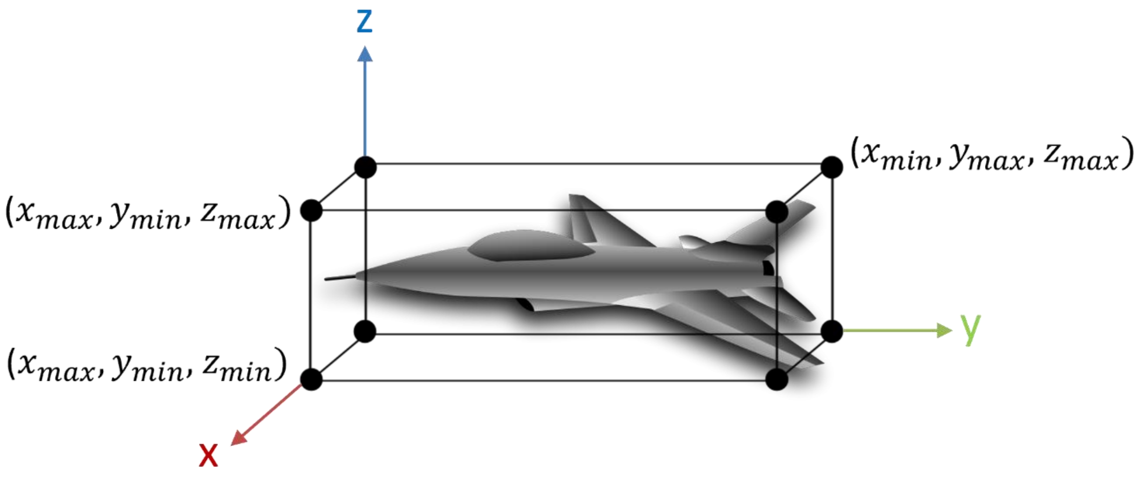

z
$(x_{min}, y_{max}, z_{max})$
$(x_{max}, y_{min}, z_{max})$
y
$(x_{max}, y_{min}, z_{min})$
x

Figure 3. 3 Identifying coordinates of the MRBM

Since rays extend beyond the extreme coordinates of the cube would not intersect the target, we take the projection of vertices of the cube thereby determining the boundaries of the ray plane with consideration of the incident angles θ_i and φ_i where φ is the angle in the xy-plane from the x-axis and θ is the angle from the positive z-axis down to xy-plane.

Figure 3.4 presents the constructed F15 model from the implementation, showing the ray plane in white and the MRBM in purple.

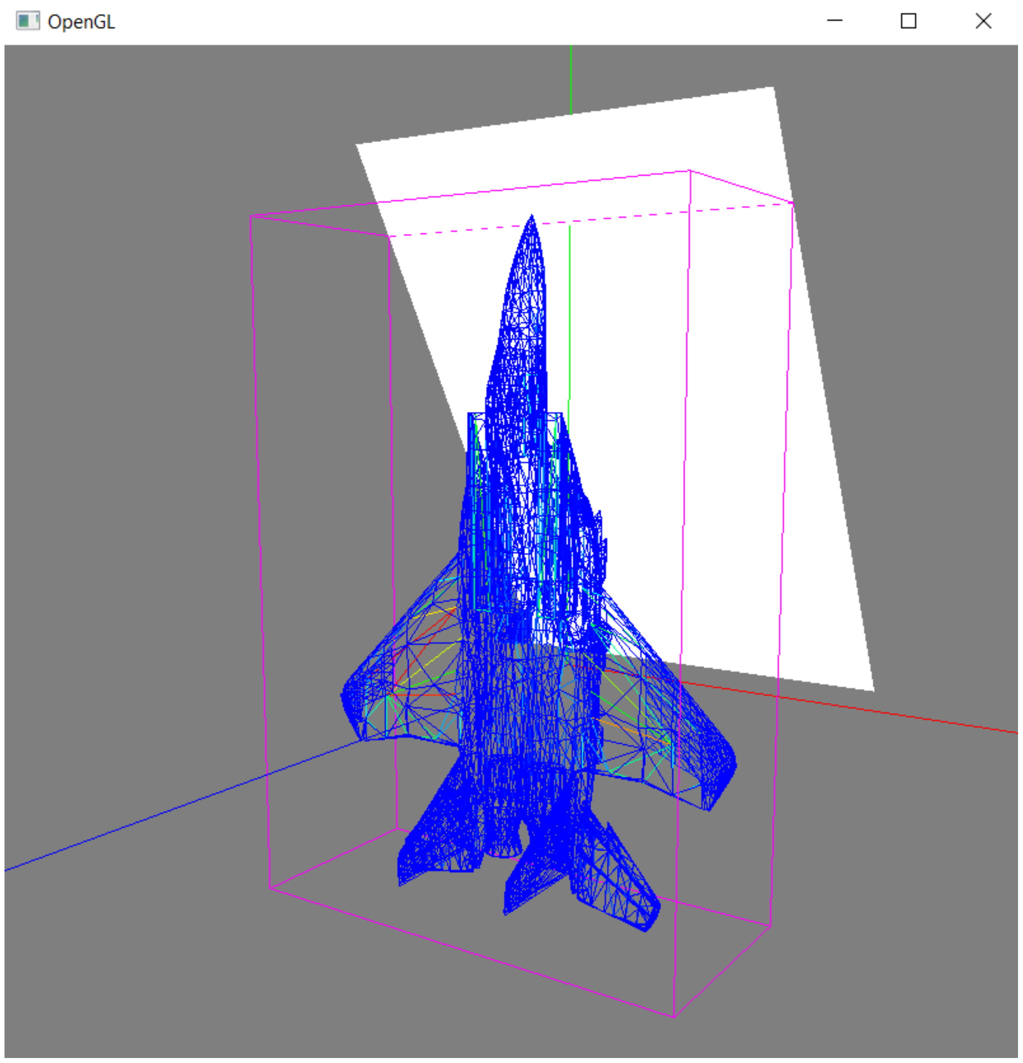


Figure 3. 4 Simulation environment for F15 model for $\theta_i = 150°$ and $\varphi_i = 90°$ with the MRBM and the incident ray plane in white

Now that the ray plane has been formed, we proceed to create the rays, using the wavelength to determine the distance between each ray on the ray plane.

$$\Delta = \frac{\lambda}{b} = \frac{c}{f \cdot b} \tag{3.8}$$

where λ is the wavelength, c is the speed of light, f is the frequency of the simulation and b is a user defined value that is used to determine the number of rays. Notably, a higher value of b results in a smaller distance between rays, leading to a greater number of rays and, consequently, more accuracy in the simulation.

The calculations for incident vertical and horizontal polarizations, and the incident direction of a ray are provided respectively in Equations 3.9, 3.10, and 3.11.

$$\mathbf{E^i}_\theta = \sin\theta_i \sin\varphi_i \,\hat{x} + \cos\theta_i \sin\varphi_i \,\hat{y} + \cos\varphi_i \,\hat{z} \tag{3.9}$$

$$\mathbf{E^i}_\varphi = \cos\theta_i \,\hat{x} - \sin\theta_i \,\hat{y} \tag{3.10}$$

$$\hat{k}_i = \frac{\sin\theta_i \cos\varphi_i \,\hat{x} + \cos\theta_i \cos\varphi_i \hat{y} - \sin\varphi_i \,\hat{z}}{\sqrt{|\sin\theta_i \cos\varphi_i \,\hat{x} + \cos\theta_i \cos\varphi_i \hat{y} - \sin\varphi_i \,\hat{z}|}} \tag{3.11}$$

Finally, the power of each incident ray, denoted as Ai, is calculated as the square root of Δ value and remains consistent across all rays.

3.1.3. Ray Tracing

After setting up the ray plane and placing the rays on it, we trace each ray until they escape the object. Ray tracing consist of two separate parts. First, using the Möller-Trumbore intersection algorithm [11], we check each ray against all the triangles in the object to find which triangles are in the ray's path. After finding all the triangles that lie on a ray's path, we select the triangle that is closest in distance for the intersection and update a ray's coordinates accordingly. Next, we call the bounce ray function to update the remaining properties of a ray. This includes its new direction and the polarization of the electric field.

3.3.3.1 Möller–Trumbore Intersection Algorithm

Möller-Trumbore intersection algorithm [11] is a fast method for determining where a ray intersects a triangle in three-dimensional space without needing precomputation of the plane equation of the plane containing the triangle. A ray $\mathbf{R(t)}$ with an incident position $\mathbf{O^i}$ and normalized direction $\hat{k}_i$ is defined as

$$\mathbf{R(t)} = \mathbf{O^i} + t\hat{k}_i \tag{3.12}$$

where the parameter t represents the distance from the ray's incident point to the point of interest, measured along the ray's direction. Meanwhile, the triangle is defined by three vertices, $\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}$. Then, a point on a triangle is given by

$$\mathbf{T}(u, v) = (1 - u - v)\mathbf{v_1} + u\mathbf{v_2} + u\mathbf{v_3} \tag{3.13}$$

where (u, v) are the barycentric coordinates, which must satisfy, $u \geq 0$, $v \geq 0$ and $u + v \leq 1$. Computing the intersection between the ray and the triangle is equivalent to $\mathbf{R(t)} = \mathbf{T}(u, v)$, which yields:

$$\mathbf{O^i} + t\hat{k}_i = (1 - u - v)\mathbf{v_1} + u\mathbf{v_2} + u\mathbf{v_3} \tag{3.14}$$

Rearranging the terms gives:

$$\begin{bmatrix} -\hat{k}_i & \mathbf{v_2} - \mathbf{v_1} & \mathbf{v_3} - \mathbf{v_1} \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{O^i} - \mathbf{v_1} \tag{3.15}$$

This indicates that by solving the above linear equations, one can determine the barycentric coordinates (u, v), as well as the distance t from the ray's incident position to the point of intersection.

Then, the objective is to determine whether the ray intersects with the triangle, and if so, to locate the point of intersection. The following cross product gives a vector perpendicular to both the ray's direction and the second edge of the triangle.

$$\mathbf{P} = \mathbf{E_2} \times \hat{k}_i \tag{3.16}$$

Then, the determinant can be calculated by taking the dot product of **P** and the first edge of the triangle. If the determinant is zero, it indicates that the ray lies in the plane of the triangle or is parallel to it. In both cases, it's not useful for the intersection test. Alternatively, if the determinant is greater than zero, its inverse is used to test the satisfying conditions for barycentric coordinates.

3.3.3.2 Ray Bouncing

After a triangle selected for the intersection and the ray's position vector is updated with the intersection point, the next step is to update the ray's new direction vector as well as the scattered electric field.

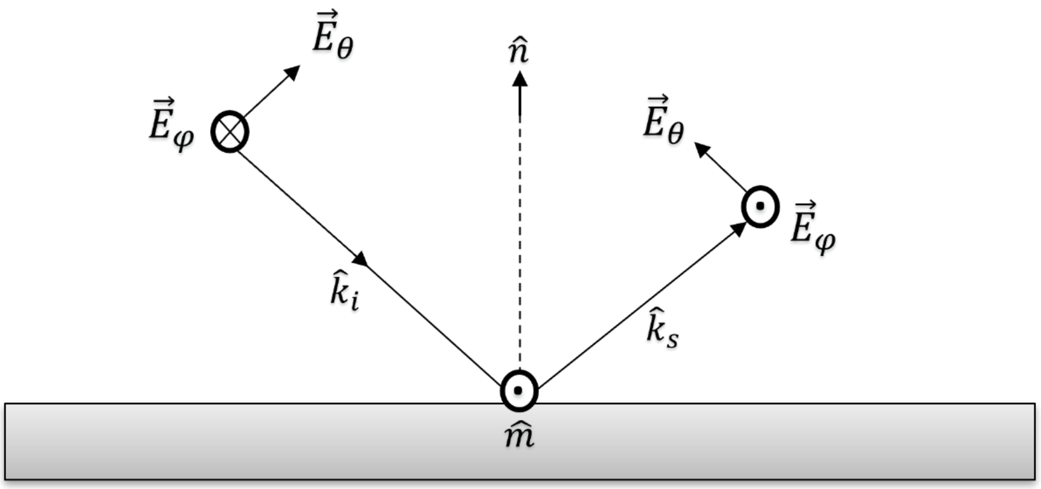


Figure 3. 5 Ray bouncing from PEC

Then, referring to Figure 3.5, define a unit vector

$$\boldsymbol{m} = \hat{k}_i \times \hat{n} \tag{3.17}$$

Also, the new direction of the ray can be found as

$$\hat{k}_s = \hat{k}_i - 2\hat{n}(\hat{n} \cdot \hat{k}_i) \tag{3.18}$$

Then, we can define two unit vectors $\hat{P}_\perp$ and $\hat{P}_\parallel$

$$\hat{P}_\perp = \hat{m} \tag{3.19}$$

$$\hat{P}_\parallel = \hat{k}_s \times \hat{m} \tag{3.20}$$

Hence, the scattered electric field vectors can be calculated as

$$\mathbf{E^s}_{\boldsymbol{\theta}} = \rho(\hat{P}_\perp \cdot \mathbf{E^i}_{\boldsymbol{\theta}}) \tag{3.21}$$

$$\mathbf{E^s}_{\boldsymbol{\varphi}} = \rho(\hat{P}_{\parallel} \cdot \mathbf{E^i}_{\boldsymbol{\varphi}}) \tag{3.22}$$

Where ρ is the reflection coefficient of the surface.

3.3.4 Ray Tube Integration and RCS Calculation

After a ray escapes the object, a ray tube integration formula in [4] is performed to sum up its contribution to the total scattered field. Hence, one possible representation of the time-dependent scattered far-field with monochromatic frequency e^{jwt} is

$$\mathbf{E^s}(r, \theta, \varphi) \approx e^{-jkr}(\hat{\theta}A_\theta + \hat{\varphi}A_\varphi) \tag{3.23}$$

in which the ‘‘EH formulation’’ in [4] can be used to obtain

$$\begin{bmatrix} A_\theta \\ A_\varphi \end{bmatrix} = \left(\frac{jk}{4\pi}\right) \int\int_{tube} e^{j\boldsymbol{k}\cdot r'} \times \left\{ \begin{bmatrix} -\hat{\varphi} \\ \hat{\theta} \end{bmatrix} \times \mathbf{E_{ap}}(\mathbf{r}') + Z_0 \begin{bmatrix} \hat{\theta} \\ \hat{\varphi} \end{bmatrix} \times \mathbf{H}_{\boldsymbol{ap}}(\mathbf{r}') \right\} \cdot \hat{\boldsymbol{n}}\, dx'\, dy' \tag{3.24}$$

where $\boldsymbol{k}$ is the propagation direction, k is the wave number, Z_0 is the characteristic impedance of the medium; $\mathbf{E_{ap}}(\mathbf{r})$ and $\mathbf{H}_{\boldsymbol{ap}}(\mathbf{r})$ are the electric and magnetic fields on the exit aperture or wave front of the ray tube, respectively. Passage to the approximation

$$\begin{bmatrix} A_\theta \\ A_\varphi \end{bmatrix} \approx \begin{bmatrix} B_\theta \\ B_\varphi \end{bmatrix} \left(\frac{jk}{4\pi}\right) S(\theta, \varphi)\, \Delta A_{exit} e^{j\boldsymbol{k}\cdot\boldsymbol{r_A}} \tag{3.25}$$

where $\mathbf{r_A}$ is the last bounce point of the ray. B_θ, B_φ, and shape function $S(\theta, \varphi)$ derived in [1][4] as

$$\begin{aligned} B_\theta = 0.5[&-s_1 \cos\varphi\, E_3 - s_2 \sin\varphi\, E_3 \\ &+ s_3(\cos\varphi\, E_1 + \sin\varphi\, E_2)] \\ &+ 0.5Z_0[s_1(\cos\theta \sin\varphi\, H_3 + \sin\theta\, H_2) \\ &+ s_2(-\sin\theta\, H_1 - \cos\theta \cos\varphi H_3) \\ &+ s_3(\cos\theta \cos\varphi\, H_2 - \cos\theta \sin\varphi\, H_1)] \end{aligned} \tag{3.26}$$

$$B_\varphi = 0.5[s_1(\cos\theta\sin\varphi\, E_3 + \sin\theta\, E_2) + s_2(-\sin\theta\, E_1 - \cos\theta\cos\varphi\, E_3) + s_3(\cos\theta\cos\varphi\, E_2 - \cos\theta\sin\varphi\, E_1)] + 0.5Z_0[s_1\cos\varphi\, H_3 + s_2\sin\varphi\, H_3 + s_3(-\cos\varphi\, H_1 - \sin\varphi\, H_2)] \tag{3.27}$$

$$S(\theta,\varphi) = 1\ (if\ the\ ray\ tube\ area\ is\ small) \tag{3.28}$$

where $\boldsymbol{E}(\boldsymbol{A}) = E_1\hat{x} + E_2\hat{y} + E_3\hat{z}$ and $\boldsymbol{H}(\boldsymbol{A}) = H_1\hat{x} + H_2\hat{y} + H_3\hat{z}$ are, respectively, the electric and magnetic field associated with each ray at $\boldsymbol{A}$, and $\hat{s} = s_1\hat{x} + s_2\hat{y} + s_3\hat{z}$ is the exit ray direction.

Finally, by summing the contribution from each ray directly, the total scattered field of the target can be obtained as

$$\mathbf{E}^s(r,\theta,\varphi) = \sum_{\boldsymbol{i\,rays}} \mathbf{E}_i^s(r,\theta,\varphi) \tag{3.29}$$

After obtaining the total scattered field, the RCS for a look angle is calculated as

$$\sigma(m^2) = 4\pi|\mathbf{E}^s(\mathrm{r},\theta,\varphi)| \tag{3.30}$$

The best way to validate an RCS simulation is to compare simulated radar cross-section values with analytically derived equations for simple objects. For instance, following the RCS equation derived in [12] for a flat rectangular plate with height a and width b will be utilized for the validation process in Section 6.

$$\sigma_{3-D} = \lim_{r\to\infty} 4\pi r^2 \left|\frac{\mathbf{E^{scat}}}{\mathbf{E^{inc}}}\right|^2 = 4\pi(\frac{ab}{\lambda})^2(\cos^2\theta_s \sin^2\varphi_s + \cos^2\varphi_s)\left[\frac{sin(X)}{X}\right]^2\left[\frac{sin(Y)}{Y}\right]^2 \tag{3.31}$$

where

$$X = \frac{\beta a}{2}\sin\theta_s\cos\varphi_s \tag{3.32}$$

$$Y = \frac{\beta b}{2}(\sin\theta_s\sin\varphi_s - \sin\theta_i) \tag{3.33}$$

3.2. Surface Current Induction

Furthermore, the total current density induced on a triangle's surface by a ray can be calculated as

$$\mathbf{J^s} = \hat{n} \times (\mathbf{H^s} - \mathbf{H^i}) \tag{3.34}$$

When dealing with a Perfectly Electrically Conducting (PEC) surface, the equation simplifies under the PO approximation to

$$\mathbf{J^s} = 2\hat{n} \times \mathbf{H^i} \tag{3.35}$$

Then, the total current density induced on a triangle is found by summing up the contributions of each ray incident on a triangle.

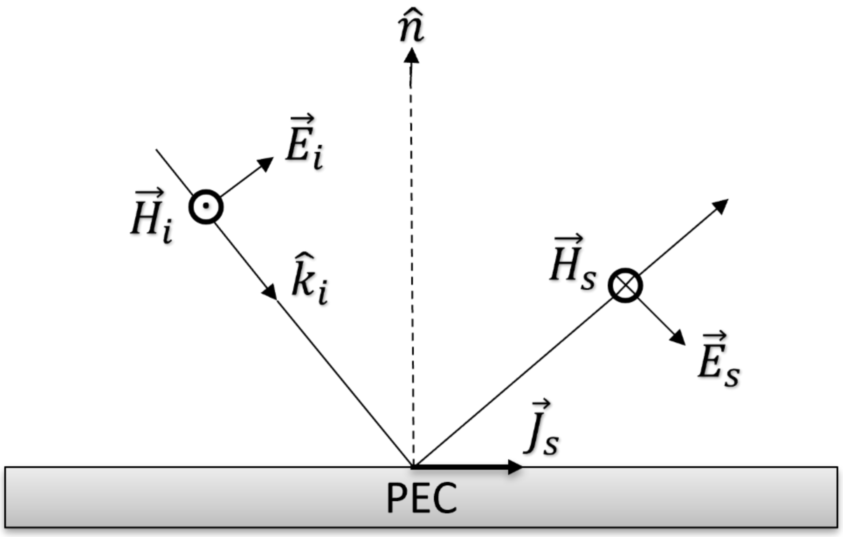


Figure 3. 6 Induced surface current during ray reflection from PEC

$$\mathbf{J^s}_{\mathbf{tri}} = \sum_{i\,rays} \mathbf{J^s}_{\mathbf{i}} \tag{3.36}$$

3.3. Design

3.3.1. Sequential Design

The sequential simulation will be developed in C++, beginning with the reading of the triangles that constitute the object of interest. Using MRBM coordinates, a ray plane will be constructed, followed by the generation of rays on this plane. These rays will be projected onto the object and traced until they escape. The backscattered electric field will be computed, and after tracing each ray, the RCS will be calculated for a given look angle. This process is illustrated in Figure 3.7.

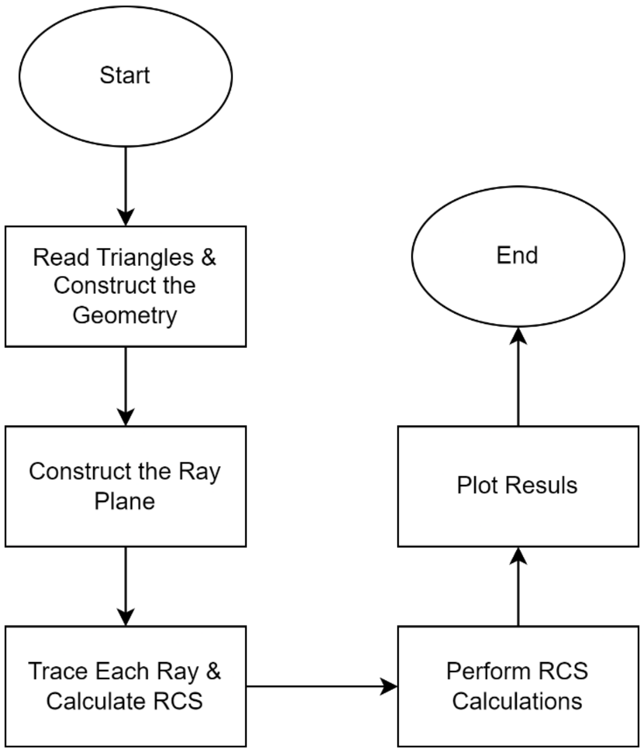

Start
Read Triangles & Construct the Geometry
Construct the Ray Plane
Trace Each Ray & Calculate RCS
Perform RCS Calculations
Plot Resuls
End

Figure 3. 7 Flowchart of the sequential implementation

3.3.2. Parallelized Design

In the parallelized version of the simulation the computational workload is divided between the CPU and GPU. On the CPU side, the object construction, determination of ray-plane boundaries, and the generation of rays take place. Whereas, on the GPU side, rays are traced in parallel, and upon completion of tracing a ray, its contribution to the scattered electric field is calculated and added immediately according to Equation 3.29. To prevent race conditions during parallel addition, atomic operations will be used. After tracing all rays and summing up their contributions to the scattered electric field, the RCS values are computed on the CPU side at the end of the simulation. This entire process is illustrated in Figure 3.8.

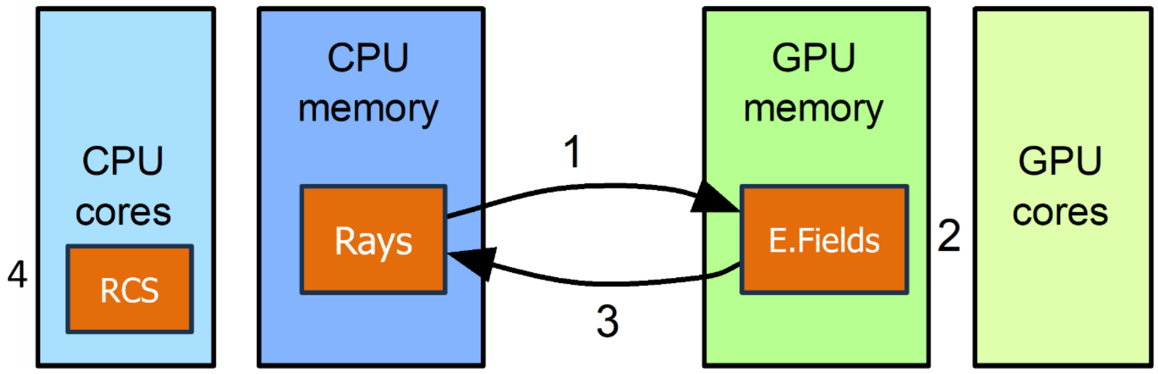


Figure 3. 8 CPU-GPU workflow

The flowchart of the parallelized design is given in Figure 3.9.

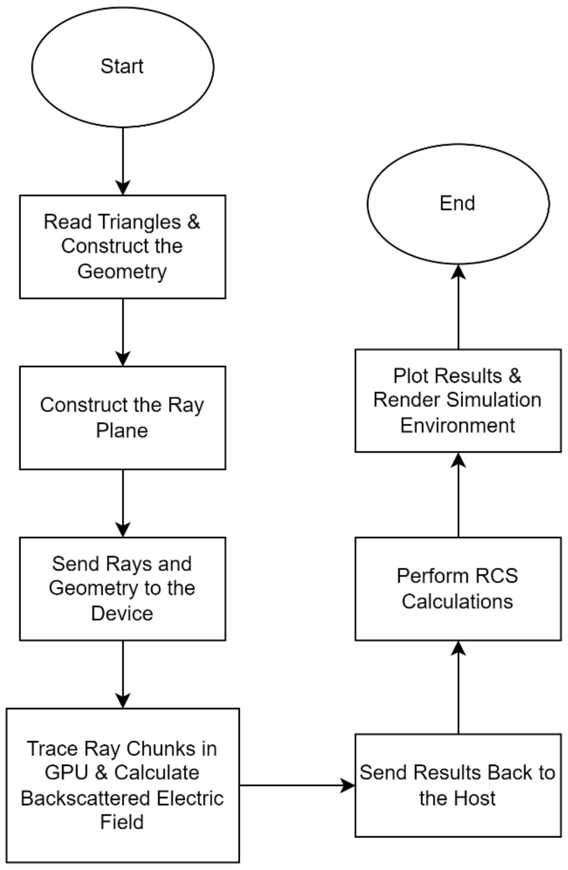

Start
Read Triangles & Construct the Geometry
Construct the Ray Plane
Send Rays and Geometry to the Device
Trace Ray Chunks in GPU & Calculate Backscattered Electric Field
Send Results Back to the Host
Perform RCS Calculations
Plot Results & Render Simulation Environment
End

Figure 3. 9 Flowchart of the parallelized design

The API utilization of the flowchart is illustrated in Figure 3.8. In this context, "host" refers to the CPU, while "device" refers to the GPU.

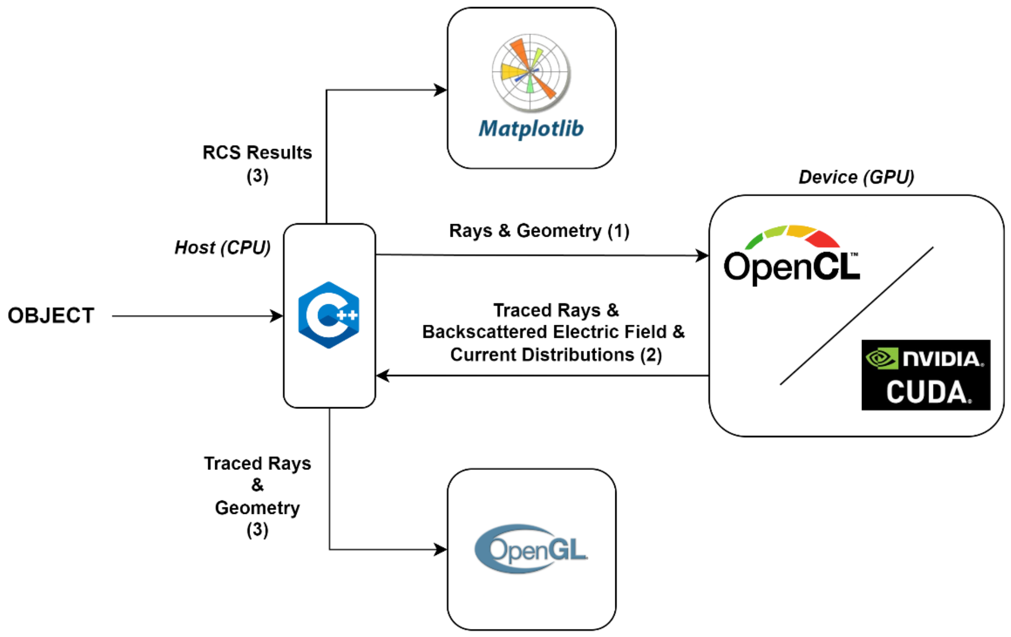


Figure 3. 10 API utilization breakdown

Memory coalescing is essential for optimizing memory access in GPU computing. It refers to the process of combining multiple memory accesses into a single transaction to minimize memory latency and maximize bandwidth utilization. To achieve efficient coalescing, several criteria must be met. From Section 1.6, it's shown that threads within a SM are organized into warps. To achieve efficient coalescing, threads within a warp should ideally access memory locations in sequence. This means that neighboring threads in a warp should access consecutive memory locations, ensuring that memory transactions can be merged into a single, contiguous block. When threads access memory in a scattered or non-sequential manner, multiple memory transactions may be required, leading to decreased bandwidth utilization and increased latency.

In order to achieve memory coalescing, the parallel implementation will adopt the Structure of Arrays (SoA) model instead of the conventional Array of Structures (AoS) model. In contrast to creating distinct instances of ray objects stored within an array, the SoA model linearizes the data, creating arrays of size 'raySize' for each member. This architectural shift optimizes memory access for GPU threads and improves memory coalescing, contributing to enhanced computational performance in GPU implementations. Figure 3.11 shows the memory organization in two different models.

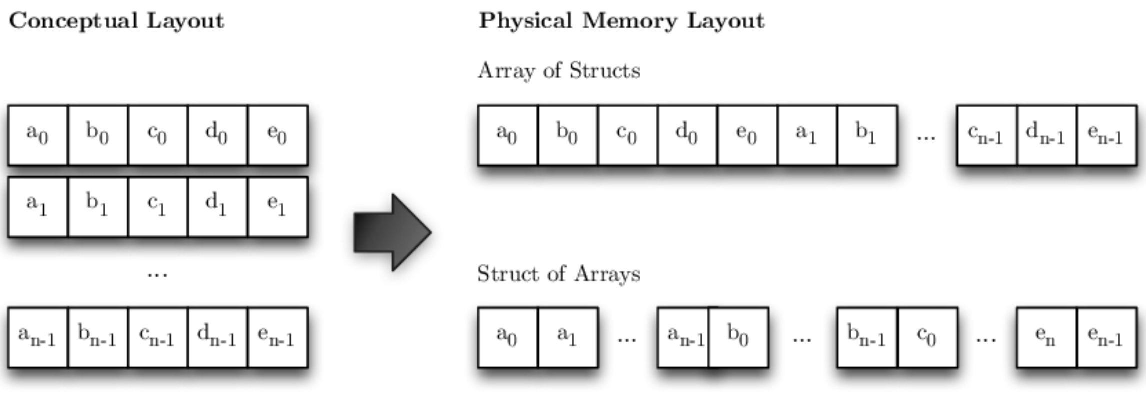
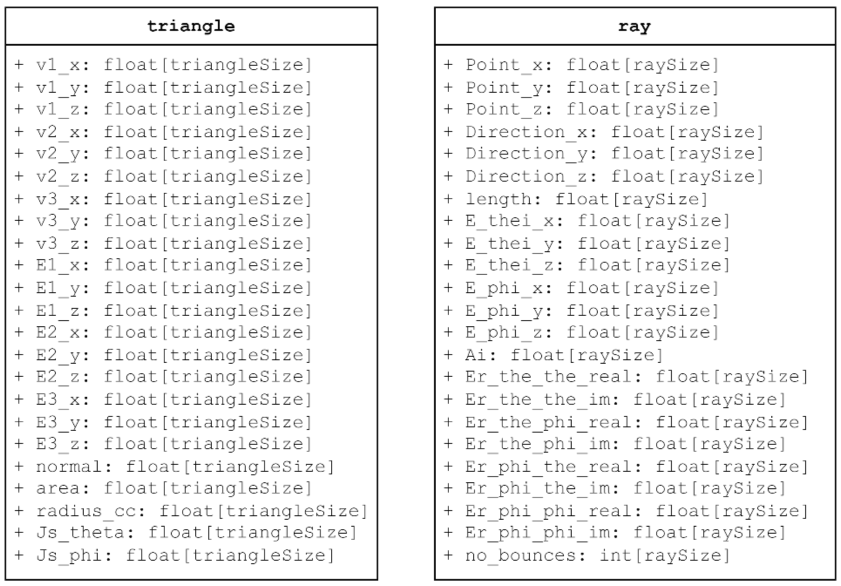

Conceptual Layout
Physical Memory Layout
Array of Structs
Struct of Arrays
a_0 b_0 c_0 d_0 e_0
a_1 b_1 c_1 d_1 e_1
...
a_{n-1} b_{n-1} c_{n-1} d_{n-1} e_{n-1}
a_0 b_0 c_0 d_0 e_0 a_1 b_1 ... c_{n-1} d_{n-1} e_{n-1}
a_0 a_1 ... a_{n-1} b_0 ... b_{n-1} c_0 ... e_n e_{n-1}

Figure 3. 11 Memory organization in SoA and AoS models

In conclusion, the data structures for the ray and triangle classes are shown in Figure 3.12. As previously explained, only one triangle and one ray object will be created in the simulation. Each array within these objects will have a number of members equal to raySize or triangleSize, respectively. By consolidating data into single objects with appropriately sized arrays, the memory access patterns become more contiguous.



```
triangle
+ v1_x: float[triangleSize]
+ v1_y: float[triangleSize]
+ v1_z: float[triangleSize]
+ v2_x: float[triangleSize]
+ v2_y: float[triangleSize]
+ v2_z: float[triangleSize]
+ v3_x: float[triangleSize]
+ v3_y: float[triangleSize]
+ v3_z: float[triangleSize]
+ E1_x: float[triangleSize]
+ E1_y: float[triangleSize]
+ E1_z: float[triangleSize]
+ E2_x: float[triangleSize]
+ E2_y: float[triangleSize]
+ E2_z: float[triangleSize]
+ E3_x: float[triangleSize]
+ E3_y: float[triangleSize]
+ E3_z: float[triangleSize]
+ normal: float[triangleSize]
+ area: float[triangleSize]
+ radius_cc: float[triangleSize]
+ Js_theta: float[triangleSize]
+ Js_phi: float[triangleSize]
```

```
ray
+ Point_x: float[raySize]
+ Point_y: float[raySize]
+ Point_z: float[raySize]
+ Direction_x: float[raySize]
+ Direction_y: float[raySize]
+ Direction_z: float[raySize]
+ length: float[raySize]
+ E_thei_x: float[raySize]
+ E_thei_y: float[raySize]
+ E_thei_z: float[raySize]
+ E_phi_x: float[raySize]
+ E_phi_y: float[raySize]
+ E_phi_z: float[raySize]
+ Ai: float[raySize]
+ Er_the_the_real: float[raySize]
+ Er_the_the_im: float[raySize]
+ Er_the_phi_real: float[raySize]
+ Er_the_phi_im: float[raySize]
+ Er_phi_the_real: float[raySize]
+ Er_phi_the_im: float[raySize]
+ Er_phi_phi_real: float[raySize]
+ Er_phi_phi_im: float[raySize]
+ no_bounces: int[raySize]
```

Figure 3. 12 Data structures for ray and triangle classes

4. IMPLEMENTATION

The simulation for both sequential and parallelized versions start with reading the object of interest from a file stream. Then constructing the ray plane as well as the rays on it. The common functions used in both simulations are explained below.

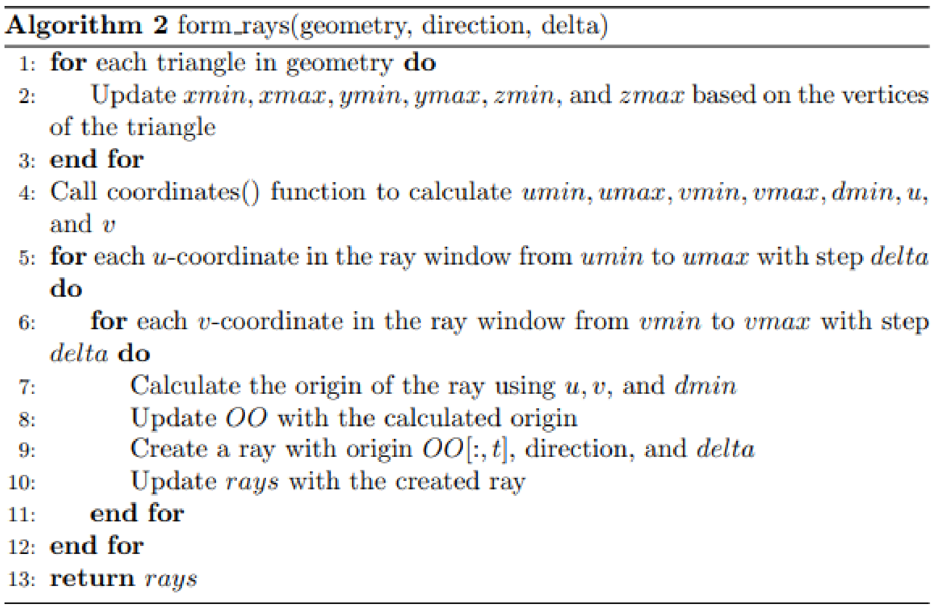
Common functions used in both types of simulation:

- **load_raw():** Reads triangle vertices and calculates the circumscribing circle radius for each triangle from a file stream. It stores the vertices and radius values in vectors and returns a triangle geometry object containing this data. Additionally, it updates the minimum radius encountered during the process. The pseudocode of the function is given in Figure 4.1.

```
Algorithm 1 load_raw(FileStream myfile)
while there are lines in myfile do
    if $has\_float$ is true then
        Read vert1, vert2, vert3
        Calculate side lengths a, b, c
        Calculate circumscribing circle radius radius
        Update minRadius with the minimum of minRadius and radius
    end if
end while
Create a triangle geometry
Return geometry
```

Figure 4. 1 Pseudocode of the load_raw function

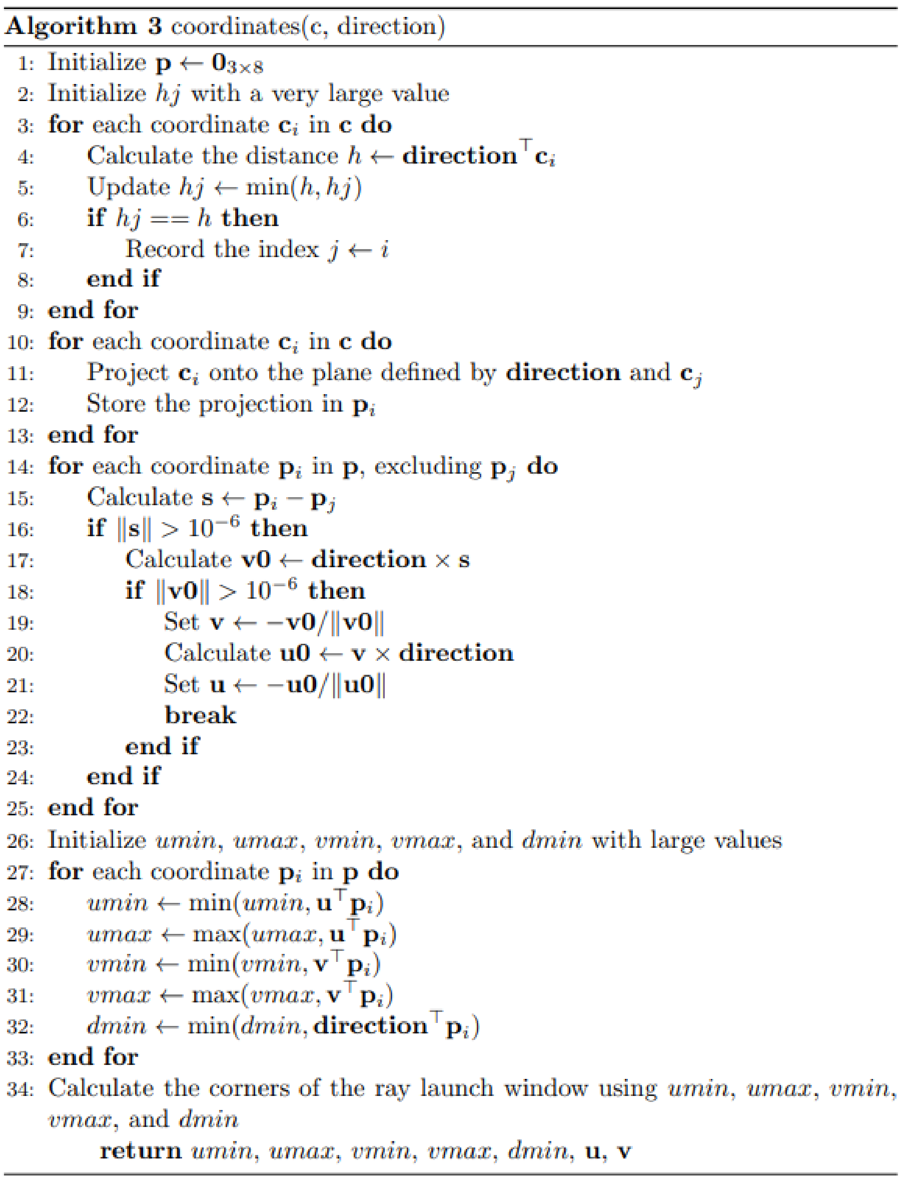
- **form_rays():** Computes rays originating from a plane and passing through a bounding box enclosing a set of triangles. It calculates the minimum and maximum coordinates of the bounding box, then generates rays within this box with specified intervals. The pseudocode of the function is given in Figure 4.2.



```
Algorithm 2 form_rays(geometry, direction, delta)
1:  for each triangle in geometry do
2:      Update $xmin, xmax, ymin, ymax, zmin$, and $zmax$ based on the vertices of the triangle
3:  end for
4:  Call coordinates() function to calculate $umin, umax, vmin, vmax, dmin, u$, and $v$
5:  for each $u$-coordinate in the ray window from $umin$ to $umax$ with step $delta$ do
6:      for each $v$-coordinate in the ray window from $vmin$ to $vmax$ with step $delta$ do
7:          Calculate the origin of the ray using $u, v$, and $dmin$
8:          Update $OO$ with the calculated origin
9:          Create a ray with origin $OO[:, t]$, direction, and $delta$
10:         Update $rays$ with the created ray
11:     end for
12: end for
13: return $rays$
```

Figure 4. 2 Pseudocode of form_rays function

- **coordinates():** Calculates the coordinates of a bounding box in a specified direction. Also, it determines the maximum and minimum coordinates as well as the distance along the direction vector for the reference plane. The pseudocode of the function is given in Figure 4.3.

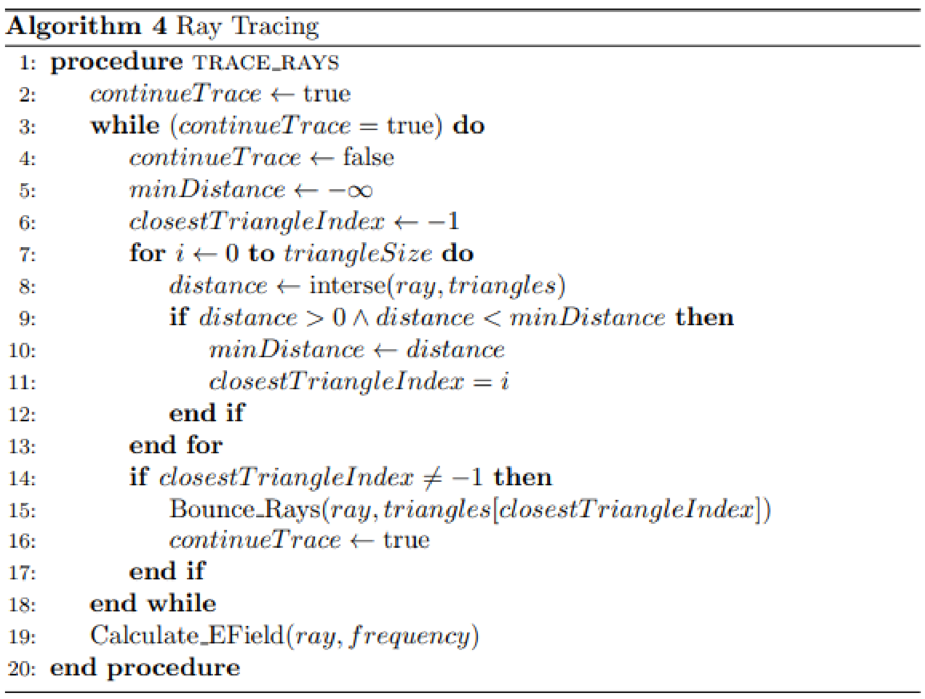


Algorithm 3 coordinates(c, direction)

```
Initialize $\mathbf{p} \leftarrow \mathbf{0}_{3\times 8}$
Initialize $hj$ with a very large value
for each coordinate $\mathbf{c}_i$ in $\mathbf{c}$ do
    Calculate the distance $h \leftarrow \mathbf{direction}^\top \mathbf{c}_i$
    Update $hj \leftarrow \min(h, hj)$
    if $hj == h$ then
        Record the index $j \leftarrow i$
    end if
end for
for each coordinate $\mathbf{c}_i$ in $\mathbf{c}$ do
    Project $\mathbf{c}_i$ onto the plane defined by direction and $\mathbf{c}_j$
    Store the projection in $\mathbf{p}_i$
end for
for each coordinate $\mathbf{p}_i$ in $\mathbf{p}$, excluding $\mathbf{p}_j$ do
    Calculate $\mathbf{s} \leftarrow \mathbf{p}_i - \mathbf{p}_j$
    if $\|\mathbf{s}\| > 10^{-6}$ then
        Calculate $\mathbf{v0} \leftarrow \mathbf{direction} \times \mathbf{s}$
        if $\|\mathbf{v0}\| > 10^{-6}$ then
            Set $\mathbf{v} \leftarrow -\mathbf{v0}/\|\mathbf{v0}\|$
            Calculate $\mathbf{u0} \leftarrow \mathbf{v} \times \mathbf{direction}$
            Set $\mathbf{u} \leftarrow -\mathbf{u0}/\|\mathbf{u0}\|$
            break
        end if
    end if
end for
Initialize $umin$, $umax$, $vmin$, $vmax$, and $dmin$ with large values
for each coordinate $\mathbf{p}_i$ in $\mathbf{p}$ do
    $umin \leftarrow \min(umin, \mathbf{u}^\top \mathbf{p}_i)$
    $umax \leftarrow \max(umax, \mathbf{u}^\top \mathbf{p}_i)$
    $vmin \leftarrow \min(vmin, \mathbf{v}^\top \mathbf{p}_i)$
    $vmax \leftarrow \max(vmax, \mathbf{v}^\top \mathbf{p}_i)$
    $dmin \leftarrow \min(dmin, \mathbf{direction}^\top \mathbf{p}_i)$
end for
Calculate the corners of the ray launch window using $umin$, $umax$, $vmin$,
$vmax$, and $dmin$
    return $umin$, $umax$, $vmin$, $vmax$, $dmin$, $\mathbf{u}$, $\mathbf{v}$
```

Figure 4. 3 Pseudocode of the coordinates function

- **trace_rays():** This function traces a ray by iterating over the triangles of the object and calling the interse() function to test for intersections with the ray. The pseudocode of the algorithm is given in Figure 4.4.



```
Algorithm 4 Ray Tracing
procedure TRACE_RAYS
    $continueTrace \leftarrow$ true
    while ($continueTrace =$ true) do
        $continueTrace \leftarrow$ false
        $minDistance \leftarrow -\infty$
        $closestTriangleIndex \leftarrow -1$
        for $i \leftarrow 0$ to $triangleSize$ do
            $distance \leftarrow$ interse($ray, triangles$)
            if $distance > 0 \wedge distance < minDistance$ then
                $minDistance \leftarrow distance$
                $closestTriangleIndex = i$
            end if
        end for
        if $closestTriangleIndex \neq -1$ then
            Bounce_Rays($ray, triangles[closestTriangleIndex]$)
            $continueTrace \leftarrow$ true
        end if
    end while
    Calculate_EField($ray, frequency$)
end procedure
```

Figure 4. 4 Pseudocode of the trace_rays function

- **interse():** Function returns the intersection point between the ray and the geometric object if the test is successful. The pseudocode of the function is given in Figure 4.5.

```
Algorithm 5 interse()
Require: Ray origin $\mathbf{O}$, Ray direction $\mathbf{D}$, Triangle vertices $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$
Ensure: Intersection point $\mathbf{P}$ or no intersection
 1: $\epsilon \leftarrow$ small positive number
 2: $\mathbf{E}_1 \leftarrow \mathbf{V}_1 - \mathbf{V}_0$
 3: $\mathbf{E}_2 \leftarrow \mathbf{V}_2 - \mathbf{V}_0$
 4: $\mathbf{H} \leftarrow \mathbf{D} \times \mathbf{E}_2$
 5: $a \leftarrow \mathbf{E}_1 \cdot \mathbf{H}$
 6: if $|a| < \epsilon$ then
 7:     return no intersection                    ▷ Ray is parallel to the triangle
 8: end if
 9: $f \leftarrow \frac{1}{a}$
10: $\mathbf{S} \leftarrow \mathbf{O} - \mathbf{V}_0$
11: $u \leftarrow f(\mathbf{S} \cdot \mathbf{H})$
12: if $u < 0$ or $u > 1$ then
13:     return no intersection
14: end if
15: $\mathbf{Q} \leftarrow \mathbf{S} \times \mathbf{E}_1$
16: $v \leftarrow f(\mathbf{D} \cdot \mathbf{Q})$
17: if $v < 0$ or $u + v > 1$ then
18:     return no intersection
19: end if
20: $t \leftarrow f(\mathbf{E}_2 \cdot \mathbf{Q})$
21: if $t > \epsilon$ then
22:     $\mathbf{P} \leftarrow \mathbf{O} + t\mathbf{D}$                          ▷ Intersection point
23:     return $\mathbf{P}$
24: else
25:     return no intersection           ▷ Intersection is behind the ray origin
26: end if
```

Figure 4. 5 Pseudocode of the interse() function

- **bounce_rays():** This function simulates the behavior of a ray as it bounces off triangles. When a ray undergoes a bounce, this function computes and adjusts its new direction, updates the polarization of the electric field, and recalculates the power remaining of a ray. The pseudocode of the function is given in Figure 4.6.

```
Algorithm 6 bounce_rays
procedure GPU_BOUNCE_RAYS(Input: direction vectors, points, edge vectors, vertices, normals, electric fields, ray size, triangle size, ...)
    Compute reflected wave coefficients $refCoefPer$ and $refCoefPar$
    Compute vector $m$ representing the reflected direction
    Normalize $m$ vector
    Compute perpendicular and parallel polarization vectors $PerPol$ and $ParPol$
    Compute reflected direction and update $new\_direction$
    Compute reflected electric field components for theta polarization
    Compute reflected electric field components for phi polarization
    Increment number of bounces
    Update point coordinates
    Update direction vectors
    Update ray lengths
    Compute current distribution for theta polarization
    Compute current distribution for phi polarization
end procedure
```

Figure 4. 6 Pseudocode of bounce_rays function

- **calculate_EField():** This function is called when a ray stops bouncing and escapes the object. It calculates and adds the backscattered electric field to a shared array.
- **plot_results():** Generates plots displaying the simulation results and facilitates comparison between different simulation outcomes.

4.1. Sequential Implementation

After constructing the object and creating rays the sequential implementation traces each ray sequentially and calculates the backscattered electric field. The function names and pseudocodes of the functions are given below.

<u>Function used in the sequential implementation:</u>

- **main():** The main function calls sub-functions to construct the object, generate and trace the rays, and calculate the backscattered field. The pseudocode of the function is given in Figure 4.4.

```
Algorithm 7 main()
  call load_raw("fileName")
  call form_rays()
  for each ray in rays do
    call trace_ray()
    call calculate_EField()
  end for
  call plot_results()
```

Figure 4.7 Main function of the sequential implementation

4.2. Parallelized Implementation

After constructing the object and creating rays the parallelized implementation sends ray chunks to the GPU and traces each ray in a chunk in parallel. The function names and pseudocodes of the functions are given below.

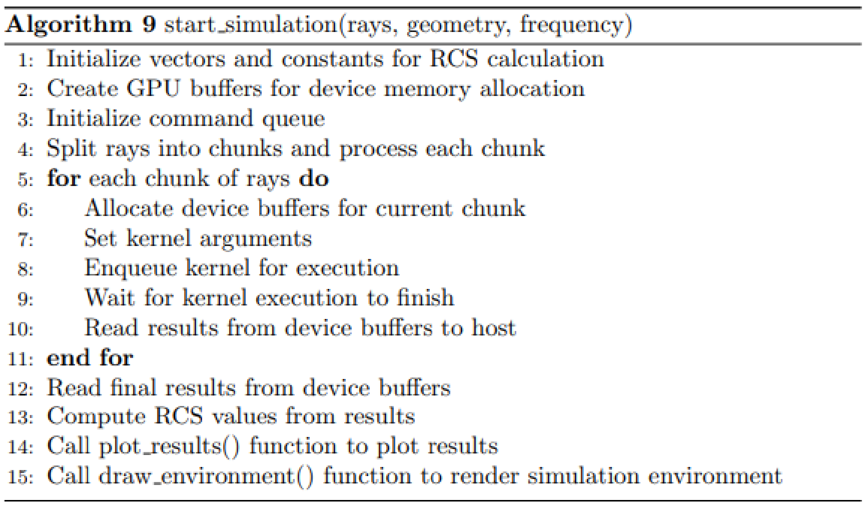
<u>Function used in the parallelized implementation:</u>

- **main():** The main function calls sub-functions to construct the object, generate and trace the rays, and calls start_simulation() function to organize GPU memory and call GPU kernels. The pseudocode of the main function is given in Figure 4.8.

```
Algorithm 8 main()
1: call load_raw("fileName")
2: call form_rays()
3: call start_simulation()
```

Figure 4. 8 Pseudocode of the main function

- **start_simulation():** Initiates data transfer to the GPU and starts parallel kernel on the GPU. After obtaining the results, it creates CPU threads responsible for plotting and rendering the simulation environment. The pseudocode of the function is given in Figure 4.9.



```
Algorithm 9 start_simulation(rays, geometry, frequency)
1: Initialize vectors and constants for RCS calculation
2: Create GPU buffers for device memory allocation
3: Initialize command queue
4: Split rays into chunks and process each chunk
5: for each chunk of rays do
6:     Allocate device buffers for current chunk
7:     Set kernel arguments
8:     Enqueue kernel for execution
9:     Wait for kernel execution to finish
10:    Read results from device buffers to host
11: end for
12: Read final results from device buffers
13: Compute RCS values from results
14: Call plot_results() function to plot results
15: Call draw_environment() function to render simulation environment
```

Figure 4. 9 Pseudocode of the start_simulation function

4.2.1. CUDA Implementation

In the CUDA implementation, GPU memory is allocated from the CPU side using the `cudaMalloc()` function. This function allocates memory on the GPU and returns a pointer to the allocated memory space. Below an example of how memory is allocated for an array `d_ray_dir_x`:

```
    float* d_ray_dir_x;

    cudaError_t   err   =   cudaMalloc((void**)&d_ray_dir_x,   raySize   *
sizeof(float));

    if (err != cudaSuccess) {

    // Handle error

    }
```

The first argument `(void**)&d_ray_dir_x` is the address of the pointer that will hold the allocated memory's address. The second argument `raySize * sizeof(float)` specifies the amount of memory to allocate. `raySize` is the number of elements in the array, and `sizeof(float)` gives the size of each element in bytes. After calling `cudaMalloc`, it's good practice to check if the allocation was successful. `cudaSuccess` indicates a successful allocation. If the allocation fails, `err` will contain an error code that can be used to handle the error appropriately. After allocating GPU memory for all arrays inside the ray instance and geometry instance, we proceed to launch the GPU kernel using the syntax `GPU_trace_rays<<<gridSize, blockSize>>>(d_ptr_dir_x, …)`. Following this kernel launch, two crucial CUDA runtime functions are employed for error checking and synchronization. First, `cudaPeekAtLastError()` is called to retrieve the last error that occurred on the current device, ensuring there were no issues during kernel launch. This step is important for diagnosing and addressing any potential errors early in the execution. Subsequently, `cudaDeviceSynchronize()` is invoked to synchronize the host thread with the device, effectively waiting for all previously launched CUDA kernels and associated streams to complete their execution.

To determine the `gridSize` and `blockSize` for launching a CUDA kernel, the `cudaOccupancyMaxPotentialBlockSize()` function can be used. This function calculates the maximum occupancy for a CUDA kernel given a specified block size limit and provides recommended values for `gridSize` and `blockSize`. Below is an example of this usage.

```
    int minGridSize; // Minimum grid size needed to achieve maximum occupancy

    int blockSize; // Recommended block size for maximum occupancy

    cudaOccupancyMaxPotentialBlockSize(&minGridSize,&blockSize,GPU_trace_rays,0,raySize);

    // Calculate gridSize based on the total number of elements (raySize) and the blockSize

    int gridSize = (raySize + blockSize - 1) / blockSize;

    // Launch the kernel with the determined gridSize and blockSize

    GPU_trace_rays<<<gridSize, blockSize>>>(d_ptr_dir_x, ...);
```

4.2.2. OpenCL Implementation

In the OpenCL implementation, GPU memory is allocated from the host side using the `clCreateBuffer()` function. This function allocates memory on the GPU and returns a buffer object that represents the allocated memory space. Below is an example of how memory is allocated for an array `d_ray_dir_x`:

```
    cl_mem d_ray_dir_x;
```

```
d_ray_dir_x = clCreateBuffer(context, CL_MEM_READ_WRITE, raySize *
sizeof(float), NULL, &err);

if (err != CL_SUCCESS) {

// Handle error

}
```

The first argument `context` is the OpenCL context in which the buffer is created. The second argument `CL_MEM_READ_WRITE` specifies the memory flags, indicating that the buffer can be read and written by the kernel. The third argument `raySize * sizeof(float)` specifies the size of the buffer in bytes. `raySize` is the number of elements in the array, and `sizeof(float)` gives the size of each element in bytes. The fourth argument is a pointer to host memory used to initialize the buffer (NULL in this case). The last argument `&err` returns an error code if the function fails. It's good practice to check if the allocation was successful by ensuring `err` is `CL_SUCCESS`. If the allocation fails, `err` will contain an error code that can be used to handle the error appropriately. After allocating GPU memory for all arrays inside the ray instance and geometry instance, we proceed to launch the GPU kernel using the syntax:

```
err=queue.enqueueNDRangeKernel(kernel,cl::NullRange,cl::NDRange(rayS
ize), cl::NullRange);

if (err != CL_SUCCESS) {

// Handle error

}
```

Following this kernel launch, `queue.finish()` function is called to block until all previously queued OpenCL commands in the given command-queue are issued to the associated device and have completed.

4.3. Rendering Part of the Implementation

On the rendering part of the simulation, the OpenGL classes are explained below.

- **shaderClass:** This object is responsible for managing shader programs within the OpenGL context. The class provides functionality to load vertex and fragment shaders from files, compile them, link them into a shader program, activate the shader program for rendering, and delete the shader program when it's no longer needed.

- **Camera:** This class is designed to manage the camera within an OpenGL environment. It handles camera movements and transformations, allowing for dynamic viewing perspectives.
- **VAO:** This class provides functionality to work with Vertex Array Objects (VAOs) in OpenGL.
- **VBO:** This class represents a Vertex Buffer Object (VBO) in OpenGL, which stores vertex data.
- **texture:** Texture class encapsulates the functionality related to texture handling in OpenGL.

Lastly, below an explanation of the functions used to render the environment.

- **draw_coordinate_system():** This function is responsible for drawing the coordinate system in the scene.
- **draw_ray_plane():** This function draws the ray plane where the rays are shot from.
- **draw_rays():** This function handles the visualization of rays in the scene.
- **draw_bounding_box():** This function draws bounding boxes around objects or regions of interest within the scene.
- **imgui_buttons():** This function deals with the rendering of buttons or interactive elements using ImGui, a user interface library for OpenGL.

Finally, Figure 4.10 gives the function call hierarchy for the proposed parallel implementation.

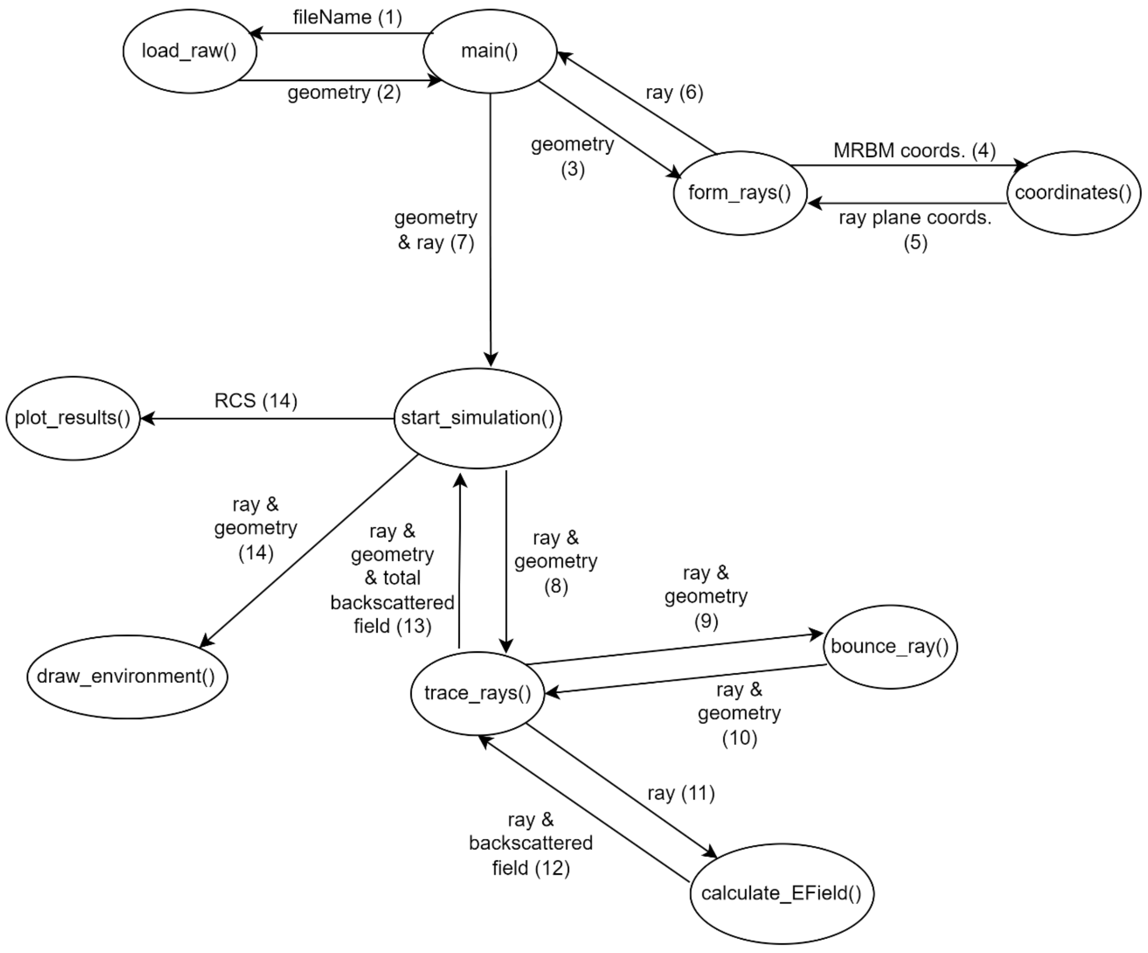


Figure 4. 10 Function call hierarchy of the simulation

5. TESTS AND RESULTS

In this section, an evaluation of the developed simulation will be presented. First, we validate the accuracy of the developed simulation using a calibration target. Next, we analyze the results from the induced surface current density described in section 3.2 and validate its accuracy. Next, to understand computational improvement achieved the timing performance of CPU and GPU implementations are compared. Lastly, timing performance of two different GPU computing APIs is compared.

The equations from 3.31 to 3.33 will be used for the given flat plate in Figure 5.1 for the validation of the RCS simulation. Figure 5.1 illustrates the simulation environment for $a = b = 2$, $\theta_i = 30°$, $\varphi_i = 270°$, $\varphi_s = 90°$ and $0° \leq \theta_s \leq 90°$. Figure 5.2 displays the simulation result for horizontal polarization, and Figure 5.3 displays the simulation result for vertical polarization.

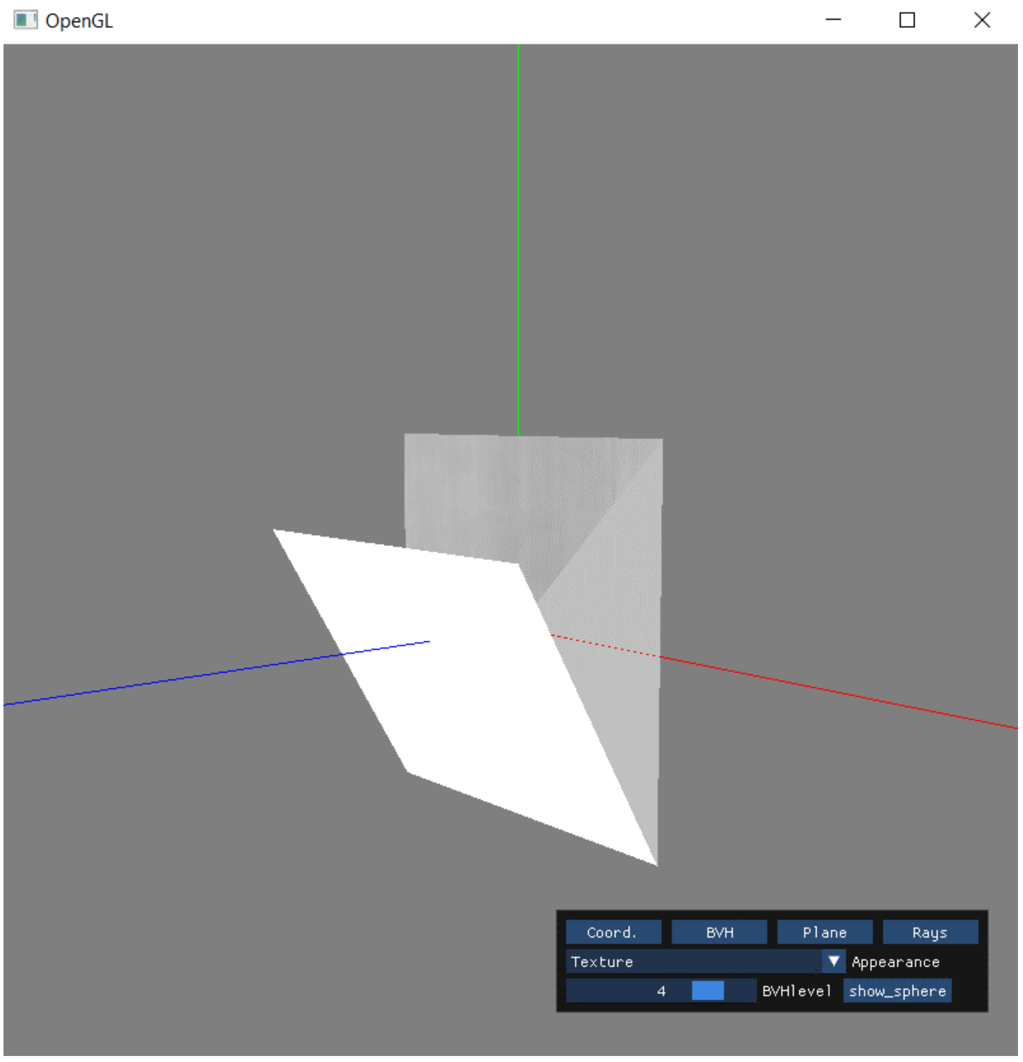


Figure 5. 1 Simulation environment for flat rectangular plate with the incident plane wave

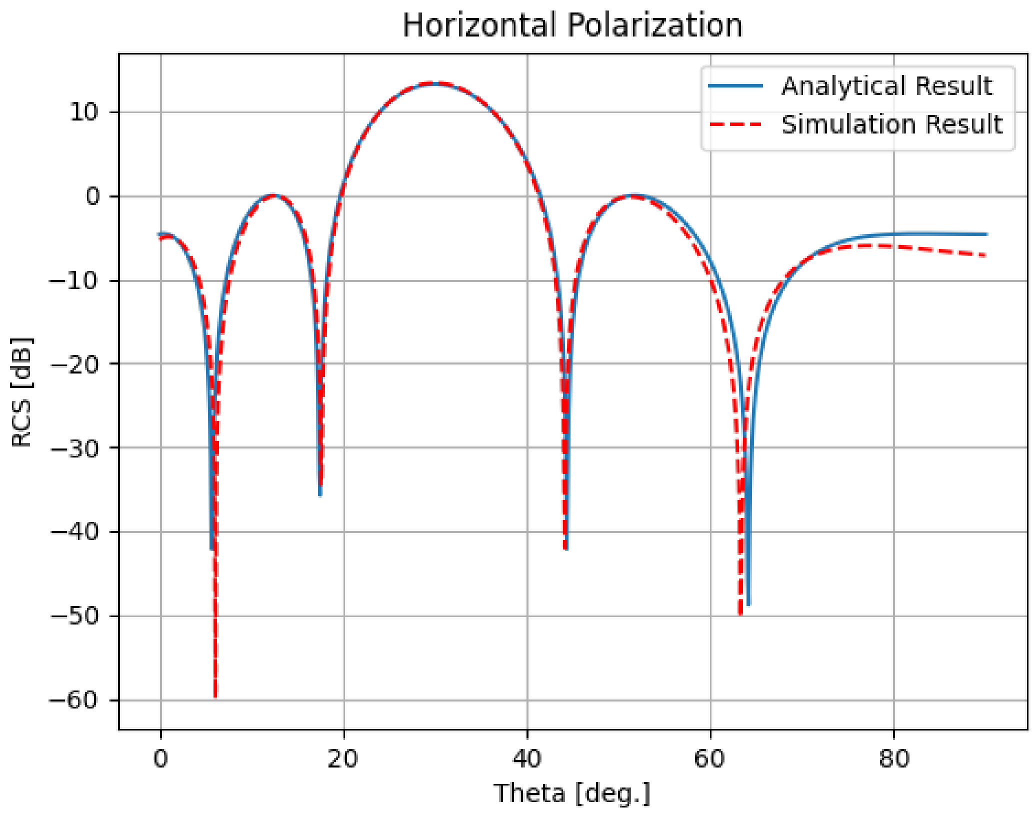


Figure 5. 2 Bistatic RCS of flat rectangular plate with horizontal polarization

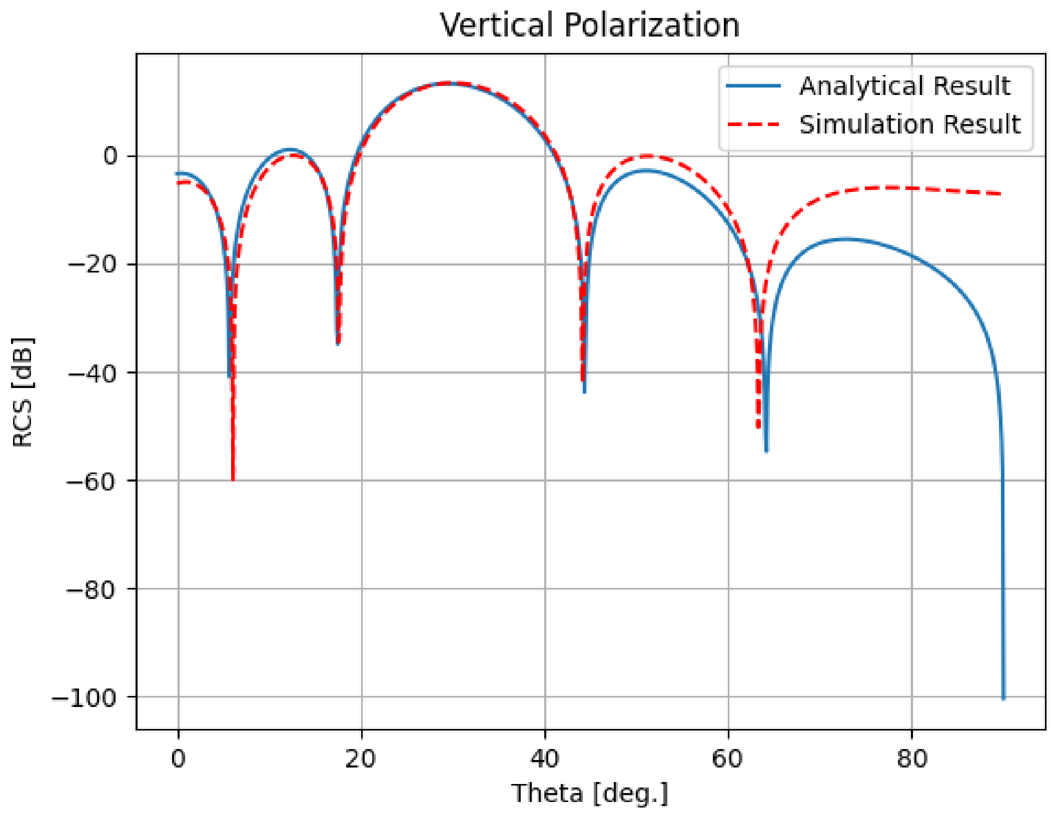


Figure 5. 3 Bistatic RCS of flat rectangular plate with vertical polarization

Then, in Section 3.2, the induced current on the surface of the object is derived when a ray hits the object. Figure 5.4 presents an HSV-coded visualization of an F15 model subjected to incident angles of $\theta_i = 45°$ and $\varphi_i = 90°$ at 5 GHz frequency with 6,551,055 rays. In this representation, the highest surface current density, measured at 150.731 $\frac{A}{m^2}$, is observed on the largest triangles.

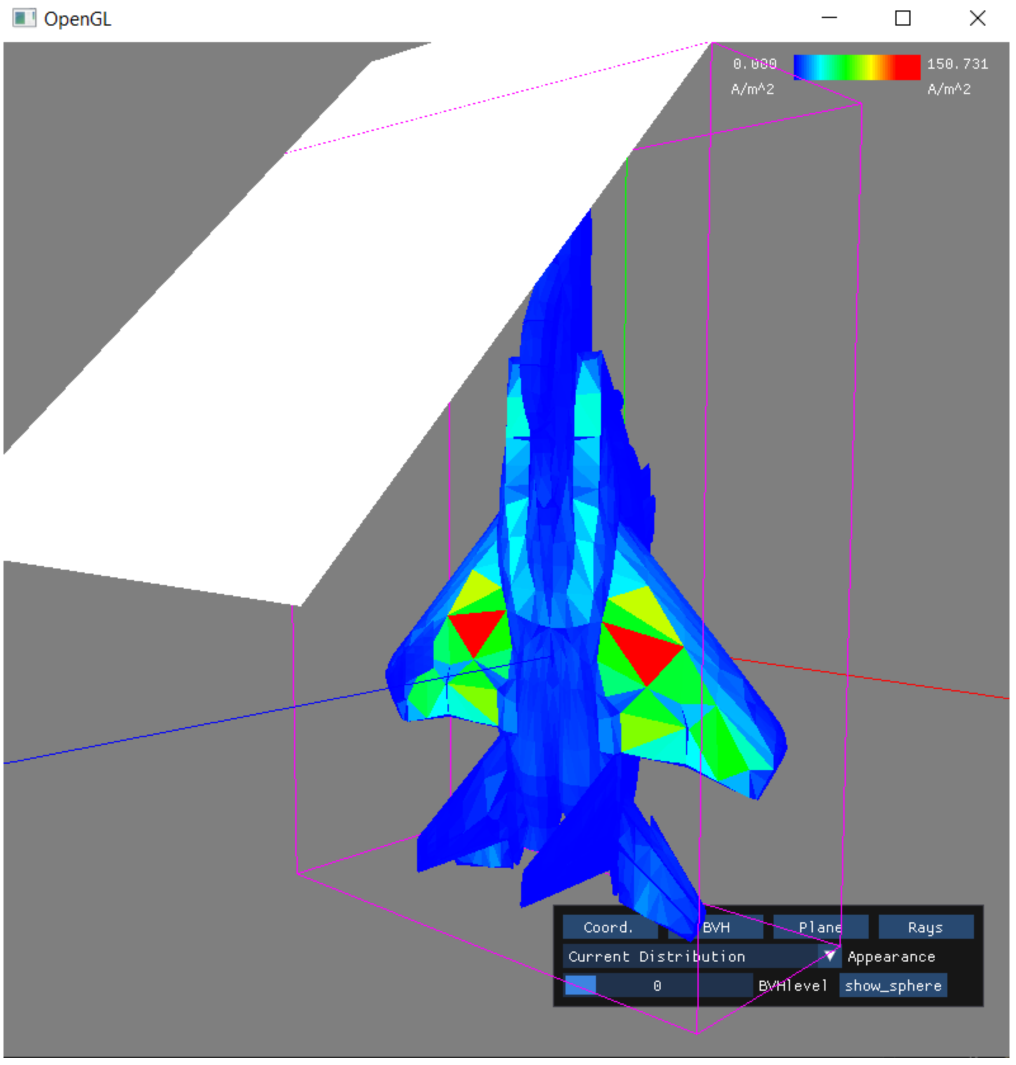


Figure 5. 4 HSV-coded visualization of surface current densities of triangles of an F15 model for $\theta_i = 45°$ and $\varphi_i = 90°$

In Section 3.2, we identified surface current densities for each triangle. we identified the surface current densities for each triangle. To further validate the simulation and examine the effect of shadowing, we can create a cylindrical object on a flat plate and mesh it with small triangles. This setup will allow us to analyze the shadowing affect. Figure 5.5 shows the setup.

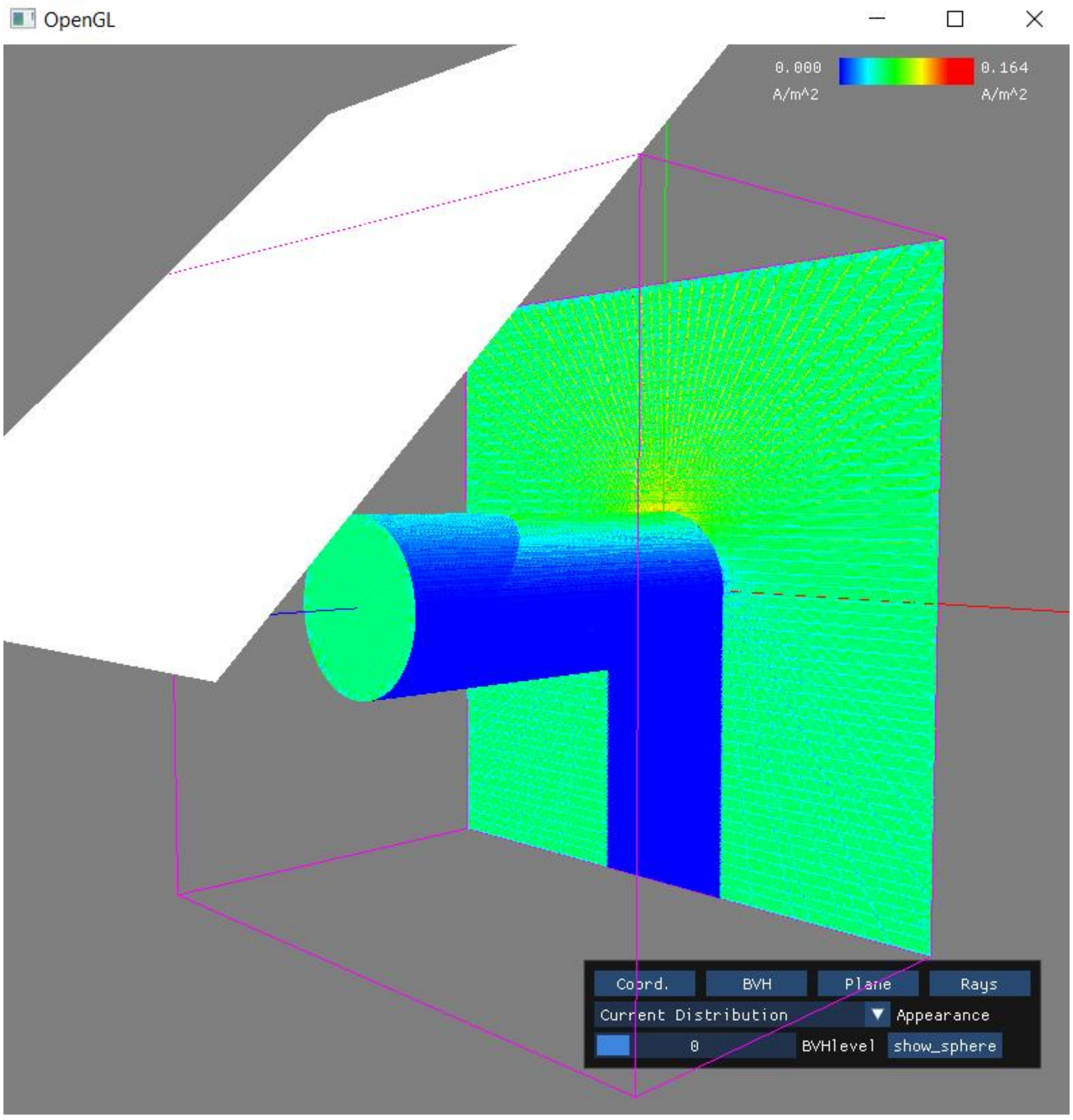


Figure 5. 5 Shadowing effect on the flat plate

The figure shows that there is no current induced on the triangles behind the cylinder on the flat plate due to the cylinder casting a shadow on that area. In front of the cylinder, lines are created as rays bounce off the cylinder and interact with the finely meshed triangles on the plate.

Notably, in some cases, some triangles can be defined significantly larger than others, leading to a greater surface area for potential interactions. Consequently, more rays can hit these larger triangles and larger current may induce on the surface of these triangles.

Section 3.1 provided details of the SBR method. In Figure 5.5, a timing comparison between the CPU and GPU implementations of the SBR method across a frequency ranging from 2 GHz to 512 GHz for the flat plate is shown. The performance analysis was conducted using an NVIDIA RTX 2060 GPU and an Intel i7-9750H CPU with 12 cores.

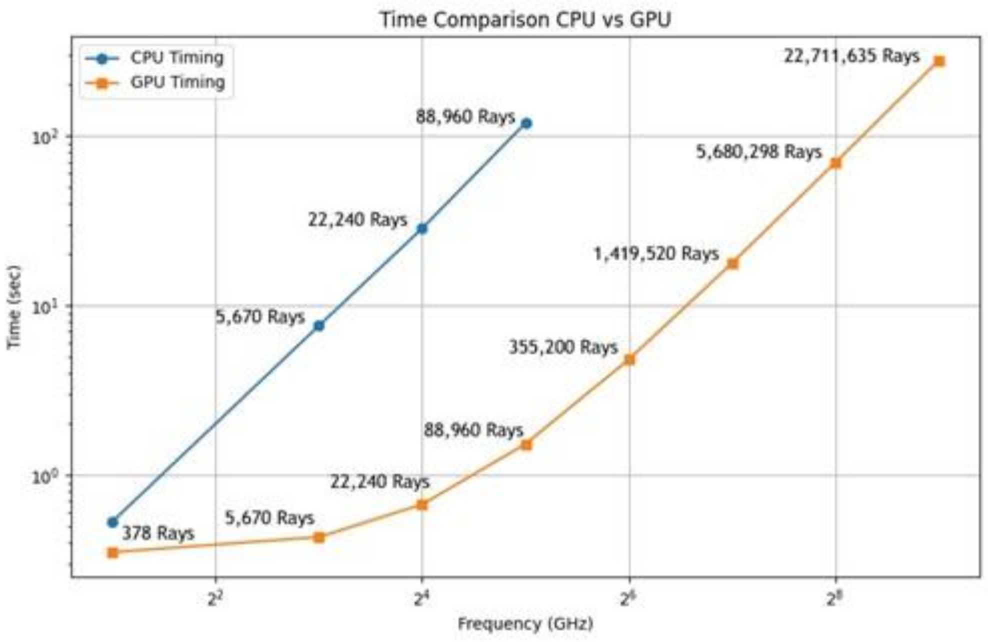


Figure 5. 6 Timing comparison of CPU and GPU implementations of the SBR method

It is evident from figure that the GPU implementation consistently outperforms the CPU implementation across all frequencies. Notably, the timing comparison reveals a logarithmic increase in computation time for the GPU implementation, which transitions to a linear trend as the ray size increases. In contrast, the CPU implementation exhibits a linear increase in timing, consistently lagging behind the GPU implementation in terms of computational speed. Finally, the CPU began to take significantly longer processing times after 88,960 rays. As a result, only GPU timings are displayed beyond this point. Table 5.1 provides the data used to construct Figure 5.5.

Table 5. 1 Data used to construct Figure 5.5

Frequency (GHz)	Ray Size	CPU Time (sec)	GPU Time (sec)
2	378	0.53	0.01
8	5,670	8.42	0.05
16	22,240	25.31	0.31
32	88,960	115.42	1.32
64	355,200	-	4.58
128	1,419,520	-	15.1
256	5,680,298	-	70.52
512	22,711,635	-	255.63

As mentioned before there are different GPU programming APIs can be used for parallelization. Similar to OpenCL, CUDA is a framework developed by Nvidia for programming graphics processing units. In the benchmark presented in Figure 5.7, an Nvidia RTX2060 graphics card was used to compare the performance of CUDA and OpenCL implementations for the F15 model at frequencies ranging from 2 GHz to 8 GHz.

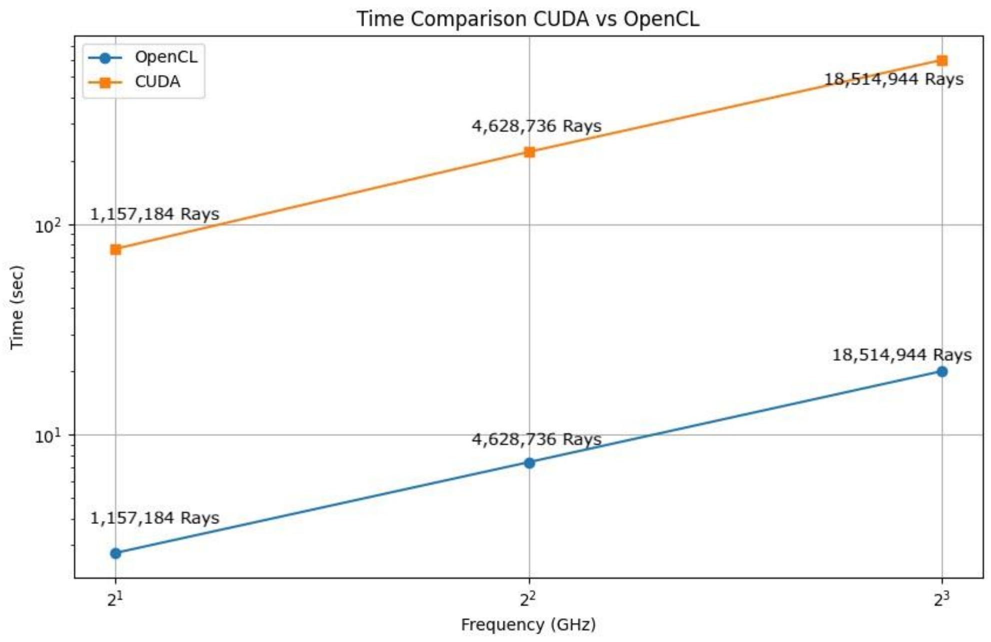


Figure 5. 7 Timing comparison of CUDA and OpenCL implementations of parallelized SBR method

Another metric to consider is the level of detail in object definition. When an object is represented with fewer, larger triangles, its accuracy may decrease compared to real-world proportions. Conversely, defining a model with numerous smaller triangles results in a representation closer to reality. However, increasing triangle count necessitates more ray-triangle tests, consequently extending computation time. Figure 5.8 illustrates the timing disparities observed in a cylinder model when defined with varying numbers of triangles and illuminated in front of it at X-band.

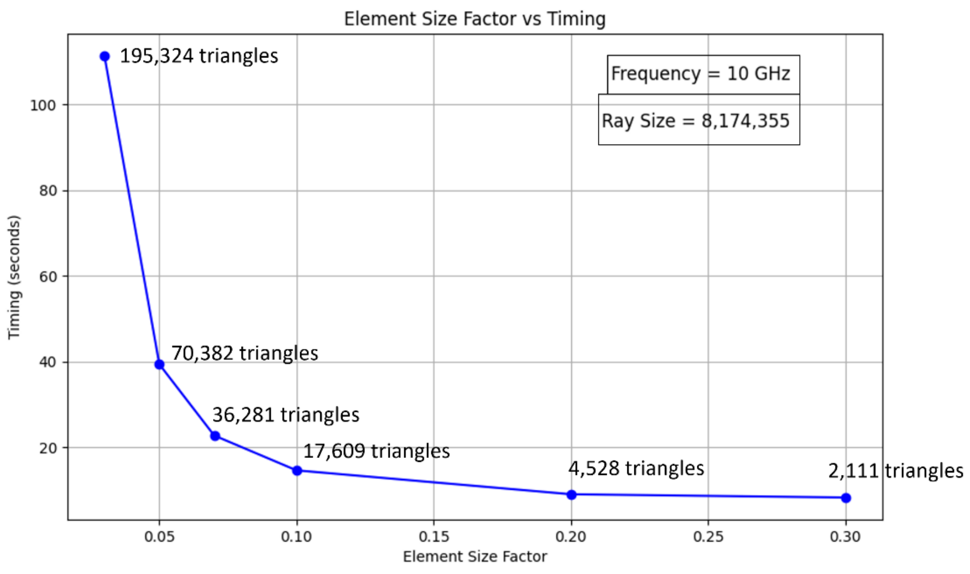


Figure 5. 8 Timing analysis with varying triangle size

The term "element size factor" in the context of Figure 5.8 refers to the relative size of the triangles used to define a 3D model. This factor is an indicator of the level of detail and accuracy of the model. When the element size factor is large, the model is represented using fewer, larger triangles. Conversely, a smaller element size factor indicates that the model is represented with many smaller triangles.

Since each ray is assigned to a GPU thread, in certain simulation scenarios the number of rays might exceed the available GPU threads or, due to each ray object having a large amount of data, there may not be enough GPU memory. In either case, we divide rays into chunks and trace each chunk in parallel. After the completion of each ray chunk, the RCS value is calculated for the desired look angle. Figure 5.9 illustrates the timing results for different chunk sizes when tracing approximately 10.5 million rays at 5 GHz.

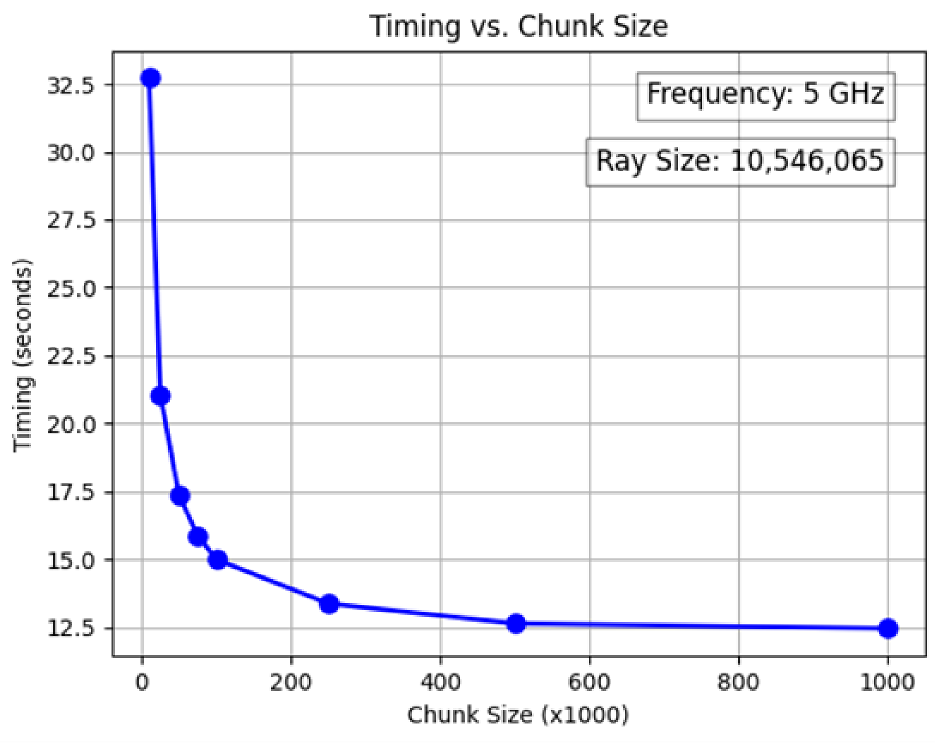


Figure 5. 9 Timing analysis with varying chunk size

6. CONCLUSION AND FUTURE WORK

6.1. Conclusion

In Section 3.1 a detailed description of the theory behind the simulation is given. Then, in section 4, the implementation and functions of the program are explained. The timing comparison in Figure 5.5 shows that the parallelized GPU-based implementation is significantly faster than the CPU-based implementation. However, the speed of both implementations highly depends on the CPU and GPU units available in the test machine. Stronger units would result in a lower derivation in both timings.

Then, in Figure 5.7, the timing comparison of CUDA and OpenCL frameworks are discussed. The specific graphics card model and its compatibility with each API implementation play significant roles in performance. One API may perform better with a particular GPU model, while the other may excel with a different model.

Lastly, it is important to keep in mind that since, ray tracing is performed once for the incident direction and only the ray tube integration is needed for every look angle, the bistatic formation results in time savings in all implementations. On the other hand, in monostatic formation, each look angle requires a reconstruction of the ray plane.

6.2. Future Work: Bounding Volume Hierarchy

Bounding Volume Hierarchy (BVH), Octrees, and K-D Trees are all spatial data structures used in computer graphics to organize geometric data and accelerate ray tracing algorithms. The most generalized version is the BVH, which begin with a bounding volume that covers the entire object. The volume is then repeatedly subdivided again and again until certain conditions are met. In this way, BVHs can accelerate intersection tests by quickly culling large portions of the scene that are unlikely to intersect with a given ray. Figure 6.1 shows an example of a BVH constructed using rectangles as bounding boxes. Here, each node in the hierarchy represents a portion of space enclosed by a bounding volume. These volumes can be constructed by using spheres or rectangles, each having advantages or disadvantages.

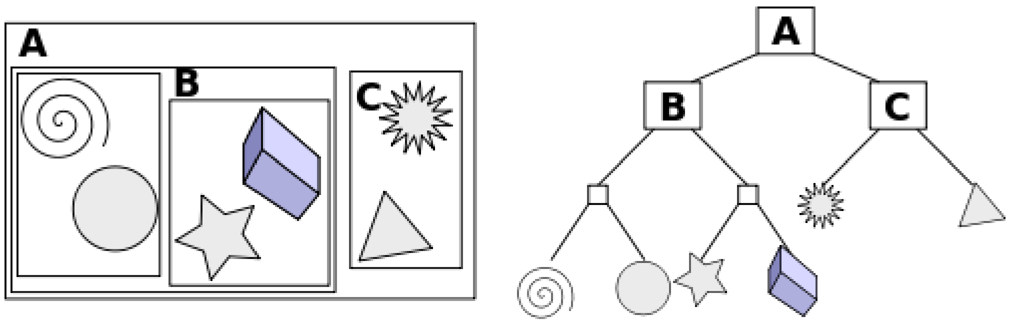


Figure 6. 1 An example of BVH using rectangles as bounding volumes

KD-Trees and octrees are both specialized form of BVHs. These structures also aim to efficiently partition the object for reducing ray-triangle checks and accelerating ray tracing algorithms. In octrees, each internal node has exactly eight children, corresponding to the eight equally-sized volumetric regions known as octants. This ensures there is no overlap between subdivisions, and each triangle is assigned to one subdivision. Whereas, in KD-trees, the volume is divided by a hyperplane perpendicular to one of the spatial dimensions (x, y, or z) at each level. This creates two child nodes, and the splitting plane alternates between dimensions as you traverse down the tree. Unlike octrees, KD-trees can have overlap between subdivisions, where a triangle might intersect the dividing hyperplane and belong to both child nodes.

In this thesis, a hybrid approach combining octree and KD-tree will be employed. The root of the tree is the MRBM, mentioned in Section 3.1.2. At each level of the tree, the volume will be partitioned into segments based on user-defined value along the longest axis. To determine which volume each triangle belongs to at each level, we will calculate the center of gravity of the triangles and check in which volume they are located using Equation 6.1.

$$\boldsymbol{Centroid}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \left(\frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3}, \frac{z_1+z_2+z_3}{3}\right) \quad (6.1)$$

After determining which triangle belongs to each volume, the volumes will be offset so that each volume tightly covers all its assigned triangles. These processes will result in overlapping volumes, but each triangle will be assigned to only one volume. Here, the complexity arises from having to run the Möller-Trumbore intersection algorithm up to 12 times per box, as each box has 6 faces, and each face can be represented by 2 triangles. This can be time-consuming for many scenes. To address this, we can use circumscribing spheres around the rectangular box and perform a single ray-sphere intersection test using a simple equation. This significantly reduces the number of intersection checks compared to iterating through each triangle of a box. The equation used for this test is given in Equation 6.2.

$$\|O + tD - C\|^2 = r^2 \tag{6.2}$$

where O is the origin point of the ray, D is the direction vector of the ray, t is a parameter representing the distance along the ray from its origin, C is the center of the sphere and r is the radius of the sphere. By solving this equation for t, we can determine if and where the ray intersects the sphere. If there are solutions for t, the ray intersects the sphere. If there are no real solutions, the ray does not intersect the sphere.

The bounding rectangular boxes for the F15 model, with a depth of four and the branching factor of two at each level, are illustrated in Figure 6.2. Additionally, Figure 6.3 depicts the bounding spheres corresponding to the same model.

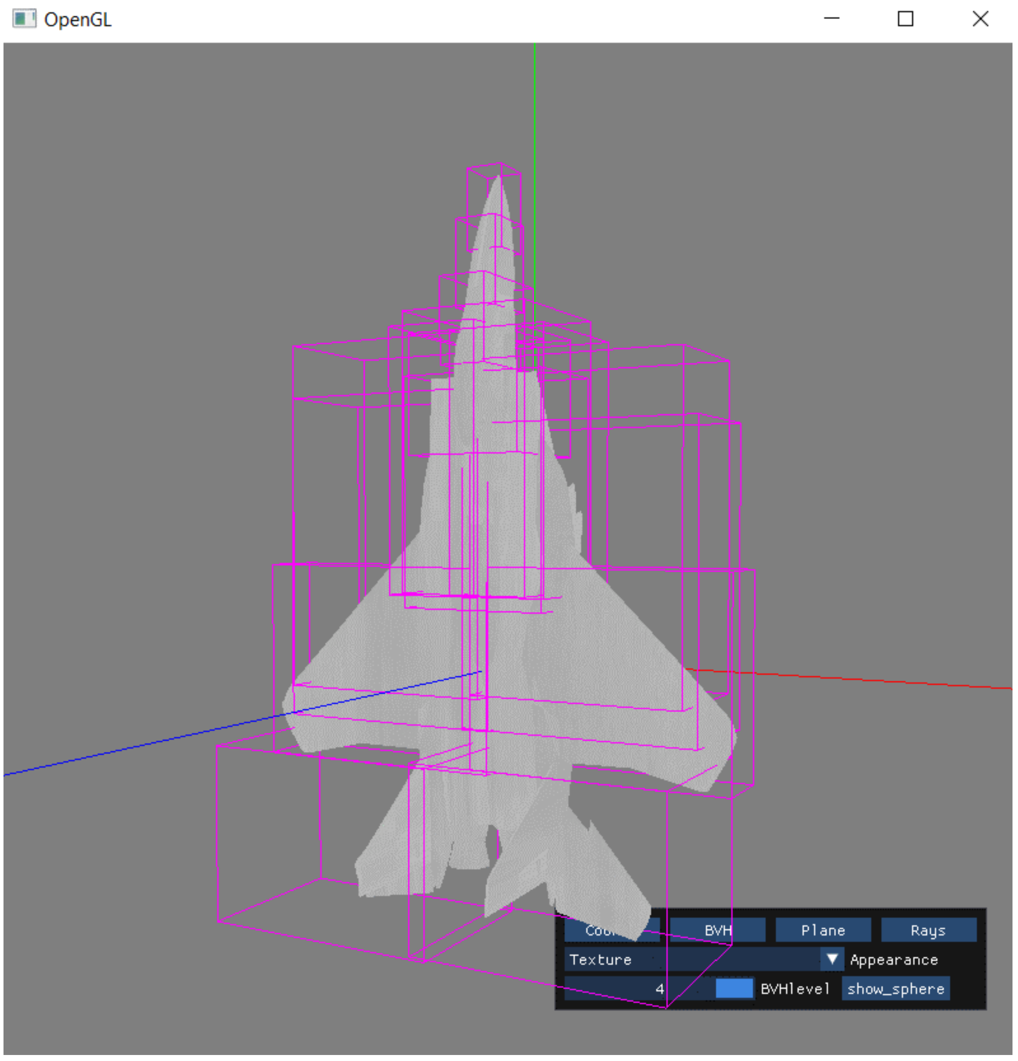


Figure 6. 2 Bounding rectangular boxes for F15 model

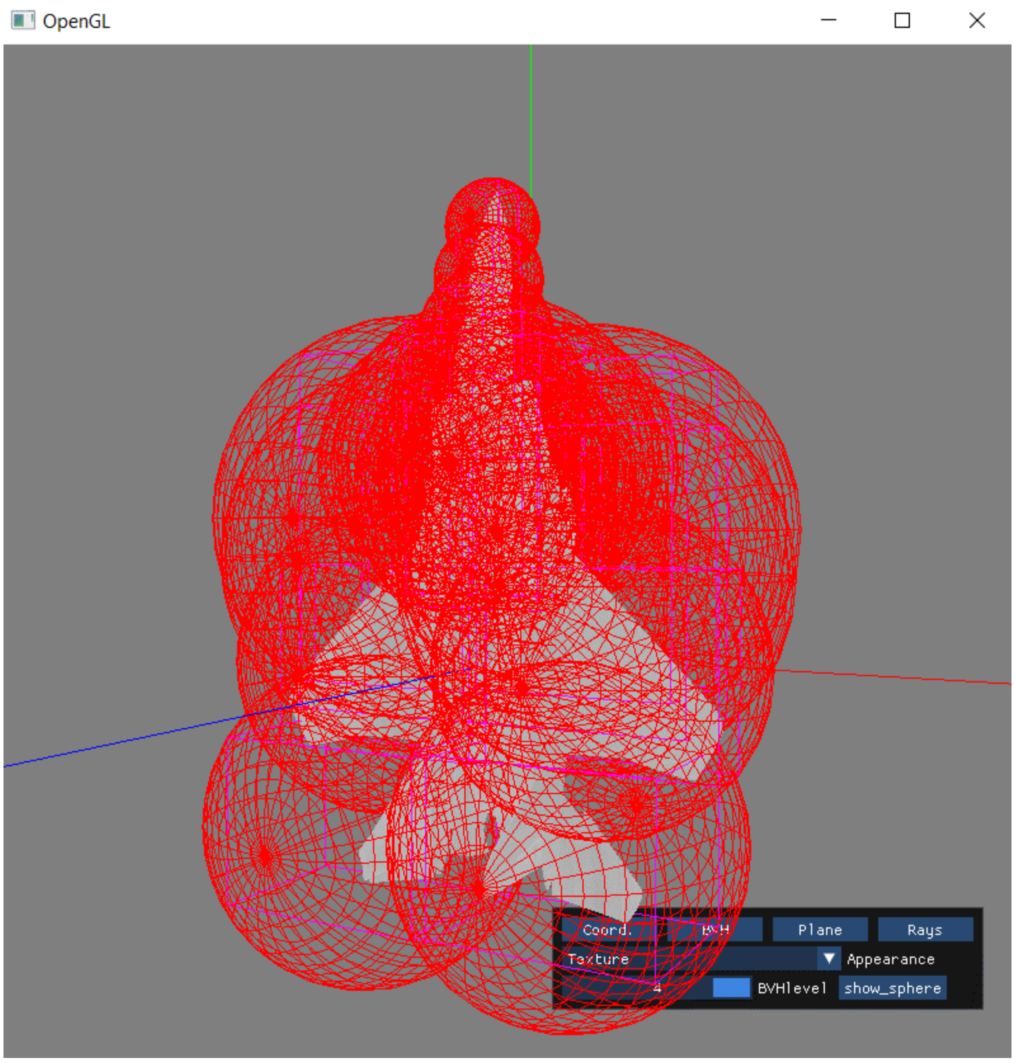


Figure 6. 3 Bounding spheres for F15 model

Finally, for efficient tree generation based on CAD model properties, reference [5] introduces a formula utilizing the radius of enclosing spheres. The radius of the sphere, r, can be used to determine the level, L, with which a triangle or rectangular box is associated, with the formula:

$$L = \left\lfloor \frac{\log(\frac{r_0}{r})}{\log 2} \right\rfloor \tag{6.3}$$

where r_0 is the radius of the sphere of MBRB. Then, the number of levels H for the CAD model can be determined as

$$H = \left\lfloor \frac{\log(\frac{r_0}{r_{patch}^{min}})}{\log 2} \right\rfloor \tag{6.4}$$

where r_{patch}^{min} is the radius of the smallest enclosing sphere.

Bibliography

[1] H. Lee, R.C. Chou, and S.W. Lee, Shooting and bouncing rays: Calculating the RCS of an arbitrarily shaped cavity, IEEE Trans Antennas Propag 37 (1989), 194–205.

[2] "IEEE Standard Dictionary of Electrical and Electronics Terms," in IEEE Transactions on Power Apparatus and Systems, vol. PAS-99, no. 6, pp. 37a-37a, Nov. 1980.

[3] "IEEE Standard Letter Designations for Radar-Frequency Bands," in *IEEE Std 521-2019 (Revision of IEEE Std 521-2002)* , vol., no., pp.1-15, 14 Feb. 2020.

[4] S.W. Lee, H. Ling, and R. Chou, Ray-tube integration in shooting and bouncing ray method, Microwave Opt Technol Lett 1 (1988), 286–289.

[5] Dikmen, F., Ergin, A.A., Sevgili, A.L. and Terzi, B. (2010), Implementation of an efficient shooting and bouncing rays scheme. Microw. Opt. Technol. Lett., 52: 2409-2413.

[6] Y. Tao, H. Lin and H. Bao, "GPU-Based Shooting and Bouncing Ray Method for Fast RCS Prediction," in IEEE Transactions on Antennas and Propagation, vol. 58, no. 2, pp. 494-502, Feb. 2010.

[7] C. Y. Kee, C. -F. Wang and T. T. Chia, "Optimizing high-frequency PO-SBR on GPU for multiple frequencies," 2015 IEEE 4th Asia-Pacific Conference on Antennas and Propagation (APCAP), Bali, Indonesia, 2015, pp. 132-133.

[8] P. Gao, X. Wang, Z. Liang and W. Gao, "Efficient GPU implementation of SBR for fast computation of composite scattering from electrically large target over a randomly rough surface," 2015 IEEE International Symposium on Antennas and Propagation & USNC/URSI National Radio Science Meeting, Vancouver, BC, Canada, 2015, pp. 1666-1667.

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

[10] B. Bural, O. Ozgun, A. E. Yilmaz and M. Kuzuoglu, "GPU-Accelerated Shooting and Bouncing Ray Method for Inverse Synthetic Aperture Radar Imaging," 2022 32nd International Conference Radioelektronika (RADIOELEKTRONIKA), Kosice, Slovakia, 2022, pp. 01-04.

[11] Möller, Tomas; Trumbore, Ben (1997). "Fast, Minimum Storage Ray-Triangle Intersection". Journal of Graphics Tools. 2: 21–28

[12] C. A. Balanis, "Advanced Engineering Electromagnetics, Second Edition," Wiley-IEEE Press, 2012.