Immersive Visualization of Satellite Tracking at ESA's Cebreros Ground Station Using SPICE Kernels and Virtual Reality

Sofia Cunha Avelino
Master in Computer Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Immersive Visualization of Satellite Tracking at ESA's Cebreros Ground Station Using SPICE Kernels and Virtual Reality

Sofia Cunha Avelino

Dissertation carried out as part of the Master in Computer Science

Department of Computer Science

2025

Supervisor
Prof. Luís Lopes, Associate Professor, Faculty of Sciences, University of Porto

Acknowledgements

First and foremost, I would like to thank my supervisor, Luís Lopes, for his invaluable guidance, continuous support, and readiness to help throughout the development of this thesis. His willingness to assist and his insightful expertise were deeply appreciated during this process.

I would also like to thank my co-supervisor, Jorge Fauste, for placing his trust in me for the development of this work, and for the guidance and support he provided throughout my internship at the European Space Agency. I am truly grateful for this opportunity and hope to have met his expectations with this work.

To my family, I could not have achieved all that I have without your unwavering support in every area of my life. I thank you dearly for always believing in me and standing by me in everything I do.

To my friends at ESA, thank you for making my time there so incredible and for always offering your opinions and assistance with my work whenever I needed it.

Finally, last but by no means least, I would like to thank my partner, Paulo Ferreira, who has been my pillar not only throughout this process but in life. You give me the strength to keep going when things become hard or frustrating, and I could never thank you enough for all the love and support you give me.

This thesis is not only a reflection of my hard work, but also a testament to all those who have stood by me throughout my academic journey and supported me along the way. To all of these people, including those not mentioned here, I am deeply grateful.

Thank you.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado em Ciência de Computadores

Visualização Imersiva do Acompanhamento de Satélites na Estação de Cebreros da ESA Utilizando SPICE Kernels e Realidade Virtual

por Sofia AVELINO

Esta dissertação apresenta o desenvolvimento de ferramentas de visualização interativa concebidas para reforçar o envolvimento do público com as missões espaciais acompanhadas pela antena DSA-2 da ESA na Estação de Cebreros. O projeto focou-se em dois objetivos principais: fornecer informação precisa e atualizada, acessível a visitantes independentemente da sua formação técnica, e enriquecer a experiência do visitante através de exibições interativas inovadoras, incluindo ambientes de realidade virtual imersiva.

Para alcançar estes objetivos, foram implementadas plataformas de visualização utilizando a plataforma Cesium e a aplicação Cosmographia, complementadas por modelos 3D criados tanto para aplicações digitais como físicas. Estas ferramentas encontram-se atualmente em utilização operacional em Cebreros, onde apoiam visitas guiadas e iniciativas de divulgação. Além disso, os modelos foram integrados em outras atividades da ESA, incluindo um vídeo comemorativo do 50.º aniversário da agência.

As soluções desenvolvidas demonstram o potencial de técnicas de visualização de baixo custo e adaptáveis para apoiar a comunicação institucional e as atividades de divulgação pública. Ao traduzir o acompanhamento de missões espaciais em experiências acessíveis e envolventes, este trabalho contribui para reduzir a distância entre a ciência espacial especializada e o público em geral, estimulando a curiosidade e inspirando futuras gerações a seguir carreiras na ciência e na exploração espacial.

A dissertação conclui com recomendações para trabalho futuro, incluindo a otimização do desempenho, o aumento da interatividade no ambiente de realidade virtual e a inclusão de informação contextual mais rica e de modelos de infraestruturas. Estas melhorias aumentariam ainda mais o valor educativo e de divulgação das visualizações e

apoiariam a sua adaptação a outras estações de solo da ESA e a diferentes contextos de comunicação.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Computer Science

Immersive Visualization of Satellite Tracking at ESA's Cebreros Ground Station Using SPICE Kernels and Virtual Reality

by Sofia AVELINO

This thesis presents the development of interactive visualization tools designed to enhance public engagement with the space missions tracked by ESA's DSA-2 antenna at Cebreros Station. The project pursued two main objectives: to provide accurate, up-to-date information accessible to visitors regardless of technical background, and to enrich the visitor experience through innovative interactive displays, including immersive virtual reality environments.

To achieve these goals, visualization platforms were implemented using Cesium and Cosmographia, complemented by 3D models created for both digital and physical applications. These tools are currently in operational use at Cebreros, where they support guided tours and outreach initiatives. In addition, the models have been integrated into broader ESA activities, including a commemorative video for ESA's 50th anniversary.

The solutions developed demonstrate the potential of low-cost, adaptable visualization techniques to support institutional communication and public outreach. By translating complex spacecraft tracking into accessible, engaging experiences, this work helps bridge the gap between specialized space science and the general public, fostering curiosity and inspiring future generations to pursue careers in science and exploration.

The thesis concludes with recommendations for future work, including performance optimization, enhanced interactivity in the VR environment, and the inclusion of richer contextual information and infrastructure models. These improvements would further increase the educational and outreach value of the visualizations and support their adaptation to other ESA ground stations and communication contexts.

Contents

List of Figures

Glossary

AOS	Aquisition of Signal
AR	Augmented Reality
ASI	Italian Space Agency
DLR	German Aerospace Center
DSA-2	Deep Space Antenna 2
ECEF	Earth-Centered, Earth-Fixed
ESA	European Space Agency
ESAC	European Space Astronomy Centre
ESOC	European Space Operations Centre
ESTRACK	ESA Tracking Stations
ET	Ephemeris Time
FEC	Front End Controller
INTA	National Institute for Aerospace Technology
KSAT	Kongsberg Satellite Services
LOS	Loss of Signal
MER	Main Equipment Room
NASA	National Aeronautics and Space Administration
NNO	New Norcia Station
OPS	Operations
SSC	Swedish Space Corporation
STC	Station Computer

Chapter 1

Introduction

1.1 Motivation

In an era of rapidly expanding technological advancement and increasing interconnection of scientific fields, it has become crucial to inspire and prepare the next generation of scientists for the opportunities and challenges that lie ahead. As science and technology evolve at extraordinary rates, it becomes critical to captivate young minds with the possibilities of innovation and foster their curiosity, creativity, and drive to contribute meaningfully to society. Studies have shown that exposing students and aspiring scientists to science beyond the traditional classroom environment, through visits to museums, research centers, and institutions like space agencies, has a profound and lasting impact on their academic trajectories and career aspirations [1, 2]. These experiences provide an opportunity to witness firsthand the applications of science and the ways in which it shapes our understanding of the world, kindling a passion for exploration and discovery that goes beyond what can be obtained in a classroom.

Beyond sparking individual interest, such initiatives are essential in helping to build skilled professionals in science, technology, engineering, and mathematics (STEM) fields. They help students envision their potential role in driving technological innovation and contributing to solving some of humanity's greatest challenges, from climate change to space exploration. As a result, encouraging engagement with science at an early stage and in interactive, impactful ways is a crucial investment in the future of scientific progress. This is why many scientific institutions continue to prioritize outreach efforts. For instance, in 2016, NASA launched the Science Activation (SciAct) program to connect NASA Science with people of all ages through active, engaging learning experiences about our

planet and the universe. In 2023 alone, NASA allocated approximately $50 million to support this initiative [3].

1.2 Goals

This motivation has driven the work undertaken in this thesis, which focuses on the development and enhancement of interactive visualization software designed to showcase the missions tracked by ESA's DSA-2 antenna at the Cebreros station, located near Madrid, Spain. By leveraging these tools, the project seeks to create meaningful connections between visitors and the complex missions being carried out at the station, showcasing some of the work developed in space exploration in an attempt to expand humanity's frontiers.

The goal of this project is twofold: first, to ensure that the information presented to visitors is accurate, consistently up-to-date, and accessible to audiences of diverse technical backgrounds and ages. Second, to create new interactive displays that offer a richer and more engaging experience. This includes incorporating modern techniques, such as virtual reality scenes, to dynamically and immersively visualize the antenna and spacecraft. These innovative approaches aim to capture visitors' interest, inspire their imagination, and leave a lasting impression of the immense potential of science and technology.

Ultimately, this project not only seeks to enhance visitors' understanding of the missions tracked by ESA's DSA-2 antenna but also aligns with the broader goal of using public outreach to spark enthusiasm for space science and its interdisciplinary connections. By bridging computer science, space science, and public engagement, the work undertaken in this thesis exemplifies how these disciplines can converge to inspire future generations and highlight the boundless opportunities within STEM fields.

1.3 Contribution

In this work, I contributed to the development and enhancement of multiple visualization systems aimed at improving the real-time monitoring and understanding of spacecraft tracking by the Cebreros deep-space antenna. These systems were mainly designed for public outreach and include web-based 3D displays, virtual reality experiences, and desktop applications. My role involved extending existing tools and creating new components to enable more accurate, automated, and dynamic visualizations of antenna activity and spacecraft motion.

The work resulted in significant architectural improvements across several platforms, including increased automation, expanded compatibility with real-time data, and the integration of realistic 3D models. Several of these visualization systems are now actively used at the Cebreros station to help visitors better understand how ESA antennas track spacecraft in deep space. Overall, my contributions helped transform a collection of static or manual tools into a unified and interactive suite that offers a clearer and more engaging view of ongoing space operations.

1.4 Organization

This thesis is structured to guide the reader through the development, integration, and enhancement of visualization tools for the Cebreros Deep Space Antenna (DSA-2) and spacecraft tracking. The remainder of this document is organized as follows:

- **Chapter 2: Background** provides the technical context and foundational components relevant to the project. It is divided into 4 main sections:
  - **Section 2.1: ESA Ground Stations and Tracking Network** presents an overview of ESA's ground station network (ESTRACK), with a focus on the Cebreros station;
  - **Section 2.2: SPICE Kernels** introduces the concept of SPICE kernels and its key components and applications;
  - **Section 2.3: Software Components** describes the software components used in the development of this work and their main features;
  - **Section 2.5: Previous Work** describes the preliminary system that was set up and running at the Cebreros station prior to this thesis.
- **Chapter 3: Method** presents the core technical contributions and is divided into 4 main sections:
  - **Section 3.1: Development of 3D Models of the Cebreros DSA-2 and New Norcia NNO-3 Antennas** describes the development of two detailed 3D model of the Cebreros DSA-2 and the New Norcia NNO-3 antennas, created from scratch using Blender, which serve as the foundation for improved visualization accuracy;

  - **Section 3.2: Antenna Model Integration in Cesium** focuses on the integration of the antenna models into the Cesium platform, replacing a previous generic model and enhancing the visual representation of the station;
  - **Section 3.3: Leveraging SPICE Kernels for Real-Time Spacecraft Visualization** details the use of SPICE kernels to enable real-time spacecraft visualization. This includes methods to obtain precise positional and orientation data necessary for dynamic tracking and display;
  - **Section 3.4: A Virtual Reality Cebreros View** discusses the adaptation and enhancement of the Cebreros antenna and spacecraft views for a virtual reality (VR) environment, enabling immersive real-time interaction and visualization.

- **Chapters 4–6** detail the technical implementation of the contributions outlined in Chapter 3, presenting the resulting improvements in the visualization tools and user experience as well as discussions of challenges encountered and how they were addressed:
  - **Chapter 4: Cesium Display and Antenna Models** focuses on the creation of 3D models of the Cebreros and New Norcia 3 antennas and their integration in the station's Cesium-based visitor display;
  - **Chapter 5: Cosmographia and Trajectory Display** details the improvements made to the station's Cosmographia Display;
  - **Chapter 6: A VR Cebreros View** describes the development of a Virtual Reality application featuring views of the Cebreros Station, the Solar System, and the spacecraft currently tracked at Cebreros.

- **Chapter 7: Conclusions and Future Work** provides an overview of the challenges encountered and the accomplishments achieved throughout this thesis, and explores potential directions for future development and improvement.

Chapter 2

Background

To enhance visitors' understanding and engagement with the work carried out at Cebreros and provide a more enriching experience, it is crucial to gain a deeper insight into the purpose and functionalities of ESA's ground stations, particularly focusing on the role of the DSA-2 deep space antenna. This involves understanding their operations, exploring what they do and how they function, and becoming acquainted with the existing software suite and tools in use.

This chapter provides a brief overview of ESA's tracking station network, the Cebreros control room where the original software suite for visitors was established (Section 2.1), and the tools and software utilized in enhancing this suite (Sections 2.2 and 2.3, respectively).

2.1 ESA Ground Stations and Tracking Network

2.1.1 ESTRACK

ESTRACK, ESA's tracking station network, is the backbone of the agency's space communication infrastructure. Since its establishment in 1975, ESTRACK has evolved into a comprehensive global system of ground stations designed to provide world-wide spacelink connectivity for a wide range of missions. These include Low Earth Orbit (LEO) missions, Deep Space (DS) missions, and Near-Earth (NE) missions such as Geostationary Transfer Orbits (GTO), Medium Earth Orbits (MEO), Geostationary Orbits (GEO), Highly Eccentric Orbits, Lunar Orbits, and Lagrangian Orbits.

The ESTRACK system is composed of three nested networks of ground stations, as illustrated in Figure 2.1. The core network comprises seven antennas across six stations,

including three Deep Space Stations located in Cebreros (Spain), Malargüe (Argentina), and New Norcia (Australia). The augmented network includes several non-ESA stations that complement the ESTRACK core infrastructure. These include stations operated by Kongsberg Satellite Services AS (KSAT, Norway) and Swedish Space Corporation (SSC) in both the northern and southern hemispheres, as well as equatorial stations managed by the Italian Space Agency (ASI), German Aerospace Center (DLR), and Spain's National Institute of Aerospace Technology (INTA). In addition, a cooperative network has been established to support collaboration with other space agencies, such as NASA [4].

FIGURE 2.1: ESTRACK Network Map

The primary roles of ESA's ground tracking stations are [5]:

- Maintaining communication with spacecraft by transmitting commands and receiving scientific data as well as spacecraft status updates;
- Collecting radiometric data to enable mission controllers to determine the spacecraft's position, trajectory, and velocity;
- Providing support throughout all mission phases, including:
  - Launch and Early Orbit Phase (*LEOP*) , a critical stage of the mission;
  - Commissioning and routine operations;

- Deorbiting and ensuring the spacecraft's safe disposal.

This extensive network delivers over 15,000 hours of tracking support annually to more than 20 missions, maintaining a service availability rate exceeding 99% [6].

2.1.2 Cebreros Station

The Cebreros station is home to the DSA-2 antenna, a 35-meter deep-space dish that operates in the X-band for transmission and in the X-, K-, and Ka-bands for reception. With a transmission power of 20 kW, DSA-2 is capable of sending commands to spacecraft millions of kilometers away while simultaneously receiving scientific data and telemetry [7]. Its advanced design enables support for multiple missions, including high-profile endeavors such as Gaia [8], BepiColombo [9], and Mars Express [10].

While the antenna is typically operated remotely from the European Space Operations Centre (ESOC), its entire system can be locally monitored and controlled from the Main Equipment Room (MER). Consoles in the MER provide access to the STC and FEC monitoring and control systems.

To provide additional context about the missions, including the spacecraft being tracked, orbital positions, and distances, three large screens were installed in the MER to display this mission-specific information (Figure 2.2). One screen (right) presents a 3D model of the spacecraft currently being tracked by the station, another (center) displays its orbital and trajectory view, and a third (left) offers a realistic 3D visualization of the station area, which initially included a placeholder model of a generic NASA antenna to represent the DSA-2 antenna.

The MER is often included in station tours due to its impressive setup, leaving a strong impression on visitors.

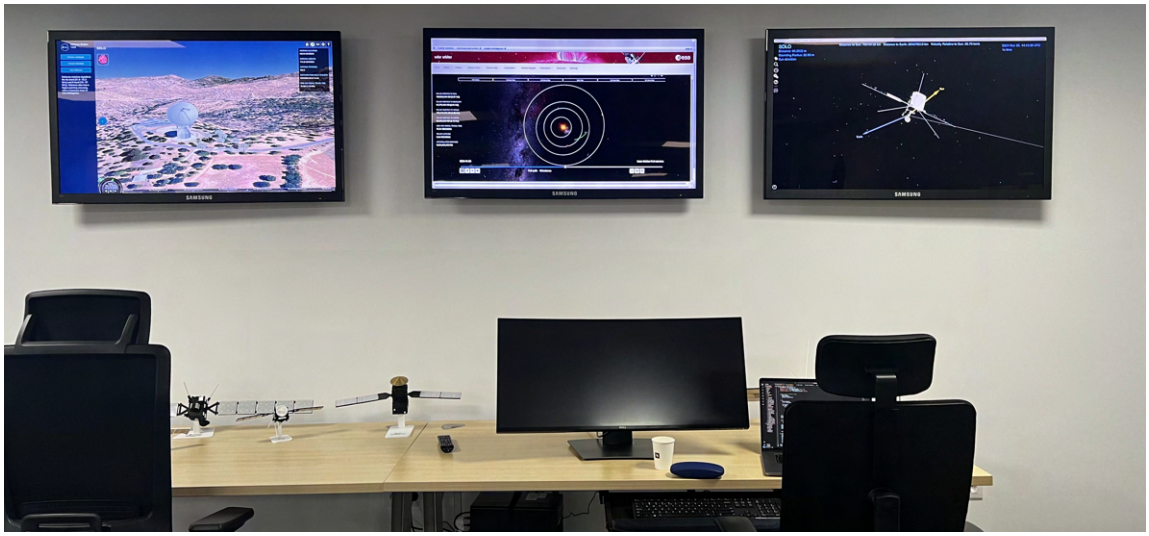


FIGURE 2.2: MER Visitor Display

2.2 SPICE kernels

SPICE *(Spacecraft, Planet, Instrument, C-matrix, Events)* is an information system developed by NASA's Navigation and Ancillary Information Facility (NAIF) to support space mission planning, analysis, and operations. It provides a robust suite of tools and data files that enable the calculation of relevant scientific information such as spacecraft positions, orientations, and the trajectories of planets and other celestial bodies [11]. These parameters are crucial for mission planning, data analysis, and operational support, facilitating accurate navigation and ensuring precise targeting during space missions.

2.2.1 Key Components of SPICE

SPICE is made up of several key components, namely [12]:

- **Kernels:** These are the primary data files used within the SPICE system. They contain crucial mission-specific information such as spacecraft position and velocity, planet and moon ephemerides, and spacecraft orientation (attitude). There are different types of kernels, including:
  - **SPK (Spacecraft and Planet Kernel):** Contains data about the positions and velocities of spacecraft and planetary bodies;
  - **CK (Coordination Kernel):** Provides the spacecraft's orientation or attitude data;

  - **PCK (Planetary Constants Kernel):** Contains information on planetary body characteristics, such as radii and gravitational parameters;
  - **FK (Frame Kernel):** Defines the coordinate systems used for the position and orientation data;
  - **EK (Event Kernel):** Used to store event-related data for missions, such as specific events or milestones during the spacecraft's journey.

- **Toolkit:** The toolkit includes software libraries and tools that allow mission planners and engineers to read, write, and analyze the data contained within the kernels [13]. The toolkit provides functions for tasks such as converting between different coordinate systems, calculating distances between celestial bodies, and determining the orientation of spacecraft in space. In this work, I specifically use the `SpiceyPy` module, a Python wrapper for the SPICE toolkit, to perform these calculations within Python.

2.2.2 Applications of SPICE

The SPICE system is an essential tool in both scientific and operational sectors of space exploration. Its applications include:

- **Trajectory Analysis and Navigation:** SPICE kernels provide the data necessary to track spacecraft positions and velocities in real-time, enabling precise navigation and maneuver planning for space missions;
- **Spacecraft Orientation:** The system's ability to calculate and provide spacecraft orientation (attitude) data ensures that instruments onboard the spacecraft, such as cameras or scientific instruments, are properly aligned and able to gather accurate measurements;
- **Mission Design and Simulation:** SPICE data supports mission planners in designing and simulating the spacecraft's trajectory and interactions with celestial bodies. By using accurate ephemerides, mission planners can predict spacecraft encounters with planets, moons, and other objects, ensuring mission success;

- **Scientific Research:** SPICE is also used in post-mission analysis for scientific research. By reconstructing the spacecraft's path and orientation throughout the mission, scientists can interpret scientific data in relation to specific events, such as flybys or planetary observations.

2.3 Software Components

2.3.1 Blender 3D

Blender 3D is a powerful open-source software suite for 3D modeling, animation, rendering, and simulation [14]. It is commonly used in various industries for purposes ranging from game development to scientific visualization. In the context of this work, Blender is a crucial tool for creating realistic 3D models of spacecraft and station components, such as the DSA-2 antenna.

Blender provides the following key features:

- **Modeling:** Blender's wide set of modeling tools allows for the creation of highly detailed and articulated 3D models, suitable for use in games, simulations, and various other applications;
- **Animation and Rigging:** Blender provides a robust rigging system that allows users to create skeletal structures and deformable rigs for characters and objects. This provides a foundation for producing fluid and dynamic animations, seamlessly integrated with the software's advanced animation tools;
- **Exporting Models:** Blender allows for model exporting in a range of different formats compatible with other visualization platforms, such as glTF [15] or obj [16], which is essential for integration into systems like Cosmographia or Unity.

2.3.2 Cosmographia

Cosmographia is a 3D visualization software suite designed to represent the solar system and its celestial bodies. It was later enhanced for real-time mission simulation and accurate visualization of spacecraft along with their paths, orbits, and other related data. It is a valuable tool for both scientific and engineering analysis, as well as public outreach [17].

Cosmographia provides the following key features:

- **Real-Time Visualization:** Cosmographia allows for the real-time visualization of spacecraft trajectories, orbital positions, and mission data;
- **Integration with SPICE:** Cosmographia is tightly integrated with the SPICE system, allowing for precise plotting of spacecraft positions, velocities, and orientations based on SPICE kernels;
- **Multi-Dimensional Data:** It supports the visualization of various data layers, including mission-specific parameters like spacecraft attitude, distance to celestial bodies, and mission milestones;
- **Customizable Visual Displays:** The software offers various options for displaying mission data, including 3D models, trajectory lines, and dynamic annotations that help interpret complex data.

2.3.3 Cesium

Cesium is an open-source geospatial visualization platform used for creating 3D globes and maps. It allows users to visualize space-related data in real-time and has applications across various industries, including space exploration. Cesium can also be integrated with other tools, such as Unity and Unreal [18], for use in diverse projects [19].

Cesium provides the following key features:

- **Geospatial Accuracy:** Having been created initially for use in aerospace missions, Cesium is known for its high precision and support for geospatial data. This tool now offers elevated accuracy for geospacial ecosystem building accross land, sky, space, and sea;
- **Data Integration:** It supports the integration of a wide range of data formats, including SPICE kernels, allowing for seamless import and visualization of mission parameters;
- **Web-Based Visualization:** Cesium offers the advantage of being accessible in a web browser, enabling remote access to mission visualizations and data.

2.3.4 Unity

Unity is a leading real-time development platform used for creating 3D interactive applications and games. It is widely used in both the gaming industry and for creating immersive simulations, which makes it well-suited for the creation of interactive visualizations of space missions [20].

Unity provides the following key features:

- **Real-Time Rendering:** Unity provides powerful real-time rendering capabilities, allowing the creation of high-quality graphics and visualizations that can be rendered instantly during the simulation, making it ideal for dynamic and interactive environments;
- **Interactivity:** Unity supports the development of interactive experiences by allowing users to manipulate objects, control cameras, and modify scenes in real-time, enabling a fully engaging and user-controlled simulation experience;
- **Multi-Platform Support:** Unity offers cross-platform compatibility, meaning that projects can be deployed across multiple devices and platforms such as PC, consoles, mobile devices, VR/AR systems, and web browsers;
- **Physics Engine:** Unity's built-in physics engine simulates real-world behaviors, including gravity, collisions, and material properties, allowing for realistic space simulations and interactions between objects in a virtual environment.

2.4 Hardware Components

2.4.1 Oculus Quest 2

The Oculus Quest 2 is a standalone virtual reality (VR) headset developed by Meta, designed to provide immersive VR experiences without the need for a connected PC or external sensors (Figure 2.3). It is widely used for gaming, simulation, and educational applications, making it well-suited for interactive visualizations of space missions [21].

The Oculus Quest 2 offers the following key features:

- **Wireless Standalone Operation:** The headset operates independently without requiring connection to a PC or console, allowing users to move freely in VR environments.

- **High-Resolution Display:** It features high-resolution LCD panels with fast refresh rates, providing clear and smooth visuals that enhance immersion.
- **Inside-Out Tracking:** The Quest 2 uses onboard cameras for inside-out tracking, enabling precise motion tracking without external sensors.
- **Hand and Controller Input:** Supports both handheld controllers and hand tracking, allowing for natural and intuitive interaction within virtual environments.
- **Cross-Platform Compatibility:** While standalone, it can also be connected to a PC to access more complex VR applications via Oculus Link.
- **Comfort and Ergonomics:** Designed with adjustable straps and lightweight materials to enable extended use during simulations and visualization tasks.

FIGURE 2.3: Oculus Quest 2

In this project, the Oculus Quest 2 headset was used to create a Virtual Reality visualization of the Cebreros ground station and the Solar System. The VR scenes integrate real-time tracking data of spacecraft along with a detailed 3D model of the Cebreros antenna, providing an immersive and interactive experience of space mission operations.

2.5 Previous Work

Prior to the start of this project, a first mock-up and proof-of-concept system was set up and running inside the MER at the Cebreros station, developed by Rafael Andrés from the Science Directorate at ESAC in collaboration with Jorge Fauste from the Operations (OPS) Department. This display corresponds to the one mentioned in Subsection 2.1.2, composed of three screens exhibiting relevant mission information.

2.5.1 Preliminary Cosmographia Setup

One of the original programs of the initial exhibit at Cebreros features a view of the solar system in Cosmographia, showcasing a 3D model of the spacecraft being tracked by the DSA-2 antenna at each moment, or the Earth when no spacecraft is being tracked.

To achieve this display, a dedicated Python script was created to extract the *planview* file from the ESOC Scheduling office once per day. This file contains the names of all spacecraft expected to be tracked over the following two weeks, along with their respective AOS and LOS times, and is then converted by the script into a CSV file containing only the data relevant to the Cebreros missions.

In this initial version, the spacecraft position and attitude are obtained from SPICE kernels containing data initially projected during the planning stages of the missions, and therefore do not reflect the real-time, up-to-date values of these parameters.

2.5.2 Preliminary Cesium Scene Setup

The other main display from the original exhibit at Cebreros features a 3D view of the Cebreros station terrain, including a free mock-up 3D model of an antenna created by NASA, which is significantly different from the DSA-2 antenna.

The antenna's 3D model is divided into three sections: the base, the azimuth platform, and the disk. Its azimuth and elevation are adjusted using SPICE kernels in combination with the *planview* file mentioned in Subsection 2.5.1, determining the position of the tracked spacecraft at each moment and rotating the antenna parts to point at it, while returning to its idle position when no spacecraft is being tracked.

In the scene, the tracked spacecraft is also shown in its position in the sky, using SPICE kernels to accurately identify its correct placement within the view.

2.5.3 Preliminary Trajectory Page Setup

The last display from the original exhibit at Cebreros serves as an auxiliary display to the Cosmographia view described in Subsection 2.5.1.

For some of the spacecraft tracked by DSA-2 at Cebreros, the ESA Science Department has created web pages that visualize their position and trajectory. These web pages are manually displayed on one of the screens when the respective spacecraft is being tracked at Cebreros and shown in the Cosmographia view. When no spacecraft are being tracked, or if the tracked spacecraft does not have a corresponding web page, nothing is displayed on the screen.

Chapter 3

Proposed Work

This chapter outlines the work proposed in this thesis. The starting point was a preliminary visualization mock-up and proof-of-concept system developed for visitors at ESA's Cebreros station, primarily by Rafael Andrés (Science Directorate, ESAC) in collaboration with Jorge Fauste (OPS Department). The system consists of three screens connected to a single computer housed within a dedicated ESA IT Security Zone, allowing secure Internet access as well as limited access to ESOC intranet pages, ESTRACK web resources, and Science Department sites.

The aim of this work was to further develop and enhance this system by improving its visual accuracy and appeal, as well as by extending its functionality through the addition of new interactive tools for station visitors.

3.1 Enhancement of the Cesium Display

Originally, the Cesium visualization of the Cebreros station integrated Python scripts for calculating spacecraft positions and antenna orientation and Angular for 3D rendering, using a generic NASA antenna model.

The proposed enhancements of this display involved creating a detailed 3D model of the Cebreros antenna from scratch in Blender and replacing the generic model, increasing the visualization's accuracy and realism.

This new 3D model was later approved for use and subsequently integrated into the Cesium scene, which was then officially launched. Following this, I presented the updated Cesium display to ESA staff members, who requested the creation of a second antenna model—this time representing the New Norcia (NNO-3) antenna in Australia. This

model was to be incorporated into an enhanced version of the Cesium display, allowing visualization of all three ESA deep-space stations: Cebreros, New Norcia, and Malargüe.

The impact of my work on the evolution of the visualization system is illustrated in Figure 3.1.

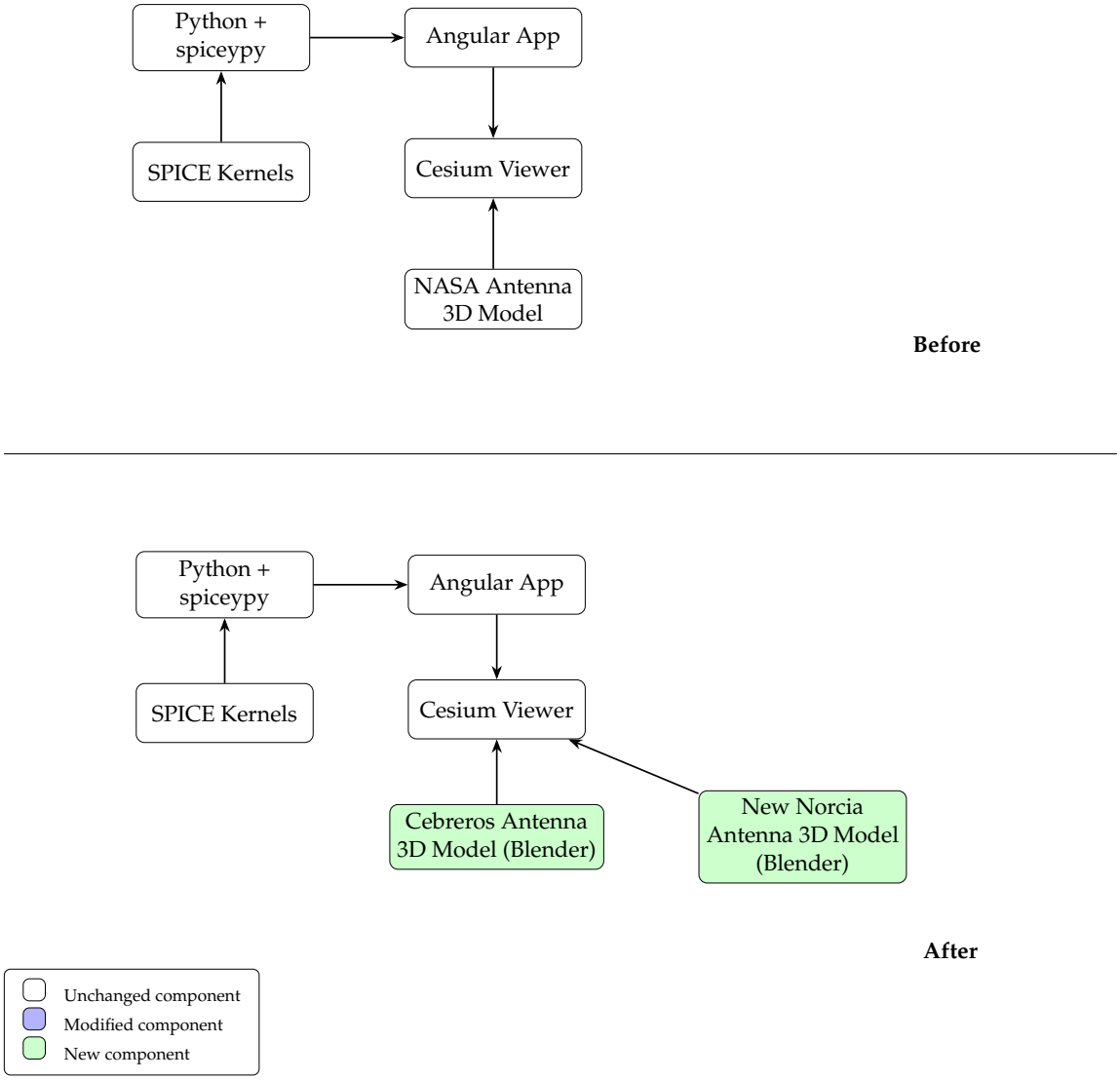


FIGURE 3.1: Architecture of Cesium Display: Before and After Model Integration

3.2 Enhancement of the Cosmographia Display

The original Cosmographia system used daily-updated spacecraft tracking schedules and plan-based SPICE kernels to position and visualize spacecraft in Cosmographia.

The proposed enhancements to this system were the following:

- Display the real-time position and orientation of the spacecraft currently being tracked by the Cebreros antenna;
- Introduce camera dynamics by enabling automatic rotation around the tracked spacecraft;
- Automate spacecraft management through scripts for adding and removing spacecraft from the display;
- Improve informational content by dynamically displaying ephemeris data associated with the tracked spacecraft.

The corresponding diagram depicts these architectural changes and new components (Figure 3.2).

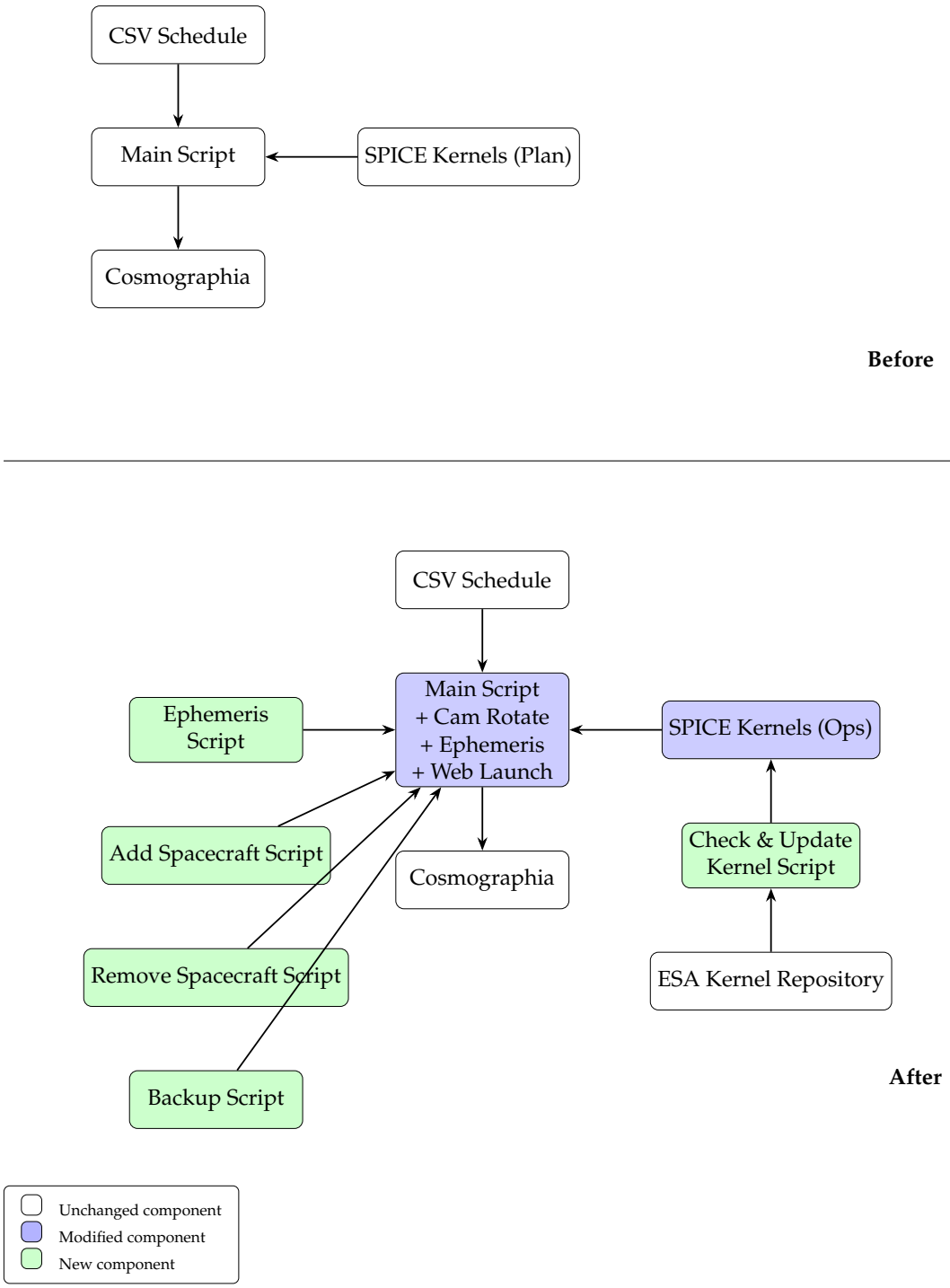


FIGURE 3.2: Architecture of Cosmographia Display: Before and After Script Integration

3.3 Enhancement of the Trajectory Display

Previously, spacecraft trajectories were viewed manually via web browser.

The proposed enhancement to this display was to automate this process by integrating trajectory webpage launching directly from the Cosmographia script during tracking

periods, ensuring seamless and timely visualization (Figure 3.3).

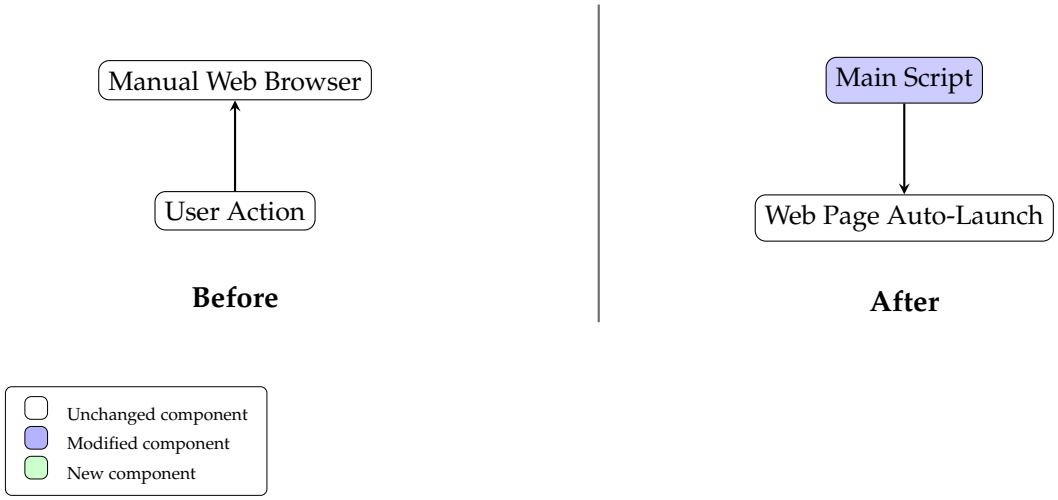


FIGURE 3.3: Architecture of Trajectory Display: Before and After Automation

3.4 Development of a Virtual Reality Application

I developed a real-time VR application using Unity and a Flask backend. This system visualizes the Cebreros antenna and spacecraft positions and rotations live, leveraging SPICE kernel data calculated by the Flask app and requested via HTTP from Unity. This application provides an immersive and intuitive way to observe antenna tracking and spacecraft dynamics in the context of the Solar System, complementing the previous visualization tools. The architecture of this VR system is summarized in Figure 3.4.

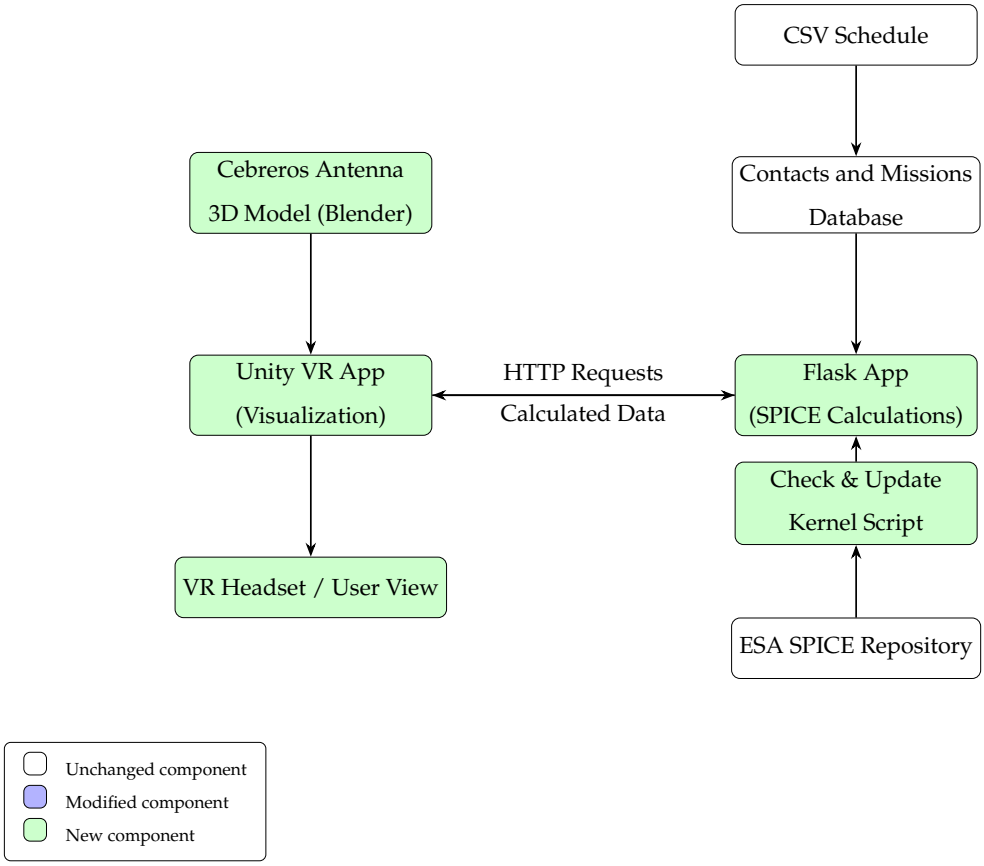


FIGURE 3.4: Architecture of the Virtual Reality App for Real-Time Visualization

Together, these contributions provide a comprehensive and improved suite of visualization tools that enable more accurate, real-time, and immersive monitoring of spacecraft tracked by the Cebreros antenna, facilitating both operational use and public outreach. Currently, the enhanced versions of the first three displays mentioned are in use at Cebreros for visitors to see and learn more about the activities of the antenna.

Chapter 4

Cesium Display and Antenna Models

The following three chapters, Chapters 4–6, describe the implementation of the work outlined in the previous chapter, namely, the development and integration of real-time antenna and spacecraft visualization tools for outreach and educational purposes. The work spans multiple technologies, including 3D modeling software, web-based visualization platforms, and virtual reality environments, which utilize SPICE kernels to obtain accurate positional data.

The development process involved the enhancement and further expansion of existing tools, as well as the creation and integration of new systems to enhance interactivity and realism. Each major component of this project is addressed in its own chapter, detailing the progression of the work from design to implementation and integration.

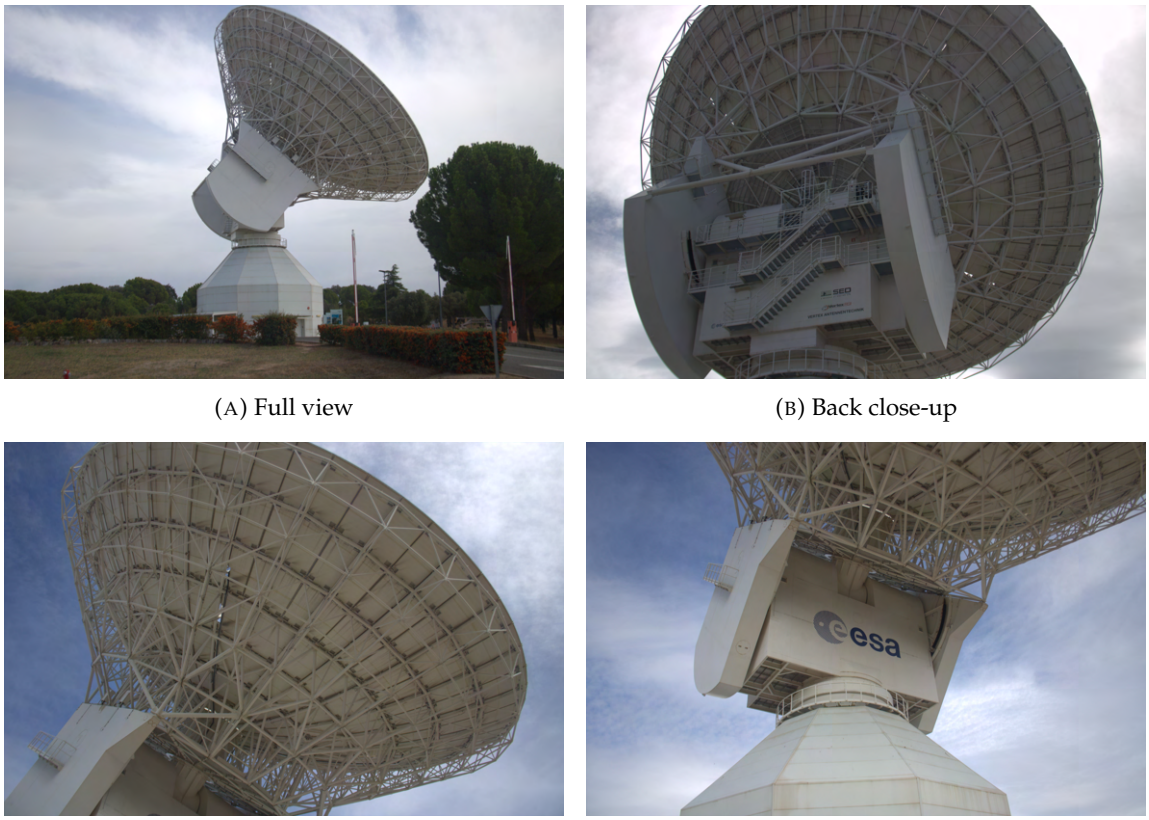
This chapter focuses on the first stage of development, which aimed to create realistic and dynamic 3D models of the Cebreros and New Norcia 3 antennas for integration into the stations' Cesium-based visitor display.

4.1 Data Collection

DSA-2

In order to gather both visual and dimensional data for the Cebreros antenna model, I conducted an on-site visit to the station, where I took photographs from multiple angles to ensure coverage of all relevant sections of the antenna (Figure 4.1). These images served as reference material during the modeling process. In addition to the photographic documentation, I obtained architectural project images of the antenna, provided by Jorge

Fauste, which were instrumental in determining the dimensions of several components of the structure.



(A) Full view

(B) Back close-up

(C) Dish close-up

(D) Front close-up

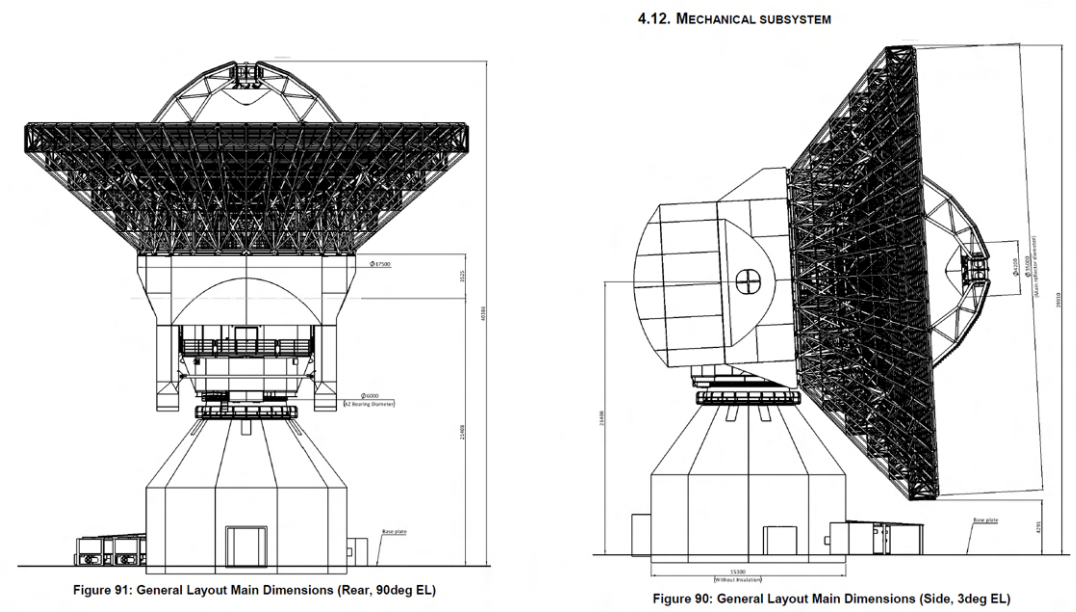
FIGURE 4.1: Photographs of the DSA-2

NNO-3

Due to the remote location of the New Norcia station, I was unable to visit the site in person to observe and photograph the NNO-3 antenna. However, Jorge Fauste kindly provided a selection of photographs taken by staff at the station (Figure 4.2), along with a few architectural project images of the antenna (Figure 4.3). These resources served as the primary references for the development of the 3D model.

(A) Full front view (B) Full back view

FIGURE 4.2: Photographs of the NNO-3



(A) Front view (B) Side view

FIGURE 4.3: Structural Plans of the NNO-3

4.2 Modeling Process

The 3D modeling of the antennas was done using Blender, with the structures built from the bottom up using the previously gathered reference images for guidance. No reference planes or blueprints were used in Blender for this project. Instead, the measuring tool in Blender was used to ensure consistent scaling across all parts of each model, in accordance with the structural plans. The modeling process was carried out iteratively, beginning with simple geometric shapes and gradually refining them to capture finer details.

4.2.1 Model Segmentation

This process was structured into three main parts, each corresponding to a key component of the antenna's physical architecture:

- **Base:** The foundation structure of the antenna;
- **Azimuth structure:** The middle rotating section that allows horizontal movement;
- **Elevation structure:** The dish and yoke assembly that enables vertical tilting.

The sections were separated into three distinct objects to allow for later articulation of the antenna's rotational and tilting motions. To support this, the objects were parented hierarchically as follows:

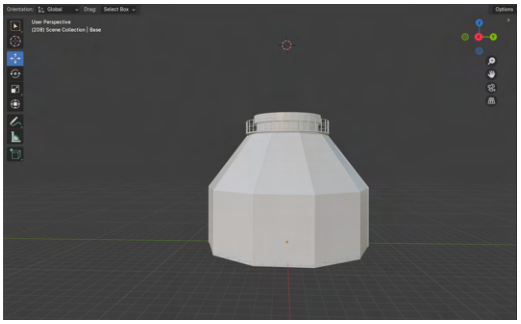
$$\text{Base} \rightarrow \text{Azimuth Structure} \rightarrow \text{Elevation Structure}$$

This hierarchy ensures that any transformation applied to a parent object is inherited by its children, but not the other way around. Additionally, the pivot points of the movable structures were adjusted to ensure correct alignment with their intended axes of rotation.

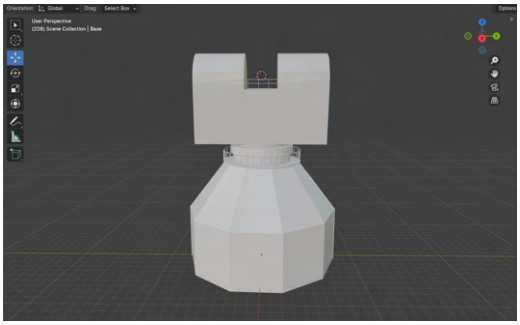
The construction of the DSA-2 3D model, section by section from bottom to top, is illustrated in Figure 4.4. The completed model created for the NNO-3 is shown in Figure 4.5.

4.2.2 Model Complexity

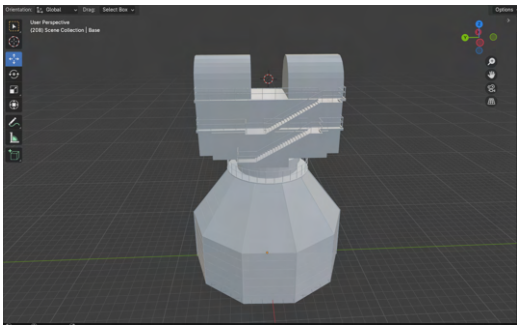
While achieving good performance when integrating the models into the Cesium scene was an important consideration, the primary focus of this work was to ensure the realism and visual credibility of the antenna models.



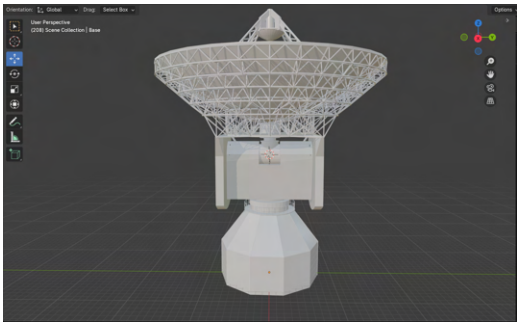
(A) DSA-2 Base in Blender



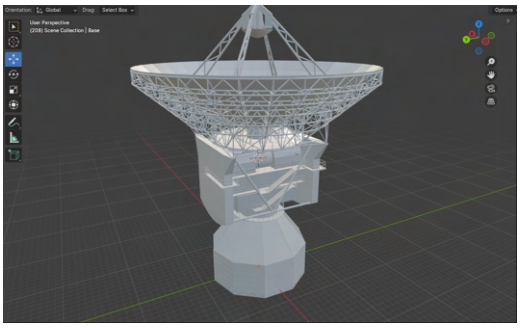
(B) DSA-2 Base and Azimuth (Front)



(C) DSA-2 Base and Azimuth (Back)



(D) Full DSA-2 Model (Front)



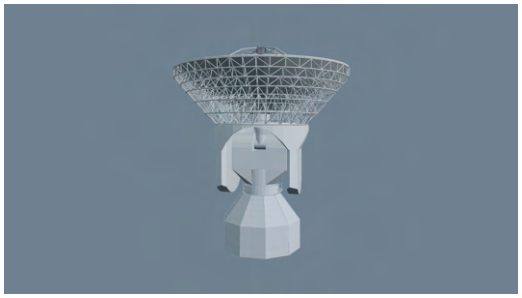
(E) Full DSA-2 Model (Back)

FIGURE 4.4: Bottom-to-top construction of the DSA-2 3D model in Blender

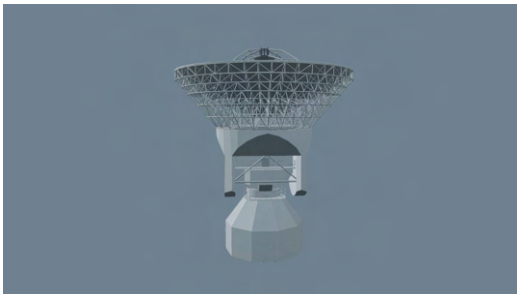
The final DSA-2 model consists of approximately 290,551 triangles, which provides a high level of geometric detail suitable for virtual reality and scientific visualization purposes. This resolution represents a balance between visual accuracy and real-time rendering performance on modern platforms.

4.2.3 Export Format

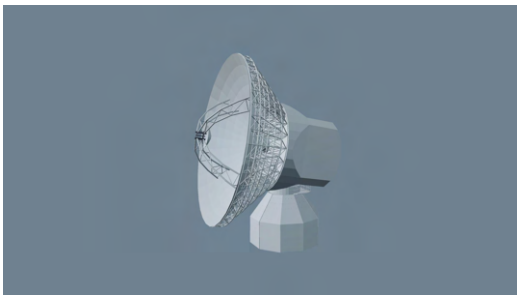
After completing the modeling in Blender, both models were exported to the glTF format. This format was chosen due to its ease of integration with the pre-existing code for the Cesium scene, where the models would be used (the previously employed NASA antenna model was also in this format). This choice is well suited for the task, as glTF is



(A) NNO-3 3D Model (Front)



(B) NNO-3 3D Model (Back)



(C) NNO-3 3D Model (Side)

FIGURE 4.5: Full NNO-3 3D Model in glTF Viewer

an industry-standard format that is lightweight and compatible with Cesium, as well as other software used in this project, such as Unity.

4.2.4 Articulations

The articulations defining control over the antenna's azimuth and elevation angles were embedded directly within the glTF files using the following extension snippet:

```
"extensions": {
    "AGI_articulations": {
        "articulations": [
            {
                "name": "AntennaAzimuth",
                "stages": [
                    {
                        "name": "AzimuthRotation",
                        "type": "yRotate",
                        "minimumValue": 0,
                        "initialValue": 0,
                        "maximumValue": 360
```

```
                    }
                ]
            },
            {
                "name": "AntennaElevation",
                "stages": [
                    {
                        "name": "ElevationRotation",
                        "type": "zRotate",
                        "minimumValue": -90,
                        "initialValue": 0,
                        "maximumValue": 0
                    }
                ]
            }
        ]
    }
},
"extensionsUsed": [
    "AGI_articulations"
]
```

This extension was included to enable the model's use within the Cesium scene, where it must accept azimuth and elevation inputs and update its orientation accordingly.

4.3 Antenna Model Integration in Cesium

The integration of the articulated 3D models into the Cesium environment required minimal effort, due to the compatibility of the glTF format and its articulation extensions with the pre-existing implementation (Figure 4.6).

DSA-2

The integration of the DSA-2 model was straightforward: the generic NASA model previously in use was replaced with the new model, and the relevant scripts were updated

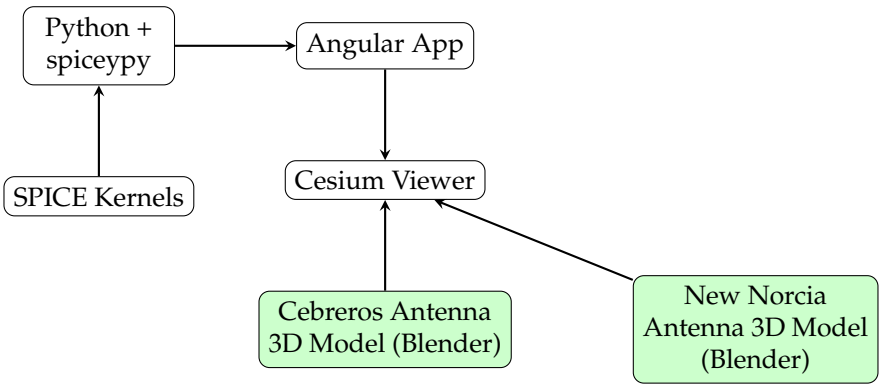


FIGURE 4.6: Architecture of Cesium Display: After Model Integration

accordingly. Additionally, it was necessary to calibrate the model's rotation to accurately reflect the tracking motion of the real antenna. This involved taking the real azimuth and elevation values, obtained using SPICE kernels and related functions, and transforming them into input values that would correctly reproduce the antenna's physical orientation within the scene.

To achieve this, I first identified the model's idle position, which corresponds to the azimuth structure being aligned with the base and the dish pointing directly upwards, with no inclination. I then tested which input values matched this neutral configuration and determined the direction of positive rotation for both azimuth and elevation inputs. Based on this, I adjusted the transformation logic accordingly. For example, the elevation input required by the model was computed as `el - 90`, where `el` is the real elevation angle of the antenna.

Figure 4.7 shows the Cesium scene following the integration of the DSA-2 model.

NNO-3

The New Norcia station was incorporated into the Cesium scene some time after the DSA-2 model had been created and integrated into the original display. This expansion of the scene, including the integration of the NNO-3 antenna, was carried out by other ESA staff. The integration process followed a similar approach to that used for the DSA-2 model.

FIGURE 4.7: Cesium View After DSA-2 Integration

Chapter 5

Cosmographia and Trajectory Display

The main objective of this part of the work was to enhance the pre-existing Cosmographia display—originally used to visualize the spacecraft tracked by the Cebreros antenna within the Solar System—by making it more dynamic and capable of receiving real-time operational data on spacecraft positions, rather than relying on static planned trajectories (Figure 5.1).

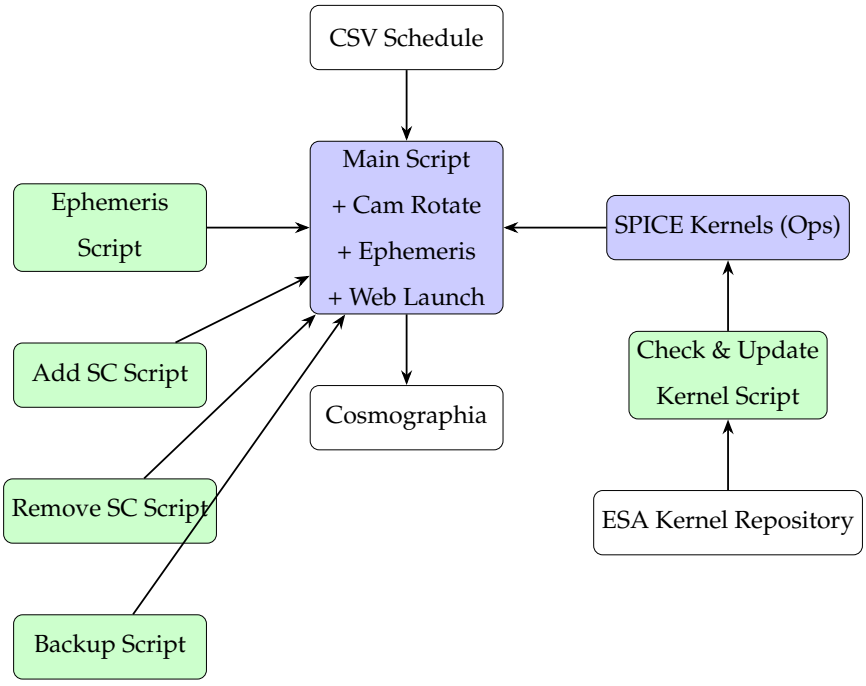


FIGURE 5.1: Architecture of Cosmographia Display: After Script Integration

5.1 Automating SPICE Kernel Updates

The initial Cosmographia display relied on static, plan-based SPICE kernels to determine the position and orientation of the spacecraft being tracked. However, these kernels only contained predicted data rather than real-time information. To achieve accurate, up-to-date spacecraft positioning, it was therefore necessary to switch to operational SPICE kernels, which include actual tracking data generated during mission operations.

Each spacecraft typically requires a dedicated metakernel, which is a text file (with extension .tm) that lists the paths to all necessary SPICE kernels relevant to that mission. These usually include:

- SPK (Spacecraft and Planet Kernel) – for position (ephemeris) data
- CK (C-Kernel) – for spacecraft orientation (attitude) data
- SCLK (Spacecraft Clock Kernel) – to convert between spacecraft clock time and standard time
- FK (Frame Kernel) – to define the reference frames used
- IK (Instrument Kernel) – describing onboard instruments' geometry
- LSK (Leapseconds Kernel) – to handle leap seconds in time conversions
- PCK (Planetary Constants Kernel) – optional, contains physical and orientation constants

A sample content of a metakernel might look like Listing 5.1.

```
KERNELS_TO_LOAD = (

  `$KERNELS/ck/gaia_sc_ssm_20240101_20250101_f20191030_v01.bc'

  `$KERNELS/fk/gaia_v01.tf'

  `$KERNELS/lsk/naif0012.tls'

  `$KERNELS/pck/de-403-masses.tpc'
  `$KERNELS/pck/pck00010.tpc'

  `$KERNELS/sclk/gaia_fict_20191030.tsc'

  `$KERNELS/spk/de432s.bsp'
  `$KERNELS/spk/gaia_rec_20240101_20250101_v01.bsp'

)
```

LISTING 5.1: Excerpt from a Gaia metakernel

Locally, the structure of the kernel directories for a spacecraft like Gaia may look like Listing 5.2.

```
gaia_kernels/
├── ck/
│   └── gaia_sc_ssm_20240101_20250101_f20191030_v01.bc
├── fk/
│   └── gaia_v01.tf
├── lsk/
│   └── naif0012.tls
├── pck/
│   ├── de-403-masses.tpc
│   └── pck00010.tpc
├── sclk/
│   └── gaia_fict_20191030.tsc
├── spk/
│   ├── de432s.bsp
│   └── gaia_rec_20240101_20250101_v01.bsp
└── mk/
    └── gaia_ops.tm
```

LISTING 5.2: Kernel directory structure

Only the metakernel (`.tm` file) is loaded directly into the software. It then loads the referenced kernels automatically. The organization and completeness of these files are essential for accurate visualization and analysis, especially when real-time or near-real-time mission tracking is involved.

Additionally, with the update of the kernels used, it is often necessary to update the associated resource files used in Cosmographia for each spacecraft. These resources typically include 3D models of the spacecraft, configuration files, and scenario files. The configuration files define how the spacecraft and its components are visualized and behave within the environment (e.g., articulated parts like solar arrays), while the scenario files specify what is shown in a particular visualization, such as which spacecraft, ground stations, or annotations are included in the scene, and over what time interval the visualization runs. Keeping these resources consistent with the new kernels ensures a correct and coherent visual representation of the updated trajectory data.

These kernels are frequently updated as each mission progresses, and it is therefore essential to keep them up to date in order to ensure the accuracy of the results. To facilitate this, I developed the `load_kernels.py` script.

Listing 5.3 provides an overview of the script, outlining the relationships between the algorithms discussed in this section to facilitate understanding. While some functions are only briefly described, others are presented in full through their corresponding pseudo-code.

```
load_kernels.py
└── main() (Algorithm 1)
    ├── getlatestkernel() (Algorithm 2)
    ├── checkfileupdate() (Algorithm 3)
    │   └── getlastupdate()
    ├── deletereferencedfiles() (Algorithm 4)
    ├── deletefileswkey() (Algorithm 5)
    ├── downloadfile() (Algorithm 6)
    ├── editmkpath()
    ├── extractfilesfrommk() (Algorithm 7)
    │   └── downloadfile() (Algorithm 6)
    ├── updateresources() (Algorithms 8-10)
    │   ├── updateresourcelinux()
    │   ├── deletefileswkey()
    │   ├── downloadfile()
    │   ├── updatecebrerosload()
    │   ├── editscenarioscript()
    │   └── extractconfigfiles()
    ├── updatecebrerosspice()
    ├── updategetall()
    └── registerupdate()
```

LISTING 5.3: load_kernels.py structure

The script's main function is defined in Algorithm 1.

Algorithm 1 Updating SPICE kernels Script

```
1:  function MAIN(spacecraft, esa_rep, mk_dir, resource_dir)
2:      for sc in spacecraft do
3:          sc_rep ← esa_rep+sc
4:          sc_rep_mk ← sc+'/kernels/mk/'
5:          latest_mk ← GETLATESTKERNEL(sc_rep_mk,sc)
6:          isUpdated ← CHECKFILEUPDATE(latest_mk,sc)
7:          new_mk ← latest_mk.split('/')[-1]
8:          new_mk ← mk_sc_id.split('_')[0]
9:          if isUpdated == False then
10:             DELETEREFERENCEDFILES(textttmk_sc_id)
11:             DELETEFILESWKEY(resource_dir,mk_sc_id)
12:             sc_resources ← esa_rep+sc
13:             downloaded_mk ← DOWNLOADFILE(latest_mk,mk_dir)
14:             mk_path ← mk_dir+downloaded_mk
15:             EDITMKPATH(mk_path)
16:             EXTRACTFILESFROMMK(mk_path,sc)
17:             UPDATERESOURCES(sc_rep,sc_resources,sc,new_mk)
18:             UPDATECEBREROSSPICE(mk_path)
19:             UPDATEGETALL(mk_path,sc)
20:             REGISTERUPDATE(sc)
21:         end if
22:     end for
23: end function
```

Algorithm 1 automates the update process by performing the following steps for each spacecraft:

- Checks whether the spacecraft's metakernel on ESA's repository has changed since the last local update (Algorithm 1, lines 5–6);
- If no changes are detected, the script proceeds to the next spacecraft;

- If the spacecraft is being processed for the first time (i.e., it has no prior local data), the script downloads all relevant files, including the metakernel, associated kernels, configuration files, and 3D models, and loads the spacecraft accordingly (Algorithm 1, lines 9–21);
- If updates to the metakernel are detected, the script deletes the previously stored data for that spacecraft and replaces it with the newly downloaded metakernel, all referenced kernels, and corresponding configuration and model files (Algorithm 1, lines 9–21).

This script is intended to be run on a regular basis (e.g., every two weeks) to ensure that the visualization remains up-to-date with the latest kernel and configuration data.

5.1.1 Checking Available Updates

To check whether updated metakernels are available in ESA's SPICE repository, I created a tracking file that stores the last local kernel update date done for each mission, as well as the URL pointing to the metakernel currently in use in the ESA repository. I then implemented two functions, `getlatestkernel()` (Algorithm 1, line 5) and `checkfileupdate()` (Algorithm 1, line 6) to evaluate the need for a new update. These functions are defined in Algorithms 2 and 3, respectively.

Algorithm 2 Retrieving the Last Available Metakernel

```
1: function GETLATESTKERNEL(rep_url, sc)
2:     response ← HTTPGET(rep_url)
3:     if sc == 'SOLAR-ORBITER' then
4:         files ← [filename | for filename in response if 'solo_ANC_soc-pred-mk' in filename]
5:         retrieved_file ← MIN(files,key=length)
6:     else
7:         files ← [filename | for filename in response if 'ops' in filename]
8:         if not files then
9:             files ← [filename | for filename in response if 'plan' in filename]
10:        end if
11:        if not files then
12:            files ← [LATESTMODIFIEDMK(rep_url)]
13:        end if
14:    end if
15:    retrieved_file ← MIN(files,key=length)
16:    return '{rep_url}{retrieved_file}'
17: end function
```

Algorithm 3 Checking if Update is Required

```
1:  function CHECKFILEUPDATE(file_url, sc)
2:      filename ← file_url.split('/')[-1]
3:      base_url ← file_url.remove(filename)+'/'
4:      response ← HTTPGET(base_url)
5:      for row in response.rows do
6:          cols ← response.columns
7:          if length(cols) ≥ 2 then
8:              file_link ← cols[1]
9:              if file_link == filename then
10:                 last_modified_date ← cols[2]
11:             end if
12:         end if
13:     end for
14:     if last_modified_date then
15:         last_update ← GETLASTUPDATE(sc)
16:         if last_update == None or last_modified_date > last_update then
17:             return False
18:         end if
19:         return True
20:     end if
21: end function
```

When invoked in Algorithm 1, Algorithms 2 and 3 carry out the following steps for each spacecraft:

- Algorithm 2 sends an HTTP request to the ESA SPICE repository to retrieve the list of all available metakernel files for the spacecraft;
- It then parses the HTML response and searches for files containing the key `ops` in their names. If none are found, it searches for files containing the key `plan`. As a last fallback, it selects the most recently added metakernel. In specific cases where the naming convention differs, such as for Solar Orbiter, the code is adjusted to search for a custom pattern like `solo_ANC_soc-pred-mk` (Algorithm 2, lines 3–14);

- Among the matching files, it selects the one with the shortest filename to account for cases where multiple versions exist (e.g., `gaia_ops.tm` vs. `gaia_ops_v1.tm`), and registers its URL. This is done because multiple versions of a metakernel may coexist and, in such cases, the versioned files are used during development or validation stages, while the unversioned file is typically updated to always point to the latest validated version. This unversioned file acts as the "current" operational metakernel, maintained by the mission's SPICE kernel provider (Algorithm 2, lines 5&15);

- Algorithm 3 then extracts the modification date of the selected metakernel file from the repository (Line 10);

- It then retrieves the last recorded update date for the spacecraft from the local tracking file (Algorithm 3, line 15);

- Finally, it compares both dates, and if the repository's file is more recent, flags that an update is available (Algorithm 3, lines 16–19).

5.1.2 Deleting Previously Stored Kernels

If an update is available for a given spacecraft, the script begins by deleting all local kernels associated with that mission, by calling the functions `deletereferencedfiles()` (Algorithm 1, line 10) and `deletefileswkey()` (Algorithm 1, line 11). These functions are defined in Algorithms 4 and 5, respectively.

Algorithm 4 Deleting Files Referenced in a Metakernel

```
1:  function DELETEREFERENCEDFILES(mk_id, mk_dir)
2:      metakernel_path ← None
3:      for filename in mk_dir do
4:          file_path ← mk_dir+filename
5:          if ISFILE(file_path) and mk_id in filename then
6:              metakernel_path ← mk_dir+filename
7:          end if
8:      end for
9:      file ← OPENFILE(metakernel_path)
10:     referenced_files ← file.findExpression('KERNELS_TO_LOAD\s*=\s*(.*?\.b?\S+)')
11:     for file_path in referenced_files do
12:         if ISFILE(file_path) then
13:             DELETEFILE(file_path)
14:         end if
15:     end for
16: end function
```

Algorithm 5 Deleting Files with a given Key

```
1: function DELETEFILESWKEY(directory, keyword)
2:     for filename in directory do
3:         file_path ← directory+filename
4:         if ISFILE(file_path) and keyword in filename then
5:             DELETEFILE(file_path)
6:         end if
7:     end for
8: end function
```

Algorithm 4 first identifies the correct local metakernel by looping through all local metakernel files in the kernel directory and looking for one whose filename contains a keyword unique to the spacecraft. This keyword is consistently the first word in the filename (before the first underscore), which matches the beginning of the latest metakernel filename of that mission from the ESA repository (Algorithm 4, lines 3–8).

Once the corresponding local metakernel is located, the script opens the file and extracts all kernel files listed in the `KERNELS_TO_LOAD` section (Listing 5.1).

It then iterates through each referenced file path, checks if the file exists locally, and deletes it (Algorithm 4, lines 11–15).

After all referenced kernel files are removed, Algorithm 5 deletes the metakernel file itself.

If no metakernel is found locally for a given spacecraft, this means that it is being processed for the first time and there are no files to delete, so this step is skipped.

5.1.3 Downloading and Storing Updated Spacecraft Data Files

Once all local kernels for a given spacecraft have been deleted, Algorithm 1 calls an auxiliary function, `downloadfile()`, in order to download the necessary updated files from the ESA SPICE repository. This function is defined in Algorithm 6.

Algorithm 6 Dowloading File

```
1:  function DOWNLOADFILE(url, download_dir)
2:      response ← HTTPGET(url)
3:      filename ← BASENAME(url.rstrip('/'))
4:      if '.' not in filename and '?' not in filename then
5:          folder_name ← download_dir+filename
6:          CREATEFOLDER(folder_name)
7:          files ← [filename | for filename in response]
8:          for file in files do
9:              if file == '../' then
10:                 continue
11:             end if
12:             file_url ← url + file
13:             DOWNLOADFILE(file_url,folder_name)
14:         end for
15:     else if '.' in filename
16:         full_path ← download_dir+filename
17:         file ← OPENFILE(full_path)
18:         WRITETOFILE(file, response)
19:     end if
20:     return filename
21: end function
```

To begin, Algorithm 6 receives the metakernel URL returned by Algorithm 2 and downloads this file via an HTTP request into the local `mk` (metakernel) folder. In this case, since the received URL points to a file, the function proceeds directly to the *else* branch of the *if* statement, retrieving the file's contents and writing them to a new file in the designated local directory (Algorithm 6, lines 15–19). However, the function is also designed to handle URLs pointing to folders, from which we may wish to recursively download all contained files. This behavior is implemented in the first branch of the *if* statement (Algorithm 6, lines 4–14).

After downloading the metakernel, Algorithm 1 calls the `editmkpath()` function, which simply opens the file and modifies the line defining `PATH_VALUES` to:

```
PATH_VALUES = ( 'LOCAL_PATH' )
```

This edit ensures that all relative paths defined in the metakernel correctly resolve to local directories, rather than expecting the kernel files to exist at the original locations used by ESA (which would typically be absolute paths specific to ESA's internal systems).

Next, Algorithm 1 invokes the `extractfilesfrommk()` function, which is defined in Algorithm 7.

Algorithm 7 Extracting Files from Metakernel

```
function EXTRACTFILESFROMMK(mk, sc, kernel_dir, esa_rep)
    mk_file ← OPENFILE(mk)
    for folder_name in kernel_dir do
        if folder_name == 'mk' then
            continue
        end if
        files ← []
        sc_rep ← esa_rep+sc+'/kernels/{folder_name}/'
        folder_dir ← kernel_dir+folder_name
        pattern ← COMPILEREGEX("\s*([^\s]+/{folder_name}/[^\s]+)")
        for line in content do
            match ← pattern.search(line)
            if match then
                filename ← line.split('/')[-1].replace("'","")
                files.append(filename)
            end if
        end for
        for file in files do
            download_url ← sc_rep+file
            DOWNLOADFILE(download_url,folder_dir)
        end for
    end for
end function
```

Algorithm 7 loops through each subfolder inside the local `kernels` directory, each representing a different kernel type (e.g., `spk`, `ck`, `lsk`, `sclk`). For each folder, it scans the metakernel content for entries that reference that folder name, thereby identifying all

kernel filenames required for this spacecraft. These filenames are then used to construct download URLs from the ESA repository, and the corresponding files are retrieved and saved into their respective local kernel subfolders.

Then, Algorithm 1 proceeds to call the auxiliary function `updateresources()` in order to update the resource files associated with the given spacecraft. This function is defined in Algorithms 8–10.

Algorithm 8 Updating Resource Files

```
1: function UPDATERESOURCES(sc_rep, resource_dir, sc, mk_sc)
2:     config_dir ← resource_dir+'/config/'
3:     models_dir ← resource_dir+'/models/'
4:     scenarios_dir ← resource_dir+'/scenarios/'
5:     config_rep ← sc_rep+'/misc/cosmo/config/'
6:     models_rep ← sc_rep+'/misc/cosmo/models/'
7:     scenarios_rep ← sc_rep+'/misc/cosmo/scenarios/'
8:     folder_paths ← [scenarios_dir, models_dir]
9:     rep_links ← [scenarios_rep, models_rep]
10:    mk_id ← mk_sc.split('_')[0]
11:    mk_keywords ← "".join(x | for x in mk_sc.split('_')[1:]).split('.')[0]
12:    for folder_path in folder_paths do
13:        if not EXISTS(folder_path) then
14:            UPDATERESOURCELINUX(sc)
15:            CREATEFOLDER(folder_path)
16:        end if
17:    end for
18:    for i,link in enumerate(rep_links) do
19:        DELETEFILESWKEY(folder_paths[i],"")
20:        response ← HTTPGET(link)
21:        files ← [filename | for filename in response]
22:        mk_schema ← ''
23:        default ← ''
24:        pred ← ''
25:                                          // (...) Continues in next algorithm
26:    end for
27: end function
```

Algorithm 9 Updating Resource Files (continuation)

```
1: function UPDATERESOURCES(sc_rep, resource_dir, sc, mk_sc)
2:     for i,link in enumerate(rep_links) do
3:                                          // (...) Continuation of previous algorithm
4:         for file in files do
5:             if '.' not in file then
6:                 continue
7:             end if
8:             url ← link+file
9:             download_dir ← folder_paths[i]
10:            if i!=0 then
11:                DOWNLOADFILE(url)
12:            end if
13:            if i==0 and mk_keywords.lower() in file.lower() then
14:                if mk_schema=='' or Length(mk_schema) > Length(file) then
15:                    mk_schema ← file
16:                end if
17:            else if i==0 and 'plan' in file.lower()
18:                if mk_schema=='' then
19:                    mk_schema ← file
20:                end if
21:            end if
22:            if i==0 and '{mk_id.lower()}_001' in file.lower() then
23:                if default=='' or Length(default) > Length(file) then
24:                    default ← file
25:                end if
26:            end if
27:            if i==0 and 'pred' in file.lower() then
28:                if pred=='' or Length(pred) > Length(file) then
29:                    pred ← file
30:                end if
31:            end if
32:        end for
33:                                          // (...) Continues in next algorithm
34:    end for
35: end function
```

Algorithm 10 Updating Resource Files (continuation)

```
1: function UPDATERESOURCES(sc_rep, resource_dir, sc, mk_sc)
2:     for i,link in enumerate(rep_links) do
3:                                          // (...) Continuation of previous algorithm
4:         schemas ← [mk_schema, default, pred]
5:         scenario_extracted ← False
6:         if i==0 then
7:             for schema in schemas do
8:                 if schema!='' then
9:                     url ← link+schema
10:                    DOWNLOADFILE(url,download_dir)
11:                    UPDATECEBREROSLOAD(sc,schema)
12:                    scenario_path ← scenarios_dir+schema
13:                    EDITSCENARIOSCRIPT(scenario_path)
14:                    EXTRACTCONFIGFILES(scenario_path,config_rep,config_dir)
15:                    scenario_extracted ← True
16:                end if
17:            end for
18:            if scenario_extracted==False then
19:                latest_scenario ← GETLATESTSCENARIO(scenarios_rep)
20:                url ← link+latest_scenario
21:                DOWNLOADFILE(url,download_dir)
22:                UPDATECEBREROSLOAD(sc,latest_scenario)
23:                scenario_path ← scenarios_dir+latest_scenario
24:                EDITSCENARIOSCRIPT(scenario_path)
25:                EXTRACTCONFIGFILES(scenario_path,config_rep,config_dir)
26:            end if
27:        end if
28:    end for
29: end function
```

Algorithm 8 starts by defining the relevant local resource directories, specifically, the configuration (`config`), models (`models`), and scenarios (`scenarios`) folders, under the spacecraft's resource directory. It also constructs the corresponding paths to these folders

in the ESA SPICE repository, hosted under the `misc/cosmo/` directory of the spacecraft's remote archive (Algorithm 8, lines 2–7). Listing 5.4 illustrates the structure of the repository for a given spacecraft, using GAIA as an example.

```
GAIA/
├── kernels/
│   └── ...
└── misc/
    ├── git_hooks
    ├── release_notes
    ├── reports
    ├── skd
    └── cosmo
        ├── config
        ├── models
        └── scenarios
```

LISTING 5.4: Kernel directory structure

To ensure the local directory structure is in place, the script checks whether the required folders exist locally. If not, it creates the missing directories (Algorithm 8, lines 12–17).

Next, the function iterates through the repository URLs for the `models` and `scenarios` folders. For each of these, it begins by deleting all existing local files in the corresponding folder using the helper function defined in Algorithm 5, `delete_files_w_key()` (Algorithm 8, line 19). It then sends an HTTP request to retrieve the list of files available in the current remote folder, parsing the HTML content to extract the filenames (Algorithm 8, lines 20–21).

For the `models` folder, all listed files are downloaded directly into the local `models` directory (Algorithm 9, lines 10–12).

For the `scenarios` folder, the script performs a more selective procedure: it attempts to identify the most relevant scenario file by comparing the filenames with keywords derived from the current metakernel (e.g., spacecraft name and version). It prioritizes files that (1) contain the metakernel keyword but exclude `ops`, (2) match a pattern ending in

_001, or (3) contain pred in the name. This prioritization was derived by several tests using different scenario files to check which worked properly in Cosmographia (Algorithm 9, lines 13–31).

If a match is found, that scenario file is downloaded. If none of these patterns yield a result, the script instead downloads the latest available file in the scenarios folder (Algorithm 10).

Once a scenario file has been selected and downloaded, the function invokes a series of helper functions to finalize the update process: it updates the configuration file that controls which scenario is loaded for each mission (update_cebreros_load) and extracts any embedded configuration files from the scenario archive (extract_config_files) into the local config directory (Algorithm 10, lines 11–15 & 22–25).

Finally, Algorithm 1 ensures consistency across all local files by updating any files that reference the spacecraft's metakernels to reflect the new metakernel filenames. It also updates the tracking file, which logs the last update date for each mission, to record the new timestamps corresponding to the completed updates (Algorithm 1, lines 29–31).

5.2 Creating a Backup File

As is the case with any code that depends on external data sources, updating SPICE kernels can fail for various reasons, such as network issues or unanticipated changes in the structure or availability of files in the ESA SPICE repository. Recognizing this, it is important to prepare for such eventualities, including failures that may require manual intervention or modifications to the code.

Given that many critical files are deleted, replaced, or edited during the update process, a failed update could leave the system in an unstable state with no straightforward way to revert to a previously working configuration. To mitigate this risk, I developed a script, update_backup.py, that creates a full backup of the files used for the Cosmographia display prior to any updates. This script's main function is defined in Algorithm 11.

Algorithm 11 Creating/Updating a Backup File

```
1:  function UPATEBACKUP(source_dir, backup_dir, temp_backup_dir)
2:      if EXISTS(backup_dir) and ISFOLDER(backup_dir) then
3:          if EXISTS(temp_backup_dir) then
4:              DELETETREE(temp_backup_dir)
5:          end if
6:          COPYTREE(backup_dir,temp_backup_dir)
7:          DELETETREE(backup_dir)
8:      end if
9:      for folder in source_dir do
10:         folder_dir ← source_dir+folder
11:         backup_folder_dir ← backup_dir+folder
12:         if ISFOLDER(folder_dir) then
13:             COPYTREE(folder_dir,backup_folder_dir)
14:         end if
15:     end for
16:     DELETETREE(temp_backup_dir)
17:     if ERROROCURRED( ) then
18:         if EXISTS(temp_backup_dir) then
19:             if EXISTS(backup_dir) then
20:                 DELETETREE(backup_dir)
21:             end if
22:             COPYTREE(temp_backup_dir,backup_dir)
23:         end if
24:     end if
25: end function
```

Algorithm 11 follows these steps:

- It first checks whether a backup folder already exists. If it does, the script copies its contents into a temporary folder using `shutil.copytree()` as a safeguard, then deletes the original backup directory using `shutil.rmtree()` to make room for the new backup (Algorithm 11, lines 2–8);

- It then creates a new backup directory in the same location and copies all current Cosmographia-related files into it using `shutil.copytree()` (Algorithm 11, lines 9–15);

- If the new backup is created successfully, the temporary folder is deleted. If the backup fails, the script restores the temporary backup to its original location to preserve the previous working state (Algorithm 11, lines 16–24).

This way, whenever the `load_kernels.py` script is executed, the updated files should first be tested in Cosmographia by running `cebreros_main.py` within the application. Then, based on the outcome:

- If everything functions as expected, the `update_backup.py` script should be executed to update the backup folder with the latest working version of the spacecraft files;

- If an error occurs, `cebreros_main.py` from the backup directory can be executed to restore the last stable configuration of the Cosmographia display. This ensures that the application remains functional while the issue is diagnosed and resolved, preventing system-wide failure.

5.3 Simplifying Spacecraft Management (Adding and Removing)

When working with real-life dynamic systems, such as the Cebreros antenna, it is important to account for potential changes in these systems and to design flexible solutions that can help adapt to them with ease as they occur.

With this in mind, I developed two scripts (`add_spacecraft.py` and `remove_spacecraft.py`) to help simplify spacecraft management in Cosmographia, specifically for scenarios where the antenna begins tracking a new mission or stops tracking an existing one, requiring spacecraft to be added to or removed from the Cosmographia display.

5.3.1 Adding a Spacecraft

Adding a new spacecraft to Cosmographia involves downloading all required kernels and resource files, as well as modifying several local files to incorporate the new mission. Performing these steps manually each time can be tedious and error-prone, omitting a single file or edit can lead to issues that are difficult to diagnose.

To address this, I structured the `load_kernels.pyscript`, whose main function is defined in Algorithm 1, in such a way that:

- It begins by identifying which spacecraft are currently being tracked by the antenna, as specified in a separate text file;
- When checking for updates, if no entry exists for a given spacecraft in the update tracking file, the script interprets this as a new addition and proceeds to download all required kernels and resource files for that mission;
- During the step where the spacecraft's entries are updated in other configuration files (such as `cebreros_spice.json`, which stores the paths to each spacecraft's metakernel), the script checks whether an entry already exists. If it does not, a new entry is automatically created.

Given the structure of the `load_kernels.py` script, the final step was to implement an `add_spacecraft.py` script that allows the user to specify a new spacecraft to be added to the display.

Listing 5.5 offers a structured summary of the script, highlighting how the algorithms in this subsection are connected in order to support clearer understanding.

```
add_spacecraft.py
└── main() (Algorithm 12)
    ├── checkscname() (Algorithm 13)
    │   ├── addspacecraft() (Algorithm 14)
    └── addtrajectorypage() (Algorithm 15)
```

LISTING 5.5: add_spacecraft.py structure

The script's main function is defined in Algorithm 12.

Algorithm 12 Adding a Spacecraft to the Display

```
1: function MAIN(input)
2:     spacecraft ← input[0]
3:     sc_cosmo ← input[1]
4:     CHECKSCNAME(spacecraft,sc_cosmo)
5:     if Length(input) > 2 then
6:         url ← input[2] ADDTRAJECTORYPAGE(sc_cosmo,url)
7:     end if
8: end function
```

Algorithm 12 receives the name of the spacecraft the user wishes to add to the display (using the official ESA SPICE repository name) and its corresponding name for Cosmographia, as defined in the spacecraft's configuration file in the resources folder. Including this second name ensures the connection between ESA's naming conventions and those used by the Cosmographia application, which will be necessary later. Optionally, a URL for the spacecraft's trajectory page is also passed as input. The script then calls the auxiliary function `checkscname()`, which is defined in Algorithm 13.

Algorithm 13 Checking if Spacecraft exists in ESA Repository

```
1: function CHECKSCNAME(spacecraft, sc_cosmo)
2:     sc_rep ← esa_rep+spacecraft
3:     response ← HTTPGET(sc_rep)
4:     if ISSUCCESSFUL(response) then
5:         spacecraft_added ← ADDSPACECRAFT(spacecraft, sc_cosmo)
6:         return spacecraft_added
7:     end if
8: end function
```

Algorithm 13 checks whether the spacecraft exists in the ESA SPICE repository, and if so, invokes the `addspacecraft()` function, defined in Algorithm 14.

Algorithm 14 Adding Spacecraft to dedicated text file

```
1: function ADDSPACECRAFT(spacecraft, spacecraft_path, sc_cosmo)
2:     file ← OPENFILE(spacecraft_path)
3:     lines ← file.readlines
4:     sc_exists ← False
5:     for i,line in enumerate(lines) do
6:         if spacecraft in line then
7:             sc_exists ← True
8:         end if
9:     end for
10:    if sc_exists == False then
11:        file.addLine('{spacecraft} {sc_cosmo}\n')
12:    end if
13: end function
```

Algorithm 14 opens the dedicated text file that lists all spacecraft currently being tracked by the antenna and checks whether the selected spacecraft is already listed. If not, it adds the spacecraft to the file.

Once the spacecraft is added, Algorithm 12 checks whether a url was passed as part of the input. If so, it calls the function `addtrajectorypage()`, defined in Algorithm 15.

Algorithm 15 Adding Trajectory url to dedicated text file

```
1:  function ADDTRAJECTORYPAGE(sc_cosmo, url)
2:      file ← OPENFILE(spacecraft_path)
3:      lines ← file.readlines
4:      sc_exists ← False
5:      for i,line in enumerate(lines) do
6:          parts ← line.split(' ')
7:          if parts[0] == sc_cosmo then
8:              parts[1] ← url
9:              lines[i] ← '{sc_cosmo} {url}\n'
10:             sc_exists == True
11:             break
12:         end if
13:     end for
14:     file.writelines(lines)
15:     if sc_exists == False then
16:         file.addLine('{sc_cosmo} {url}\n)
17:     end if
18: end function
```

Algorithm 15 opens the text file containing spacecraft names and their trajectory page URLs, used for display when those spacecraft are being tracked. It then checks whether an entry exists for the selected spacecraft. If it does, the corresponding URL is replaced with the provided one. Otherwise, a new line is added with the spacecraft's Cosmographia name and input URL.

Once the `add_spacecraft.py` script has been executed, all that is required is to run the `load_kernels.py` script, and all necessary changes will be applied automatically.

5.3.2 Removing a Spacecraft

Removing a spacecraft from the Cosmographia display may be necessary when a mission concludes and the antenna is no longer tracking that spacecraft. This process involves, at a minimum, removing the spacecraft's entries from configuration files that instruct Cosmographia which missions to load (such as `cebreros_load.json`). However, if the associated SPICE kernels and resource files are not deleted, they will continue to occupy

significant disk space that could be better used elsewhere. Moreover, if the spacecraft is not removed from the text file that lists all missions currently tracked by the antenna, the `load_kernels.py` script will continue to update its files, resulting in unnecessary time and storage consumption. For this reason, it is important to ensure that all files and references associated with the spacecraft are fully removed in order to maintain an efficient and clean working environment.

With this in mind, I created a script, `remove_spacecraft.py`, whose main function is defined in Algorithms 16&17.

Algorithm 16 Removing a Spacecraft from the Display

```
1: function REMOVESPACECRAFT(input, spacecraft_path, webpages_path,
   last_updated_dir, mk_dir, resource_dir)
2:     spacecraft ← input
3:     file ← OPENFILE(spacecraft_path)
4:     lines ← file.readlines
5:     sc_cosmo ← ''
6:     for line in lines do
7:         if spacecraft not in line then
8:             file.write(line)
9:         else
10:            sc_cosmo ← line.split(' ')[1]
11:        end if
12:    end for
13:    if sc_cosmo != '' then
14:        f ← OPENFILE(webpages_path)
15:        lines ← f.readlines
16:        for line in lines do
17:            if sc_cosmo not in line then
18:                f.write(line)
19:            end if
20:        end for
21:    end if
22:    updatesfile ← OPENFILE(last_updated_dir)
23:    lines ← updatesfile.readlines
24:    latest_mk ← ''
25:    for line in lines do
26:        sc ← line.split(' ')[0]
27:        if sc == sc_cosmo then
28:            latest_mk ← line.split(' ')[2]
29:        end if
30:    end for                          // (...) Continues in next algorithm
31: end function
```

Algorithm 17 Removing a Spacecraft from the Display (Continuation)

```
1: function REMOVESPACECRAFT(input, spacecraft_path, webpages_path,
   last_updated_dir, mk_dir, resource_dir)   // (...) Continuation of previous algorithm
2:     if latest_mk != '' then
3:         new_mk ← latest_mk.split('/')[-1]
4:         mk_sc_id ← new_mk.split('_')[0]
5:         mk_name ← FINDMETAKERNEL(mk_sc_id)
6:         mk_path ← mk_dir+mk_name
7:         sc_resources ← resource_dir+spacecraft
8:         DELETEFILESFROMMK(mk_path,spacecraft)
9:         DELETEFILE(mk_name,mk_dir)
10:        DELETERESOURCES(sc_resources)
11:        UPDATECEBREROSSPICE(mk_path)
12:        UPDATEGETALL(mk_path,spacecraft)
13:        UPDATECEBREROSLOAD(spacecraft)
14:        UPDATERESOURCELINUX(spacecraft)
15:        DELETETIMESTAMP(spacecraft)
16:    end if
17: end function
```

Algorithms 16&17 perform a series of automated steps to completely remove a spacecraft from the local Cosmographia environment:

- Algorithm 16 receives as input the name of the spacecraft to be removed (matching the ESA SPICE repository designation) and checks whether it is listed in the text file that tracks currently monitored missions. If the spacecraft is found, its corresponding Cosmographia name (from the file entry) is recorded as `sc_cosmo`, and the entry is then deleted (lines 2–12);
- Then, the function checks whether the dedicated text file containing the spacecraft trajectory pages contains an entry for the selected spacecraft. If so, that entry is also deleted (Algorithm 16, lines 13–21);
- The script then retrieves the spacecraft's most recent metakernel file directory url from the dedicated text file containing the information regarding the last local kernel updates(Algorithm 16, lines 22–30);

- If no entry is found in the file for this spacecraft, this means that no files have been downloaded for this spacecraft yet, and no further changes need to be made. Otherwise, Algorithm 17 locates the spacecraft's local metakernel and first calls the auxiliary function `deletefilesfrommk()` to extract the paths of all referenced kernels and delete each of them. This function is identical to the `extractfilesfrommk` function, defined in Algorithm 7, up until line 17. From there, the function simply deletes all files from the files list.

- Once all reference files are deleted, it deletes the spacecraft's metakernel and all its resource files. It then deletes the spacecraft's entry from the update tracking file and updates any remaining relevant configuration files (such as `cebreros_load.json`) by removing references to the spacecraft (Algorithm 17, lines 9–15).

5.4 Calculating Spacecraft Ephemeris Data

In order to display ephemeris information of the tracked spacecraft in Cosmographia, such as its velocity relative to the Sun and its distances from both the Sun and the Earth, I developed a script named `spice_calculations.py`, whose main function is defined in Algorithm 18.

Algorithm 18 Calculating Ephemeris Data

```
1:  function CALCULATEEPHEMERIS(spacecraft,kernel_directory,current_time,ephemeris_path)
2:      for file in kernel_directory do
3:          if ISMETAKERNEL(file) then
4:              FURNSH(file)
5:          end if
6:      end for
7:      state_sun ← SPKEZR(spacecraft,current_time,'J2000','NONE','SUN')
8:      dist_sun ← state_sun[0][:3]
9:      vel_sun ← state_sun[0][3:]
10:     dist_earth, _← SPKPOS(spacecraft,current_time,'J2000','NONE','EARTH')
11:     dist_sun_magnitude ← GETNORM(dist_sun)
12:     dist_earth_magnitude ← GETNORM(dist_earth)
13:     vel_sun_magnitude ← GETNORM(vel_sun)
14:     f ← OPENFILE(ephemeris_path)
15:     line ← '{dist_sun_magnitude};{dist_earth_magnitude};{vel_sun_magnitude}'
16:     f.writeline(line)
17: end function
```

When called with a specific spacecraft name, Algorithm 18 begins by loading all relevant metakernels using the `furnsh()` function. It then retrieves the current time and uses it in combination with `SpiceyPy` functions, specifically `spkezr()` and `spkpos()`, to compute the required values.

The function `spkezr(spacecraft, current_time, 'J2000', 'NONE', 'SUN')` returns a state vector and a light-time value, where the first three components represent the spacecraft's position relative to the Sun, and the last three represent its velocity relative to the Sun. Similarly, `spkpos(spacecraft, current_time, 'J2000', 'NONE', 'EARTH')` returns the spacecraft's position vector relative to the Earth, along with the corresponding light-time value (Algorithm 18, lines 7–10).

From the resulting position and velocity vectors, the script computes the magnitudes in order to obtain the final scalar values of interest: velocity relative to the Sun, distance to the Sun, and distance to the Earth. These values are then saved to a separate text file, `ephemeris_data.txt`, for subsequent use (Algorithm 18, lines 11–16).

5.5 Updates to the Main Script

The original main script of the Cosmographia display utilized the `cosmoscripting` module for Cosmographia and was structured as follows:

- It begins by loading all necessary spacecraft catalog files, which provide the spacecraft models, positional information, and other required resources;
- It retrieves the current simulation time in Cosmographia and reads the `passes.csv` file, which contains the antenna's tracking schedule for the upcoming two weeks. From this, it constructs a timeline of events starting at the current time;
- For each event in the timeline, the script determines which spacecraft is being tracked and the duration of the tracking window;
- If the target is `IDLE`, indicating that no spacecraft is currently being tracked, an auxiliary function `idle()` is called. This function moves the camera to point at the Earth for the duration of the idle period using the `cosmo.waitSim()` function;
- If the target is a specific spacecraft, the auxiliary function `contact()` is called, which moves the camera toward the target spacecraft and displays it for the duration of the event;
- After each event concludes, the script continues to the next event in the timeline, repeating the same logic.

The main modifications I made to this script focused on enhancing the visualization aspects of the display, aiming to create a more dynamic and engaging experience for visitors.

5.5.1 Implementing camera dynamics

To enhance the visual engagement of the Cosmographia display during active spacecraft tracking, I modified the `contact()` function to include continuous, smooth camera motion around the target spacecraft. The camera initially moves to the spacecraft's position using the `cosmo.gotoObject(target, 2)` function, centering the view on the object of interest.

To avoid a static and potentially uninteresting visualization, I implemented a continuous rotation using the `cosmo.circleCenterRight(60, 60)` function within a loop that

runs for the duration of the tracking event. This rotates the camera around the spacecraft at a steady rate, producing a cinematic effect that simulates the camera orbiting the target in real time. The goal is to make the visualization more dynamic and captivating for visitors, especially during public outreach events or educational demonstrations.

5.5.2 Displaying Ephemeris Data

In order to enrich the visualization and provide more informative content during spacecraft tracking, I modified the main script to dynamically display key ephemeris data, namely, the spacecraft's velocity relative to the Sun, and its distances from both the Sun and the Earth. These changes were implemented as follows:

- When a spacecraft is being tracked and the `contact()` function is called, it invokes the `calculate_ephemeris()` function from the `spice_calculations.py` script, defined in Algorithm 18. This function calculates the spacecraft's ephemeris data using SPICE and saves the results to the `ephemeris_data.txt` file, as explained in Section 5.4;

- The main script then checks whether this file exists and contains valid data. If the expected output is present, the script extracts the relevant values and constructs a formatted string, such as:

```
info_text = (
f`Distance to Sun: {float(ephemeris_data[0]):.0f} km \n'
f`Distance to Earth: {float(ephemeris_data[1]):.0f} km \n'
f`Velocity Relative to Sun: {float(ephemeris_data[2]):.2f} km/s'
)
```

- This information is then displayed on the Cosmographia screen for the duration of the tracking event using the `cosmo.displayNote()` function, thereby providing both an engaging and educational view for observers.

5.5.3 Trajectory Webpage Launch

Some of the spacecraft tracked by the Cebreros antenna have dedicated ESA webpages that display their mission trajectories and current positions in real time. Previously, these

pages were opened manually whenever the corresponding spacecraft was being tracked and visualized in Cosmographia.

To streamline the workflow and eliminate the need for manual intervention, I updated the main script to automatically open the relevant webpage when such a spacecraft begins being tracked (Figure 5.2).

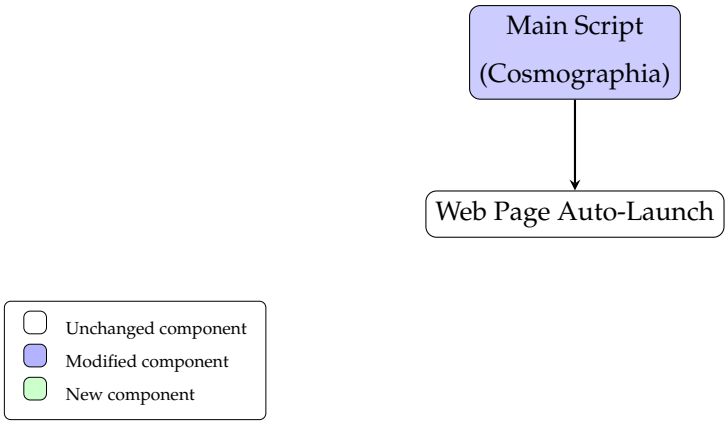


FIGURE 5.2: Architecture of Trajectory Display: After Automation

I began by creating a separate text file to register the URLs of webpages alongside the names of their corresponding spacecraft, mentioned in Subsection 5.3.1. I then implemented a script, `open_webpage.py`, which takes a URL and a duration as arguments and uses a `webdriver.Chrome()` instance to:

- Open the provided URL with `driver.get();`
- Maximize the window via `driver.maximize_window();`
- Wait for the specified duration using `time.sleep();`
- Close the webpage with `driver.quit()`.

In the main script, I modified the event-handling loop so that for each timeline event, it checks if the currently tracked spacecraft has a corresponding entry in the webpage URL file. If such an entry exists, the script retrieves the URL and uses `subprocess.Popen()` to launch the `open_webpage.py` script in a separate process, passing the relevant URL and duration as arguments. This enables the automatic display of a spacecraft's webpage on another screen during the tracking window, which then closes automatically once the event ends.

Chapter 6

A VR Cebreros View

The goal of this portion of the work was to extend the visualization system by creating an immersive and interactive view of the Cebreros antenna and the tracked spacecraft. To achieve this, I developed a virtual reality application that integrates much of the work presented in the previous chapters with additional development efforts and new tools (Figure 6.1).

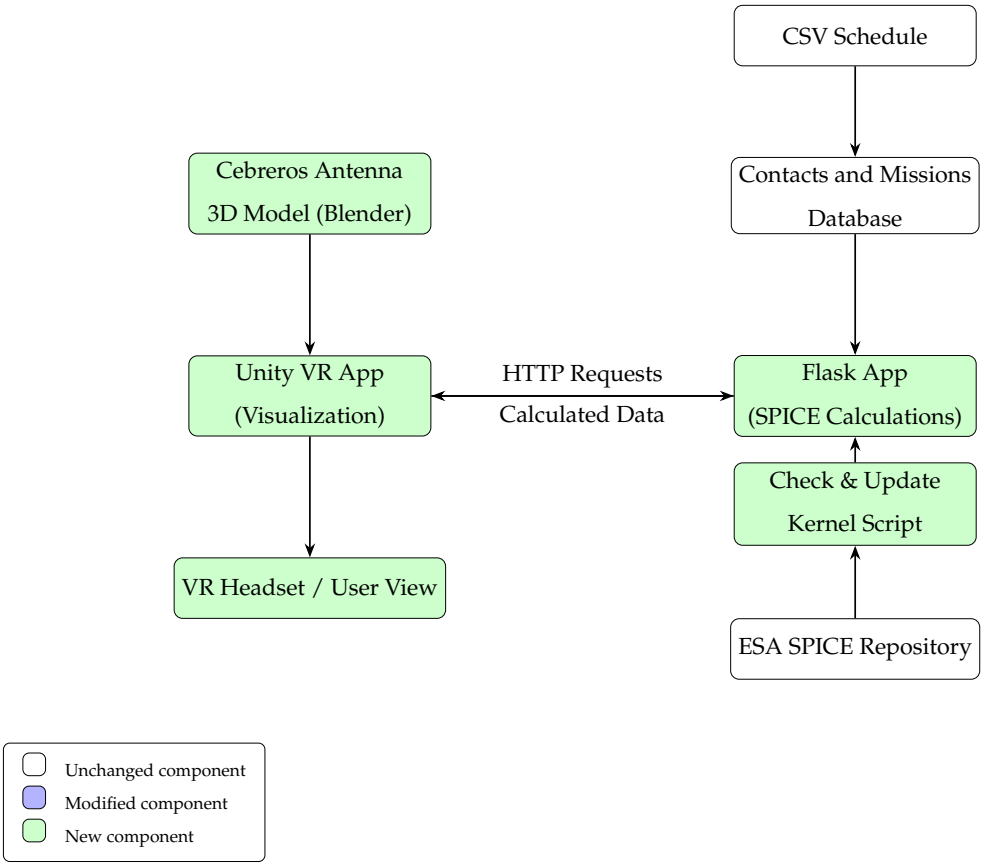


FIGURE 6.1: Architecture of the Virtual Reality App for Real-Time Visualization

I chose to build this application in Unity, as it is open-source and I had prior experience with the engine. It proved to be well-suited to the objectives of this project. The development began with the adaptation of the Cesium-based Cebreros visualization to a Unity Virtual Reality environment, aiming to create a realistic and immersive representation of the antenna and the tracked spacecraft. This view was then expanded by introducing a second scene that provides a broader perspective of the Solar System, displaying the spacecraft in their actual positions.

The system architecture is illustrated in Figure 6.2, and its physical setup is presented in Figure 6.3.

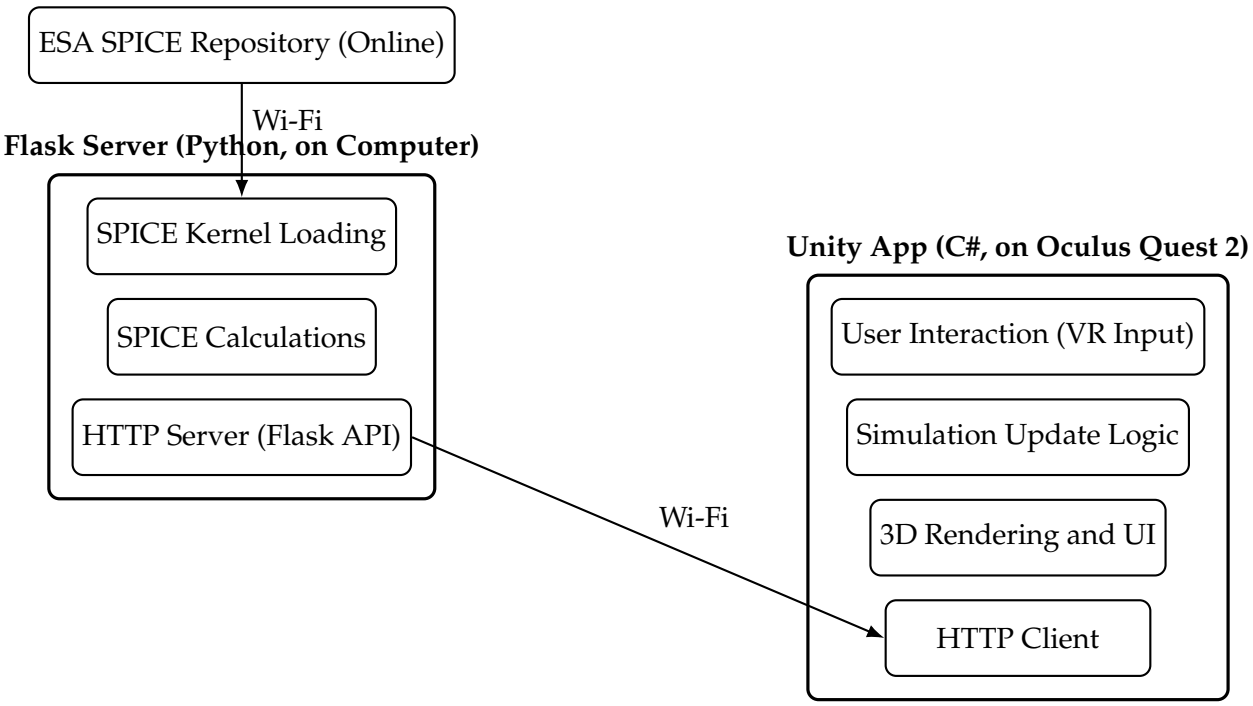


FIGURE 6.2: System Architecture

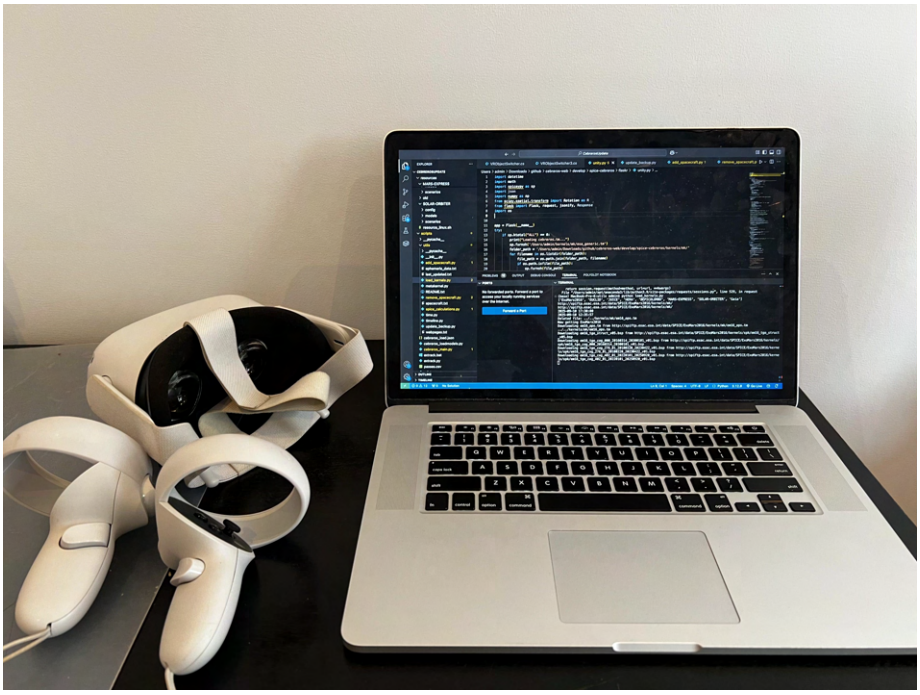


FIGURE 6.3: System Setup

6.1 Creating a VR Cebreros View

In order to begin creating a view of the Cebreros station in Unity, I first downloaded a sample project for Cesium in Unity [22], which enabled the use of Cesium functionalities within the engine to display the area surrounding the Cebreros station. This sample project includes example scenes featuring a `CesiumGeoreference` object, which specifies the location on Earth being rendered by defining the origin coordinates of the georeference. To display the intended view of Cebreros, I replaced the default coordinates in the sample with the actual geographic coordinates of the station.

In order to create a Virtual Reality scene suitable for the Oculus Quest 2, the VR headset targeted for this application, I installed the `OpenXR Plugin` and `Oculus XR Plugin` packages in Unity. These packages provide the necessary interface between Unity and the Oculus hardware, ensuring proper tracking and rendering for the headset.

Additionally, I added two essential GameObjects to the scene: the `XR Origin` and the `XR Interaction Manager`. The `XR Origin` serves as the root for the player's position and orientation in the virtual environment, handling head and hand tracking by managing the camera and input controllers relative to a defined origin point. The `XR Interaction Manager` is responsible for coordinating interactions within the VR environment, facilitating communication between interactable objects and the user's input devices by enabling

features like object grabbing, selection, and UI interaction.

Next, in order to add my antenna model to the scene, I installed the `GLTFUtility` package from GitHub [23]. This package enabled the import of my antenna model in `glTF` format into Unity, preserving its structural hierarchy. Maintaining these partitions was essential for enabling independent rotation of different components in later development stages. Once imported, the model was added to the scene and positioned correctly on the terrain of the Cebreros station.

I also added game objects to the Unity scene for each of the tracked spacecraft. Each of these game objects contained two child objects:

- A label, created using a TextMesh component, displaying the name (or an abbreviated name) of the spacecraft (e.g., `MPO` for BepiColombo);
- A 3D model of the spacecraft, imported from a previously developed ESA Unity project.

This initial setup can be seen in Figure 6.4.

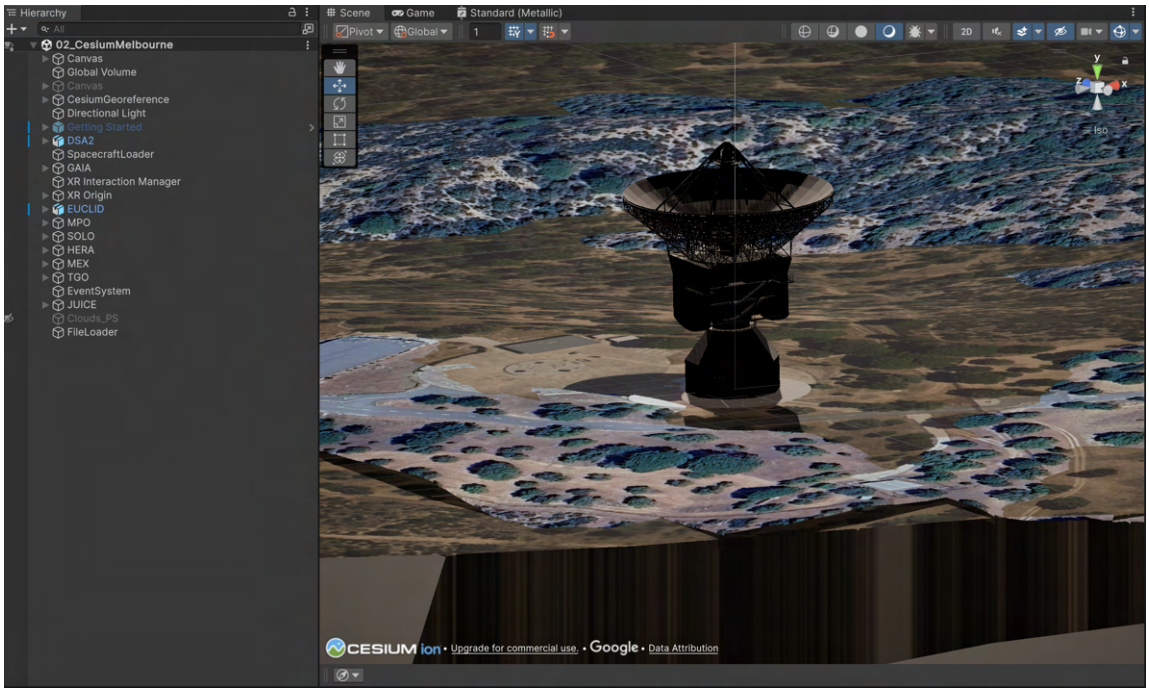


FIGURE 6.4: Initial Cebreros Station View Setup

With all the primary components set up in Unity, the next step was to implement a method for providing the antenna and spacecraft with the necessary values to simulate their real-life movements. Since there is no module that allows for the use of SPICE

kernels in Unity, and using web services like `WebGeoCalc` [24] proved to be too time-consuming for rendering the application, I created a Flask server that can be queried for specific SPICE calculations, whose results are then used to update the scene in Unity accordingly.

6.1.1 Flask Server Development

To create the Flask server for SPICE calculations, I adapted some of the code from the Cesium Display background Flask application, as much of the data required for the VR application overlaps with that used in the Cesium scene.

To support realistic tracking simulation, I programmed the Flask server to receive a date via HTTP request and compute the positions throughout that day of all spacecraft tracked by the Cebreros station, along with the antenna's corresponding azimuth and elevation values.

Given the density of algorithms in this subsection, Listing 6.1 serves as a guide to the relationships among the algorithms presented.

```
Flask Server
└── main() (preparedata() + generatedata()) (Algorithms 19&20)
    ├── getpositions() (Algorithm 21)
    ├── generatefile() (Algorithm 22)
    │   └── generatecartesians() (Algorithm 23)
    ├── antenna() (Algorithm 24)
    │   ├── idleadapter() (Algorithm 25)
    │   ├── getidledata() (Algorithm 26)
    │   ├── contactadapter() (Algorithm 27)
    │   └── getgroundstationdata() (Algorithm 28)
    └── generateantennafile() (Algorithm 29)
```

LISTING 6.1: Flask Server structure

The main server routine is split below into two pseudocode blocks (Algorithms 19 and 20).

Algorithm 19 Prepare spacecraft and antenna input data

```
1: function PREPAREDATA(step_size=60)
2:     date ← request.args.get('date')
3:     start_time ← CONVERTOISO(date)
4:     start_date ← PARSE(start_time)
5:     end_date ← start_date + 1 day
6:     ets ← GENERATETIMESTEPS(start, start+1, step_size)
7:     missions ← query all missions from database
8:     contacts ← query all contacts between start_date and end_date from database
9:     GENERATEDATA(start_time, start_date, end_date, step_size, missions, ets,
   contacts)
10: end function
```

Algorithm 20 Generate tracking and spacecraft output data

```
1: function GENERATEDATA(date, start_time, start_date, end_date, step_size, missions,
   ets, contacts)
2:     for mission in missions do
3:         pos ← GETPOSITIONS(mission.spacecraft, ets)
4:         filename ← 'sc_{spacecraft}_{date}.txt'
5:         GENERATEFILE(mission.spacecraft, pos, step_size, filename)
6:     end for
7:     (antenna_times, azimuths, elevations, contact_list) ← ANTENNA(contacts,
   step_size, start_date, end_date)
8:     GENERATEANTENNAFILE(antenna_times, azimuths, elevations, contact_list,
   date)
9:     contents ← {}
10:    for mission in missions do
11:        file_name ← 'sc_{spacecraft}_{date}.txt'
12:        f_sc ← OPENFILE(file_name)
13:        contents[file_name] ← GETCONTENTS(f_sc)
14:    end for
15:    file_az ← 'azimuths_{date}.txt'
16:    file_el ← 'elevations_{date}.txt'
17:    f_az ← OPENFILE(file_az)
18:    f_el ← OPENFILE(file_el)
19:    contents[file_az] ← GETCONTENTS(f_az)
20:    contents[file_el] ← GETCONTENTS(f_el)
21:    return JSONIFY(contents)
22: end function
```

The Flask server is configured to handle HTTP `GET` requests that include a query parameter: `date`. This parameter specifies the day for which data should be generated. Upon receiving a request, the server returns a JSON object containing a dictionary where the keys are filenames — formatted as `sc_{spacecraft}_{date}.txt` for spacecraft positional data, or `azimuth_{date}.txt` and `elevation_{date}.txt` for antenna rotation data— and the values contain the corresponding data for the specified date.

Upon receiving an HTTP request, the script first checks whether the required argument is present by using the function `request.args.get('date')` (Algorithm 19, lines 2).

The script then converts the date from its original `YYYYMMDD` format into an ISO 8601 date-time string in the format `YYYY-MM-DDT00:00:00Z`. It then defines the end of the requested interval by incrementing the received date by one day, ensuring that only data from the specified day is retrieved. The time resolution of the collected data is determined by the step size of 60 seconds (Algorithm 19, lines 3–5).

Subsequently, the script queries all existing Cebreros missions and retrieves all contact windows occurring during the requested day from the local `cebreros` database (Algorithm 19, lines 7–8).

After preparing all the necessary data, the script proceeds to construct the spacecraft positional data files by calling first the `getpositions()` function, followed by the `generatefile()` function for each mission (Algorithm 20, lines 3–7). The latter function internally relies on an additional auxiliary function: `generatecartesians()`. The definitions of these three functions are provided in Algorithms 22, 21, and 23, respectively.

Algorithm 21 Get spacecraft positions in Unity coordinate system

```
1:  function GETPOSITIONS(name, ets, unity_origin_ecef=CERBREROS_COORDINATES)
2:      positions, _ ← SPKPOS(name, ets, 'IAU_EARTH', 'NONE', 'EARTH')
3:      unity_up ← NORMALIZE(unity_origin_ecef)
4:      ecef_up ← [0, 0, 1]
5:      ecef_x ← [1, 0, 0]
6:      unity_east ← NORMALIZE(cross(ecef_up, unity_up))
7:      unity_north ← NORMALIZE(cross(unity_up, unity_east))
8:      base_change_matrix ← NEWMATRIXFROMBASE(unity_east, unity_north,
    unity_up)
9:      relative_positions ← positions − unity_origin_ecef
10:     rotated_positions ← DOT(relative_positions,transpose(base_change_matrix))
11:     SWAPMATRIXCOLUMNS(rotated_positions, 2, 3)
12:     return rotated_positions
13: end function
```

Algorithm 22 Generate data file

```
1: function GENERATEFILE(name, positions, step_size, filename)
2:     cartesians ← GENERATECARTESIANS(positions, step_size)
3:     f ← OPENFILE(filename)
4:     f.Write(name)
5:     for t in range(0,length(cartesians)) do
6:         f.Write(cartesians[t])
7:     end for
8:     return
9: end function
```

Algorithm 23 Generate Cartesian coordinates for spacecraft positions

```
1: function GENERATECARTESIANS(positions, step_size=60, scale=10^3)
2:     cartesians ← []
3:     for index, position in enumerate(positions) do
4:         ct ← index * step_size
5:         cartesians.append(ct)
6:         cartesians.append(position[0] * scale)
7:         cartesians.append(position[1] * scale)
8:         cartesians.append(position[2] * scale)
9:     end for
10:    return cartesians
11: end function
```

In Algorithm 21, and for each mission, we retrieve the corresponding spacecraft name and the list of Eastern Time (ET) timestamps (`ets`) and compute the positions of the spacecraft relative to Earth using `sp.spkpos(spacecraft, ets, 'IAU_EARTH', 'NONE', 'EARTH')` (Algorithm 21, Line 2). Then, it converts each of these positions to Unity's reference frame, centered at the Cebreros station. This reference frame is left-handed, with X to the right, Y up, and Z pointing forward (out of the screen).

The transformation involves:

- Using the Unity origin in Earth-Centered, Earth-Fixed (ECEF) reference frame (`CEBREROS_COORDINATES = [4846725, -370172.3, 4116859]`, the Cebreros Station

coordinates used in Unity) to compute the Unity *up* direction by normalizing the vector from the Earth's center to the station (Algorithm 21, Line 3);

- Calculating the Unity *east* direction as the cross product of the global ECEF up vector `[0, 0, 1]` and the Unity up vector (Algorithm 21, Line 6);
- Deriving the Unity *north* direction as the cross product of Unity up and Unity east (Algorithm 21, Line 7);
- Constructing a base-change matrix (Algorithm 21, Line 8):
  - Row 1: Unity east (new X-axis)
  - Row 2: Unity north (new Y-axis)
  - Row 3: Unity up (new Z-axis)
- Subtracting the Unity origin from all ECEF positions (Algorithm 21, Line 9);
- Applying the rotation matrix to get local coordinates (Algorithm 21, Line 10);
- Swapping Y and Z to follow Unity's left-handed system (Algorithm 21, Line 11).

Once the position vectors are obtained in the desired reference frame, the script flattens them into a list of numerical values. Each component is converted from kilometers to meters by multiplying by 10^3. If a time step is specified (in this case, 60 seconds), a corresponding time offset is added to each entry. The final list is organized into groups of four rows: the first row contains the time offset from the beginning of the day, followed by three rows with the spacecraft's x, y, and z coordinates at the corresponding time. (Algorithm 23).

In Algorithm 22, the script opens an output file named `sc_{name}_{date}.txt` and writes the spacecraft name as the first line (Algorithm 22, lines 2–4). It then iterates through the list of Cartesian values, writing one value per line (Algorithm 22, lines 5–7). After all values have been written, the output file is closed.

An example of the output file generated at this stage, for ESA's BepiColombo spacecraft (also known as MPO), is shown below (Listing 6.2).

```
MPO
0
-12130497528.594152
-202576638651.9123
71267737378.94995
60
-11278059674.920973
-202615112841.2239
71301655985.96262
```

LISTING 6.2: Excerpt from a MPO's positional data file

After generating all spacecraft positional data files, the script proceeds to build the files containing antenna azimuth and elevation data. This process begins with a call to the `antenna()` function, which in turn relies on four auxiliary functions: `idleadapter()`, `getidledata()`, `contactadapter()`, and `getgroundstationdata()`. The definitions of these five functions are presented in Algorithms 24–28.

Algorithm 24 Generate antenna rotation data files

```
1: function ANTENNA(contacts, step_size, start_date, end_date)
2:     antenna_times ← []
3:     azimuths ← []
4:     elevations ← []
5:     contact_list ← []
6:     idle_start ← start_date
7:     for contact in contacts do
8:         (needs_idle, idle_start_str, nsteps) ← IDLEADAPTER(idle_start,
   contact.start, step_size)
9:         if needs_idle then
10:            (times, azs, els) ← GETIDLEDATA(idle_start_str, nsteps, step_size)
11:            antenna_times.append(times)
12:            azimuths.append(azs)
13:            elevations.append(els)
14:        end if
15:        (cstart, cend, nsteps, mission) ← CONTACTADAPTER(contact, start_date,
   end_date, step_size)
16:        (times, azs, els, _) ← GETGROUNDSTATIONDATA(mission.spacecraft, 'CE-
   BREROS', cstart, nsteps, step_size, 0) antenna_times.append(times)
17:        azimuths.append(azs)
18:        elevations.append(els)
19:        contact_list.append((mission.spacecraft, times))
20:        idle_start ← PARSE(cend)
21:    end for
22:    (needs_idle, idle_start_str, nsteps) ← IDLEADAPTER(idle_start, end_date,
   step_size)
23:    if needs_idle then
24:        (times, azs, els) ← GETIDLEDATA(idle_start_str, nsteps, step_size)
25:        antenna_times.append(times)
26:        azimuths.append(azs)
27:        elevations.append(els)
28:    end if
29:    return (antenna_times, azimuths, elevations, contact_list)
30: end function
```

Algorithm 25 Check if antenna is idle

```
function IDLEADAPTER(idle_start, contact_start, step_size, offset=10 minutes)
    if idle_start ≥ contact_start then
        return (False, None, None)
    end if
    corr_idle_start ← idle_start + offset
    corr_idle_end ← contact_start - offset
    corr_start ← CONVERTTOISO(corr_idle_start)
    duration ← TOTALSECONDS(corr_idle_end - corr_idle_start)
    nsteps ← FLOOR(durationstep_size)
    return (True, corr_start, nsteps)
end function
```

Algorithm 26 Generate Idle Azimuth and Elevation Values

```
function GETIDLEDATA(utc_time, nsteps, step_size, az_idle=0, el_idle=90)
    et ← CONVERTTOET(utc_time)
    azimuths ← []
    elevations ← []
    times ← []
    for step in range(0,nsteps) do
        ct ← et + (step ∗ step_size)
        az ← az_idle
        el ← el_idle
        azimuths.append(az)
        elevations.append(el)
        ct_utc ← CONVERTTOUTC(ct)
        times.append(ct_utc)
    end for
    return (times, azimuths, elevations)
end function
```

Algorithm 27 Adapt Contact Interval to Requested Date

```
1: function CONTACTADAPTER(contact, start_date, end_date, step_size)
2:     if start_date < contact.start then
3:         contact_start ← contact.start
4:     else
5:         contact_start ← start_date
6:     end if
7:     if end_date < contact.end then
8:         contact_end ← contact.end
9:     else
10:        contact_end ← end_date
11:    end if
12:    duration ← TOTALSECONDS(contact_end - contact_start)
13:    nsteps ← FLOOR(duration)/step_size
14:    cstart ← CONVERTTOISO(contact_start)
15:    cend ← CONVERTTOISO(contact_end)
16:    return (cstart, cend, nsteps, contact.mission)
17: end function
```

Algorithm 28 Generate Ground Station Azimuth and Elevation Values

```
1: function GETGROUNDSTATIONDATA(sc_id, gs_name, utc_time, nsteps, step_size,
   min_elev=0)
2:     et ← CONVERTTOET(utc_time)
3:     azimuths ← []
4:     elevations ← []
5:     times ← []
6:     for step in range(0,nsteps) do
7:         ct ← et + (step * step_size)
8:         pos, _ ← SPKPOS(sc_id, ct, gs_name + '_TOPO', 'NONE', gs_name)
9:         _, lat, lon ← RECLAT(pos)
10:        lat ← RADTODEGREES(lat)
11:        lon ← RADTODEGREES(lon)
12:        azimuths.append(lat)
13:        elevations.append(max(lon, min_elev))
14:        ct_utc ← CONVERTTOUTC(ct)
15:        times.append(ct_utc)
16:    end for
17:    return (times, azimuths, elevations)
18: end function
```

In Algorithm 25, for each contact established by DSA-2 on the selected date, we first determine whether an idle period exists between the end of one contact and the start of the next (or, in the case of the first contact, between the start of the day and the beginning of the contact). If such an idle interval is found, the script generates intermediate time steps using the specified step size to fill the gap.

Algorithm 26 comes into action when an idle period is present. In this case, its boundaries are adjusted to exclude the first and last 10 minutes. The script then calculates the number of time steps that fit within this adjusted idle duration and it computes the timestamps starting from the beginning of the idle interval, incrementing by the defined step size until reaching the end of the interval. For each of these timestamps, the script registers the time and assigns corresponding azimuth and elevation values. Since the antenna is idle during this interval, the azimuth is consistently set to 0 degrees and the elevation to 90 degrees (representing the model pointing directly upward).

Next, Algorithm 27 is used to calculate the number of time steps that fit within the current contact interval. Then, finally, Algorithm 28 calculates, for each timestamp—starting from the beginning of the contact and incrementing by the defined step size until reaching the end of the contact period—the position of the spacecraft with respect to the Cebreros ground station (Algorithm 28, Line 8). From these positions, the script computes the corresponding latitude and longitude, and then converts them to degrees. Finally, it records each timestamp along with the corresponding azimuth (latitude) and elevation (longitude) values (Algorithm 28, lines 9–15).

After calculating and recording the times and azimuth and elevation values from the beginning of the day until the end of the last contact, the script checks if there is still idle time left between the end of the last contact and the end of the day, and if so retrieves the corresponding idle times, azimuths and antenna values (Algorithm 24, lines 22–28).

Once the antenna's rotational data is all collected for the requested date, the script calls one last auxiliary function to generate two files from this data: `generateantennafile()`. This function is defined in Algorithm 29.

Algorithm 29 Generate azimuth and elevation files

```
1:  function GENERATEANTENNAFILE(antenna_times, azimuths, elevations, contact_list,
    date)
2:      antenna_elevations ← [−(90−el) | for el in elevations]
3:      antenna_azimuths ← [az if az > 0 else az +360 | for az in azimuths]
4:      azimuths_file ← OPENFILE('azimuths_{date}.txt')
5:      elevations_file ← OPENFILE('elevations_{date}.txt')
6:      for each index t from 0 to length(times) − 1 do
7:          azimuths_file.Write(times[t])
8:          azimuths_file.Write(antenna_azimuths[t])
9:          elevations_file.Write(times[t])
10:         elevations_file.Write(antenna_elevations[t])
11:     end for
12:     return
13: end function
```

This function writes all the previously calculated azimuth and elevation values into two text files, adjusted to accurately reflect the tracking motion of the real antenna in

Cesium (Algorithm 29).

These files are structured as follows:

```
2025-03-03T00:00:00Z -41.70571136993129
2025-03-03T00:01:00Z -41.89560171831371
2025-03-03T00:02:00Z -42.08554970671231
2025-03-03T00:03:00Z -42.275553035895406
2025-03-03T00:04:00Z -42.465609434313684
2025-03-03T00:05:00Z -42.655716658438074
```

LISTING 6.3: Excerpt from an elevations file

Then, finally, the script initializes an empty dictionary and, for each previously created data file, adds an entry with the filename as the key and its contents as the value. It then converts this dictionary to a JSON object and returns it (Algorithm 20, lines 9–21).

6.1.2 Unity-Side Data Integration

Once the server was operational, I implemented Unity scripts to fetch and apply the data within the Cebreros environment.

I began by developing a script, `LoadFiles.cs`, responsible for managing the retrieval and storage of external data within Unity. This script performs three main tasks:

- It clears Unity's *Persistent Data Path* by deleting any previously stored files. This prevents the accumulation of outdated data files, ensuring that storage is used efficiently and only relevant files are retained;
- It initiates a coroutine that executes once per day, first when the scene is loaded and subsequently at every midnight for as long as the application remains active. This coroutine constructs an HTTP request to the Flask server, using the current date embedded in the URL. Upon receiving a successful response, it parses the returned JSON dictionary and, for each key–value pair, creates a corresponding file (with the key as the filename and the value as the file contents), which it stores in the *Persistent Data Path*;
- Once all files have been retrieved and written, the script proceeds to initialize two additional Unity scripts: `SpacecraftLoader.cs` and `AntennaUpdater.cs`. These

scripts use the newly stored data to load the spacecraft into the scene and rotate the antenna in real time to match the orientation of the actual antenna at Cebreros.

The `SpacecraftLoader.cs` script was developed to load the spacecraft models into their correct positions in the Cebreros sky. These positions are continuously updated to reflect the real-time locations of the spacecraft, as defined in the positional data files, which contain coordinates for every minute of the day.

Listing 6.4 presents a concise outline of the script, showing the interconnections between the algorithms described in this subsection to aid comprehension.

```
SpacecraftLoader.cs
└── LoadSpacecraft() (Algorithm 30)
    ├── TryParsePosition()
    ├── FindStartIndex()
    ├── UpdatePosition() (Algorithm 31)
    └── AddLabelRotationUpdater()
```

LISTING 6.4: SpacecraftLoader.cs structure

This script's main function is defined in Algorithm 30.

Algorithm 30 Load spacecraft in scene

```
1: function LOADSPACECRAFT(targets, currentDay, spacecraftObjects)
2:     for target in targets do
3:         dataPath ← PersistentDataPath
4:         filePath ← First file in dataPath matching sc_{target}_{currentDay}.txt
5:         lines ← ReadLines(filePath)
6:         scName ← Trim(lines[0])
7:         if scName not in spacecraftObjects then
8:             spacecraftObject ← FIND(scName)
9:             if EXISTS(spacecraftObject) then
10:                modelTransform ← FindChild(spacecraftObject, 'model')
11:                labelTransform ← FindChild(spacecraftObject, 'label')
12:                if modelTransform and labelTransform exist then
13:                    model ← modelTransform.gameObject
14:                    label ← labelTransform.gameObject
15:                    SETACTIVE(model, false)
16:                    SETACTIVE(label, true)
17:                    times ← empty list
18:                    positions ← empty list
19:                    if TRYPARSEPOSITION(lines, scaleFactor, times, positions) then
20:                        startIndex ← FINDSTARTINDEX(times)
21:                        STARTCOROUTINE(UpdatePosition(label, times, positions,
   startIndex))
22:                        ADDLABELROTATIONUPDATER(label)
23:                        spacecraftObjects[scName] ← spacecraftObject
24:                    end if
25:                end if
26:            end if
27:        end if
28:    end for
29: end function
```

For each spacecraft tracked at Cebreros, Algorithm 30 begins by retrieving the corresponding positional data file and extracting the spacecraft name from the first line. If this spacecraft has not yet been loaded (i.e., it has not been added to the `spacecraftObjects` dictionary), the script searches the scene for a `GameObject` with a matching name. If the object is found, the script locates its children, specifically the spacecraft model and its label. If both are present, the model is deactivated and the label is activated, so only the label remains visible (Algorithm 30, lines 7–16).

Next, the script initializes two lists: `times` and `positions`. It then calls the function `TryParsePosition()`, which parses the spacecraft's positional data from the file. This function applies appropriate scaling for visualization in Unity by setting a target distance of $1.5 * 10^5$ units and computing the necessary scaling factor as `targetDistance / originalDistance`. The resulting time values and position vectors are then stored in the `times` and `positions` lists (Algorithm 30, Line 19).

If the parsing is successful, the script calls the function `FindStartIndex()` to identify the value in the `times` list that is closest to the current time (measured in seconds since midnight). This value's index is then used to determine the starting point in the `positions` list from which the spacecraft's position should begin being updated. The two lists are one-to-one mapped, meaning that each position vector in the `positions` list corresponds to the time at the same index in the `times` list (Algorithm 30, Line 20).

Once the starting index is identified, the script starts a coroutine by calling the `UpdatePosition()` function, which, for the remainder of the day, updates the position of the spacecraft every minute. The implementation of this function is presented in Algorithm 31.

Algorithm 31 Update spacecraft position in real time

```
1:  function UPDATEPOSITION(object, times, positions, startIndex)
2:      for i in range(startIndex, |times| −1) do
3:          duration ← times[i + 1] − times[i]
4:          start ← positions[i]
5:          end ← positions[i + 1]
6:          elapsedTime ← 0
7:          while elapsedTime < duration do
8:              elapsedTime ← elapsedTime + DELTATIME
9:              t ← elapsedTime / duration
10:             interpolatedPosition ← LERP(start, end, t)
11:             object.position ← interpolatedPosition
12:         end while
13:         object.position ← end
14:     end for
15: end function
```

In Algorithm 31, for each consecutive pair of time-position values in the list, the function calculates the duration between them and performs a smooth linear interpolation (using Unity's `Vector3.Lerp`) between the start and end positions over the course of that interval. The interpolation is updated every frame using `Time.deltaTime`, ensuring a smooth transition between positions in sync with Unity's frame updates.

During each interpolation segment, the spacecraft object is moved continuously along the interpolated path. At the end of each segment, it is snapped exactly to the final position before continuing to the next segment. This update process continues until the final position in the list is reached. This results in the spacecraft smoothly following its predicted trajectory in the Unity scene, updated every minute, in sync with the real spacecraft's movement in the sky.

Back in Algorithm 30, once the coroutine is running, the script invokes an auxiliary function, `AddLabelRotationUpdater()`, which ensures that the label continuously faces the camera, improving its readability (Algorithm 30, Line 22).

Finally, once the spacecraft is loaded and correctly positioned, the script adds it to the `spacecraftObjects` dictionary using its name as the key, thereby registering that it has finished loading (Algorithm 30, Line 23).

While `SpacecraftLoader.cs` is running, a separate script, `AntennaUpdater.cs`, executes in parallel.

This script's internal functions and their relationships are outlined in listing 6.5 for a clearer understanding of the algorithms presented in the remainder of this subsection.

```
AntennaUpdater.cs
└── LoadRotations() (Algorithm 32)
    └── SmoothRotationRoutine() (Algorithm 33)
```

LISTING 6.5: AntennaUpdater.cs structure

The implementation of its main function is presented in Algorithm 32.

Algorithm 32 Load Antenna Rotation Files

```
1: function LOADROTATIONS(rotationType, initialRotation, initialAzimuth, initialElevation)
2:     schedule ← {}
3:     currentTime ← GETCURRENTUTCTIME( )
4:     currentIndex ← 0
5:     currentRotation ← 0
6:     for line in rotationFile do
7:         time, rotation ← line.split(' ')
8:         schedule.Add(time, rotation)
9:     end for
10:    for i in range(0,length(schedule) - 1) do
11:        if schedule[i].time ≤ currentTime < schedule[i+1].time then
12:            currentIndex ← i
13:            break
14:        end if
15:    end for
16:    if currentIndex < |schedule| - 1 then
17:        if rotationType == 'azimuths' then
18:            currentRotation ← 360 - schedule[currentIndex].value
19:            antennaBase.rotation ←EULER(0, currentRotation + initialAzimuth, 0)
20:        else
21:            currentRotation ← schedule[currentIndex].value
22:            antennaDisk.rotation ←EULER(0, 0, currentRotation + initialElevation)
23:        end if
24:    end if
25:    STARTCOROUTINE(SmoothRotationRoutine(rotationType,currentIndex,
       currentRotation,schedule))
26: end function
```

For each of the antenna rotation data files, one containing azimuth values and the other elevation values, Algorithm 32 performs the following steps:

- It begins by parsing the contents of the data file into a list of key–value pairs, where

each key represents a specific time of day and each value corresponds to the antenna's rotation angle (either azimuth or elevation, depending on the file being processed) (Algorithm 32, lines 6–9).

- The script then identifies the time interval in the schedule that contains the current UTC time. Specifically, it finds the most recent time entry in the schedule that precedes the current time of day. Based on this match, it determines the appropriate rotation angle for the current time. If the file corresponds to azimuth data, the value is inverted (by subtracting it from 360) and added to the initial azimuth offset before being applied to the antenna base. If the file corresponds to elevation data, the value is applied directly with the initial elevation offset to the antenna disk (Algorithm 32, lines 10–24).

- The script then finally starts a coroutine, calling an auxiliary function, `SmoothRotationRoutine()`, which updates the antenna rotations consistently throughout the rest of the day. This function is defined in Algorithm 33.

Algorithm 33 Smooth Rotation Routine

```
1: function SMOOTHROTATIONROUTINE(rotationType, currentIndex, currentRotation, schedule)
2:     for i in range(currentIndex,length(schedule) - 1) do
3:         startTime ← schedule[i].time
4:         endTime ← schedule[i+1].time
5:         if rotationType == 'azimuths' then
6:             startRotation ← 360− schedule[i].value
7:             endRotation ← 360− schedule[i+1].value
8:         else
9:             startRotation ← schedule[i].value
10:            endRotation ← schedule[i+1].value
11:        end if
12:        duration ← SECONDS(endTime - startTime)
13:        elapsed ← 0
14:        while elapsed < duration do
15:            elapsed ← elapsed + DELTATIME
16:            t ← elapsed / duration
17:            interpolatedRotation ← LERP(startRotation, endRotation, t)
18:            if rotationType == 'azimuths' then
19:                antennaBase.rotation ← EULER(0, interpolatedRotation + initialAzimuth, 0)
20:            else
21:                antennaDisk.rotation ←EULER(0, 0, - (interpolatedRotation + initialElevation))
22:            end if
23:            currentRotation ← interpolatedRotation
24:        end while
25:        if rotationType == 'azimuths' then
26:            antennaBase.rotation ←EULER(0, endRotation + initialAzimuth, 0)
27:        else
28:            antennaDisk.rotation ←EULER(0, 0, - (endRotation + initialElevation))
29:        end if
30:        currentRotation ← endRotation
31:        currentIndex ← i + 1
32:    end for
33: end function
```

For each consecutive pair of time-rotation values in the list, the function smoothly applies rotation *interpolation* between the current and next rotation values. It does this by:

- Retrieving the time span between the two scheduled entries (Algorithm 33, Line 12);
- Determining the start and end rotation angles, reversing the values in the case of azimuths (to match the coordinate system) (Algorithm 33, Line 5–11);
- Gradually interpolating the rotation using a linear interpolation function `Lerp()`, based on the elapsed time fraction (Algorithm 33, Line 17);
- Applying the interpolated value at each frame using Unity's `DeltaTime()` and updating the antenna transform accordingly (Algorithm 33, Line 14–24);
- Ensuring that at the end of each interval, the exact target rotation is applied to avoid interpolation errors (Algorithm 33, Line 23);
- Updating the `currentRotation` and `currentIndex` variables, before moving on to the next interval in the schedule (Algorithm 33, Line 29–31).

This coroutine continues looping through the schedule, ensuring smooth transitions throughout the day as time progresses, whether it is controlling the azimuth or elevation of the antenna.

These Unity-side scripts, together with the Flask server, enabled the creation of an accurate simulation of the Cebreros station view, featuring real-time antenna rotation and spacecraft movement.

6.1.3 Interactability in VR

Once the Cebreros view was complete, the final step was to make it interactive for users in the Oculus Quest. To achieve this, I developed a script called `VRObjectSwitcher.cs`, whose main function is defined in Algorithm 34.

Algorithm 34 VR Object Switcher Script

```
1:  function ONSELECTED(model, label, camera, distanceFromCamera)
2:      if ISACTIVE(model) then
3:          DISABLE(model)
4:          ENABLE(label)
5:      else
6:          positionInFront ← camera.position+camera.forwardDir*distanceFromCamera
7:          model.position ← positionInFront
8:          model.rotation ← LOOKROTATION(camera.forwardDir)
9:          DISABLE(label)
10:         ENABLE(model)
11:     end if
12: end function
```

Algorithm 34 allows users to click on a spacecraft label to display its corresponding 3D model. The label is hidden, and the model appears in front of the user at a larger scale, slowly rotating to allow close inspection (Algorithm 34, lines 5–11). If the user clicks on the model again, the model disappears and the label reappears (Algorithm 34, lines 2–4). This interaction enables users to identify spacecraft in the sky and visualize what they look like up close.

To detect input from the Oculus Quest controllers, I integrated Unity's XR interaction system. Specifically, I added the `XRSimpleInteractable.cs` component from Unity's XR Interaction Toolkit to each clickable object. I then linked the `Select` event in the Inspector to my custom script. This setup allows the system to detect user selections and trigger the corresponding logic in `VRObjectSwitcher.cs`.

6.2 Creating a Solar System View

To further expand the Virtual Reality application, I created a second scene focused on the Solar System.

This scene presents a visual representation of the planets and their orbits, with the spacecraft tracked by Cebreros shown in their actual positions in space. This allows users to gain a deeper understanding of the spacecraft's locations and distances, beyond simply observing their position in the sky from the Cebreros ground station.

To build the scene, I began by downloading the `Planets of the Solar System 3D` package from Unity's Asset Store [25]. This package includes 3D models of the planets and the Sun, along with shaders that enhance their visual resemblance to the celestial bodies they represent.

Next, I added the spacecraft GameObjects, each consisting of a label and a 3D model, just as I had done in the Cebreros view scene.

This initial setup can be seen in Figure 6.5.

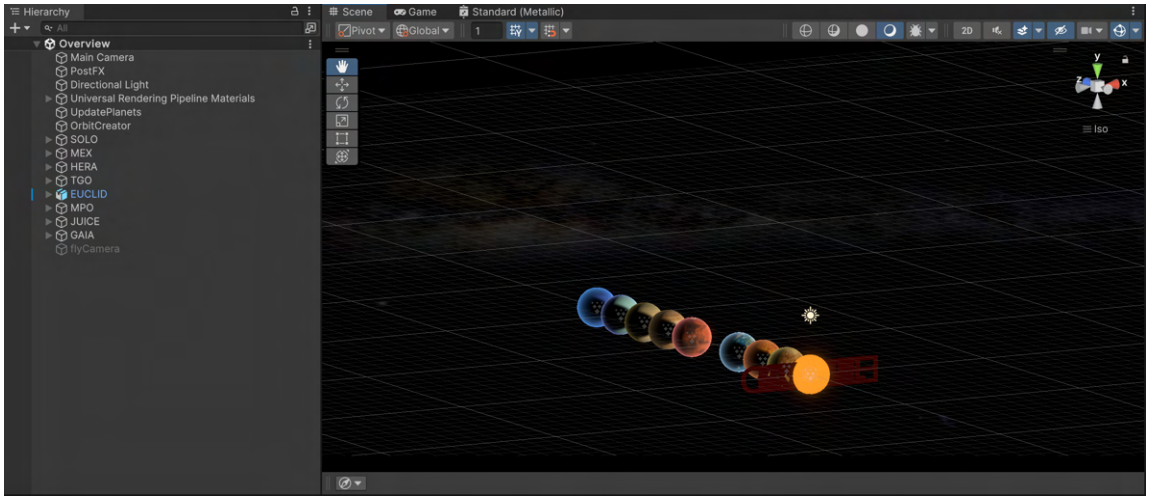


FIGURE 6.5: Initial Solar System View Setup

With all the main objects included in the scene, the next step was to position them accurately in space, relative to the Sun at the center, and ensure that their locations remained correct over time. To achieve this, I updated the Flask server to handle a second type of HTTP request, this one including a date and an indicator specifying that the data was intended for the Space View scene.

6.2.1 Flask Server Update

This update to the Flask server was designed to retrieve the positions of the planets and spacecraft relative to the Sun for a given date. Additionally, it computes the trajectories of each planet for visualization in the scene. As part of this, I also calculated Earth's velocity relative to the Sun to determine an optimal camera position, perpendicular to Earth's trajectory, so that both the orbits and the spacecraft could be clearly viewed from this perspective.

The updated version of the server's main function is shown in Algorithms 35 and 36. For simplicity, only the newly added section, which runs when the additional indicator is

present in the HTTP request, is included here, as the original implementation remains unchanged. The relationships between the functions implemented in this updated version are outlined in Listing 6.6

```
Flask Server Update
└── main() (GenerateSpaceData()) (Algorithm 35&36)
    ├── GetPositionsSpace() (Algorithm 37)
    ├── GenerateFile() (Algorithm 22)
    └── GetVelocity() (Algorithm 38)
```

LISTING 6.6: Flask Server Update structure

Algorithm 35 Generate planet and spacecraft output data

```
1: function GENERATESPACEDATA(date, start_time, start_date, end_date, step_size, missions, ets, planets, planets_barycenters, planet_orbit_years, orbit_start)
2:     if indicator is present then
3:         for mission in missions do
4:             pos ← GETPOSITIONSSPACE(mission.spacecraft, ets)
5:             filename ← 'sc_{mission.spacecraft}_{date}_space.txt'
6:             GENERATEFILE(mission.spacecraft, pos, step_size, filename)
7:         end for
8:         start_et ← orbit_start to ET time
9:         end_dates ← [start_et + years * SECONDS_PER_YEAR | for years in planet_orbit_years]
10:        trajectory_steps ← SECONDS_PER_MONTH / 2 * planet_orbit_years
11:        for i, planet in enumerate(planets) do
12:            pos_planet ← GETPOSITIONSSPACE(planet, ets)
13:            filename_planet ← '{planet}_{date}.txt'
14:            GENERATEFILE(planet, pos_planet, step_size, filename_planet)
15:            traj_ets ← GENERATETIMESTEPS(start_et,end_dates[i],trajectory_steps[i])
16:            pos_traj ← GETPOSITIONSSPACE(planets_barycenters[i], traj_ets)
17:            filename_traj ← '{planet}_trajectory.txt'
18:            GENERATEFILE(planet, pos_traj, trajectory_steps[i], filename_traj)
19:        end for
20:        vel ← GETVELOCITY(ets)
21:        filename ← 'Earthvelocity_{date}.txt'
22:        GENERATEFILE('Earthvelocity', vel, step_size, filename)
23:                                                    // Code continues in next algorithm
24:    end if
25: end function
```

Algorithm 36 Generate planet and spacecraft output data (continuation)

```
1: function GENERATESPACEDATA(date, start_time, start_date, end_date, step_size, missions, ets, planets, planets_barycenters, planet_orbit_years, orbit_start)
2:                                                          // Continuation of previous function
3:     contents ← {}
4:     for mission in missions do
5:         file_name ← 'sc_{spacecraft}_{date}_space.txt'
6:         f_sc ← OPENFILE(file_name)
7:         contents[file_name] ← GETCONTENT(f_sc)
8:     end for
9:     for planet in planets do
10:        file_planet ← '{planet}_{date}.txt'
11:        file_traj ← 'planettrajectory.txt'
12:        f_planet ← OPENFILE(file_planet)
13:        f_traj ← OPENFILE(file_traj)
14:        contents[file_planet] ← GETCONTENT(f_planet)
15:        contents[file_traj] ← GETCONTENT(f_traj)
16:    end for
17:    file_name ← 'Earthvelocity_{date}.txt'
18:    f_vel ← OPENFILE(file_name)
19:    contents[file_name] ← GETCONTENT(f_vel)
20:    return JSONIFY(contents)
21: end function
```

In this case, Algorithm 35 begins by generating the spacecraft positional data in the Solar System by calling the function `getpositionsspace()` defined below in Algorithm 37, for each mission tracked by the antenna.

Algorithm 37 Get spacecraft positions relative to the Sun

```
1: function GETPOSITIONSSPACE(name, ets)
2:     Initialize: positions as empty list
3:     for each et in ets do
4:         pos, _ ← SPKPOS(name, et, 'J2000', 'NONE', 'SUN')
5:         positions.append(pos)
6:     end for
7:     return positions
8: end function
```

Algorithm 37 uses SPICE toolkit tools to calculate each spacecraft's position relative to the Sun.

Algorithm 35 then generates a filename in the format `sc_{spacecraft}_{date}_space.txt` and calls the `generatefile()` function, previously defined in Algorithm 22, to save the data.

Next, it takes the defined start date for orbital calculations, `orbit_start`, and converts it to Eastern Time. It then constructs a list of end dates for each planet's orbital period by adding to the start date the number of seconds in a year multiplied by the number of years required for each planet to complete a full orbit (values given in the `planet_orbit_years` list). To determine the step size for trajectory calculations, the script assumes that for a planet completing one orbit per year, two position points per month are sufficient for accurate visualization. Thus, it sets the step size to two weeks (in seconds) times the number of years in each planet's orbital period (Algorithm 35, lines 8– 10).

For each planet in the Solar System, Algorithm 35 first calculates its position relative to the Sun by calling `getpositionsspace()`, defined in Algorithm 37, and saves the result using `generatefile()`, defined in Algorithm 22. It then generates a list of ET timestamps spanning the planet's full orbital period, sampled at intervals of `trajectory_steps` seconds. The orbital trajectory points are computed by once again calling the `getpositionsspace()` function and saving the output with `generatefile()` (Algorithm 35, lines 11– 19).

Finally, the script calculates the velocity of Earth relative to the Sun throughout the day by calling the `getvelocity()` function, defined below in Algorithm 38, and saves the resulting data using `generatefile()` (Algorithm 35, lines 20– 22).

Algorithm 38 Get Earth's velocities relative to the Sun

```
1: function GETVELOCITY(ets)
2:     Initialize: velocities as empty list
3:     for each et in ets do
4:         state, _ ← SPKEZR('EARTH', et, 'J2000', 'NONE', 'SUN')
5:         velocities.append(state[3:])
6:     end for
7:     return velocities
8: end function
```

All the files generated in this subsection follow the same structure as the spacecraft positional data files from the Cebreros View scene (Listing 6.2).

Once all the intended data files have been created, the script proceeds to initialize an empty dictionary, `contents`. For each previously created file, it reconstructs the filename and adds an entry to the dictionary with the filename as the key and its contents as the value. The dictionary is then converted to a JSON object and returned (Algorithm 36).

6.2.2 Unity-Side Data Integration

Once the updates to the Flask server were in place, I moved on to focus on the integration of the retrieved data in unity.

I started by updating the `LoadFiles.cs` script to generate a second HTTP request to generate the Solar System scene files. This second request includes not only the date argument, but also the indicator that it's these scene's files that are being requested. This script runs when the main scene (the Cebreros view) first opens, so when the scene change occurs between the two, the necessary files will already be loaded.

To load the spacecraft and place them correctly in the scene, I reused the `SpacecraftLoader.cs` script (whose main function is shown in Algorithm 30). In this case, the script retrieves data from the spacecraft positional data files for the Solar System view and calls the `UpdatePosition()` function on the entire spacecraft object, ensuring that both the label and the model are moved together.

To place the planets in their correct positions and keep them updated throughout the day, I created a similar script called `UpdatePlanets.cs`. Like `SpacecraftLoader.cs`, this script processes each planet by first constructing the path to its positional data file and extracting the planet's name from the file's first line. If the planet has not yet been loaded,

it locates the corresponding GameObject, initializes two lists, `times` and `positions`, and attempts to parse the file using `TryParsePosition()`. It then calls `FindStartIndex()` to find the timestamp closest to the current time and finally launches a coroutine with `UpdatePosition()`, defined in Algorithm 31, to continuously update the planet's position.

Once the spacecraft and the planets were loaded into the scene, I created a script, `OrbitVisualizer.cs`, to visualize the planet orbits in Unity. The main function of this script is defined in Algorithm 39.

Algorithm 39 Generate planet orbits

```
1:  function LoadAndDrawOrbits(planets, orbitMaterial, properties)
2:      for planet in planets do
3:          dataPath ← PersistentDataPath
4:          filePath ← First file in dataPath matching {planet}trajectory.txt
5:          lines ← ReadLines(filePath)
6:          positions ← []
7:          if TryParsePosition(lines, positions) then
8:              orbitObj ← NewGameObject('{planet}_Orbit')
9:              lr ← AddComponent(orbitObj, LineRenderer)
10:             lr.positionCount ← Count(positions)
11:             lr.SetPositions(positions.ToArray())
12:             lr.widthMultiplier ← properties.wmult
13:             lr.useWorldSpace ← properties.useWSpace
14:             lr.material ← properties.material
15:             lr.startColor ← properties.startColor
16:             lr.endColor ← properties.endColor
17:         end if
18:     end for
19: end function
```

For each planet, the script begins by constructing the path to the planet's orbital data file and reading its contents into the `lines` array. It then attempts to parse the data using the `TryParsePosition()` function. If the parsing is successful, a new GameObject is created for the planet's orbit, named `planet_Orbit`, and a `LineRenderer` component is added to it. The number of points on the line is set to match the number of parsed

positions, and `SetPositions()` is called to assign these positions to the line. Finally, the appearance of the orbit line is configured: the width is set to 0.05, world space is enabled (ensuring the orbit is positioned in the global coordinate system), the material is set to `orbitMaterial`, and the color is set to white from start to end.

These scripts, along with the updates to the Flask server, enabled the creation of a semi-realistic view of the Solar System, with spacecraft integrated into the scene and additional visual information representing the planets' orbits. To enhance the visual effect, I also added a graphical representation of the asteroid belt between Mars and Jupiter. While the positions of the planets and spacecraft are to scale, the planet sizes are not — all planets are rendered at the same size to ensure they remain clearly visible.

6.2.3 Interactability in VR

Once the scene was set up, I added interactive functionalities to create a more dynamic visualization.

I developed the `VRObjectSwitcher2.cs` script, which enables the user to click on any spacecraft label and view its model up close in the corresponding position. Its main function is presented in Algorithm 40 and calls an auxiliary function, `ResolveOverlap()`, shown in Algorithm 41.

Algorithm 40 VR Object Switcher Script

```
function ONSELECTED(model, label, camera, moveDistance, XRrig, planets, spacecraft, safeDistance, initialCamPos, initialCamRot)
    if ISACTIVE(label) then
        modelCenter ← GETCENTER(model)
        directionToLabel ← NORMALIZE(modelCenter - camera.position)
        targetCameraPosition ← modelCenter - directionToLabel*moveDistance
        offset ← targetCameraPosition - camera.position
        XRrig.position ← XRrig.position + offset
        lookDirection ← modelCenter - camera.position
        lookDirection.y ← 0
        XRrig.rotation ← LOOKROTATION(lookDirection)
        DISABLE(label)
        ENABLE(model)
     RESOLVEOVERLAPS(model, safeDistance)
    else
        for sc in spacecraft do
            m ← GETMODEL(sc)
            l ← GETLABEL(sc)
            m.localPosition ← (0,0,0)
            l.localPosition ← (0,0,0)
            DISABLE(m)
            ENABLE(l)
            XRrig.position ← initialCamPos
            XRrig.rotation ← initialCameRot
        end for
    end if
end function
```

Algorithm 41 Resolve Overlaps

```
1:  function RESOLVEOVERLAPS(model, safeDistance, planets, spacecraft)
2:      for planet in planets do
3:          if INTERSECTS(model, planet) then
4:              pushDirection ← NORMALIZE(model.position - planet.position)
5:              while INTERSECTS(model, planet) do
6:                  model.position ← pushDirection * (safeDistance/2)
7:              end while
8:          end if
9:      end for
10:     for sc in spacecraft do
11:         m ← GETMODEL(sc)
12:         l ← GETLABEL(sc)
13:         if ISACTIVE(l) and INTERSECTS(l, label) then
14:             pushDirection ← NORMALIZE(model.position - l.position)
15:             while INTERSECTS(model, l) do
16:                 model.position ← pushDirection * (safeDistance/2)
17:             end while
18:         end if
19:         if ISACTIVE(m) and INTERSECTS(m, label) then
20:             pushDirection ← NORMALIZE(model.position - m.position)
21:             while INTERSECTS(model, m) do
22:                 model.position ← pushDirection * (safeDistance/2)
23:             end while
24:         end if
25:     end for
26: end function
```

Algorithm 40 is triggered when a spacecraft label is selected in the Oculus. It moves the camera to a closer position facing the selected spacecraft, makes the spacecraft model visible, and disables the label (Algorithm 40, lines 2–12). The auxiliary function `ResolveOverlaps()` is then called to check whether the selected model is overlapped by another object in the scene, such as a planet or another spacecraft, and to slightly reposition it until the overlap

is resolved. Such overlaps occur because the planets and spacecraft are not shown to scale (Algorithm 41).

When a model is selected, it also rotates slowly to provide the user with a better view.

If the same model is selected again, the label is re-enabled, the model is hidden, and the camera returns to its original position. The label and model are also restored to their initial positions if they had been moved due to overlap correction (Algorithm 40, lines 13–24).

This way, the user is able to interact with the scene and see the spacecraft up close in their real positions in the Solar System.

Additionally, in both scenes, I included a button that allows the user to switch between them, enabling full exploration of the visualization.

6.3 User VR Experience

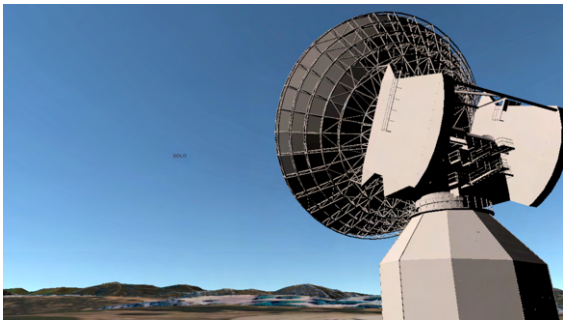
This Virtual Reality application provides users with a dynamic visualization of the Cebreros Station and the missions it tracks.

In the Cebreros View, the antenna can be observed moving in real time, mirroring the actual antenna's rotation as it tracks spacecraft. Users can freely move around the antenna to view it from different angles and see the spacecraft labels positioned in the sky. When a spacecraft is selected, its model is displayed up close and slowly rotates, allowing the user to examine its appearance in detail.

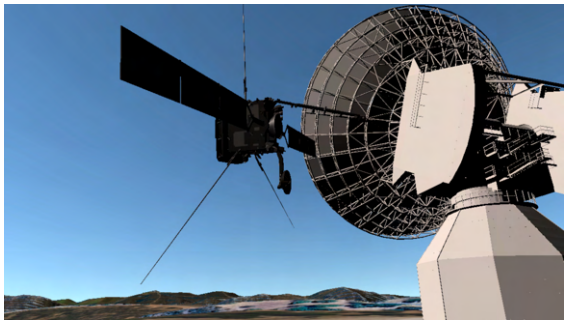
In Figure 6.6a, the antenna is shown pointing at the SOLO spacecraft as it tracks it. Figure 6.6b displays the spacecraft model as it appears after selection. Finally, Figure 6.6c presents the antenna from a different angle, with the label of the JUICE spacecraft also visible in the sky.

In the Solar System View, the user can see the planets and their orbits around the Sun, along with the spacecraft labels in their real positions. When a spacecraft is selected, the camera moves closer to it and displays its model. The user can then look around to observe its position within the Solar System.

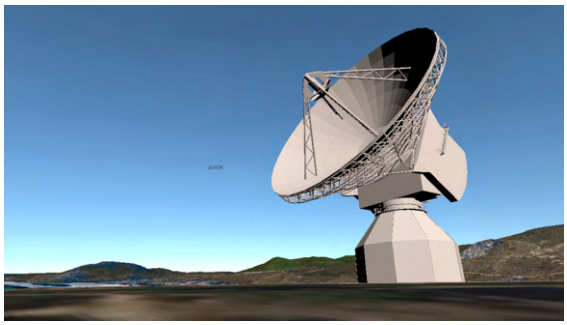
Figure 6.7a shows the initial view when the user enters the Solar System scene. Figure 6.7b displays the SOLO spacecraft model at its position within the Solar System, while Figure 6.7c shows the Euclid spacecraft model in its respective position.



(A) Antenna tracking SOLO

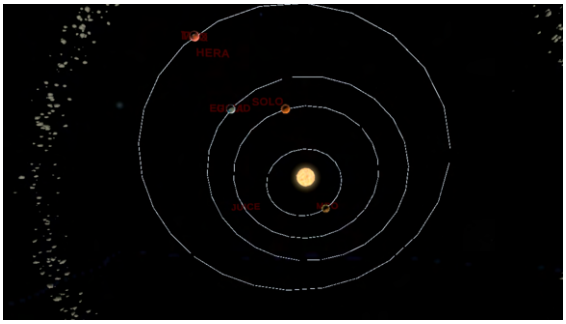


(B) SOLO model in Cebreros View



(C) Antenna Alternative View

FIGURE 6.6: Cebreros View



(A) Solar System View

(B) SOLO Model in Solar System

(C) Euclid Model in Solar System

FIGURE 6.7: Solar System View

The Virtual Reality application offers a broader and more immersive view of the scenes, enabling users to look around from different perspectives and positions, thereby providing a detailed understanding of the placement of each object.

Chapter 7

Conclusions and Future Work

This thesis set out to develop and improve interactive visualization tools aimed at communicating the space missions tracked by ESA's DSA-2 antenna at the Cebreros station. The initial objectives were twofold: to ensure that the information displayed to visitors is accurate, up-to-date, and accessible regardless of technical background, and to enhance the visitor experience through new interactive displays, including immersive virtual reality environments.

These goals reflect a broader ambition to foster public engagement with space science, using technological innovation to build a deeper connection between the public and ESA's ongoing missions.

The objectives proposed in this thesis were successfully achieved. The Cesium and Cosmographia visualizations developed during this project are currently in use at Cebreros Station as part of guided tours for external visitors. Additionally, the 3D models I created have been integrated into multiple initiatives at the station, including not only the Cesium display but also 3D-printed physical models and a semi-realistic video commemorating ESA's 50th anniversary.

These contributions have not only enhanced the station's public engagement tools but have also demonstrated the potential for low-cost, adaptable visualization techniques to be integrated into ESA outreach activities.

Beyond the technical contributions, this work plays a meaningful role in advancing scientific outreach and public engagement. By making complex space mission data visually accessible and engaging through interactive tools, this project helps bridge the gap between the general public and the highly specialized work carried out by ESA. Such visualization platforms not only enhance the educational value of visits to Cebreros Station

but also foster curiosity and inspire future generations to pursue careers in science and space exploration.

Moreover, the use of immersive technologies like Virtual Reality enables a more intuitive understanding of orbital dynamics, spacecraft tracking, and planetary motion, concepts that are often difficult to grasp through traditional media. The flexibility of the solutions developed also ensures that they can be extended or adapted to other ESA ground stations or outreach contexts with minimal cost and technical effort.

As space agencies increasingly recognize the importance of public engagement in maintaining support for scientific missions, tools like the ones developed in this thesis are becoming essential components of institutional communication strategies.

7.1 Main Challenges

One significant challenge encountered during this project was related to the use of SPICE kernels. These files often lack strict naming conventions and are subject to unexpected changes, making them difficult to manage programmatically. Furthermore, not all spacecraft share the same types of metakernels, which complicated efforts to automate kernel retrieval and updating. In several cases, hardcoded filenames were necessary, and the system remained sensitive to unforeseen naming changes that are difficult to generalize.

Another notable challenge was the integration of SPICE kernels into the Virtual Reality visualization. Since Unity does not natively support SPICE or related modules, it was necessary to develop a separate Flask server to manage the kernel-based data processing. This introduced the additional requirement of maintaining a consistently running backend service, adding to the system's complexity. Despite the extra effort this entailed, the solution proved effective and was successfully implemented.

7.2 Future Work

While the current implementation has successfully achieved the initial goals, several areas remain open for future improvement and expansion.

One direction for future work is the optimization and refinement of the scripts used, particularly those handling SPICE kernels, object loading, and position updates, to improve performance and simplify maintainability.

In the Virtual Reality (VR) environment, several enhancements could further improve the user experience. These include:

- Adding interactive time controls to allow users to move forward and backward in time during the simulation;
- Creating simple interfaces for adding or removing spacecraft, similar to the functionalities created for the Cesium display;
- Making the visualization more engaging by incorporating contextual information, such as basic facts about the satellites or visual overlays of constellations, which would help users relate satellite positions to the night sky as seen from Cebreros.

Additionally, the level of interactivity in the VR environment could be increased through features such as clickable elements, annotations, or immersive storytelling mechanisms.

Both the VR and Cesium views would also benefit from the inclusion of even more detailed 3D models of the Cebreros station infrastructure, including buildings and other equipment. This would enhance the realism and immersion of the visualizations and provide visitors with a better spatial understanding of the station's layout.

These extensions would not only elevate the aesthetic and educational quality of the visualizations but also strengthen their value as tools for scientific outreach and public engagement.

The code developed during this project cannot be made publicly available, as much of it is in active use at ESA and therefore constitutes ESA property. Portions of the code were also developed in collaboration with ESA staff. However, it remains accessible for further development within ESA.

Bibliography

[1] NASA, "IV&V STEM Engagement," https://www.nasa.gov/ivv-stem-engagement/, 2025, accessed: 2025-07-04. [Cited on page 1.]

[2] Dana Foundation, "Measuring the Impact of Scientific Outreach and Engagement," https://dana.org/article/measuring-the-impact-of-scientific-outreach-and-engagement/, 2025, accessed: 2025-07-04. [Cited on page 1.]

[3] National Aeronautics and Space Administration, Science Mission Directorate, "2023 Science Activation Impact Report," NASA, Tech. Rep., May 2024. [Online]. Available: https://science.nasa.gov/wp-content/uploads/2024/05/2023-sciact-impact-report-tagged.pdf [Cited on page 2.]

[4] Y. Doat, M. Lanucara, P. Besso, T. Beck, G. Lorenzo, and M. Butkovic, "ESA Tracking Network – A European Asset," pp. 1–3, Jun. 2018. [Online]. Available: https://arc.aiaa.org/doi/pdf/10.2514/6.2018-2306 [Cited on page 6.]

[5] L. Butcovic, "ESTRACK: ESA's Ground Station Network," 2017, accessed: 2025-01-27. [Online]. Available: https://www.mpe.mpg.de/events/593-heraeus-seminar/Talks-and-Posters/1-Monday/PDF/1.%20Butcovic%20-%20ESTRACK.pdf [Cited on page 6.]

[6] European Space Agency, "Estrack: ESA's Global Ground Station Network," n.d., accessed: 2025-01-27. [Online]. Available: https://www.esa.int/Enabling_Support/Operations/ESA_Ground_Stations/Estrack_ESA_s_global_ground_station_network [Cited on page 7.]

[7] ——, "Cebreros - DSA 2," https://www.esa.int/Enabling_Support/Operations/ESA_Ground_Stations/Cebreros_-_DSA_2, n.d., accessed: 2025-01-27. [Cited on page 7.]

[8] Gaia Collaboration, "The Gaia mission," *Astronomy Astrophysics*, vol. 595, p. A1, 2016. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2016A%26A...595A...1G/abstract [Cited on page 7.]

[9] J. Benkhoff, J. van Casteren, H. Hayakawa, M. Fujimoto, H. Laakso, M. Novara, P. Ferri, H. R. Middleton, and R. Ziethe, "BepiColombo—Comprehensive exploration of Mercury: Mission overview and science goals," *Planetary and Space Science*, vol. 58, no. 1-2, pp. 2–20, 2010. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2010P%26SS...58....2B/abstract [Cited on page 7.]

[10] A. Chicarro, P. Martin, and R. Trautner, "The Mars Express mission: An overview," in *Mars Express: The Scientific Payload*, A. Wilson and A. Chicarro, Eds. Noordwijk, Netherlands: ESA Publications Division, 2004, pp. 3–13. [Cited on page 7.]

[11] C. H. Acton, "Ancillary data services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, vol. 44, no. 1, pp. 65–70, 1996. [Cited on page 8.]

[12] NASA Navigation and Ancillary Information Facility, "SPICE Concept," n.d., accessed: 2025-01-27. [Online]. Available: https://naif.jpl.nasa.gov/naif/spiceconcept.html [Cited on page 8.]

[13] C. Acton, N. Bachman, J. Diaz del Rio, B. Semenov, E. Wright, and Y. Yamamoto, "SPICE: A Means for Determining Observation Geometry," EPSC Abstracts, Vol. 6, EPSC-DPS2011-32, 2011, ePSC-DPS Joint Meeting 2011. [Online]. Available: https://meetingorganizer.copernicus.org/EPSC-DPS2011/EPSC-DPS2011-32.pdf [Cited on page 9.]

[14] Dovramadjiev, Tihomir, "Modern accessible application of the system Blender in 3D design practice," *Publishing House "Union of Scientists - Stara Zagora"*, p. 10 – 13, 06 2015. [Cited on page 10.]

[15] Khronos Group, "glTF: The 3D Asset Delivery Format," https://www.khronos.org/gltf/, 2025, accessed: 2025-07-03. [Cited on page 10.]

[16] Adobe Inc., "What Are OBJ Files?" https://www.adobe.com/products/substance3d/discover/what-are-obj-files.html, 2025, accessed: 2025-07-03. [Cited on page 10.]

[17] NASA Navigation and Ancillary Information Facility, "Cosmographia: A 3D Visualization Tool for Space Science Missions," n.d., accessed: 2025-01-27. [Online]. Available: https://naif.jpl.nasa.gov/naif/cosmographia.html [Cited on page 10.]

[18] Epic Games, "Unreal Engine," https://www.unrealengine.com/, 2025, accessed: 2025-07-03. [Cited on page 11.]

[19] Cesium, "About Cesium," n.d., accessed: 2025-01-27. [Online]. Available: https://cesium.com/about/ [Cited on page 11.]

[20] Unity Technologies, "Unity Real-Time Development Platform," https://unity.com, 2024, accessed: 2025-07-03. [Cited on page 12.]

[21] Meta Platforms, Inc., "Learn about Meta Quest 2," 2020, accessed: 2025-08-04. [Online]. Available: https://www.meta.com/help/quest/930077757778362/ [Cited on page 12.]

[22] CesiumGS, "Cesium for Unity Samples," https://github.com/CesiumGS/cesium-unity-samples, 2024, accessed: 2025-07-03. [Cited on page 69.]

[23] Siccity, "GLTFUtility: Importer for glTF files in Unity," 2023, accessed: 2025-07-03. [Online]. Available: https://github.com/Siccity/GLTFUtility [Cited on page 70.]

[24] European Space Agency, "ESA WebGeocalc Tool," http://spice.esac.esa.int/webgeocalc/#NewCalculation, n.d., accessed: 2025-07-03. [Cited on page 71.]

[25] Evgenii Nikolskii, "Planets of the Solar System 3D," https://assetstore.unity.com/packages/3d/environments/planets-of-the-solar-system-3d-90219, 2017, unity Asset Store. [Cited on page 94.]