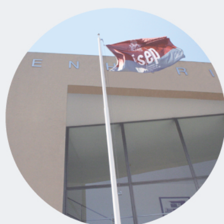
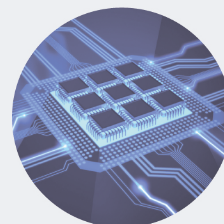
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS





Real-Time Digital Twin Visualization and Simulation for the KARVEL Satellite Platform

DANIEL FILIPE PIMENTEL DA ROSA LOPES

outubro de 2025

P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Real-Time Digital Twin Visualization and Simulation for the KARVEL Satellite Platform

Daniel Filipe Pimentel da Rosa Lopes

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: Tiago Diogo Ribeiro De Carvalho

Evaluation Committee:

President:

Luis Lino Ferreira, Instituto Superior de Engenharia do Porto

Members:

Tiago Carvalho, Instituto Superior de Engenharia do Porto

António Barros, Instituto Superior de Engenharia do Porto

Porto, September 22, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, September 22,2025

Abstract

With advances in computational power and software, real-time data processing for digital twin rendering has become viable in aviation and the aerospace sector. This capability enables scientists and engineers to simulate real objects and their environments within a virtual space, allowing for accurate trajectory simulations, obstacle detection, and action planning before real-world implementation—thus reducing risks and optimising resources.

In this study, data from the satellite's onboard instruments are transmitted to mission control using network protocols optimised for real-time space communication. Mission control processes and visualises the telemetry, validates system behaviour, and issues telecommands to adjust spacecraft operations. A high-fidelity simulation, powered by a modern game engine, renders the satellite's state and orbital behaviour in real time, based on telemetry data and telecommands.

This work presents a proof-of-concept prototype for a digital twin application, aiming to explore and validate key functionalities that may be valuable in future satellite monitoring systems. The objective is to test the feasibility and usefulness of features such as real-time telemetry visualisation, telecommand execution, orbital simulation, and historical data playback in a virtual environment. The proposed architecture is designed to support seamless integration with telemetry-based systems and is aligned with aerospace standards wherever applicable. By simulating satellite behaviour and interactions with mission control, the prototype serves as a foundation for improving spacecraft monitoring, anomaly detection, and mission planning workflows.

Keywords: Telemetry, Telecommand, Spacecraft, Mission Control System, Digital Twin, Embedded system, Game Engine

Resumo

Com os avanços no poder computacional e no software, o processamento de dados em tempo real para a renderização de gémeos digitais tornou-se viável na aviação e no setor aeroespacial. Esta capacidade permite que cientistas e engenheiros simulem objetos reais e os seus ambientes num espaço virtual, possibilitando simulações precisas de trajetórias, deteção de obstáculos e planeamento de ações antes da implementação no mundo real, reduzindo riscos e otimizando recursos.

Neste estudo, os dados provenientes dos instrumentos de bordo do satélite são transmitidos para o controlo da missão através de protocolos de comunicação de rede otimizados para operações espaciais em tempo real. O controlo da missão processa e visualiza a telemetria, valida o comportamento do sistema e emite telecomandos para ajustar as operações da espaçonave. Uma simulação de alta-fidelidade, suportada por um motor de jogo moderno, representa em tempo real o estado e o comportamento orbital do satélite com base nos dados de telemetria e nos telecomandos.

Este trabalho tem como objetivo desenvolver um sistema robusto que garanta uma transmissão contínua de telemetria, a execução de comandos e a simulação em tempo real do gémeo digital num ambiente virtual. A estrutura proposta foi concebida para ser interoperável com outros sistemas, alinhando-se com os padrões aeroespaciais existentes sempre que aplicável, e visa melhorar a monitorização da espaçonave, a deteção de anomalias e o planeamento da missão.

Table of Contents

1 Introduction 1
1.1 Problem statement 1
1.2 Relevance of the subject 2
1.3 Goal and objectives of the work 2
1.4 Chapter resume 3
2 State of the Art 5
2.1 Mission Control 5
2.1.1 YAMCS 7
2.1.2 Lighthouse 8
2.2 Satellite platforms 9
2.2.1 NASA cFS 9
2.2.2 Karvel 13
2.3 Game engine 14
2.4 Digital twin concept 17
2.4.1 Digital twin frameworks with game engines 19
2.4.2 Mission Specific platforms 21
2.5 Communication protocols 22
2.5.1 MQTT 22
2.5.2 CoAP 25
2.5.3 DDS and DDSI-RTPS 27
2.6 Chapter resume 31
3 Fundamentals of Orbital Mechanics 33
3.1 Astronomical models of the solar system 33
3.1.1 Geocentric 33
3.1.2 Heliocentric 34
3.1.3 Heliocentric in a Geocentric reference frame 35
3.2 Ellipse 36
3.3 Reference Plane and Reference Frame 38
3.3.1 Reference Plane 38
3.3.2 Reference Frame 39
3.4 Orbital elements 40
3.5 Chapter resume 41
4 Methodology 43
4.1 Requirements 43
4.1.1 Functional Requirements 44
4.1.2 Non-functional Requirements 47

4.2 High-level architecture 48

5 Proposed approach 51

5.1 System Overview and Technology Stack 51

5.2 Design 52

5.3 Functional Requirements Mapping 53

5.3.1 Functional Requirements 54

5.4 Non-Functional Requirements Mapping 56

5.4.1 Non-functional Requirements 56

5.5 Sequence diagrams 58

5.5.1 Send telemetry 58

5.5.2 Send Telecommand 58

5.6 Unreal Engine plugins 59

5.6.1 MaxQ plugin – Integration of NASA SPICE toolkit 60

5.6.2 UE4SimpleSQLite – Database manager 63

5.6.3 MqttUtilities – MQTT protocol 63

5.7 Kernel files and 3D models 64

5.8 Reference frame and plane 65

6 Implementation 67

6.1 Object types 67

6.1.1 Blueprint Class 67

6.1.2 Blueprint Function Library 67

6.1.3 Level and Level Blueprint 67

6.1.4 Structure 68

6.1.5 Widget Blueprint 68

6.2 Unreal Engine features and constraints 68

6.2.1 UGameInstance 68

6.2.2 Scale and Units 69

6.3 Development 70

6.3.1 Widget Manager Architecture 71

6.3.2 BP_SharedData variables 72

6.3.3 GraphicPoint Struct 73

6.3.4 Loading Kernel files 73

6.3.5 Solar System simulation 73

6.3.6 Lighting 74

6.3.7 Karvel orbit 76

6.3.8 Orbital Values - Orbital values input widget 76

6.3.9 Time Scale – TimeMods widget 77

6.3.10 Lighthouse – Lighthouse widget 77

6.3.11 MQTT – MQTT widget 78

6.3.12 Compare Orbit - Copy_Satellite widget 79

6.3.13 Synoptics – Synoptics widget 81

6.3.14 Graph – Graph widget 83

6.3.15 Settings 84

6.3.16 Select Satellite 84
6.3.17 Main menu level 85

7 Conclusion 87

7.1 Requirement Fulfilment Overview 87
7.1.1 Functional requirements status 88
7.1.2 Non-functional requirements status 90

7.2 Future work 91
7.2.1 Time-series database 91
7.2.2 In-depth orbit and force calculations........ 92
7.2.3 AI........ 92
7.2.4 Satellite federation 92
7.2.5 Communication protocols 93
7.2.6 Housekeeping and event warning 93
7.2.7 Customization and UI/UX........ 93

7.3 Reflections 94

8 Bibliography 96

List of Figures

Figure 1 - Basic Space flight mission diagram 6
Figure 2 – Lighthouse (Critical Software) 8
Figure 3 – NASA cFS application stack diagram 10
Figure 4 – Karvel architecture stack diagram (Critical Software) 13
Figure 5 – Geocentric diagram (Largegreenbird) 34
Figure 6 – Heliocentric diagram (Pressbooks) 35
Figure 7 – Geocentric reference frame in a heliocentric model (Njit) 36
Figure 8 – Important points and segments in an Ellipse 37
Figure 9 - Reference planes and markers used in celestial coordinate systems 39
Figure 10 – ECI reference frame (Wikipedia) 40
Figure 11 – Diagram of the Keplerian elements that describe an orbit (Wikipedia) 41
Figure 12 – Detailed system architecture 52
Figure 13 – Sequence Diagram: Send Telemetry 58
Figure 14 - Sequence Diagram: Send Telecommand 59
Figure 15 – Accessing and storing a game instance object reference 69
Figure 16 – Accessing game instance data 69
Figure 17 – Solar system render 70
Figure 18 – Custom light logic 75
Figure 19 – Karvel orbit and Orbital values input widget 77
Figure 20 – Example of the position of several celestial bodies in the future 77
Figure 21 – Lighthouse open in digital twin 78
Figure 22 – MQTT widget 79
Figure 23 – Compare orbit popup 80
Figure 24 – Compare orbit completion 81
Figure 25 – Synoptics widget 82
Figure 26 – Graph widget 83
Figure 27 – Select satellite model 85
Figure 28 – Main menu 85

List of Tables

Table 1 – Functional and Non-functional requirements 44
Table 2 – Functional requirements mapping 54
Table 3 – Non-functional requirements mapping.. 56

Acronyms and Symbols

List of Acronyms

3D	Three-Dimensional
AI	Artificial Intelligence
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
BSP	Board Support Package
CAD	Computer-Aided Design
CAM	Camera control
CCSDS	Consultative Committee for Space Data Systems
CFDP	CCSDS File Delivery Protocol
CI	Command Ingest
cFE	Core Flight Executive
cFS	Core Flight System
CoAP	Constrained Application Protocol
CPU	Central processing Unit
CRT	Cathode Ray Tube
DAP	Data Acquisition and Processing
DCPS	Data-Centric Publish-Subscribe
DDS	Data Distribution Service
DTDL	Digital Twins Definition Language
DUP	Duplicate delivery of a Package
EDSAC	Electronic Delay Storage Automatic Calculator
EPP	Encapsulation Packet protocol
ES	Executive Services
ESA	European Space Agency
EUD4MO	ESA Unified Data System for Mission Operations
EVS	Event Services

FSW	Flight software
GDPR	General Data Protection Regulation
GDS	Global Data Space
GMSEC	Goddard Mission Services Evolution Center
GPU	Graphics Processing Unit
GUI	Graphical User interface
HDSW AL	Hardware/Software Abstraction Layer
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IAU	International Astronomical Union
ID	identifier
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Internet Protocol
ISEP	*Instituto Superior de Engenharia do Porto*
JSON	JavaScript Object Notation
LAN	Local Area Network
LOS	Loss of Signal
MOC	Mission operation Centre
MCS	Mission Control System
MDS	Mission Design Simulator
MO	Mission Operations
MQTT	Message Queuing Telemetry Transport
NASA	National Aeronautics and Space Administration
NDA	Non-Disclosure Agreement
NISA	Nano Immersive Situational Awareness
OBCP	On-Board Control Procedure
OBS	Observation control
OBSW	On-Board SoftWare
OS	Operating System
OSAL	Operating System Abstraction Layer

PDU	Protocol Data Unit
PIM	Platform Independent Model
PROM	Programmable Read-Only Memory
PSM	Platform Specific Model
PSP	Platform Support Package
PUS	Packet Utilization Standard
PVN	Packet Version Number
QoS	Quality of Service
REST	Representational State Transfer
RTOS	Real-time Operating System
RTPS	Interoperability Real-Time Publish-Subscribe
SB	Software Bus Services
SCH	Scheduler
SDK	Software Development Kit
SDLP	Space Data Link Protocol
SPP	Space Packet Protocol
SQL	Structured Query Language
stdin	Standard input
SWDB	Software Database
TBL	Table Services
TCP	Transmission Control Protocol
TIME	Time Services
TO	Telemetry Output
U6	Unity 6
UDP	User Datagram Protocol
UE5	Unreal Engine 5
URI	Uniform Resource Identifier
VGA	Video Graphics Array
WHL	Filter Wheel Control
XML	Extensible Markup Language
XTCE	XML Telemetry and Command Exchange

YAMCS Yet Another Mission Control System

List of Symbols

α alpha

a Semi-major axis

b Semi-minor axis

e Eccentricity

1 Introduction

This chapter presents an overview of the problems and objectives present in developing a digital visualizer for a embed satellite platform.

1.1 Problem statement

In projects where satellites are involved, one of the most important and critical components is a stable communication with satellites, which are constantly monitored by operators in the Mission Operation Centre (MOC) on Earth for any issues that might arise. A stable and reliable communication channel is imperative for a successful mission (1) (2) (3). To improve mission outcomes, providing operators with the ability to test changes before deploying them would also be highly beneficial.

As some spacecrafts, such as satellites, are not physically human-operated, it is crucial for the control centre to remotely collect the necessary data regarding the current state of the spacecraft. This data enables operators to monitor key aspects of the spacecraft, such as location, velocity, rotation, and the status of both its software and hardware. Consequently, it is essential for scientists and engineers in the control centre to utilize mission control software that facilitates remote control operations and telemetry reception. This capability allows them to monitor the spacecraft's behaviour, implement remote changes as needed, and identify potential issues and take pre-emptive measures to prevent them.

Another significant challenge in such projects is the effective visualization of data. While telemetry and statistics can be displayed on a screen in raw and processed form, this approach limits the number of users who can interpret the data effectively. Even those who understand the data may require significant time to analyse it and derive actionable insights. This issue becomes particularly problematic when operators need to test different variables and observe their effects on the spacecraft in real-time.

Ideally, and if processing power is present, the data should be processed and interpreted by a three-dimensional (3D) render engine capable of simulating the spacecraft in real-time. This simulation would run parallel to the inputted data, providing a visual representation of the spacecraft's behaviour. Such a system would simplify the process of data analysis and enhance decision-making efficiency.

The development of a framework that can request, submit, and track data shared between the spacecraft and the control centre would significantly enhance operational efficiency. When paired with a real-time data visualization tool with simulation capabilities, this framework would empower operators to simulate potential changes in a 3D environment before deploying them, visualize the spacecraft's behaviour in real-time based on

telemetry data, and quickly and effectively assess the impact of adjustments. By enabling operators to visualize and test changes in a simulated environment, this proposed solution would improve mission outcomes and reduce the risk of errors during critical operations.

1.2 Relevance of the subject

Across the scientific and engineering fields, numerous tools have been developed and continue to be updated to address many of the challenges that are also associated with this project.

The overall objective of this dissertation is to integrate several freely available frameworks and tools with proprietary software designed specifically for this project. This integration aims to assist similar projects by developing and documenting how communication can be achieved between a spacecraft and control centres, and how the same data can be used to monitor current mission state. Achieving this objective could significantly enhance the efficiency and reliability of space exploration efforts, simplifying spacecraft monitoring processes.

Most importantly, the primary objective is the inclusion of a 3D engine to simulate and display relevant data would provide operators with valuable assistance in analysing complex datasets. Such a tool would enable operators to test changes and assess their impacts quickly before deploying them, and, as importantly, to detect errors and fix them during an active missing to avoid mission failure. By improving resource management - including time, money, and personnel - this approach could streamline mission operations and contribute to advancements in space exploration technologies.

1.3 Goal and objectives of the work

This project is tasked with development of a framework capable of facilitating communication between a satellite and a mission control centre while simulating the mission state in a virtual environment. This communication involves the exchange of spacecraft data, like speed and rotation data, and commands to update spacecraft status, like thruster force for path correction, ensuring real-time simulation of the satellite's behaviour in a virtual environment to enhance accessibility and usability.

To achieve this overarching goal, the following objectives have been identified:

- **Satellite Framework**: Configure an already existing embedded software platform for the spacecraft. This will involve:
    - Configuration of the communication protocols to send telemetry and receive telecommands.
    - Setup variables representing satellite payload and status. Depending on their state, they can trigger events in real-time.
- **Mission Control Centre**: Establish control and monitoring software capable of:
    - Receiving telemetry data from the satellite.

  - Sending telecommands to modify specific satellite parameters and functionalities.
- **Virtual Simulator**: Design and implement a 3D simulation engine capable of:
  - Tracking telemetry and telecommands exchanged between the satellite and the mission control centre.
  - Simulating the satellite's behaviour in real time.
  - Testing operator-submitted changes in a virtual environment before deploying them to the actual satellite.
  - Store telemetry data locally for further analysis.
- **Full System Integration**: Combine the three components (Satellite Framework, Mission Control Centre, and Virtual Simulator) into a cohesive system capable of real-time operation. This will involve:
  - Ensuring seamless communication between the components within a reasonable time frame.
  - Conducting extensive testing to validate data transmission and processing.
  - Implementing error recovery mechanisms to enhance system robustness.

If successfully implemented, the project will be capable of performing three main tasks:

- **System Monitoring**: The satellite framework will transmit vital telemetry data and events, enabling operators to monitor the spacecraft's status in real time.
- **Orientation Control**: The satellite will respond to telecommands, allowing operators to modify its orientation and trajectory remotely.
- **Manoeuvre Planning**: By integrating System Monitoring and Orientation Control, operators will be able to track the satellite's current state, simulate planned manoeuvres in the 3D engine and possibly communicate them to the craft, and identify and rectify potential issues before executing changes on the actual satellite

1.4 Chapter resume

This chapter summarises the motivation and goals behind developing a digital twin application for satellite mission visualization and control, highlighting the importance of reliable communication between satellites and mission control centres, the need for real-time telemetry monitoring and remote command capabilities.

One issue is the difficulty in interpreting raw telemetry data. To improve how operators can read and interpret it, the project proposes an application with a 3D environment that visualizes and simulates satellite behaviour in real time, allowing operators to test changes before deployment and overwatch satellite status.

The chapter also notes the need to combine existing tools with software developed specifically for the purpose connecting to external satellite monitoring applications. The main objective is

to create a system that connects a satellite, mission control, and a virtual simulator and enables real-time data exchange, visualization, and testing. If successful, the system will support real-time monitoring, remote orientation control, and manoeuvre planning.

2 State of the Art

This chapter explores current technologies relevant to this project, including potential tools, frameworks, and methodologies that could be utilized. Additionally, it examines similar projects, analysing the challenges they encountered and the solutions they implemented. By reviewing existing approaches, this study aims to identify best practices, anticipate potential problems, and improve the proposed implementation.

This chapter will start by first explaining important concepts addressed in the project, to later bridge them into already existing technologies and how other projects used similar technologies and concepts to achieve their own goals.

2.1 Mission Control

Managing an active spacecraft from a mission control centre on Earth is no easy task. The critical nature of such projects necessitates meticulous planning and control of various elements, including the spacecraft itself, communication protocols, and the expertise of the engineers responsible for monitoring the craft and managing the data transmitted between systems. As such, a mission control system (MCS) that is secure, user-friendly, easily integrable with other applications, and equipped with robust resources to address foreseeable challenges is an essential requirement (4). For such, it will be analysed some of the requirements in space flight projects and what some mission control systems that already exist have to offer.

Figure 1 shows a basic diagram of a space flight mission architecture, illustrating the exchange of information between all involved parties:

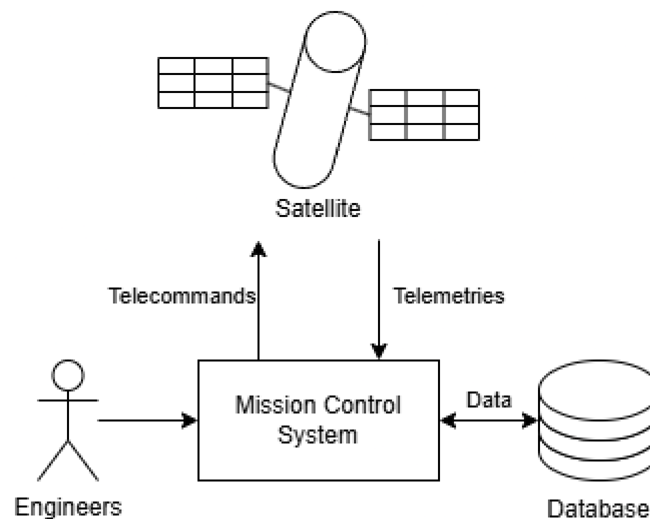


Figure 1 - Basic Space flight mission diagram

As shown in the architecture, MCS is responsible for managing all communication and interactions throughout the mission. Its primary role is to facilitate communication between the spacecraft and the ground station and its users. Engineers rely on the MCS to access mission data and monitor the satellite, making stable and reliable communication essential for accurate spacecraft state assessment, while the satellites rely on it to receive updates.

To ensure the successful operation of a mission, the MCS must achieve the following three main functionalities (5) (6) (7) (8):

- **Telemetry**: The satellite transmits mission-specific information such as data from spacecraft instruments and the status of various components, including battery levels, rocket throttle levels, and other vital systems. This real-time flow of information enables engineers to monitor the health and performance of the spacecraft.
- **Tracking**: The MCS must be capable of tracking the spacecraft's angular location and range. This tracking ensures that data can be accurately transmitted between the spacecraft and the control centre, and that the spacecraft's position is precisely known for navigation and communication purposes.
- **Command**: When necessary, telecommands are sent from the MCS to the spacecraft to modify its behaviour remotely. Examples of these commands include course corrections, software updates, and adjustments to onboard systems to address anomalies or optimize performance.

The MCS serves as the backbone of any spaceflight operation, enabling seamless communication and control throughout the mission. Therefore, we will analyse the requirements of spaceflight projects and review several existing mission control systems and frameworks.

2.1.1 YAMCS

This mission control software attempts to offer an open-source solution containing a scalable and flexible framework compliant with European Space Agency (ESA) and Consultative Committee for Space Data Systems (CCSDS) MO service definitions for any kind of mission and craft (9) (10), providing secure access to telemetry and telecommands through its architecture. Additionally, offers a way to store and review payloads by using its parameter tracking service, a service that stores those values for a limited amount of time.

YAMCS enables parallel operations for multiple commanding and monitoring, meaning it can track, monitor, and send instructions to payloads of several crafts within a single YAMCS instance, showcasing its scalability (9). During a mission, storing telemetry data is vital, as it can be used post-mission to troubleshoot errors, analyse and visualize information, or simply for long-term record-keeping. The archiving system provides a Graphical User interface (GUI) that allows easy access to telemetry packets and other mission parameters (10).

As for communication, YAMCS operates on a client-server architecture, where the server runs as a single process incorporating an embedded Hypertext Transfer Protocol (HTTP) server, providing an HTTP Application Programming Interface (API) for its service to be used by clients in the same network or external communications (9) (10), further helping with its integration in other projects. This embedded HTTP server supports static file serving, authentication, and Application Programming Interface (API) requests, and is integrated with YAMCS's security system, serving as the default interface for external tools and clients. Within this architecture, YAMCS utilizes Data Links to connect with target systems such as instruments, ground stations, or laboratory equipment. These Data Links handle telemetry packets, telecommands, and parameters, easing data exchange between the spacecraft and the ground control system (9). To manage data flow, YAMCS uses Streams that transport data tuples, decoupling data producers from consumers, and allowing for Structured Query Language (SQL)-like queries to be performed (9). This design allows YAMCS to be better integrated with other systems and filter data with queries in real-time, useful in analysing telemetry in critical missions. The system's Processors manage telemetry and telecommand data as per mission database definitions, supporting concurrent processing of parallel data streams. This setup allows YAMCS to handle multiple payloads or satellites simultaneously.

Sometimes, data must be reviewed during a mission rather than afterward. For example, issues may occur during transmission, and it is important to quickly analyse the situation. YAMCS includes a replay functionality (9) that is particularly useful in Loss of Signal (LOS) situations to check for anomalies in telemetry and telecommands. This feature is also essential after reconnection, when catch-up data must be evaluated for troubleshooting and decision-making.

To support replay and archive functionalities, the GUI offers tools such as the Packet Viewer and the Event Viewer (9). The Packet Viewer displays a set of parameters and packets being received in real-time, while the Event Viewer logs significant payload packet information triggered by specific events, such as communication anomalies, storing them in a format similar to an event log.

In mission control, it is extremely beneficial for spacecraft and ground systems to use a standardized information model for telemetry and telecommands. To this end, YAMCS is compatible with XTCE (XML Telemetric and Command Exchange), a markup language based on Extensible Markup Language (XML) specifically developed for spacecraft telemetry and command metadata. (9) (10)

2.1.2 Lighthouse

Lighthouse is a mission control application developed by Critical Software to interface with the Karvel platform. It serves as a ground segment tool that allows operators to send telecommands and receive telemetry data in real time. Figure 2 shows how the frontend of Lighthouse looks like:

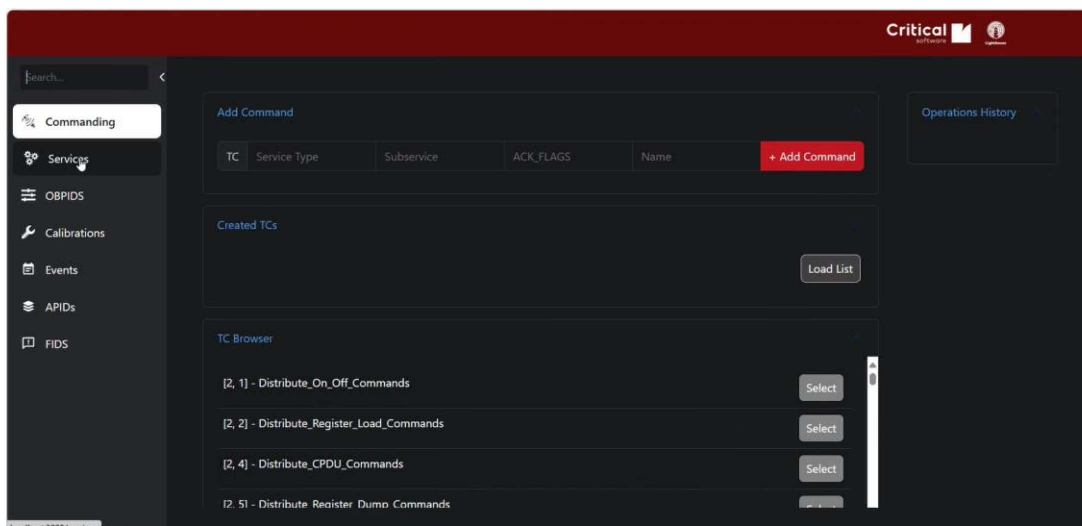


Figure 2 – Lighthouse (Critical Software)

Lighthouse is composed of two main components:

- **Ground Segment**: A web-based application built with React for the frontend and Python with Flask for the backend. It runs locally on a computer and provides the user interface for mission operations.
- **Space Segment**: Represents the satellite's on-board computer running Karvel, that can be hosted on a Raspberry Pi for demonstration purposes.

Communication between these two models can be achieved with Transmission Control Protocol (TCP), which allows both sides to share telemetries and telecommands.

The frontend is tasked with providing methods to manage telemetries and telecommands, such as the ability to create and configure telecommands, browse and select commands from a predefined list, view command history with timestamps and statuses, real-time display of incoming telemetry data and filtering and inspecting of telemetry by service and subservice.

The backend exposes a RESTful API (an implementation of the Representational State Transfer (REST) architectural) to create, list, and delete telecommands, as well as to retrieve telemetry data from Karvel, and interface with the Software Database (SWDB) to retrieve parameter definitions. Telemetry data is structured according to the Packet Utilization Standard (PUS) protocol, compatible with standard space mission communication formats.

2.2 Satellite platforms

In satellites, different kinds of applications are tasked with managing resources, hardware and communication with embedded systems and with outside communications. Due to the high standards and requirements of these kinds of systems, a platform that can provide resource and communication management is crucial for mission success.

This chapter describes platforms designed to be deployed in satellites and can manage the strict requirements of such missions.

2.2.1 NASA cFS

The National Aeronautics and Space Administration (NASA)'s Core Flight System (cFS) is a reusable software framework composed of reusable software applications (11) (12) (13) (14). The reusability of the software is important, as it saves resources - time, money, or personnel - in adapting a whole new framework for specific missions. Indeed, some components and features will need to be developed from scratch due to the requirements of certain missions, but by abstracting part of the framework, a lot of it can be reused repeatedly. With that in mind, the main key aspects of cFS's architecture are dynamic run-time capabilities, layered software, and component-based design (11) (15) (16) (17), which all contribute to decreasing the time needed to develop and test for missions while maintaining compatibility with different projects.

Figure 3 shows an overview of the architecture of cFS, split into different layers and their main components:

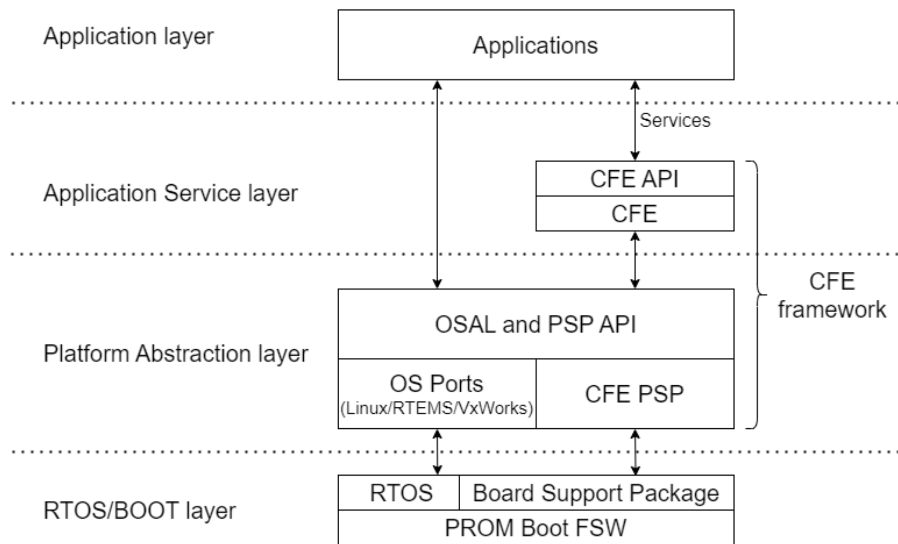


Figure 3 – NASA cFS application stack diagram

From the bottom, we begin with the Real-time Operating System (RTOS) and Boot layer. cFS requires a place to boot from, like a Programmable Read-Only Memory (PROM), which will contain early initialization and Operating System (OS) bootstraps (11). As for the OS, it is required to be an RTOS due to the necessity of a preemptive, priority-based multitasking system for resource and task management in a multi-threaded environment (11) (18). This also means handling interrupts and exceptions, message queues, and mutexes. The Boad Support Package is software layer that defines how the hardware can be accessed, as well as some additional settings such as OS kernels (19).

The Platform Abstraction API defines the portability of cFS applications (11) (13). The main advantage of this design is how it abstracts hardware access via system calls and hardware dependencies from the applications in the higher API layers, which allows the user to more easily adapt the framework to different RTOS and hardware by only updating the API. Thus, it provides hardware, operating system, and hardware platform independence. This is achieved by two components: Operating System Abstraction Layer (OSAL) and Platform Support Package (PSP) (11) (12). OSAL is a small software library with the task of separating the OS from the flight system by providing an API for applications to access OS functionalities like threads, synchronization primitives such as mutexes and semaphores, and I/O (20) (21). This way, developers can develop applications without worrying about OS dependencies. If the OS does need to be changed, then only the OSAL layer requires modifications to adapt the API to the new OS. Like the OSAL, the PSP provides the API for applications to use, but for hardware access instead, allowing portability across different hardware architectures. PSP handles hardware-specific tasks such as memory management and Central processing

Unit (CPU) architecture (20) (21). Just like the OSAL, if changes are to be made to the hardware, only the PSP should require modifications.

It is important to note that the OSAL and PSP APIs and services provided by the Core Flight Executive (cFE) should be the only methods of gaining access to the hardware and OS by the applications, as this isolates those applications from hardware and software dependencies (11) (12) (22) (23).

cFE is a framework included in the Application Service, a flight software (FSW) operational environment that provides services usually necessary for in-flight applications to develop and test embedded aerospace flight software (11) (22) (23). Its main goal is to achieve a portable embedded system software independent from RTOS and hardware, achieved by OSAL and PSP, respectively. cFE also includes a few sets of tools for software development: UTF, to unit test applications using cFE; Software Timing Analyzer, to track real-time performance; Table Builder; and Command and Telemetry utilities (11) (12) (22).

The cFE API provides five services to the applications: Executive Services (ES), Time Services (TIME), Event Services (EVS), Table Services (TBL), and Software Bus Services (SB) (11) (12) (22). ES is tasked with managing the runtime environment and tracking system resources used by those applications. TIME manages spacecraft time, allowing applications to retrieve time for purposes such as timestamping data and computing delta time. EVS allows applications to send timestamped text events with parameters, functioning as logging event messages. TBL consists of a collection of application parameters and coordinates the loading and dumping of tables between a file and an application. SB provides an application publish-and-subscribe messaging system, allowing applications to be compiled, linked, loaded, and started without requiring a full system rebuild. By also being an open-source project, anyone can create their own libraries and add them to the Application Library layer, which is important when developing applications for specific missions with specific requirements.

In addition, two applications designed for test and development but not for mission deployment are included: Telemetry Output (TO), which manages data flow from the software bus to outside clients, and Command Ingest (CI), which allows commands to be injected from the outside into the software bus (11) (12). One kind of test that takes advantage of CI and TO is Network Blind Fuzzing (11) (24), which is a technique used to find vulnerabilities in network protocols and message processing by sending unexpected, random, or malformed data to a network service and observing its behaviour. Testing can be done by injecting user telecommands into the CI and analysing the application behaviour. It is also possible to modify or replace network libraries so that inputs are read from files or Standard input (stdin).

For communication, cFS uses CCSDS Space Packet Protocol (SPP) and Encapsulation Packet protocol (EPP) for different purposes (12) (25) (26).

EPP was designed to allow transmission over ground-to-space, space-to-ground, or space-to-space communication without an authorized Packet Version Number (PVN), a field that indicates the protocol version used, by encapsulating Protocol Data Units (PDUs) from higher layers of the OSI model when these move into or out of the Data Link Layer (27). This is achieved by utilizing packet services present in the Space Data Link Protocol (SDLP) (28), standardizing encapsulated packets for data transmission over heterogeneous systems. The encapsulated packet consists of a packet header field, which contains metadata such as synchronization information and error detection, and an encapsulated data field, which contains mission-specific, privately defined data. This way, a system can share diverse data types in a single packet using a standard transmission.

SPP was primarily developed for use in space application data transfers from a user application to another, encapsulating application data into a Space Packet for a more effective transmission (29) (30). Like EPP, this is also done by SDLP services in the Data Layer.

Another key feature of cFS is its extensive use of software bus architecture, which facilitates modularity and scalability. The software bus enables a message-passing paradigm where different components communicate asynchronously, reducing interdependencies and allowing for easier integration of new software modules. This is particularly beneficial for missions requiring rapid adaptation to new hardware or mission parameters. (11) (12)

Additionally, cFS supports the development of custom applications and services (12). By leveraging the open-source nature of cFS, mission designers can tailor the framework to accommodate unique mission requirements, including specialized data handling, autonomous operations, or integration with external ground systems. One application that can be plugged in to the cFE is the Scheduler (SCH) (12) (31), also developed by NASA, which allows the user to generate software bus messages in pre-determined intervals (31).

cFS has already been used by external sources to NASA. Such example is the inclusion of cFS in a Telescope Control Software used in the observation of the 2017 total solar eclipse (32). From the beginning, it was intended to use several features of cFS: OSAL and PSP for OS and hardware independency, and cFE services used in mission-specific and reusable applications. Some of these applications added to cFS were a camera control (CAM), filter wheel control (WHL), data acquisition and processing (DAP), and observation control (OBS). CAM would configure the camera for frame acquisition, and delivery of images, WHL could control the filter-wheel motion, such as position in degrees, speed in rotations per minute, and acceleration. DAP was used in image processing, statistics and other image calculations. The OBS receives telemetry of current CAM and WHL status and generates scientifical data. Additionally, there was the inclusion of reusable applications developed by NASA such as a SCH, tasked with generating periodic commands, and usage of CI/TO to process remote commands and

telemetries via Ethernet. Control software also used cFS to communicate with the ground system by invoking its character user interface. Thanks to cFS, the team could quickly develop a portable telescope control system for coronagraph missions.

Another usage of cFS was its inclusion in a lunar surface imaging mission (33). Once again, by taking advantage of OSAL and PSP, the project can remain OS and hardware independent, as well as using cFE services, such as ES and SB, to assist several applications, like battery and camera applications, in obtaining and processing data. Like the other project, the SCH and CI/TO are used to control pre-determined messages but over Wi-Fi. Other applications were developed for this project, then added to cFS as plugins, such as an Inertial Measurement Unit (IMU) containing an accelerometer, a gyroscope, and a magnetometer to transmit their data from launch to landing on the lunar service, and a Three Nano Immersive Situational Awareness (NISA) Cameras, which take pictures of the lunar landers when landing.

2.2.2 Karvel

Karvel is a space On-Board SoftWare (OBSW) platform developed by Critical Software. Like NASA's cFS, Karvel aims to reduce development time and resource usage by promoting software reuse across missions. However, it introduces several architectural and operational differences from cFS.

Figure 4 shows an overview of the architecture of Karvel:

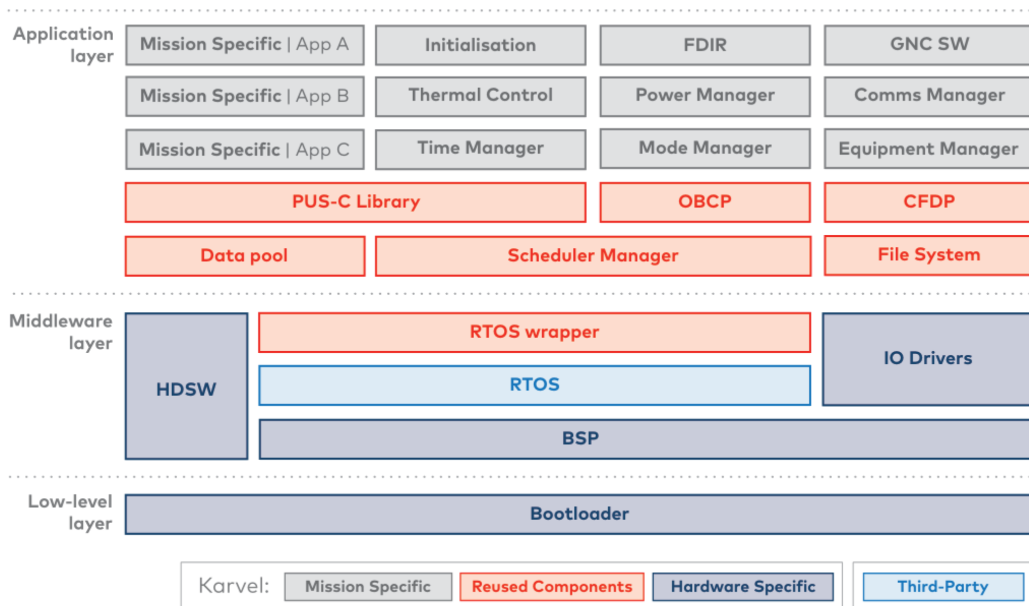


Figure 4 – Karvel architecture stack diagram (Critical Software)

At its core, Karvel is designed to be modular and portable, to let developers focus only on mission-specific applications while reusing core components. The platform is structured into three main layers:

- **Low-Level Layer**: This layer includes the bootloader, which is responsible for early system initialization and bootstrapping the operating system. It must be configured for each specific hardware platform.
- **Middleware layer**: Represents the agnostic architecture of Karvel.
    - RTOS Wrapper: Provides a unified API to interact with various types of RTOS, such as RTEMS, VxWorks, FreeRTOS, and Linux. It provides the application layer with an abstraction layer of RTOS primitives like semaphores, timers, and message queues, so the applications can be RTOS agnostic.
    - Hardware/Software Abstraction Layer (HDSW AL): Encapsulates hardware-specific logic to ensure portability.
    - I/O Drivers Abstraction: Standardizes access to input/output peripherals.
    - Board Support Package (BSP): Offers low-level access to hardware features, including kernel-level operations and device drivers.
- **Application Layer**: This layer contains both reusable and mission-specific components. All applications interact with the underlying system through the middleware, ensuring hardware and OS agnosticism. Key components include:
    - **PUS Library**: Implements the ECSS-E-ST-70-41C standard, which defines the Packet Utilization Standard (PUS), a protocol for structuring and managing telecommands and telemetry in space missions.
    - **On-Board Control Procedure (OBCP)**: Enables autonomous operations by executing predefined routines onboard, reducing reliance on ground control.
    - **CCSDS File Delivery Protocol (CFDP)**: Used for reliable file transfers between spacecraft and ground systems, supporting retransmissions and bandwidth optimization (34).
    - **Data Pool**: Tasked with storing data related to on-board parameters and manage how it is requested.

Another critical component is the Software Database (SWDB), a centralized source of project-specific information such as format of messages, on-board parameters, events, data types, and structures.

2.3 Game engine

Game engines are comprehensive software frameworks primarily designed for video game development and real-time rendering. Modern game engines provide built-in features such as scene rendering, object modelling, and coding support in specific programming languages. Additionally, they allow users to extend functionality through plugins, file imports, and scripts. Their modular architecture enables the integration of

middleware, such as sound and physics engines, while separating game logic from assets like models, sound, and animations. This modularity simplifies and automates the creation of complex projects by unifying various development tasks within a single program. (35) (36) (37)

While initially created for video game development, game engines have evolved significantly over the past few decades, finding applications in other industries, particularly in movies and television. Their ability to achieve high-fidelity real-time rendering, combined with their accessibility for creating custom 3D models, makes them valuable tools for visual effects and animation.

The concept of video games dates back as early as the 1940s, though early implementations were vastly different from modern game engines. Due to technological limitations, early video games often ran directly on hardware without the abstraction of a game engine. In 1952, A.S. Douglas developed one of the earliest games, a tic-tac-toe game named OXO on the Electronic Delay Storage Automatic Calculator (EDSAC). The game used a Cathode Ray Tube (CRT) to display its state and stored data in mercury delay lines, where sound waves were converted into electrical signals and maintained in a feedback loop as long as the system remained active. (38) (39)

As computer hardware advanced, video games became more sophisticated. The emergence of mainstream video game consoles like the Atari and personal computers like the Apple II enabled the development of more complex games. Notably, Pong utilized transistors for game logic, sound generation, and score tracking (40). By the late 1970s, Space Invaders became a popular game coded in Assembly, a low-level programming language that allowed direct manipulation of memory addresses. This approach improved efficiency by using the assembler to compile and generate executable machine code tailored to specific architectures. (41) (42) (43)

The 1990s marked the beginning of what would become the modern game engine. In 1992, *id Software* sought to innovate first-person shooters by improving gameplay dynamics, graphics, and modifiability. The company developed Wolfenstein 3D, the first game engine to use ray-casting for fast-paced gameplay. By locking the player's vertical vision, keeping all objects on the same height level, and using sprite-based graphics on a grid layout, the engine optimized rendering speed. Additionally, it took advantage of Video Graphics Array (VGA) technology to support 256 unique colours. Recognizing that players were reverse engineering their games to modify sounds and textures, *id Software* designed subsequent releases to facilitate modding by separating game logic from assets. (44) (45)

Only a year later, in 1993, *id Software* refined the acquired knowledge to further enhance future game releases, releasing *id Tech 1*, commonly known as the *Doom engine,* to run what would become *Doom*, the game. Often regarded as the first modern game engine, they learned from Wolf3D and designed it with modding in mind from the outset, fostering a strong community around the game *Doom*. The game introduced

Binary Space Partitioning (BSP), an algorithm that improved rendering efficiency by preventing the rendering of non-visible objects, allowing for larger and more complex maps without significant performance loss. Furthermore, it introduced vertical level variation, dynamic lighting provided by changes in light sources and environmental interaction (such as pressing buttons to unlock areas), and online multiplayer via peer-to-peer connections (known as *LAN parties*, which used Local Area Network(LAN) to connect with players at close proximity in the same network), further increasing player base support and engagement. (35) (45) (46) (47)

A few years later, *id Software* developed another game engine *id Tech 2*, also called *Quake*, made for the video game *Quake* of the same name which built upon Doom's advancements. *Quake* introduced true 3D environments and polygonal models, eliminating vertical restrictions on player movement. It also pioneered true client-server multiplayer over the internet and introduced QuakeC, a scripting language that allowed users to modify the engine more extensively. The later release of GLQuake, a port of Quake to OpenGL, enabled hardware acceleration, improving rendering speed and introducing visual enhancements such as transparency and shadows. (45) (48) (49) (50)

As industries beyond gaming demanded advanced real-time rendering, game engines incorporated features like physics-based rendering and seamless integration with external software platforms. The contributions of community-driven development, including user support and add-ons, further enhanced their adaptability and functionality.

In recent years, two game engines have dominated the gaming and entertainment industries due to their investment in features, graphical fidelity, and accessibility for both programmers and non-programmers: Unity and Unreal Engine. Although other engines like CryEngine have been popular, shifts in development priorities and documentation have made them less desirable for most projects. (51) (52) (53)

Unity, first launched in 2005 for macOS (then called Mac OS X), later expanded to support Windows and web browsers. Initially attracting independent developers and hobbyists, Unity gradually gained traction as new versions introduced features such as hardware acceleration, C++ compatibility, sprite rendering, plugin support, and improved graphics. Now in version 6, Unity is one of the most widely used game engines, serving both beginners and major game studios. It supports three programming languages: C# (a high-level language), UnityScript (similar to JavaScript), and Boo (similar to Python). It also includes a visual scripting tool called Bolt, allowing developers to create logic without coding. (36) (54) (55) (56) (57) (58) (59)

Unreal Engine, widely considered the most powerful game engine today, was first released in 1998 with support for Unix, Linux, macOS, and Windows. Initially relying on software rendering using only the CPU, Unreal Engine evolved to include Graphics Processing Unit (GPU)-based rendering, hardware acceleration, and modern rendering technologies such as Nanite and Lumen, and many leading advancements in ray tracing

and path tracing allows developers to create realistic environments more easily. It offers Blueprint visual scripting for non-programmers, alongside C++ support for developers. Unreal Engine's emphasis on realism has made it a staple in film and television, contributing to productions such as The Mandalorian, Westworld, and The Lion King (2019). (36) (37) (60) (59)

In this project, game engine capabilities can be leveraged to simulate spacecraft behaviour within a 3D environment. Their real-time rendering, physics engines, and customizable coding interfaces enable accurate simulations that integrate with external platforms. This allows for processing telemetry data, calculating spacecraft trajectories using mathematical models, and visualizing these behaviours in an interactive environment. Such integration demonstrates the potential of game engines beyond video game development, extending their use to scientific and industrial applications.

2.4 Digital twin concept

The concept of a digital twin involves the simulation of an object or a representation of a real object within a virtual environment, where the same laws and rules apply. In other words, the virtual object exhibits the same behaviours as its real-world counterpart. Depending on the level of accuracy and the degree of communication between the real and virtual entities—whether one-way or two-way—the concept of digital twins can be classified into three categories: digital model, digital shadow, and digital twin (61) (62) (63) (64).

- **Digital model**: This represents both the real and virtual objects along with their respective environments. It is the simplest form of representation, as the real and virtual objects operate independently but are governed by the same rules and principles. Consequently, manipulating one does not affect the other. However, if both objects are subjected to identical interactions, their outcomes are expected to align, assuming the model accurately reflects real-world behaviour. No data is shared automatically between the two, requiring manual intervention for synchronization.
- **Digital Shadow**: A digital shadow builds upon the digital model by introducing automatic one-way communication from the real object to its virtual counterpart. Any changes in the real object are automatically transmitted and reflected in the virtual model, while manual input is still required to influence the real object. With advancements in data transmission and processing, this synchronization can occur in near real-time, making the virtual object a “shadow” of the real one. The virtual object continuously replicates the real object’s state and behaviour, enabling more dynamic and responsive simulations.
- **Digital Twin**: The most advanced stage, the digital twin, extends the concept of a digital shadow by enabling automatic two-way communication between the real

and virtual objects. This bidirectional data exchange allows not only for the virtual twin to replicate the real object but also for the virtual entity to influence or control the real object remotely. The primary advantage of a true digital twin lies in its ability to perform advanced simulations and predictive analyses while executing real-time adjustments to the physical counterpart, thereby enhancing decision-making and operational efficiency.

The growing adoption of digital twins is driven by their significant advantages, including (61) (62) (63) (64):

- Cost Efficiency: Despite initial research and development costs, digital twins rely primarily on virtual development, reducing expenses related to prototyping, manufacturing, and testing. Unlike real objects, virtual twins can be instantiated quickly, modified easily, and do not require manual labour or reconstruction in case of damage during testing.
- Predictive Capabilities: When correctly implemented, a digital twin can predict and warn users of potential faults in a model or its behaviour without the need for physical testing.
- Monitoring and Maintenance: A digital twin allows for remote monitoring, control, and updates of a real object, improving maintenance efficiency and safety.

Digital twins have been successfully implemented across various fields, from agriculture to aerospace.

In agriculture, digital twins have been used in potato harvesting to optimize machinery performance (65). Sensors embedded in artificial potatoes collect data on impact and rotation caused by harvesting vehicles, helping drivers adjust harvesting equipment to reduce crop damage. Additionally, this data can be analysed to identify damage patterns for different potato varieties.

In medicine, digital twins meet a slower development due to the ethical concerns of application of engineering in the human body and side-effects in the body and psychology. With current technologies, covering a human body in sensors and camaras is not feasible for monitoring, which means many papers are purely theoretical, and only a few have applied the concept of digital twins to certain extents. Nonetheless, some tests have already been done by temporarily providing data from sensors. An example of such is using camara output from a person's head movement to detect carotid stenoses (66). In this situation, a camara is used to record and track head oscillations caused by pulsatile blood flow through the carotid artery when affected by a certain percentage of carotid occlusion. This data is then feed to a virtual model to simulate head oscillation in several states of blood flow and carotid occlusion percentage. This way, a virtual model can be trained and used to estimate severity of carotid stenosis by providing it with constant camara feedback.

As for the aerospace field, is important to mention one of the most important and often considered the earliest implementations of a digital twin, although not totally compatible with modern definitions, the overall idea and implementation was, the Apollo 13 simulators (63) (67). For the Apollo 13 mission, several physical simulators were built and configured with the same modules of the spacecraft to train the crew under several conditions, like take off, re-entry, and during several issues, to test how they can handle mistakes and recover. These physical twins could be readapted quickly to simulate and test several mission conditions. During the mission several issues appeared which meant that the mission control had to help the crew resolve them 400,000 kilometres away by using only data from telemetry and voice reports. To help resolve those issues, they used the telemetry and crew feedback to recreate the state of the craft in the simulators, and then attempt to solve it in those, so a solution could be transmitted to the apollo 13. Although, by modern standards and definitions, is questionable if it could be considered a digital twin, it is certain it was one of the earliest implementations of two important concepts in digital twins:

- Live telemetry: NASA's reliable communication network allowed the spacecraft to transmit telemetry data, enabling mission control to monitor and diagnose issues.
- Adaptability: The simulators could integrate the telemetry to simulate software conditions of the craft, which helped with troubleshooting and decision making.

Although Apollo 13's simulators may not fully meet the modern definition of a digital twin, they demonstrated the importance of real-time replication for monitoring, problem-solving, and operational resilience. The continued evolution of digital twin technology promises even greater advancements across various industries, enhancing efficiency, predictive maintenance, and decision-making processes.

2.4.1 Digital twin frameworks with game engines

To enhance fidelity of a digital twin simulation, incorporating some kind of 3D rendered environment would greatly be very beneficial, which is why a game engine will be used to simulate and visualize the satellite's digital twin. Additionally, rather than manually implementing all the complex equations related to orbit mechanics, the simulation will take advantage of existing software tools capable of handling these calculations automatically. With this approach, one can minimize development time and ensure high accuracy, only requiring the developers to do the initial setup and logical structure.

To figure out the best way to implement the digital twin, several game engines and software platforms were evaluated with the intended goal of identifying a solution that offers good 3D rendering capabilities and built-in or integrable features for orbit simulations and satellite behaviours in real-time.

The game engine will process telemetries and telecommands in real-time to closely replicate the satellite's physical behaviours. A selection of specialized tools depends on their compatibility with the game engines, ability to process certain kinds of data, and flexibility they offer for customization. Key options include:

- Unreal Engine 5 with MaxQ toolkit.
- Unreal Engine 5 or Unity 6 integrated with cloud platforms such as Azure Digital Twin or Amazon Web Services (AWS)' Internet of Things (Iot).
- Unreal Engine 5 or Unit 6 in combination with the Eclipse Ditto framework.

The goal is to properly evaluate how each option supports development and accurate digital twin simulation.

2.4.1.1 Unreal Engine 5 with MaxQ toolkit

Unreal Engine 5 (UE5) is a widely used game engine known for its realistic renders and complex game logic, featuring the Blueprint Visual Scripting system, which enables developers to create programming logic through a node-based interface, helping those with little to no programming skills to create their own projects.

For digital twin simulations, UE5 can be extended with MaxQ Toolkit plugin, which integrates NASA's SPICE Toolkit directly into the engine (68). MaxQ enables the simulation of orbital mechanics and spacecraft dynamics, making it a suitable candidate for an aerospace digital twin. Telemetry data such as satellite's position, attitude and overall system status can be simulated by injecting it via MaxQ in UE5, which is then mapped to visual and logical elements within the engine to create a visually realistic and logically accurate digital twin. Telecommands can also be input into de simulation to change craft behaviours and predict outcomes and identify potential system flaws before implementation.

Additionally, UE5 is supported by a large community, a vast library of plugins, and a licensing model that supports free use in non-commercial contexts (69).

2.4.1.2 Unity 6 or unreal Engine 5 with Azure Digital Twins or AWS IoT

Unity 6 (U6) offers similar features as UE5, including its own visual scripting language. Because a plugin like MaxQ is not available, integration with external platforms become necessary to fulfil digital twin requirements.

- Azure Digital Twins: Developed by Microsoft, this platform as a service (PaaS) models real-world systems using the Digital Twin Definition Language (DTDL), that supports components such as Interfaces, Commands, Properties, Relationships and Telemetry. Data exchange occurs through standard protocols like Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), and Hypertext Transfer Protocol Secure (HTTPS). (70) (71)

- AWS IoT: Amazon's cloud solution enables secure and scalable digital twins' implementations. Offers several services and solutions useful for satellite digital twins, such as TwinMaker that collects data from sensors and uses them to simulate digital twins in real-time, and Lambda, which can trigger events under certain satellite actions, useful for error handling and anomaly detection. (72) (73)

These cloud-based systems offer the advantage of scalability, pre-built features and security. Their workload is handled externally, allowing the game engine to focus purely on rendering and other logic. Which platform to use depends on the preferred programming language (C++ for AWS IoT and UE5, C# for Azure and Unity) and specific requirements of the digital twin that may only be available in one of them. Proper Software Development Kits (SDKs) are required to ensure good interoperability between engine and platform.

2.4.2 Mission Specific platforms

Dedicated software platforms exist specifically to support the design and operation of space missions. These platforms often come with a wide range of mission-specific features readily available out of the box. If needed, additional functionality can frequently be requested from the platform developers or developed by the user themselves. This reduces the burden on mission teams, who only need to perform the initial configuration to begin deployment.

Mission Design Simulator (MDS) by Blackswan Space is a desktop-based space mission simulator. It offers realistic environmental simulations and allows users to manage multiple satellites through a dedicated interface. Users can visualize orbits, sensor coverage, and perform analysis related to power and thermal management, as well as simulate rendezvous and docking scenarios. For advanced or custom simulations, MDS supports integration with MATLAB and Simulink, enabling users to implement their own algorithms, 3D models, and simulation logic. (74)

Sedaro is a cloud-based mission simulation platform accessible via a web client. It provides insights into key mission parameters such as satellite attitude, power consumption, thermal conditions, and orbital data. Users can either create custom mission scenarios or use predefined ones to test satellite performance in various environments. Sedaro includes an API, allowing developers to extend its functionality using high-level programming languages like Rust and Python. (75)

Aurora by D-Orbit is another cloud-native simulation platform. It utilizes services like AWS Ground Station and Leaf Space to ensure global coverage. Aurora features a 3D orbit visualization tool and a customizable interface. It also includes an automatic warning system to detect predefined anomalies, along with modern encryption and authentication protocols for robust cybersecurity. (76)

The Satellite Orbit Simulator is a free software that allows their users to setup satellite orbits in an Earth-Cantered-Inertial coordinates system, where variables such as moon gravity, earth atmosphere and sun radiation can be taken to account with a simple checkbox. It provides several features, such as highly customizable orbit parameters, import and export data, and a scripting language with several mathematical and physics equations that can be used in functions to change simulation behaviour. (77) (78)

These platforms are alternatives to game engines and other platforms that streamline the mission simulation and planning process by offering flexible, extensible environments with powerful visualization and analysis tools but often coming with high costs.

2.5 Communication protocols

In the context of satellite communication, the energy management and processing power take great priority in development, as satellites only means of energy besides the one that carries form launch is solar energy, provided by solar panels. In these environments, a reliable communication to earth must be lightweight to consume as little power as possible and reliable, to avoid wasting resources sending the same data more than once.

Therefore, a few protocols were chosen, commonly used in IoT devices – due to their hardware, energy limitations and already presence in critical systems – as data must be reliably sent and received.

2.5.1 MQTT

One of the most popular IoT protocols is the MQTT protocol (79) (80). This bi-directional protocol that uses TCP/Internet Protocol (IP) is based on a publish-subscribe model, where communication between client and server is made through a broker, a middleman server, which is one of its main features: by splitting communication between clients and servers through the broker, the server is not required to keep track of how many clients to send messages, as well as new features can be added without having to modify clients. The clients are only required to subscribe to a topic that the server is publishing data into. The published data is then sent to the broker that distributes it to any client subscribed to that topic, taking complexity and overhead from the server.

The broker is tasked with managing message queues and ensuring messages are delivered by specifying their Quality of Service (QoS) (79) (80). QoS allows both server and client to choose the effort taken to send a message, choosing between efficient or

reliable in case it needs to be retransmitted. That is, a message can either use little overhead but risk being lost or duplicated or take extra steps to ensure it is correctly delivered but consume more resources. QoS can be set to 3 levels, which with increased complexity and overhead (79) (80):

- QoS 0 – at most once: After a sending a message, no acknowledge is expected from the receiver, often compared with "fire and forget". The message is then deleted from the broker queue and cannot be retransmitted if requested.
- QoS 1 – at least once: The sender prepares a PUBLISH message and sends it, then it awaits a PUBACK response, confirming the receiver received it. If the receiver does not send the PUBACK within a time frame, the sender retransmits the message. The main disadvantage is the possibility of duplicate messages, which need to be handled accordingly.
- QoS 3 – exactly once: This communication involves a four-part handshake to ensure a message is delivered successfully once. The sender prepares a PUBLISH message, which is then stored locally in case it needs to be retransmitted and sends it to the receiver. After a receiver receives a message, responds with a PUBREC confirming the message was received. If the sender does not receive a PUBREC, it means the message failed to reach the receiver and sends the same message again with a Duplicate delivery of a Package (DUP) flag, warning the receiver that is retransmitting the message. When the receiver finally receives the message, transmits the PUBREC confirming it. When the sender receives it, can now confirm its PUBLISH was received and deleted the stored PUBLISH. Now it prepares and stores a PUBREL in case it needs to be retransmitted and transmits it. The receiver then receives the PUBREL and transmits a PUBCOMP. Now the sender can delete the stored PUBREL and the communication ends.

QoS level is an important feature, especially in the context of this project, as satellite performance is highly restricted by energy availability, choosing which QoS to use for certain payloads is critical to maintain efficiency and avoid wasted energy and performance (81).

There is also the possibility of a broker having no clients subscribed to a certain topic (79) (80). In such cases, messages transmitted form the server to the broker are discarded unless they have the flag "retained message" set to true, in which case, the last message of that topic with that flag true is stored along with its QoS in the broker to be transmitted once a client subscribes to it. This is useful to give new subscribers immediate data to process.

In situations that an unintended disconnection can occur, there is also the possibility of creating a persistence session between the client and the broker (79) (80). To achieve this, the client when first establishing a connection with CONNECT to a broker, sets the flag "cleanSession" to false, which will tell the broker to store current sessions, like client subscriptions, and messages not delivered to the client or awaiting acknowledgement. If that client becomes offline and later attempts to reconnect, the

client only needs to pass its identifier and the flag set to false, and the broker will resume from where it last was. On the other hand, by setting the flag to true all this information is discarded and the clients starts with a fresh connection to the broker each time it connects to it.

It is important to refer that MQTT does not have any kind of authentication and encryption besides security present in the OSI model layers, like Transport Layer Security (79) (80). Because it uses TCP/IP, it takes on any overhead and security present in this protocol beyond MQTT features like QoS. This overhead can take extra processing time and, consequently, energy consumption, which needs to be managed.

A MQTT packet is composed 3 parts: Fixed Header, Variable Header and Payload (79) (80). Because it uses TCP/IP, already contains several security measures and connection options natively.

The first field is the Fixed Header, which is composed of the Control Header, Flags and Remaining Length (79) (80). The Control Header contains information related to Packet Type, which tells what the message is for, such as:

- CONNECT: User for a client to connect to a server.
- PUBACK: Publish acknowledgement in QoS.
- SUBSCRIBE: Request to subscribe to a topic.

The Flag contains flags to specify behaviours of the Packet Type. This is especially important when publishing to a topic using PUBLISH Control Packet, as it is possible to specify 3 categories:

- DUP - Duplicate delivery of a Package: Used to inform if this is the first time a PUBLI SH packet is being sent or is a re-delivery.
- QoS – Quality of Service: Specify the level of Quality of Service.
- RETAIN: Informs if the current PUBLISH packet is to be retain this message in the broker for further use.

The other component of Fixe Header is Remaining Header, which stores the length of the Variable Header and Payload. Because of this, the size can vary (79) (80).

The next part of a packet is the Variable Header, which is an optional field in case extra information is required to send. It is composed of 5 fields:

- Protocol Name Length: The length of the protocol name, how many characters is composed of.
- Protocol Name: The name of the protocol being used.
- Protocol Level: The current version of the protocol being used.
- Connect Flags: Contains several flags to indicate certain presence of data in the payload and message configurations:
    - Flag Password and Flag Username: Flags that indicate if a password and a username are present in the payload.

  - Clean Start: allows the client to choose if it wants to start a new session, clearing any configurations and subscriptions, or continue an old connection, which will automatically continue from the previous session.
  - Will Retain, Will QoS and Will Flag. Clean Start: These 3 flags configure the Will message, which is a message the sender can setup in case it disconnects abruptly.
- Keep Alive: Counts the maximum time between two messages a client can send. If a server does not receive any kind of package form a client within a one and a half times the Keep Alive value, the connection is terminated.

The last content of a MQTT packet is the Payload, which is the actual data to send.

2.5.2 CoAP

The Constrained Application Protocol (CoAP) is a protocol based on User Datagram Protocol (UDP) that uses a request/response model in a specified Uniform Resource Identifier (URI), similar to HTTP, which means a client requests data directly to the server using an API, which then the server responds (82) (83). For communication, it implements a RESTful Architecture, so a client only requires to send a request to one of the endpoints.

As CoAP uses UDP, it must implement measures to guarantee correct delivery of datagrams, such to avoid duplicates, fragmentation, packets out of order or outright missing entirely (82) (83). Therefore, CoAP integrate a set of 8 fields in the data section of a UDP datagram, with the first 5 being part of the header.

The first field is Version (82) (83). It defines the CoAP version used to transmit the datagram.

The second field is Type and is used to defines the type of the CoAP message (82) (83). This is how CoAP attempts to ensure messages using UDP are received and acknowledged. There are four types:

- **Confirmable**: A message marked with this type is classified as a reliable transmission and are used for requests and responses. The recipient must answer to this type with either an Acknowledgement type, confirming the message was received and processed, or, in case the message lacks context, missing data or incorrectly formatted, sends a Reset type to inform it did receive it but there where issues. In this case, the rest of the Confirmable message is ignored. If the Reset is used, the rest of the Reset message is discarded when received. The transmitter of Confirmable will keep sending the message at exponentially increasing intervals until it receives a response or runs out of attempts.
- **Non-confirmable**: These messages do not require an acknowledgement and must not get one. In cases where a message can't be processed due to context, missing data or incorrectly formatted, the message is simply discarded. In these cases, the

recipient may send a Reset to inform it, but the rest must of the message must be discarded. Is possible to inform the sender of the non-confirmable that its message was received and processed by sending back a duplicate of the message.
- **Acknowledgement**: Used to acknowledge when a Confirmable type of message was received and processed successfully.
- **Reset**: Can be used for either Confirmable or Non-confirmable to inform the message was received but cannot be processed and got discarded accordingly.

The **Token Length** field is the third and, as the name implies, defines the length of the Token field (82) (83). Because the Token is optional, the Token Length can be zero.

The fourth field is **Code** and is used for Request/Response semantics. CoAP Requests can be divided into 4 sections (82) (83):
- **GET**: Request a data resource from the server.
- **POST**: Creates a resource with data in the server under a new URI. If that uniform resource identifier already exists, updates the data inside.
- **PUT**: Creates a resource with data in the server. If that resource already exists, replaces it with the new data.
- **DELETE**: Deletes a resource from the server.

Just like HTTP, CoAP uses HTTP response codes to inform of Code actions, and the GET, PUT and DELETE must be idempotent in their behaviour.
After receiving one of the Requests and processing it, the server can reply as such:
- **Success**: The Request was received and processed correctly.
- **Client error**: Request with incorrect syntax or not possible to complete.
- **Server error**: Request might be valid, but server failed to process it.

The fifth field and last in the header is Message identifier (ID) (82) (83). Each value is generated sequentially and identifies messages generated by the clients and thus must be unique as to void duplicated messages. If the same value is detected, the message is discarded. Message ID is also used to match messages Type, that is, when a client generates a Message ID for message of type Confirmable/Non-confirmable, it must be responded with an Acknowledgement/Reset with the same Message ID. Once they are matched, the same Message ID may be reused if its lifetime has expired.

The Token is the sixth field (82) (83), and its primary objective is to improve security of messages and to match requests and responses. If a CoAP message is absence of any security, it is important that the generated Token is nontrivial and randomized to mitigate off-path response spoofing. The Token is then included in the request message, and the same token must also be included in the response. This is important and why Message ID can’t be used for this case. Sometimes, a request may take time to process. To not leave the requester waiting, an empty Acknowledgment can be sent with the original Message ID and Token. When the original output is processed, then the server

can create a new Confirmable/Non-Confirmable with a new Message ID and the old Token, to let the client know the message it is receiving is still related to an older request.

The two last fields are Options and Payload, both optional (82) (83). Options provides extra context about the Payload in variable-length-value format, containing a delta value for the option number, and a length field, which indicates the length of the option. Some of the options possible to use are:

- **URI-Authority**: Indicates the host and port part of the URI.
- **URI-Path**: Indicates absolute path of URI.
- **Content-type**: Indicates the Internet media type, for example, plain text, JavaScript Object Notation (JSON), audio, images.

The Payload is any data in variable size to be sent and that can be specified in the Option section.

2.5.3 DDS and DDSI-RTPS

The Data Distribution Service (DDS) follows a publish-subscribe model similar to MQTT for reading and writing data (84). However, unlike MQTT, DDS does not rely on a central broker. Instead, it implements a Data-Centric Publish-Subscribe (DCPS) model, which includes defined communication semantics and is supported by a comprehensive API. These enhancements aim to ensure predictable resource usage, minimize overhead, and give developers greater control over how data is handled.

DCPS model is designed for scalability and enables objects to be shared directly between applications. In this model (84) (85):

- **Publishers** define which data objects they intend to publish and provide corresponding values.
- **Subscribers** specify which data objects they are interested in and retrieve values accordingly.

An important aspect of DDS is that applications must define their own topics and configure their QoS settings. They are responsible for publishing and retrieving data from these topics and ensuring that the topics function correctly.

Unlike MQTT, which uses a centralized broker to manage data flow and who is connected, DDS operates in a decentralized environment (84) (85). This environment, known as the "global data space" (GDS), is where all topics exist and data is exchanged between publishers and subscribers. Applications and the environment collectively manage the flow of data and connections, as there is no predefined architecture of GDS enforced by DDS.

Access to the GDS is achieved through two components (84) (85):

- **DataWriters**, used by publishers to write data to specific topics within the domain.

- **DataReaders**, used by subscribers to read data from those topics.

While the GDS can be implemented in various ways, it must be based on a predefined model that both publishers and subscribers understand. This model, referred to as the **data model**, acts as a template and includes:

- A **topic**, which serves as the unique identifier within the GDS.
- A **type**, which specifies how the data is structured and provides safety and processing options.
- Potentially other identifiers, depending on the complexity of the system.

By predefining this data model, DDS APIs can be optimized for both performance and safety, optimized to the specific type of data being exchanged.

Entities are also another important concept of DDS (84). They are an abstract class which represent active participants in the DDS environment and how they interact with it and other entities and can be configured with QoS policies, equipped with a listener, and associated with status conditions. Entities form the basis for communication. Some of those important components are:

- **DomainParticipant**: Establishes connection between an application and the DDS domain.
- **Publisher** and **Subscriber**: Responsible for managing the production and consumption of data, respectively, as well as interacting with DataWriter and DataReader.
- **DataWriter** and **DataReader**: Handle the actual transmission and reception of data for specific topics.
- **Topic**: Contains the name, type, and QoS of the data being sahred between publishers and subscribers.

The entity class being abstract means it cannot be instantiated directly. Instead, it provides a set of common behaviors that are extended by other classes like DomainParticipant, Publisher, and DataWriter.

All these components are considered DDS entities and share the following capabilities (84) (85):

- **QoS**: Policies that define how data is handled, such as durability, reliability, and latency.
- **Listener**: Allows an entity to receive asynchronous callbacks on certain events.
- **Status Conditions**: Certain status conditions can be used in conjunction with wait-sets for synchronous monitoring of events.

Entities can be either **enabled or disabled**. By default, are typically enabled automatically, unless specified otherwise via the ENTITY_FACTORY QoS policy. An entity must be enabled before it can fully participate in communication or invoke specific operations.

DDS includes 22 different QoS policies that allow developers to specify behaviours of the system. These policies define several aspects of how data is produced, distributed, and consumed. Below is a refined description of each policy (84) (85) (86):

1. **USER_DATA**: Attaches user-defined data to entities. Often used for identification, authentication, or conveying metadata relevant to discovery and security.
2. **TOPIC_DATA**: Associates additional user-defined data with a Topic. Typically used to extend topic metadata with extra features.
3. **GROUP_DATA**: Similar to USER_DATA but applied to Publisher and Subscriber entities. This data is accessible by their associated DataWriters and DataReaders and can help coordinate access or behavior.
4. **DURABILITY**: Controls whether previously published data is stored and made available to late-joining subscribers. Options include VOLATILE, TRANSIENT_LOCAL, TRANSIENT, and PERSISTENT.
5. **DURABILITY_SERVICE**: Provides configuration options for how DURABILITY is implemented, including history depth, cleanup delay, and resource usage
6. **PRESENTATION**: Specifies whether changes to multiple instances should be grouped and the order in which they are delivered. Supports coherent and ordered presentation across topics.
7. **DEADLINE**: Ensures data is updated at least once within a specified time period. Both DataWriters and DataReaders can use this to assert or expect update frequency.
8. **LATENCY_BUDGET**: Suggests a maximum limit on the delay between when data is written and when it should be received.
9. **OWNERSHIP**: Determines if data instances are shared among writers (SHARED) or controlled by one exclusive writer (EXCLUSIVE). Ownership arbitration is based on OWNERSHIP_STRENGTH.
10. **OWNSERSHIP_STRENGTH**: Used in conjunction with EXCLUSIVE OWNERSHIP to determine which DataWriter has control over a data instance.
11. **LIVELINESS**: Declares the mechanism by which a DataWriter asserts it is still alive. Helps DataReaders detect stale connections.
12. **TIME_BASED_FILTER**: Allows DataReaders to limit the frequency of data updates they receive. Only one update per specified interval will be delivered.
13. **PARTITION**: Provides a grouping mechanism for DataWriters and DataReaders using matching string identifiers. Only those in the same partition (the same string) can communicate.
14. **RELIABILITY**: Specifies the level of guarantee that data will be delivered from a DataWriter to a DataReader. Two settings are available:
15. **BEST_EFFORT:** Attempt to deliver data but offers no guarantee of delivery. Lost data will not be retransmitted. Suitable for scenarios where timeliness is more important than completeness.
16. **RELIABLE:** Ensures all data is delivered, using acknowledgments and retransmissions as needed. If data cannot be delivered immediately, like due to full

buffers, the writer will retry for a duration defined by max_blocking_time. However, delivery can still be constrained by other QoS settings like RESOURCE_LIMITS. Note that a RELIABLE DataWriter can send data to a BEST_EFFORT DataReader but a BEST_EFFORT DataWriter cannot respond to a RELIABLE DataReader.

17. **TRANSPORT_PRIORITY**: Suggests the priority of data on the transport layer. Useful when differentiated traffic handling is supported by the network.
18. **LIFESPAN**: Specifies how long data is valid after being written. Expired samples are removed from caches and not delivered.
19. **DESTINATION_ORDER**: Controls whether data is delivered in the order it was written (BY_SOURCE_TIMESTAMP) or the order it was received (BY_RECEPTION_TIMESTAMP).
20. **HISTORY**: Determines how many samples are kept for each instance. Can be KEEP_LAST (only recent N samples) or KEEP_ALL (store all samples within RESOURCE_LIMITS).
21. **RESOURCE_LIMITS**: Sets constraints on memory usage and number of samples or instances. Important for system predictability and performance.
22. **ENTITY_FACTORY**: This policy affects the behaviour of newly created entities.
23. **WRITER_DATA_LIFECYCLE**: Controls automatic disposal of data instances when a DataWriter no longer exists or unregisters them.
24. **READER_DATA_LIFECYCLE**: Configures how long a DataReader maintains information about instances without writers or valid updates.

DDS does not mandate the use of a specific transport protocol like UDP or TCP/IP (84) (85). Instead, it defines an abstract data-centric publish-subscribe model. As a result, it is the developer's responsibility to ensure that interoperability between different implementations is present.

Due to the many features of DDS, such as configurable QoS policies and automatic discovery, these challenges are addressed by the DDS Interoperability Real-Time Publish-Subscribe (DDSI-RTPS) wire protocol, commonly referred to as RTPS (86). RTPS aims to establish a standard wire protocol that enables interoperability between different DDS implementations. To achieve this, RTPS uses two foundational concepts: the **Platform Independent Model** (PIM) and the **Platform Specific Model** (PSM).

The PIM is an abstract model that defines what the system should do, without specifying the details of how it should be implemented. It is composed of four key modules (86):

- **Structure**: Defines the communication entities and endpoints. Two primary actors function as endpoints: the Reader and the Writer. These are responsible for receiving and sending RTPS messages, respectively, and for mapping those messages to DDS entities.

- **Message**: Specifies the format and structure of messages exchanged between Writers and Readers to ensure compatibility with the RTPS protocol.
- **Behaviour**: Establishes the rules and constraints governing message exchanges between endpoints, including state transitions, time constraints, and permitted message types.
- **Discovery**: Allows participants to obtain information about available endpoints and enables the automatic discovery of Readers and Writers within the DDS domain to facilitate communication.

As PIM offers the blueprint to be used by protocols, PSM is the mapping of those rules to a specific platform, to ensure the compatibility between different protocols. Although it was designed for UPD/IP, it can be adapted to other protocols. When mapping to the desired protocol, a few features not natively offered by DDS must be implemented for each of the four key modules. A few of them are the following (86):

- **HistoryCache**: For the Writer, it contains a limited history of changes to data-objects. These changes are used to send data to late joiners, resend lost data in cases where reliability is required, or when maintaining DURABILITY. The QoS being used will affect how much and what data is being stored, most notably, RELIABILITY, HISTORY, and DURABILITY will have great effect. HistoryCache will then be passed to the Reader, which will be able to read the changes and apply them based on its custom QoS.
- **AckNak**: Allows the Reader to warn the Writer if a sequence of numbers is missing.
- **Heartbeat**: Writer tells the Reader that it is available to communicate numbers stored in the HistoryCache and uses AckNak to confirm or warn that some are missing.
- **Simple Participation Discovery protocol (SPDP)**: Tasked with discovering other participants and any properties they have. For each participant, an endpoint for reading and another for writing are created.
- **Simple Endpoint Discovery Protocol (SEDP)**: When two participants exist, SEDP helps finding each other's endpoints for reading, and writing and obtain any configurations.

2.6 Chapter resume

The state of art presents and reviews technologies, frameworks, and methodologies relevant to the develop of a digital twin for satellite orbit simulation, it begins by introducing the concept of mission control system, which is important for managing spacecraft operations in conjunction with telemetry, tracking, and telecommand functionalities. Two mission control platforms are examined: YAMCS, an open-source solution compliant with ESA and CCSDSA standards, and Lighthouse, a web-

application developed by Critical Software to interface with the Karvel satellite framework.

Two satellite software platforms are then compared: NASA cFS and Karvel. Both offer a layered architecture with reusable modules and others tailorable to specific missions, and compatibility with most used standards in satellite missions. cFS has proven already its capabilities in real-life missions, but Karvel, while still in development, attempts to provide a lightweight platform, tailored to be compliant with more European and international standards.

Game engines are discussed as a tool for real-time 3D rendering and simulation, starting with the history and evolution of game engines beginning with early hardware-based video games to modern game engines like Unreal Engine and Unity, highly adaptable, full of features capable of providing solutions to both entertainment industry and scientific industry. Because of the rendering capabilities and features, they are suited for simulating and visualising satellite orbits in real time, while also integrating telemetry and telecommand transmission in real-time.

Mission platforms are also reviewed, such as MDS, Sedaro, and Aurora for their simulation features, orbit visualization, thermal analysis, and API extensibility, offering an alternative to game engines.

Communication protocols critical in satellite-ground interactions are also discussed, such as MQTT, CoAP, and DDS/DDSI-RTPS, as each provides advantages and disadvantages on reliability, ease of implementation, and features. This is important as satellite communication happens in a highly constrained environment, where energy and bandwidth are limited.

3 Fundamentals of Orbital Mechanics

This chapter discusses some important attributes of orbit mechanics necessary to comprehend how certain parts of the application were implemented and how they behave.

3.1 Astronomical models of the solar system

When simulating the solar system, is important to choose an adequate model to represent orbital dynamics accurately. Two models have been widely used for several centuries: geocentric and heliocentric. While in modern days the heliocentric is considered the most accurate, the geocentric contains a different perspective that can be advantageous in the project.

3.1.1 Geocentric

The geocentric model places Earth at the centre of the solar system, with all the other celestial bodies, including the Sun, orbiting around it (87) (88). Because this model assumes Earth as the centre of gravity of the solar system, modification add to me made to explain how certain bodies rotated around others, like other planets rotating around the Sun, causing the illusion of retrograde and prograde motion. To account for this observed retrograde and prograde motion of bodies, the concepts of deferent and epicycle had to be thought of, as to explain more complex orbital paths (87). An example of a geocentric diagram is present in Figure 5:

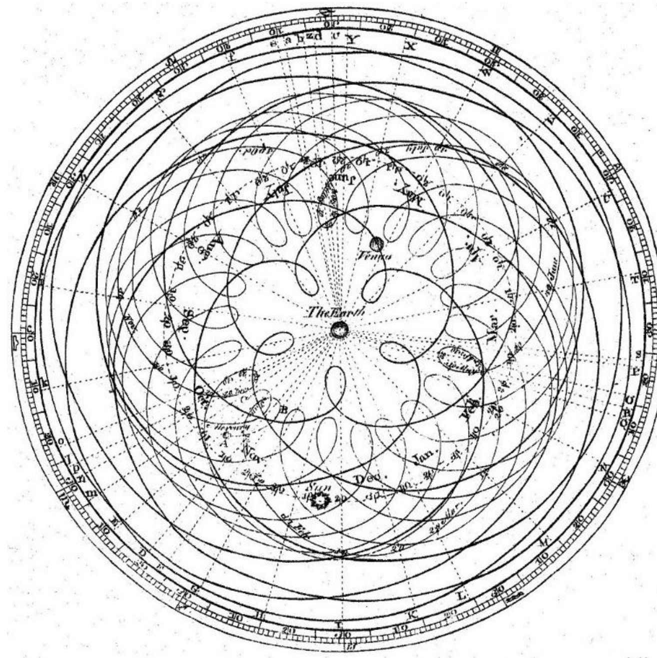


Figure 5 – Geocentric diagram (Largegreenbird)

3.1.2 Heliocentric

In a heliocentric model, the Sun is placed at the centre of the solar system, with all the other celestial bodies, including Earth, orbiting around it in elliptical paths (87) (88). In this model, deferent and epicycle motions are still present and required to explain certain celestial bodies behaviours, like satellites around their planet. This model is the most accurate representation of the solar system showing all bodies positions relative to the sun (87) (88). Figure 6 illustrates an example of heliocentric solar system orbits of the main celestial bodies:

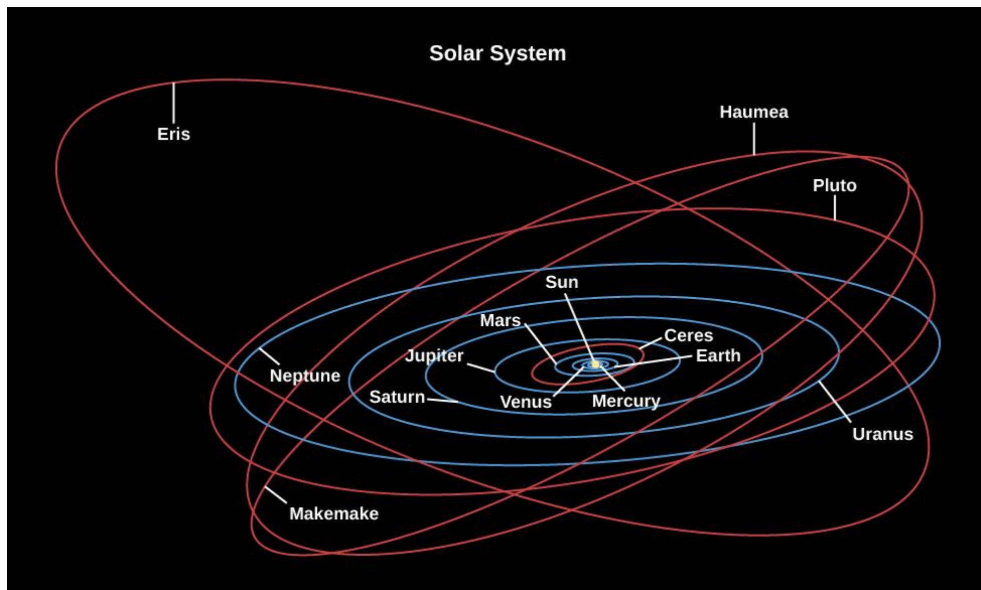


Figure 6 – Heliocentric diagram (Pressbooks)

3.1.3 Heliocentric in a Geocentric reference frame

A hybrid approach between geocentric and heliocentric models can be employed for a different but accurate model of the solar system and its behaviours. A heliocentric model in a geocentric reference frame contains all orbital mechanics based on the heliocentric model to ensure accurate body movements, but with a visual reference frame around the Earth, like the geocentric model. This configuration is similar to the Earth-Centred Inertial frame, often used for human-made satellites orbiting Earth, but applied to the solar system. In this system, is possible to see how all bodies move around the Sun and how their position changes in relation to Earth (87) (88). This concept has similarities to a geoheliocentric model called Tychonic system, where Earth is at the centre of the universe, the Sun and Moon rotate around it, and the remaining celestial bodies rotate around the sun with mathematically accurate orbits like in heliocentric model (88). In Figure 7 is visible how the gravity of two bodies interacts when using Earth as a reference frame:

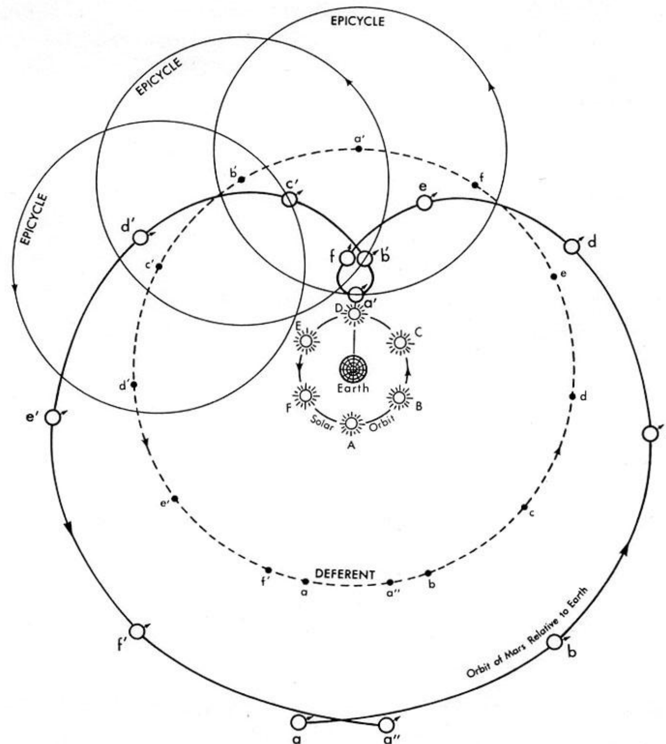


Figure 7 – Geocentric reference frame in a heliocentric model (Njit)

3.2 Ellipse

In orbital mechanics, the path followed by celestial bodies under the influence of gravity is most commonly an ellipse, in accordance with Kepler's First Law, which states that planets move in elliptical orbits with the Sun located at one of the foci (89) (90).

This section describes the fundamental characteristics of an ellipse, which form the basis for modelling orbital paths of celestial bodies.

An ellipse is defined as the set of all points for which the sum of the distances to two fixed points, called foci, is constant (91). In orbital contexts, one of these foci represents the central celestial body being orbited, such as the Sun in planetary orbits or Earth in satellite orbits, while the second focus remains unoccupied, an empty focus (92).

Figure 8 describes the key components of an ellipse (91) (93):

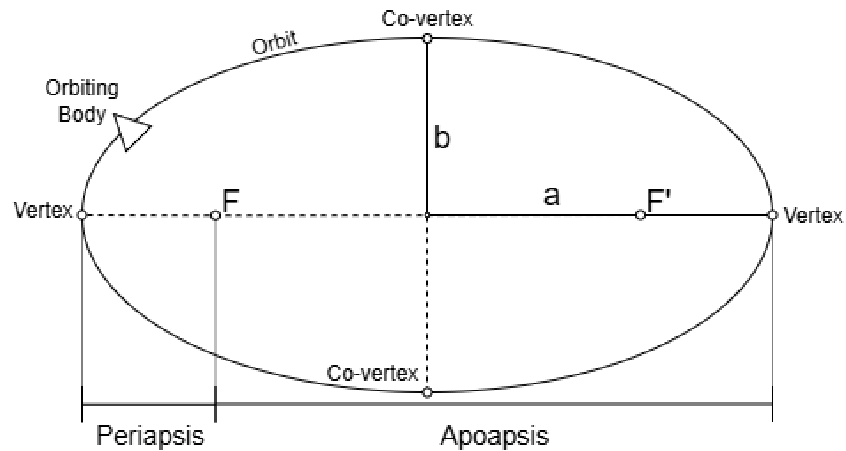


Figure 8 – Important points and segments in an Ellipse

- **F and F′ (Foci)**: The two fixed points that define the ellipse. In orbital mechanics, one focus is occupied by the central body (the body being orbited), while the other remains empty.
- **Vertex:** The points where the ellipse intersects the major axis. These represent the furthest extents of the orbit along its longest dimension.
- **Co-vertex:** The points where the ellipse intersects the minor axis, which is perpendicular to the major axis.
- **a - Semi-major Axis:** The distance from the centre of the ellipse to a vertex. The full length of the major axis is 2a, and it defines the overall size of the orbit.
- **b - Semi-minor Axis:** The distance from the centre to a co-vertex. The full length of the minor axis is 2b, and it defines the "height" or width of the ellipse.
- **Periapsis:** The point in the orbit where the orbiting body is closest to the central body. It corresponds to the vertex nearest to the occupied focus.
- **Apoapsis:** The point in the orbit where the orbiting body is farthest from the central body. It corresponds to the vertex farthest from the occupied focus.
- **Orbiting body:** The object that orbits the central body located at one of the foci. Orbital motion is generally assumed to be counterclockwise in standard diagrams.

One additional geometric property that defines the shape of an ellipse is its eccentricity (93). This is a dimensionless value that does not represent a physical quantity like distance or angle but rather serves as an indicator of how elongated or "stretched" the ellipse is compared to a perfect circle.

Eccentricity is calculated using the formula (93):

$$e = \frac{\sqrt{a^2 - b^2}}{a}$$

Where *e* is the eccentricity, *a* is the semi-major axis, and *b* is the semi-minor axis.

Depending on the value, eccentricity determinates the type of conic section (94) (95):

- $e = 0$: A perfect circle where the foci points are in the same position at the centre, so every point of the orbit measures the same distance at the centre.
- $0 < e < 1$: Ellipse with increasing degrees of eccentricity. The closer to 1, the more eccentric it becomes.
- $e = 1$: Parabola. A parabolic orbit happens when the exact energy for an orbiting body to escape the gravitational pull is reached.
- $e > 1$: Hyperbola, which happens when the orbiting body as excess energy to escape the gravitational pull.

3.3 Reference Plane and Reference Frame

The position and motion of celestial bodies or artificial satellites are expressed relative to a defined coordinate system. To accurately describe orbital behaviour, it is essential to establish both a reference plane and a reference frame. These components provide the geometric and spatial context required to define orbital elements and interpret orbital dynamics.

3.3.1 Reference Plane

A reference plane is a two-dimensional surface used as a baseline for measuring angular components in orbital elements, such as inclination and longitude of the ascending node (87) (88) (96). The choice of reference plane depends on the context of the orbit being analysed. For example (97) (98):

- The Earth's equatorial plane is typically used when working with Earth-orbiting satellites. This plane is aligned with Earth's equator and is suitable for describing satellite motion relative to Earth's rotation.
- The ecliptic plane is commonly used when analysing planetary orbits. It corresponds to the plane of Earth's orbit around the Sun and serves as a standard reference for solar system dynamics.

Figure 9 illustrates the placement of both reference planes in Earth-related orbital contexts:

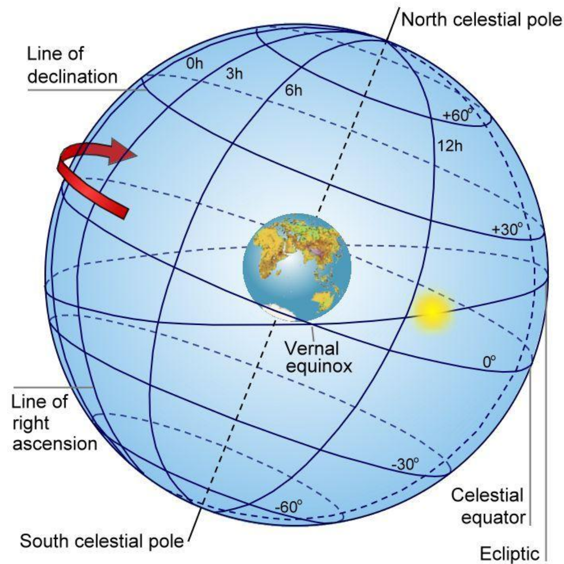


Figure 9 - Reference planes and markers used in celestial coordinate systems

Choosing an appropriate reference plane is crucial, as angular measurements such as inclination and node longitude can vary significantly depending on the plane used.

3.3.2 Reference Frame

The reference plane is a coordinate system that defines an origin of the coordinate system, the direction of the coordinate axes, and whether the frame is inertial (non-accelerating) or non-inertial (rotating or accelerating) (87) (97) (96). The inertial property determines whether the coordinate system rotates along with the central body. In an inertial frame, the axes remain fixed, while in a non-inertial frame, the coordinate system rotates with the body it is centred on.

One widely used inertial reference frame is J2000, based on Earth-Centered Inertial (ECI) frame, has its epoch at J2000.0. In this frame (97):

- Origin is at the centre of the Earth.
- Z-axis points toward the celestial north pole.
- X-axis points toward the vernal equinox.
- Y-axis completes the right-handed coordinate system.

The epoch time for J2000 is defined as 12:00 Terrestrial Time on January 1, 2000.

Another commonly used frame is Ecliptic J2000, a heliocentric ecliptic reference frame aligned with the ecliptic plane at the same J2000 epoch (99) (100):

- origin is at the centre of the Sun.
- XY-plane lies in the ecliptic plane.
- Z-axis is perpendicular to the ecliptic.

- X-axis points toward the mean vernal equinox of J2000.

Figure 10 shows how the axis of the coordinate system of J2000 are placed in relation to Earth:

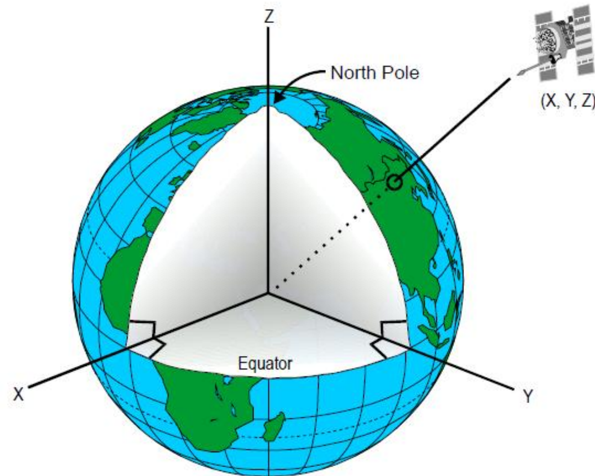


Figure 10 – ECI reference frame (Wikipedia)

3.4 Orbital elements

To properly simulate satellite behaviour in a digital twin, it is important that the parameters that define an orbit are known and understood. The orbital elements are six Keplerian elements that define the size, shape, and orientation, as well as the position of the satellite or celestial body in orbit at a given time. The name Keplerian comes from Johannes Kepler that defined 3 laws of planetary motion.

The two elements that describe the shape of an orbit are (89) (90) (101) (102):

- **Semi-Major axis(a):** Defines the size of the orbit. It is the distance from the centre of the ellipse to one of its vertices. The full length of the major axis is twice this value.
- **Eccentricity(e):** Describes the shape of the orbit. It is a dimensionless value that indicates how much the orbit deviates from a perfect circle, where an eccentricity value of 0 being a circular orbit, between 0 and 1 being an elliptic orbit, exactly 1 is a parabolic trajectory, and above 1 is a hyperbolic trajectory.

The remaining four elements define the orientation of the orbit in space and the position of the orbiting body along the orbital path (89) (90) (101) (102).These are illustrated in Figure 11:

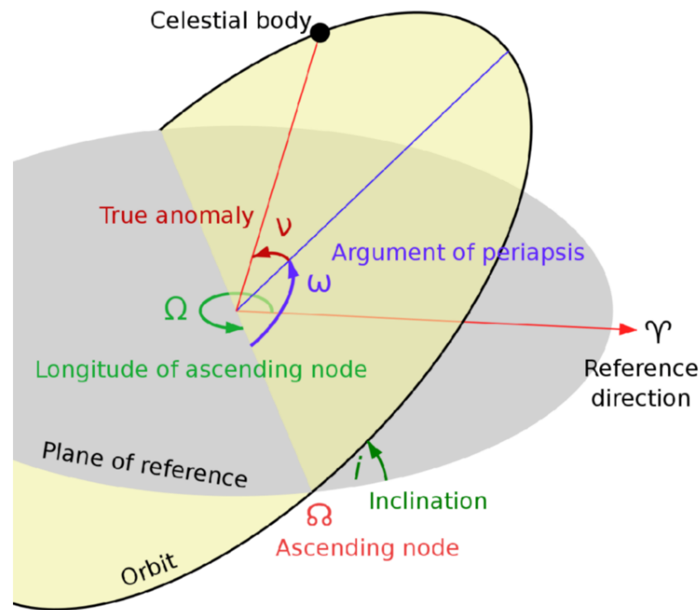


Figure 11 – Diagram of the Keplerian elements that describe an orbit (Wikipedia)

- **Inclination(i):** The angle between the orbital plane and the chosen reference plane. It determines the tilt of the orbit.
- **Longitude of Ascending Node(Ω):** Defines the horizontal orientation of the orbit. It is the angle between the vernal equinox direction and the ascending node, which is the point where the orbit crosses the reference plane from south to north. This element rotates the orbital plane around the Z-axis of the reference frame.
- **Argument of Periapsis(ω):** Specifies the orientation of the orbit within its orbital plane. It is the angle from the ascending node to the periapsis, measured in the direction of motion.
- **True Anomaly(ν):** Describes the position of the orbiting body along its orbit at a specific time. It is the angle between the periapsis and the current position of the body, measured at the focus of the ellipse.

3.5 Chapter resume

This chapter introduced the fundamentals concepts of orbital mechanics necessary to understand and simulate the motion of celestial bodies nor artificial satellites within a digital twin application.

The chapter began by exploring conceptual models of the solar system, such as the geocentric and heliocentric models. While heliocentric model describes planetary motion accurately in the solar system, a geocentric model where Earth is the centre of

the coordinate system brings advantages. This is why a hybrid approach was discussed, as a heliocentric motion visualized through a geocentric frame offers several benefits.

The geometry of orbital paths was introduced through the concept of ellipse, where the main characteristics of them were defined, which govern not just the shape of an ellipse, but if it can even be considered an ellipse, such as circles and hyperbolas.

To describe motion in a 3D space, reference plane and reference frame were required to also explain, as they serve as the baseline for measuring orbital elements, defining the origin, orientation of the coordinate system, and determining whether the system is inertial or rotating with the central body.

At the end, the six Keplerian orbital elements were presented, that collectively define an orbit's shape, orientation, position, and location of the orbiting body at a given time.

4 Methodology

This chapter outlines the architecture of the digital twin visualizer, which integrates multiple technologies working together in a coordinated system.

4.1 Requirements

Before designing the high-level architecture of the system, it was essential to define the expected functionalities and behaviours under various operational states. Accordingly, requirements gathering was carried out by professionals in the field of aerospace systems.

Each requirements follows a standardized format for better consistency and organization:

- **Name**: A unique identifier that follows a specific naming convention to distinguish each requirement and classify it by type. The format is:

  `<project>-<artifact>-<type>-<id>:Name`

  where:

  - `<project>`: The acronym of the project. In this case, DT (Digital Twin).
  - `<artifact>`: The type of artifact, such as REQ for requirement.
  - `<type>`: The category of the artifact, such as F for functional or NF for non-functional.
  - `<id>`: A four-digit identifier starting at 0000. IDs increment by tens to allow space for related requirements to be inserted later without disrupting the sequence.
  - `Name`: A small and descriptive title for the requirement, and it is separated from the identifier by a colon (:).
- **Description**: A description of a functionality or property the application must do.
- **Rationale**: A justification for why the requirement was included.
- **Priority**: A priority level to help guide implementation decisions and resource allocation. The priority levels are defined as follows, in descending order of importance.
  - **Critical:** Any feature required as per definition of a digital twin in a simulated environment. These requirements must be implemented for the project to be considered complete and to qualify as a true digital twin or digital shadow. Without them, the application fails to meet its fundamental objectives.

- **High:** Important features that are strongly advised to be implemented due to their usefulness and uniqueness. Their absence does not invalidate the project, but it may limit its effectiveness in simulating satellite operations.
- **Medium:** Features that provide additional capabilities and improve usability or flexibility. While beneficial, their absence does not significantly affect the core functionality of the digital twin concept. These requirements may be deferred or partially implemented.
- **Low:** Minor features that do not affect the functionality of the digital twin application and can be outright dismissed, that is, their absence has little to no impact in the project. Their objective is to provide a better UI/UX and visual customization.

Based on those functionalities assessment of state of art, the following requirements have been defined:

Table 1 – Functional and Non-functional requirements

Functional Requirements	Non-functional Requirements
DT-REQ-F-0010:NearRealTime	DT-REQ-NF-0010:Usability
DT-REQ-F-0020:AdaptableCommunication	DT-REQ-NF-0020:DataSecurity
DT-REQ-F-0030:OrbitViewer	DT-REQ-NF-0030:Replayability
DT-REQ-F-0040:OrbitComparer	DT-REQ-NF-0040:TelecommandCompletion
DT-REQ-F-0050:TimeManipulation	DT-REQ-NF-0050:Performance
DT-REQ-F-0060:MissionControlIntegration	DT-REQ-NF-0060:Portability
DT-REQ-F-0070:TelemetryAndTelecommand	DT-REQ-NF-0070:KnowledgeTransfer
DT-REQ-F-0080:DataVisualization	
DT-REQ-F-0090:TelemetrySelection	
DT-REQ-F-0100:DataStorage	
DT-REQ-F-0110:Customizability	

4.1.1 Functional Requirements

This section describes the functional requirements of the digital twin application, along with their rationale.

4.1.1.1 DT-REQ-F-0010:NearRealTime

- **Requirement**: The digital twin application shall receive and reflect telemetry data from the Mission Control in near real-time, immediately after processing by the OBSW.

- **Rationale**: Real-time updates ensure the digital twin remains synchronized with the actual satellite, enabling users to monitor the current state and detect potential issues promptly before any minor or catastrophic consequences occur. This is especially important to hardware, as it usually can't be replaced or fixed. Software can occasionally be upgraded, although will cost development time and other resources. Because time is still necessary to create a package, transmit it, receive it and unpack it, it is expected that data being shared won't be available in real time.
- **Priority**: High.

4.1.1.2 DT-REQ-F-0020:AdaptableCommunication

- **Requirement**: The application shall provide a graphical interface to configure communication parameters (e.g., address, port, username and password).
- **Rationale**: A configurable interface allows users to connect to different Mission Control systems without modifying the source code. This reduces resources required for post-deployment support.
- **Priority**: Medium.

4.1.1.3 DT-REQ-F-0030:OrbitViewer

- **Requirement**: The application shall include a 3D visualization environment that displays the satellite's current orbit, state, and relevant celestial bodies. The orbit calculations shall use models and standards commonly adopted in satellite and space engineering to ensure accurate orbit prediction.
- **Rationale**: The main objective of this project is to provide a realistic 3D viewer that helps users to intuitively understand the satellite's position and trajectory in relation to other celestial objects in the solar system. This assists both experienced and inexperienced users in using the application effectively. Accurate orbit modelling is essential for realism, reliability, and predictability.
- **Priority**: Critical.

4.1.1.4 DT-REQ-F-0040:OrbitComparer

- **Requirement**: The application shall support a secondary satellite simulation where users can test and compare new orbital parameters before applying them via telecommands.
- **Rationale**: This feature allows users to evaluate the impact of orbital changes in a safe, simulated environment before applying them to the real system, as well as test several hypotheses to take logical conclusions.
- **Priority**: Critical.

4.1.1.5 DT-REQ-F-0050:TimeManipulation

- **Requirement:** The application shall allow users to modify the simulation time, including fast-forwarding, rewinding, and jumping to specific dates.
- **Rationale**: Time manipulation enables users to replay past events or simulate future scenarios to assess system behaviour over time.
- **Priority**: High.

4.1.1.6 DT-REQ-F-0060:MissionControlIntegration

- **Requirement**: The application shall embed an instance of the Mission Control interface within the digital twin environment.
- **Rationale**: Integrating Mission Control into the digital twin provides a unified interface for monitoring and control tasks. The main objective is to provide an all-in-one solution, where all features can be accessed within a single application.
- **Priority**: Medium.

4.1.1.7 DT-REQ-F-0070:TelemetryAndTelecommand

- **Requirement**: The application shall display received telemetry data and provide an interface for manually sending telecommands.
- **Rationale**: Direct access to telemetry and telecommand functionality enhances user control and situational awareness, enabling them to quickly monitor satellite state and deploy changes whenever necessary.
- **Priority**: Critical.

4.1.1.8 DT-REQ-F-0080:DataVisualization

- **Requirement**: The application shall support graphical plotting of telemetry data (e.g., temperature, energy, orbital parameters) over time.
- **Rationale**: Graphical plots help users quickly identify trends, anomalies, and system performance over time.
- **Priority**: High.

4.1.1.9 DT-REQ-F-0090:TelemetrySelection

- **Requirement**: Users shall be able to select which telemetry parameters they want to receive updates for and display.
- **Rationale**: By providing methods to choose what telemetry the user wants to receive/be updated, users can more easily focus on relevant information important for their tasks. Additionally, selective telemetry reduces data clutter and bandwidth usage, further helping the satellite platform to reduce energy consumption and processing time.
- **Priority**: Medium

4.1.1.10 DT-REQ-F-0100:DataStorage

- **Requirement**: The application shall store telemetry data locally for future analysis, plotting, and simulation. The application shall log significant events (e.g., telecommand execution, anomalies, simulation changes) with timestamps.
- **Rationale**: Historical data is essential for diagnostics, replaying past events, and simulating previous system states. The digital twin must provide methods to access this information. Event logs provide traceability and support debugging and mission analysis like replaying past events of satellite state to analyse behaviours. This data can also be exported if the user requires it
- **Priority**: Medium.

4.1.1.11 DT-REQ-F-0110 :Customizability

- **Requirement**: The application shall allow users to customize non-functional aspects such as units, language, and display settings.

- **Rationale**: Customization improves user experience by adapting the interface to individual preferences.
- **Priority**: Low.

4.1.2 Non-functional Requirements

This section outlines the non-functional requirements that define the quality attributes of the system.

4.1.2.1 DT-REQ-NF-0010:Usability

- **Requirement**: The application shall provide an intuitive graphical user interface (GUI) for all major functionalities.
- **Rationale**: A user-friendly interface reduces the learning curve and enhances productivity. This helps minimize the resources required for training and reduces the time users spend performing tasks.
- **Priority**: Medium.

4.1.2.2 DT-REQ-NF-0020:SecureData

- **Requirement**: All stored and transmitted data shall be encrypted using industry-standard encryption protocols.
- **Rationale**: Encryption protects sensitive mission data from unauthorized access and tampering. This is critical, as unauthorized users could gain access to classified satellite information or even take control of the system. However, since this project is a prototype focused on demonstrating functionality and is executed in a restricted environment, security was not prioritized.
- **Priority**: Low.

4.1.2.3 DT-REQ-NF-0030:Replayability

- **Requirement**: The application shall support replaying stored telemetry data to simulate past system states.
- **Rationale**: Replay functionality allows users to simulate past events within the application, supporting post-event analysis and training.
- **Priority**: Medium.

4.1.2.4 DT-REQ-NF-0040:TelecommandCompletion

- **Requirement**: The application shall confirm the success or failure of each telecommand sent.
- **Rationale**: Acknowledgements ensure users are informed of command execution status and can respond to failures. This is critical, as not giving the correct feedback may mislead the user by taking the wrong assessment or by sending unnecessary or harmful telecommands.
- **Priority**: Critical.

4.1.2.5 DT-REQ-NF-0050:Performance

- **Requirement**: The application shall maintain responsive performance with minimal latency during real-time operations and visualizations.

- **Rationale**: High performance is critical for real-time monitoring and interaction, as delayed communications might not give enough time to deploy countermeasures. While some instability is acceptable during prototyping, the application must remain responsive enough to demonstrate key features.
- **Priority**: Medium.

4.1.2.6 DT-REQ-NF-0060:Portability

- **Requirement**: The application shall be deployable on multiple platforms (e.g., Windows, Linux) with minimal configuration.
- **Rationale**: Cross-platform support increases accessibility and ease of deployment.
- **Priority**: Medium.

4.1.2.7 DT-REQ-NF-0070:KnowledgeTransfer

- **Requirement**: The application shall be designed and documented to allow new developers to quickly understand and continue development. This includes adhering to coding standards, using a modular architecture, and providing comprehensive documentation and guides.
- **Rationale**: Providing documents and guides and using standardize coding practices assists new users and developers to lower the time required to learn the architecture of the application and how it works. This reduces onboarding time and minimizes the risk of misinterpretation or incorrect implementation.
- **Priority**: Medium.

With the main functional and non-functional requirements in place, the main functionalities of the system can be further analysed and then developed.

4.2 High-level architecture

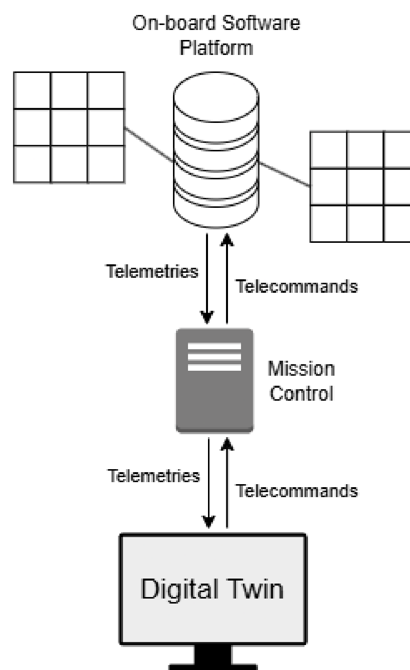
To achieve the desired functionalities, the system can be divided into three main components: the satellite platform, the mission control centre, and the digital twin. The satellite platform consists of an on-board software (OBSW) platform running on a microcontroller, simulating a satellite in orbit in real time. It continuously monitors its state and, when necessary, transmits telemetry data to the mission control centre.

The telemetry includes information such as the satellite's current state, orbital parameters, payload status, and any critical events that need to be reported. After receiving and processing this data, the mission control forwards relevant information to the digital twin for visualization and analysis.

In addition to telemetry, the system also supports telecommands. These commands can originate from either the mission control centre or the digital twin and are sent to the satellite platform. The OBSW must be capable of interpreting and executing these commands, which may alter the satellite's behaviour or payload configuration.

To ensure the system remains responsive and accurate, the simulation must operate as close to real time as possible. This is essential for promptly alerting users to any critical changes in the satellite's status.

The mission control centre plays a central role in managing communications. It tracks incoming telemetry processing while also providing an interface for sending telecommands and redirect telecommands from the digital twin to the OBSW. These commands are dispatched simultaneously to both the OBSW and the digital twin to maintain synchronization across the system.



Therefore, the high-level architecture involves the OBSW transmitting data to the Mission Control Centre, where it undergoes further processing before being forwarded to the digital twin. The digital twin then applies this data to its simulated satellite model. The digital twin also provides a real-time interface for visualizing telemetry and allows users to send telecommands via GUI that can modify the satellite's behaviour.

5 Proposed approach

This chapter describes the proposed architecture and workflow of the system, referencing technologies and communication protocols required to fulfil the requirements of the project.

5.1 System Overview and Technology Stack

The digital twin application intends to provide a realistic, interactive, and user-friendly environment and interface for monitoring and controlling a satellite platform remotely. For this project, the architecture must combine realistic visualization, real-time communication, and efficient data management to support mission-critical operations.

To deliver a realistic 3D visualization and simulate satellite behaviour, Unreal Engine was selected due to its extensive documentation, active community, and availability of relevant plugins. The engine offers cross-platform deployment to Windows, macOS and Linux, a low-code node-based interface named Blueprint Visual Scripting, that allows to quickly prototype features, and C/C++ code integration, useful for advanced logic and performance optimization. As for rendering, inbuilt features such as Lumen and raytracing are present to quickly add realistic renderization without much work.

One of the key advantages of Unreal Engine is its integration of the MaxQ plugin for simulation capabilities. This plugin integrates the NASA SPICE toolkit directly into the engine which significantly accelerates development by providing built-in models of celestial bodies, exposing orbital mechanics functions, and other astrodynamics tools essential for simulating satellite behaviour.

Communication between the digital twin and the mission control is critical. Given the nature of satellite operations, it is important to maintain control over message reliability and bandwidth usage. To support real-time data exchange between these two components, the engine uses MQTT, a publish/subscribe communication model that allows the digital twin to automatically be updated of the latest information whilst offering customizable services. This protocol can be easily integrated with other application, and works well with Unreal Engine through available plugins, making it suitable for real-time data exchange between the digital twin and mission control.

Data management is also important, as the system must be capable of storing and querying telemetry data, which is why SQLite was selected. While a time-series database would be ideal for handling temporal data, SQLite is available for free in some plugins in Unreal Engine. This solution allows the digital twin to store large volumes of telemetry data and perform queries within the simulation environment.

5.2 Design

After analysing the system requirements and evaluating available technologies, a more detailed architecture was defined, as illustrated in Figure 12:

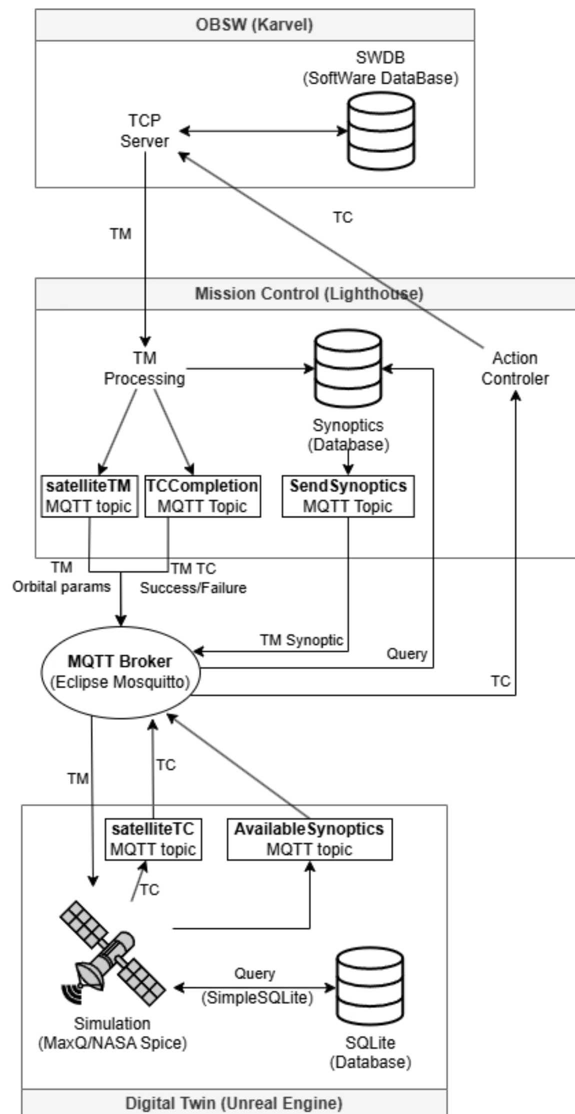


Figure 12 – Detailed system architecture

This architecture highlights the key communication pathways and the role of each system component. Like the high-level architecture, the system can be divided into three main parts:

- **OBSW**: Represented by Karvel and its components. SWDB is important for system, has it will contain the variables used for TM, and TCP Server will manage the endpoints to send TM and receive TC.
- **Mission Control**: Bridging OBSW and Digital twin is Lighthouse. Besides showing TM received and methods to send TC, Lighthouse will process TMs and select which ones

to redirect to the digital twin, as well as storing specified data in a local database, Synoptics, which can also be requested by the digital twin.

- **TM Processing**: When a telemetry arrives periodically, Lighthouse may use its contents to, for example, store it locally as logs or give an event warning.
- **Synoptics database**: Karvel contains all individual synoptics but, so far, it does not contain any information of what service each synoptic belong to. The Synoptics database intends to store and match each synoptic to its service.

- **Digital twin**: Simulates the virtual version of the satellite, as well as other functionalities present in requirements in chapter 3.1.
- **Broker**: The broker manages topics between Lighthouse and the digital twin. 4 topics are relevant for the project:
    - **`satelliteTM`**: Used to transmit telemetries from the Lighthouse to the digital twin.
    - **`TCCompletion`**: After a telecommand is applied successfully to Karvel, an acknowledgement confirming the completion of that command is sent via this topic from the Lighthouse to the digital twin.
    - **`SendSynoptics`**: A list describing all synoptics and to what service they belong. This is list can be updated dynamically, so the digital twin needs to be communicated of any modifications.
    - **`satelliteTC`**: The digital twin allows the user to send telecommands, which use this topic to transmit it to Lighthouse which redirects it to Karvel. Is also through this topic that any modifications to the Karvel's orbit are transmitted, as a telecommand.
    - **`AvailableSynoptics`**: Digital twin can request to the Lighthouse a full list services and their synoptics.

5.3 Functional Requirements Mapping

This section maps each functional requirement defined in section 4.1.1 to the technologies and components described in section 5.1.

Table 2 – Functional requirements mapping

Functional Requirement	Mapped by
DT-REQ-F-0010:NearRealTime	MQTT protocol; MQTT plugin
DT-REQ-F-0020:AdaptableCommunication	Unreal Engine; MQTT protocol; MQTT plugin
DT-REQ-F-0030:OrbitViewer	Unreal Engine; MaxQ plugin (SPICE toolkit)
DT-REQ-F-0040:OrbitComparer	Unreal Engine; MaxQ plugin (SPICE toolkit); MQTT protocol; MQTT plugin
DT-REQ-F-0050:TimeManipulation	Unreal Engine; SQLite database and plugin; MaxQ plugin (SPICE toolkit)
DT-REQ-F-0060:MissionControlIntegration	Embedded Lighthouse interface within Unreal Engine
DT-REQ-F-0070:TelemetryAndTelecommand	Unreal Engine; MQTT protocol; MQTT plugin
DT-REQ-F-0080:DataVisualization	Unreal Engine; SQLite database; SQLite plugin
DT-REQ-F-0090:TelemetrySelection	Unreal Engine; MQTT protocol; MQTT plugin
DT-REQ-F-0100:DataStorage	Unreal Engine; SQLite database; SQLite plugin
DT-REQ-F-0110:Customizability	Unreal Engine

5.3.1 Functional Requirements

This section maps each of the functional requirements to the technologies and components that fulfil it.

5.3.1.1 DT-REQ-F-0010:NearRealTime

- **Mapped by**: MQTT protocol; MQTT plugin

- **Rationale**: Once telemetry data is processed by Lighthouse, it is transmitted via an MQTT broker. The digital twin subscribes to telemetry-related topics using the MQTT plugin, receiving updates as soon as they become available.

5.3.1.2 DT-REQ-F-0020:AdaptableCommunication

- **Mapped by**: Unreal Engine; MQTT protocol; MQTT plugin
- **Rationale**: The application provides a configurable UI that allows users to connect to any MQTT broker by specifying parameters such as address and port. These settings are used to initialize the plugin accordingly.

5.3.1.3 DT-REQ-F-0030:OrbitViewer

- **Mapped by**: Unreal Engine; MaxQ plugin (SPICE toolkit)
- **Rational**: The MaxQ plugin provides 3D models of celestial bodies and functions to calculate their positions, velocities, and rotations. It also supports rendering artificial satellite orbits, enabling accurate visualization within a virtual solar system.

5.3.1.4 DT-REQ-F-0040:OrbitComparer

- **Mapped by**: Unreal Engine; MaxQ plugin (SPICE toolkit); MQTT protocol; MQTT plugin
- **Rational**: Unreal Engine can render two satellite orbits using MaxQ—one representing the current state of Karvel, and another representing a user-defined orbit. The custom parameters can be sent via MQTT to update the satellite's configuration.

5.3.1.5 DT-REQ-F-0050:TimeManipulation

- **Mapped by**: Unreal Engine; SQLite database and plugin; MaxQ plugin (SPICE toolkit)
- **Rational**: Unreal Engine allows simulation time to be fast-forwarded, rewound, or set to a specific date. MaxQ recalculates celestial body states accordingly. When simulating past states, telemetry data is loaded from the SQLite database to replay old behaviours.

5.3.1.6 DT-REQ-F-0060:MissionControlIntegration

- **Mapped by**: Embedded Lighthouse interface within Unreal Engine
- **Rational**: Unreal Engine supports embedding web interfaces, allowing Lighthouse to run inside the digital twin application and provide direct access to its mission control features.

5.3.1.7 DT-REQ-F-0070:TelemetryAndTelecommand

- **Mapped by**: Unreal Engine; MQTT protocol; MQTT plugin
- **Rational**: The application includes an interface to view incoming telemetry and manually send telecommands via MQTT, enabling direct interaction with the satellite system.

5.3.1.8 DT-REQ-F-0080:DataVisualization

- **Mapped by**: Unreal Engine; SQLite database; SQLite plugin
- **Rational**: Telemetry data is stored locally and can be visualized through dynamic graphs. Users can select parameters to track their evolution over time.

5.3.1.9 DT-REQ-F-0090:TelemetrySelection

- **Mapped by**: Unreal Engine; MQTT protocol; MQTT plugin

- **Rational**: Users can browse available telemetry parameters and choose which ones to receive and store, reducing data clutter and optimizing performance.

5.3.1.10 DT-REQ-F-0100:DataStorage

- **Mapped by**: Unreal Engine; SQLite database; SQLite plugin
- **Rational**: All telemetry data is stored in a local database for future analysis, visualization, and replay of past events.

5.3.1.11 DT-REQ-F-0110:Customizability

- **Mapped by**: Unreal Engine
- **Rational**: Unreal Engine provides extensive customization options, allowing the developer to create custom features.

5.4 Non-Functional Requirements Mapping

This section maps each functional requirement defined in section 4.1.2 to the technologies and components described in section 5.1.

Table 3 – Non-functional requirements mapping

ID	Mapped by
DT-REQ-NF-0010:Usability	Unreal Engine
DT-REQ-NF-0020:SecureData	Unreal Engine; MQTT protocol; MQTT plugin
DT-REQ-NF-0030:Replayability	Unreal Engine; SQLite database; SQLite plugin
DT-REQ-NF-0040:TelecommandCompletion	Unreal Engine; MQTT protocol; MQTT plugin
DT-REQ-NF-0050:Performance	Unreal Engine; MQTT protocol
DT-REQ-NF-0060:Portability	Unreal Engine; MQTT protocol
DT-REQ-NF-0070:KnowledgeTransfer	Unreal Engine

5.4.1 Non-functional Requirements

This section maps each of the non-functional requirements to the technologies and components that fulfil it.

5.4.1.1 DT-REQ-NF-0010:Usability

- **Mapped by**: Unreal Engine

- **Rational**: Unreal Engine provides a wide range of built-in features for designing intuitive and responsive GUI. These tools allow developers to create a custom UI for readability and ease of use, improving the overall user experience.

5.4.1.2 DT-REQ-NF-0020:SecureData

- **Mapped by**: Unreal Engine; MQTT protocol; MQTT plugin; SQLite database; SQLite plugin
- **Rational**: Both MQTT and SQLite support native security features, such as TLS/SSL encryption and file permission. Unreal Engine and its plugins allow further customization through C/C++ code, enabling the implementation of additional encryption and security measures.

5.4.1.3 DT-REQ-NF-0030:Replayability

- **Mapped by:** Unreal Engine; SQLite database; SQLite plugin
- **Rational**: The application can query stored telemetry data from the SQLite database and replay it within the simulation environment, allowing users to analyse past events and system states.

5.4.1.4 DT-REQ-NF-0040:TelecommandCompletion

- **Mapped by**: Unreal Engine; MQTT protocol; MQTT plugin
- **Rational**: When a telecommand is sent, the application can either use the same MQTT topic or a dedicated acknowledgment topic to receive confirmation of successful execution. This ensures users are informed of command status and can act accordingly.

5.4.1.5 DT-REQ-NF-0050:Performance

- **Mapped by**: Unreal Engine; MQTT protocol
- **Rationale**: Unreal Engine supports performance optimization through custom C/C++ code and adjustable rendering settings. MQTT is a lightweight protocol that minimizes bandwidth and processing overhead, contributing to responsive real-time operations.

5.4.1.6 DT-REQ-NF-0060:Portability

- **Mapped by**: Unreal Engine; MQTT protocol
- **Rationale**: Unreal Engine supports cross-platform deployment, allowing the application to run on Windows, Linux, and macOS. MQTT is widely compatible and can be integrated across different systems with minimal configuration.

5.4.1.7 DT-REQ-NF-0070:KnowledgeTransfer

- **Mapped by**: Unreal Engine
- **Rationale**: Unreal Engine projects can be documented internally using comments and annotations, and externally through guides and standardized coding practices. This facilitates onboarding and ensures future developers can understand, maintain, and extend the application efficiently.

5.5 Sequence diagrams

As to better illustrate behaviours of the system, the following sequence diagrams describe basic interactions between several components of the system.

5.5.1 Send telemetry

Sending telemetry from Karvel to the digital twin is a straightforward task. Karvel generates telemetry data and transmits it to Lighthouse, which processes the data and publishes it to the `satelliteTM` topic. The digital twin subscribes to this topic, receives the telemetry, and updates its virtual representation accordingly. Figure 13 illustrates the flow of telemetry data from Karvel to the digital twin:

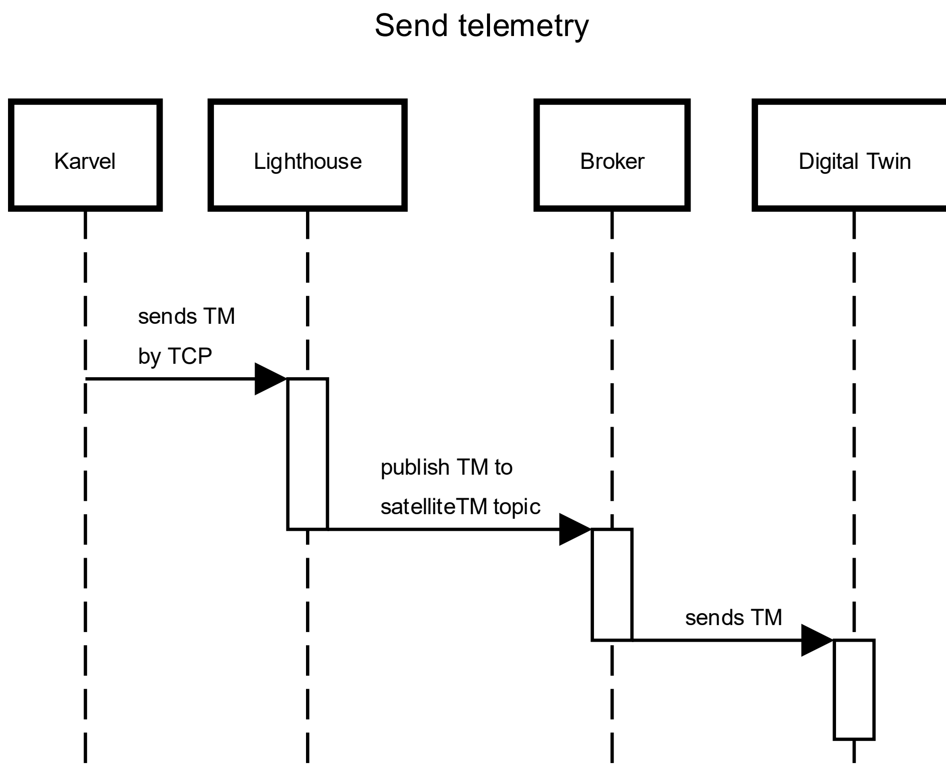


Figure 13 – Sequence Diagram: Send Telemetry

5.5.2 Send Telecommand

When a telecommand is issued from the digital twin (either by automated processes or manually by the user), it is published to the `satelliteTC` topic. The MQTT broker then forwards the telecommand to Lighthouse, which transmits it over TCP to Karvel. Karvel analyses the command and, if it is correctly formatted and valid, executes it. Upon

completion, Karvel sends a completion telemetry back to Lighthouse, which publishes an acknowledgment on the `TCCompletion` topic. The digital twin listens for this acknowledgment to confirm the telecommand's execution. This method is also shared with orbital modifications, as they are transmitted via telecommands and will require a completion response to update the digital twin. Figure 14 demonstrates the process of sending a telecommand and receiving an acknowledgment:

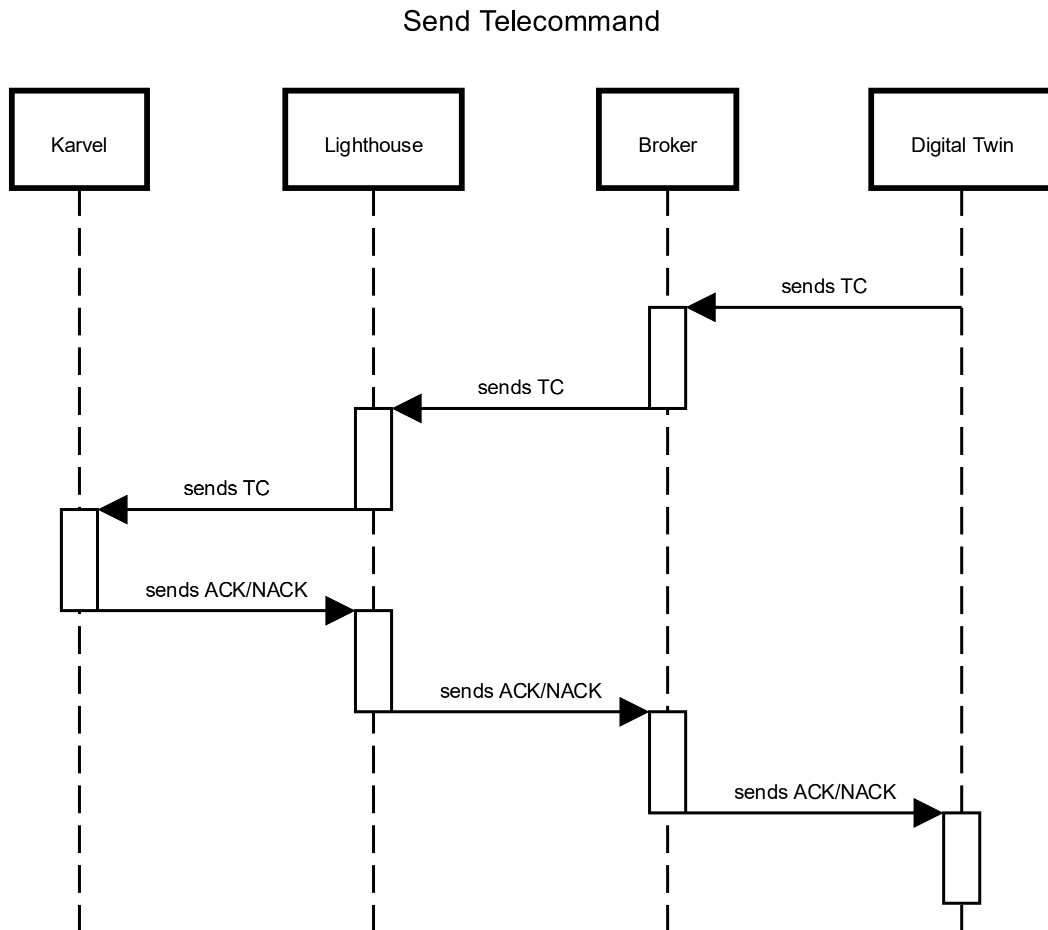


Figure 14 - Sequence Diagram: Send Telecommand

5.6 Unreal Engine plugins

Unreal Engine provides a lot of tools to build nearly everything in a 3D application but does not natively support menu of specialized features required for a telemetry-driven digital twin, such as real-time communication protocols, orbital mechanics, or data base integration. There is the option of implementing these capabilities from scratch in

C/C++ but would be very time consuming are requiring careful architectural design, which is unnecessary in a proof-of-concept prototype.

Therefore, to accelerate development and focus on validating key features, this project used third-party and community-developed plugins. These plugins add functionalities without requiring code development to create them. This section describes which plugins were used, their purpose and how they contribute to the architecture of the digital twin application.

5.6.1 MaxQ plugin – Integration of NASA SPICE toolkit

As mentioned in section 2.4.1.1, MaxQ integrates NASA SPICE toolkit in Unreal Engine, which contains several features for orbit simulations and other parameters. The plugin can be used either in the visual scripting language or called in the C/C++ code. Because MaxQ closely mirrors variables and functions (including their inputs and outputs) in SPICE, documentation and examples made by NASA can be used for MaxQ (103) (68). Such consistency makes it easier to understand and apply orbital mechanics concepts in the simulation running inside Unreal Engine. Consequently, carefully studying the resources provided in the NASA SPICE toolkit will directly increase the understanding of MaxQ.

The SPICE toolkit essentially provides an API and underlying subroutines and functions used in orbital mechanics and aerospace (103) (68). Originally developed by the Navigation and Ancillary Information Facility (NAIF), they acted under the directions of NASA's Planetary Science Division to develop a set of toolkits to support NASA scientists in planning and interpreting data from space-borne instruments, and to assist engineers in modelling, scheduling, and executing the activities required for planetary exploration missions (103) (68).

SPICE data sets are commonly referred to as "kernels" or "kernel files" (104) (105) (106). These files contain navigation and ancillary information that enable precise observation geometry, supporting both planetary science and engineering applications. To ensure usability and traceability, each kernel should include metadata that adheres to SPICE and flight project data standards, providing essential context such as data pedigree and descriptive attributes for prospective users. The SPICE kernels are what the MaxQ plugin uses to obtain data about celestial bodies, which enable accurate celestial body positioning, rotation and velocity. Depending on what kernel files are loaded, certain bodies and their characteristics become available to use.

Kernel files come in two possible formats (104) (105) (106):

- **Text kernels:** When a file contains small amounts of data, a simpler ASCII text format is used, which improves readability, and revision. They contain a description of the file and the intended use. The data follows a 'name=value(s)' (104) format.

- **Binary kernels:** Used when large amounts of data are required to be stored by instead using non-ASCII data, such as Double Precision Array Files (DAF) (107) or Direct Access Segregated (DAS) (108) formats.

Regardless of their format, MaxQ can search and obtain the necessary data from them. Some types of kernels can come in one of those formats or have both available.

The SPICE system contains several kinds of kernel types, each designed to store specific categories of data, useful in certain scenarios. The types are (104) (105) (106):

- **C-Kernel (CK):** Stores the attitude of spacecraft structures or instruments, to where they are pointing (109). This is done by expressing it in terms of a transformation matrix from a standard reference frame to a local instrument-fixed frame. In other words, it uses a rotation matrix or quaternion to find the difference between the spacecraft body frame or epoch and the instruments frame or epoch. This difference is the attitude. The letter C in CK comes from the fact that this kernel has a strong reliance on C-matrix, a 3x3 matrix used to perform algorithmics that represent the attitude of a component.
- **Digital Shape Kernel (DSK):** Contains detailed shape models of solar system bodies represented by triangular plates, such as planets, dwarf planets, natural satellites, asteroids, and comet nuclei, that is, solid bodies with defined surfaces or topologies (110). Defined topologies are required for accurate calculations with surfaces.
- **Dump binary Kernel (DBK):** Provides relational database functionalities (105) (111). Often used to support sequence components of the Event Kernel, star and image catalogues, and SPICE kernel file management.
- **Event Kernel (EK):** This kernel is tasked with storing mission event data, which is information concerning planned or unplanned activities, tailored to a specific mission (112). EK can be used to record when and how components are used, high-level descriptions of planned mission activities, and after-the-fact notes. Data is stored in a relational database, having similarities with SQL.
- **Frame Kernel (FK):** Defines the reference frames used in a mission and how they relate to one another (100). A frame is a coordinate system that specifies an origin and the orientation of its axes. By defining these 2 values, the FK provides SPICE with the context needed to build the rotation matrices that convert vectors between frames.
- **Instrumental Kernel (IK):** Consists of geometric and configuration data specific to instruments of the spacecraft, such as instruments size, shape and orientations, and parameters related to timing, optics, and optic distortion (106) (105) (113).
- **Leap Seconds Kernel (LSK):** Contains a tabulation of all leap seconds and converts time between ephemeris time and universal time coordinated (114). It is also updated with a new leap second by the International Earth Rotation Service.
- **Meta-Kernel (MK):** A file that specifies what other kernel files to load (115). Multiple MKs can be configured as a mission might require only a set of specific kernels or only for a specific time span.

- **Planetary Constant Kernel (PCK):** Provides information about solid celestial bodies (116). This data defines their physical shapes via triaxial radii, and its orientation by defining right ascension of the north pole, declination of the north pole and prime meridian angle. In cases where bodies do not have a perfect spin, information about how they precess, nutate and librate is also provided.
- **Spacecraft and Planet Kernel (SPK):** This file allows ephemerides of the solar system bodies to be present in a single file and compared with each other, which allows SPICE to compare the position and velocity of different body types, such as stars, spacecrafts, terrestrial and gas planets, in relation to some centre of mass and reference frame (117).
- **Spacecraft Clock Kernel (SCLK):** Represents the onboard time-keeping mechanism that triggers most spacecraft events and maps spacecraft clock time to standard time formats, such as ephemeris time and universal time coordinated (114).

Kernel files are available and updated in NAIF servers, which can be downloaded for free and used with MaxQ. In the servers, two categories of kernels are available (118) (119):

- **Mission specific:** Kernels classified as mission specific are kernels that were either created before, during or after a mission. Their data can vary depending on the mission, as some kernels contain mission context, commands or steps the spacecraft followed to fulfil the mission goal, or simply logs of the spacecraft for future analysis. These kernels can be used in a simulation when one wants to recreate it in a virtual environment and analyse mission behaviours.
- **Generic:** When a kernel is classified as generic it does not contain any mission-specific data, focusing on delivering a wide scope of information that can be used for most missions or simulations. Because of this, minor irregularities can be present.

One other feature of SPICE that carries over to MaxQ is how files are loaded and requested. When the application loads kernel files, it furnishes them into the kernel pool. Later, when a user requests access to variables, the system begins its search starting from the most recently furnished kernel file. This means that variables shared across multiple kernel files will be overridden by the last kernel furnished with that variable (120).

SPICE toolkit, and by consequence MaxQ plugin, are one of the most critical components of this project, as they make available all necessary tools to simulate orbits accurately and the main bodies of the solar system.

5.6.2 UE4SimpleSQLite – Database manager

In accordance with requirement DT-REQ-F-0100:DataStorage, the application must incorporate a database to store timestamped data, such as telemetry, telecommand usage or any other events or behaviours generated during simulation.

A time-series database (TSDB) is a type of database specifically designed to handle large volumes of timestamped data, because they are optimized for storing, querying, and analysing data that changes over time, making them ideal for a telemetry driven application (121). Unfortunately, there is a notable lack of free plugins that support TSDB.

Due to the lack of TSDB plugins, SQLite was selected for data storage. This decision was influenced not only by the availability of free tools like DB Browser for SQLite (122) and plugins such as UE4SimpleSQLite (123), but also because other SQLite plugins failed to work with the specific Unreal Engine version used in this project. Compatibility issues and limited plugin support for TSDB forced the adoption of SQLite as a fallback solution.

Although SQLite is not a time-series database, it serves as a practical fallback solution due to its simplicity, and widespread support (124). As a lightweight, self-contained, serverless database engine, SQLite requires minimal setup and stores all data in a single file, making it easier to manage data going into and out of the database. Using the plugin UE4SimpleSQLite enables it to be integrated in the project when other database plugins failed due to engine version mismatches. Despite lacking native time-series features such as automatic time indexing or retention policies, SQLite still supports storage and retrieval of timestamped telemetry and onboard parameters through straightforward SQL queries. This has proven sufficient for fulfilling requested features, including synoptic views, orbit-related parameter visualization, and time-rewind functionality, which allows the implementation of the requirement DT-REQ-F-0050:TimeManipulation by enabling fast-forwarding, rewinding, and jumping to specific simulation times and timestamped data queries to replay behaviours from those dates, and DT-REQ-F-0080:DataVisualization by allowing telemetry data to be plotted over time by storing and querying values. The plugin itself is free, open-source, and compatible with the current version of the project. It offers basic functionalities to manage and query the database, enough to achieve the desired functionalities from the digital twin.

5.6.3 MqttUtilities – MQTT protocol

The digital twin application needs to be updated of any new telemetry and important events soon after being created and processed by the mission control, which is why MQTT protocol was selected.

Instead of manually coding the reception and transmission of telemetry and telecommand, the plugin MqttUtilities was selected to integrate MQTT communication into the digital twin (125). The plugin natively supports Blueprint, enabling quick implementation and providing ways to manage MQTT connections, subscriptions, and message handling without writing low-level networking code. This makes it well-suited for developing a digital twin proof-of-concept prototype in real-time simulated environment.

One significant issue with the plugin is a lack of broker, which is necessary to manage topics and to who redirect data. To solve this, Eclipse Mosquitto was deployed as it provides a quick broker with minimum, which the plugin can be configured to connect.

By selecting this MQTT plugin and Mosquitto, they directly contribute to the fulfilment of several requirements:

- **DT-REQ-F-0010:NearRealTime:** Ability to subscribe to topics that publish telemetry data as soon as it is available.
- **DT-REQ-F-0020:AdaptableCommunication:** The plugin exposes connection parameters such broker address, port, username, and password and can be set at run-time, avoiding hardcoded values.
- **DT-REQ-F-0040:OrbitComparer:** Allows the user to compare and send custom telecommands with orbital parameters to modify the orbit.
- **DT-REQ-F-0070:TelemetryAndTelecommand:** The plugin allows the user to manually send telecommands and receive telemetries by publishing and subscribing to topics dynamically.
- **DT-REQ-F-0090:TelemetrySelection:** By allowing the application to publish/subscribe to topics, the user can choose what parameters to receive and be updated of.

Therefore, this plugin and broker play an important role by bridging all communications of the digital twin to external applications, like Lighthouse and Karvel.

5.7 Kernel files and 3D models

As discussed in section 5.6.1, there are several categories of kernel files. Because this project does not intend to replicate past missions, generic kernels were selected, as this allows for a more general-purpose use, like simulating the position of the main celestial bodies of the solar system.

Therefore, the following types of generic kernel files were selected that involve accurate celestial body movements:

- **Ephemeris (SPK)** - Contain planetary and lunar positions for the solar system:
    - de441_part-1.bsp

  - de441_part-2.bsp
- **Planetary Constants (PCK)** – Body orientations and sizes:
  - pck00011.tpc
- **Leapseconds (LSK)** – Latest leapseconds for time conversions:
  - naif0012.tls
- **Reference Frames (PCK)** – Reference frames for Earth and Moon:
  - earth_fixed.tf
  - moon_080317.tf
  - moon_assoc_me.tf
  - moon_assoc_pa.tf
  - moon_de440_220930.tf
- **Mass Constants (PCK)** – Gravitational parameters for Sun and planetary barycentres:
  - de-403-masses.tpc

As for 3D models of the celestial bodies, MaxQ already includes models for most bodies and natural satellites, as well as some artificial ones. These models contain the vertex mesh, textures, and materials. The solar system background texture is already present in Unreal Engine, named `StartSkySphere_Equirectengular`, which will act as the backdrop.

5.8 Reference frame and plane

As discussed in 3.3, this project requires the definition of both a reference frame and a reference plane. For this project, which focuses on simulating a satellite's orbit around Earth within the context of the solar system, the **ECLIPJ2000** reference frame and the **Earth barycentre reference plane** were selected.

This configuration places the Earth's barycentre (the centre of mass of the Earth–Moon system) at the origin of the simulation environment, with coordinates (0, 0, 0). The Earth's ecliptic plane is used as the reference plane for all orbital calculations.

6 Implementation

This chapter details the implementation of the digital twin application, showcasing both the features developed and some important configurations and components of unreal Engine that enabled them. Throughout the development, several technical challenges emerged, which required some problem-solving and adaptation. The solutions accomplished are also presented and is important to also reference that there are multiple paths to develop the same features. The current paths were taken due to several reasons, like ease of implementation, availability of resources, or simply because it was the most straightforward way thought of at the time.

6.1 Object types

Several kinds of objects exist, each containing features that bring something new to the project. In this section, the types used in the project are explained, as each were used in important roles.

6.1.1 Blueprint Class

A Blueprint Class, often simply called Blueprint, represents a Class that only accepts visual scripting (126). Often used to define variables and functions, which were the main uses in this project.

6.1.2 Blueprint Function Library

A Blueprint Function Library contains reusable code accessible to all Blueprints (127). These are not considered objects but rather a utility container containing custom made global data. Useful when creating C/C++ code instead of visual scripting brings some advantages, like easier implementation and readability.

6.1.3 Level and Level Blueprint

A Level in Unreal Engine represents a playable environment or scene, where each level may contain their own unique mechanics, objects, lighting and other elements (128).

Level Blueprints is a script associated with each Level. In it is possible to define level-specific logic, such as triggers, events, and interactions (129).

6.1.4 Structure

Structure, also referred as `Struct` in C/C++, is a data type that can contain multiple variables under a single name (130). Structures are useful when organizing related data under a single type, like the 6 orbital elements.

6.1.5 Widget Blueprint

A Widget Blueprint is used to design and implement UI. This also includes creating custom scripting logic (131). Some commonly used features are buttons, menus, text inputs, and graphs.

6.2 Unreal Engine features and constraints

During development, some issues during implementation arose. This was not necessarily due to bad implementations and logic, but rather due to unknow ways Unreal Engine works, which forced a different implementation.

This section discussed some important inbuilt features and characteristics of Unreal Engine. While most of the engine features improved development speed, sometimes certain constraints on how the engine works created limitations that required workarounds.

6.2.1 UGameInstance

During development, one of the first issues was how to share variables and functions across different Blueprints and levels. This issue was resolved by the usage of **GameInstance**, a subtype of Blueprint Class that is instantiated once the application starts and is only deleted when it closes. This persistence allows any variable and function it contains to be accessed globally across the project (132).

In the digital twin application, a custom GameInstance was created and named `BP_SharedData`. However, creating it alone is not enough to be immediately use it across the project. It must be explicitly set in the project settings. This is done by navigating to `Project Settings > Maps & Modes > Game Instance Class` and selecting `BP_SharedData` from the dropdown menu.

To access the contents of `BP_SharedData` from other Blueprints, it is necessary to store a reference to it. This is done by creating a variable of type **Object Reference to BP_SharedData** in each Blueprint where shared data or functions are needed.

The process can be done by calling the **Get Game Instance** function, which returns a generic reference to the active GameInstance. However, since this reference is not yet

recognized as a `BP_SharedData` object, it must be **cast** to `BP_SharedData`. This casting tells Unreal Engine to treat the reference as an instance of the custom class, allowing access to its variables and functions.

Once casted, the result is stored in a variable named `Ref_BP_SharedData`, which can then be used throughout the Blueprint to access shared data and logic. The name `Ref_BP_SharedData` was used in all instances where access to game instance was required.

Figure 15 shows how this logic looks like in visual scripting:

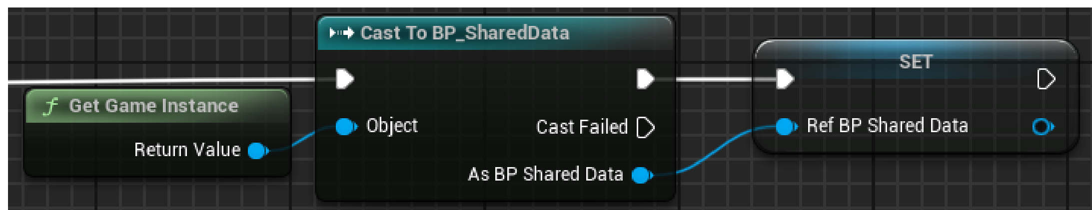


Figure 15 – Accessing and storing a game instance object reference

Afterwards, all contents inside `BP_SharedData` become available to use via `Ref_BP_SharedData`, as shown in Figure 16:

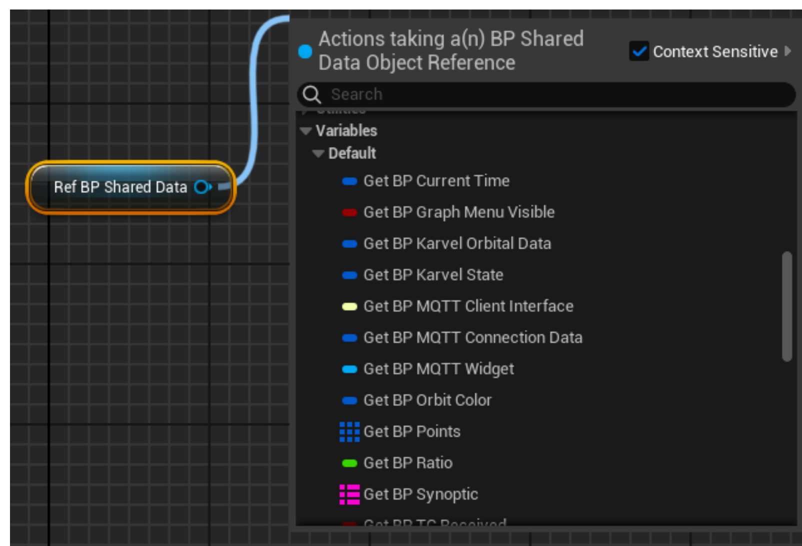


Figure 16 – Accessing game instance data

6.2.2 Scale and Units

When simulating the solar system, it is important that the virtual units do not match real-world distances and sizes. Using true-to-life values would make navigation impractical and introduce significant rendering issues, as Unreal Engine cannot accurately display visual effects at such vast scales, like lighting and shadow artifacts.

During testing, it became visible that rendering meshes and calculating lighting at large distances led to visual artifacts and performance problems. (133).

To address this, all size-related values were divided by a factor of 3000, allowing all celestial bodies to be visible within the same scene. Additionally, distances between objects were multiplied by 0.15 to bring them closer together. These scaling factors were determined through trial and error.

6.2.2.1 Sun

Another issue was the Sun's immense size when compared to the other planets. Even after reducing its dimensions by a factor of 3000, the Sun remained disproportionately large and would engulf nearby planets in the simulation. To resolve this, the Sun's size was further reduced by multiplying it by an additional factor of 0.15.

It is important to reference that all calculations done for orbits, dimensions, positions, velocities, and rotations use the original unedited values. The reduced sizes and distances are only applied afterwards, which ensures consistency and accuracy. This guarantees that the solar system remains accurate in all aspects, with only the scale of the planets and the Sun being disproportionate in the final render, as seen in Figure 17:

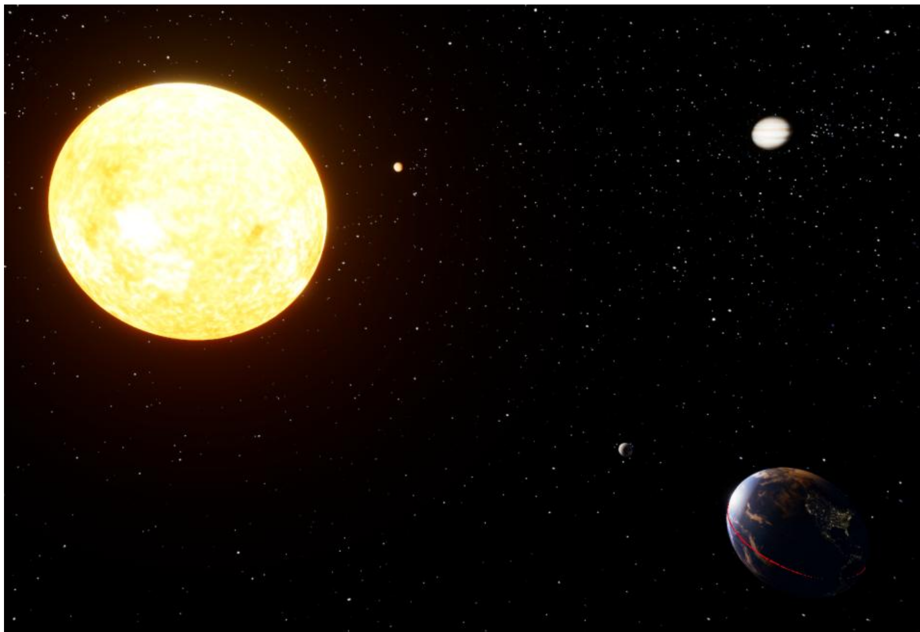


Figure 17 – Solar system render

6.3 Development

In this section is discussed all features and steps required to create the digital twin and its features.

Two important conventions are followed: all custom functions created in the visual scripting system are prefixed with **CSW** to distinguish them from Unreal Engine and plugin built-in functions, and variables stored in `BP_SharedData` follow a naming pattern:

- Those referencing a class begin with **Ref**.
- All others begin with **BP**

This help differentiate between variable types in game instance and local to the blueprint or structure file.

6.3.1 Widget Manager Architecture

To streamline UI management and better organize project structure, a **Widget Manager** was implemented. The Widget Manager is responsible for creating all main widgets used in the application and managing their visibility through a main menu.

Once the Widget Manager is instantiated by the level script, it automatically handles the creation and displays the following main widgets, each accessible through a dedicated button in the main menu:

- **Time Scale** – Adjusts the simulation speed and allows jumping to a specific date.
- **Orbital Values** – Enables modification of satellite orbital parameters (available only in offline mode)
- **MQTT** – Displays telemetry terminals and provides an input box for sending telecommands.
- **Lighthouse** – Opens the mission control application (Lighthouse) within a web browser widget, allowing direct interaction with the mission control interface.
- **Compare Orbit** – Creates a second, user-modifiable orbit. The widget sends a telecommand to apply the new orbit and await confirmation (TCCompletion).
- **Synoptics** – Lists all available synoptic values from telemetry and lets users choose which ones to monitor.
- **Graph** – Plots synoptic values using point widgets.
- **Settings** – Provides configuration options.

Each widget manages its own data and logic. The only roles of the Widget Manager are to instantiate these widgets and toggle their visibilities when interacted with. With this architecture, only the widget manager needs to be created by the level script, and all other Ui management is handled by it.

6.3.2 BP_SharedData variables

Due to the complexity of the project, multiple variables were created within BP_SharedData to support the functionalities of the digital twin. These variables allow data sharing across different Blueprints.

The following list enumerates these variables, including their names, types, and purposes:

- **BP_CurrentTime** (SEphemerisTime): Tracks the current simulation time in seconds.
- **BP_KarvelOrbitalData** (SconicElements): Stores the six Keplerian elements, epoch, and gravitational parameter. Uses Perifocal Distance instead of Semi-major axis.
- **BP_TimeScale** (SEphemerisPeriod): Represents the simulation time scale.
- **BP_KarvelState** (SStateVector): Contains distance and velocity vectors; can be used for orbit calculations.
- **BP_OrbitColor** (LinearColor): Holds RGBA values to modify orbit colour.
- **BP_MQTTClientInterface** (MQTTClientInterface): Represents the MQTT connection, enabling topic subscription and data publishing.
- **BP_MQTTConnectionData** (MQTTConnectionData): Stores login and password for MQTT broker access.
- **BP_Ratio** (Float): Adjusts the relative sizes and distances of solar system bodies.
- **Ref_Database** (SQLiteConnector): Reference to the SQLite database for data manipulation.
- **Ref_ArrayWidgetPoints** (PointWidgetBP[]): Array of point widgets for graph plotting.
- **BP_TCReceived** (Boolean): Triggers events when telemetry is received.
- **BP_MQTTWidget** (MQTTWidgetBP): Reference to the MQTT widget displaying telemetry and telecommands.
- **BP_TCReceivedData** (String): Stores received telemetry as a string.
- **BP_TM** (JsonObject): Telemetry data cast to JSON.
- **BP_Synoptics** (StringMap): Tracks new synoptic values.
- **Ref_Graph** (GraphWidgetBP): Reference to the graph widget for visualizing synoptic data.
- **BP_Points** (GraphPointArray): Array of points for graph plotting. Description is present in 6.3.3.
- **BP_GraphMenuVisible** (Boolean): Indicates whether the main menu is visible, used to trigger events.
- **Ref_ClickOnData** (SatelliteClickOnDataWidget): Reference to a widget that highlights the satellite and displays synoptic information when selected.

6.3.3 GraphicPoint Struct

To display points in a graph, a custom structure named GraphicPoint was created containing all necessary information to place and manage points within the graph widget. Information includes coordinates, visual representations, and display settings.

The structure contains the following fields:

- **SynopticName** (String): The name of the synoptic being represented.
- **Points** (2D Vector[]): An array of X and Y coordinate pairs for point.
- **Buttons** (Point Widget Blueprint[]): Widgets representing each visible point on the graph.
- **MinX** (Float): The smallest X value in the Points 2D vector array.
- **MinY** (Float): The smallest Y value in the Points 2D vector array.
- **MaxX** (Float): The largest X value in the Points 2D vector array.
- **MaxY** (Float): The largest Y value in the Points 2D vector array.
- **Colour** (Color): RGBA values used to define the colour of the point widgets.
- **isSelected** (Boolean): Indicates whether the synoptic is currently selected for display on the graph.

6.3.4 Loading Kernel files

Kernel files can be loaded using two MaxQ functions that interface with SPICE: **Enumerate Kernels** and **Furnish List**. The Enumerate Kernels function scans a specified folder for kernel files and passes them to Furnish List, which then loads them into the simulation environment. With this, all functions and parameters provided by kernel files can be used by simply searching them.

6.3.5 Solar System simulation

MaxQ provides several 3D models representing the main celestial bodies of the solar system. However, simply placing these models in the environment is not sufficient for accurate simulation, as their positions and orientations must be dynamically calculated based on real astronomical data.

To achieve this, all calculations are performed relative to the **ECLIPJ2000** reference frame and the **Earth barycentre** reference plane, with the reference epoch time serving as the basis for all transformations. This ensures that each celestial body is positioned and oriented correctly within the simulation in relation to the same origin point and plane.

The parameters required for these calculations are:

- Target body name: The celestial body to retrieve data for.
- Observing body name: The reference body from which calculations are made.
- Epoch time: The current simulation time, used as the reference epoch.
- Format: The specific value or property to retrieve from the SPICE kernel.

To retrieve physical properties, such as the radius of a body, the **Bodvrd** function is used. For example, querying the target "MOON" with the format "RADII" returns the Moon's radii, which can then be used to scale its 3D model appropriately using the Set Scale 3D function.

For dynamic positioning, the **Spkezr** function calculates the state vector (location and velocity) of the target body relative to the observing body at the given epoch. The resulting position is applied to the 3D model using the Set Actor Location function.

For orientation, the **Pxform** function computes the rotation matrix for the target body at the current epoch, relative to the reference frame. This matrix is converted to a quaternion using the **M2q** function, then cast to a rotator and applied to the model with Set Actor Rotation.

By combining these functions, each celestial body in the simulation is accurately placed and oriented according to real astronomical data, referenced to the ECLIPJ2000 frame and Earth barycentre. This is done at every tick of the simulation.

6.3.6 Lighting

With the solar system correctly simulated, one issue arose regarding the illumination of celestial bodies. The Sun's 3D model available in the plugin contained a texture with emissive properties, but they were, by default, not strong enough to eliminate the entire solar system realistically. Increasing the emission made the Sun appear unnaturally white and only strongly illuminated nearby planets, leaving those farther away in shadow.

Unreal Engine offers several types of lights. While a **point light** most closely mimics the Sun by emitting light in all directions, testing revealed that placing a point light at the Sun's position (using coordinates from the Spkezr function) still resulted in only the closest planets being illuminated, and often too intensely.

To address this, a different approach was adopted using **rectangular lights**. Unlike point lights, rectangular lights illuminate only the area in front of them. The solution involved placing one rectangular light for each celestial body, positioned between the Sun and the respective planet or moon. This setup is illustrated in Figure 18:

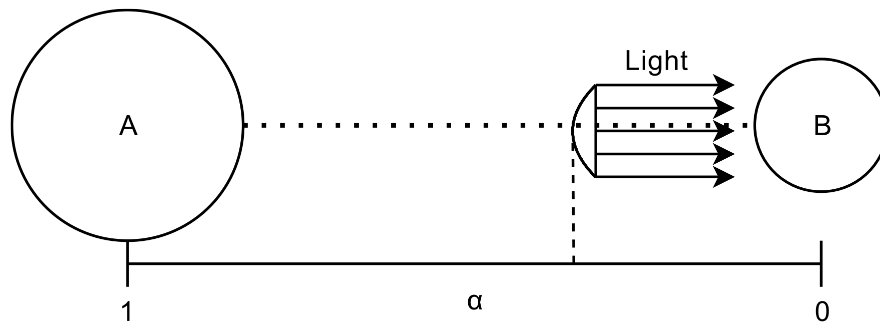


Figure 18 – Custom light logic

The placement of each light is determined using **linear interpolation (lerp)** between the Sun's position (A) and the planet's position (B). The custom lerp equation is:

$$Lerp(A, B, \alpha) = B + (A - B) \cdot \alpha$$

where:

- **A**: Position of the Sun (vector).
- **B**: Position of the planet (vector)
- **α** (alpha): Interpolation parameter (0 = at the planet, 1 = at the Sun; values in between place the light at a proportional distance between them)

By choosing a small α, the light is placed near the planet, ensuring it is properly illuminated regardless of its distance from the Sun.

Since both the Sun and planets are in motion, the lights must also be dynamically oriented. For that, the function Find Look at Rotation can be used to calculates the rotation needed for the light to face the planet. The interpolated position is applied with Set Actor Location, and the calculated rotation is set with Set Actor Rotation. Each planet and the Moon have their own dedicated light, and their positions and orientations are updated every tick using the current coordinates of the Sun and each body.

There are several limitations to the current lighting approach. First, because the alpha value in the interpolation uses the centres of the Sun and each planet, setting alpha close to 0 or 1 places the light inside the respective model, which does not illuminate the surface. This could be improved by calculating the interpolation between the surfaces of the two bodies, but this would require determining the exact points on each surface that face one another. Second, the alpha value is currently static and does not adjust based on the distance between the Sun and each planet. While a small alpha like 0.05 may work well for closer planets, it is insufficient for more distant ones, as the light may be placed too far from the planet to provide adequate illumination. Similarly, the size and strength of each rectangular light is also static and identical for all planets. If the user changes the scale of the planets, the light may become disproportionately

large which illuminates areas beyond the planet’s surface facing the Sun, or too small, failing to cover the entire visible side. Addressing these issues would require dynamically adjusting both the alpha value and the light size and strength based on the distance and scale of each celestial body.

6.3.7 Karvel orbit

Rendering a visible orbit around Earth for the Karvel satellite was made straightforward thanks to several MaxQ functions. The process relies on SConicElements struct can be used, which defines an elliptical orbit using the six Keplerian elements. In addition to these elements, the structure also requires:

- Epoch time (current simulation time).
- Gravitational parameter of the central body (in this case, Earth).
- Perifocal distance, which is the distance from the centre of the central body to the closest vertex on the orbit’s major axis.

Because Karvel orbits Earth, the functions **bodvrd** can be used to obtain Earth’s radii and gravitational parameter from SPICE kernel. With SConicElelemnts struct filled, the function Render Debug Orbit can be employed to render a visible orbit. This function requires the centre of the ellipse, current epoch time, and reference frame.

The variable **BP_KarvelOrbitalData** present in BP_SharedData contains the final value of SConicElements to simulate the orbit. This means that modifying this struct in other places will automatically update the orbit.

6.3.8 Orbital Values - Orbital values input widget

With Karvel successfully placed in orbit around Earth, a custom widget was developed to allow manual adjustment of its orbital parameters. The Orbital Values Input Widget displays the current state of the orbit and provides text input fields where users can freely modify the satellite’s orbital elements. When the user submits new values, they are parsed and applied to the BP_KarvelOrbitalData structure. This automatically recalculates the orbit, which is then visually rendered in the simulation using the updated parameters. This feature is useful for testing different orbital configurations and observing their effects in real time. These modifications only work in offline mode, that is, when not connected to the MQTT broker and thus is not communicated. If it is connected via MQTT, telemetry data will replace the modified orbit. Is possible to see the rendered orbit and the Orbital values input widget in Figure 19:

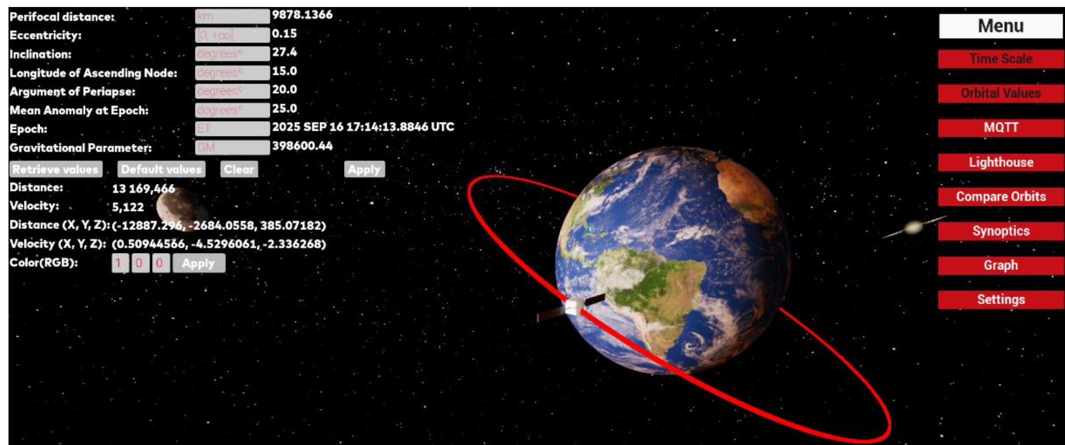


Figure 19 – Karvel orbit and Orbital values input widget

6.3.9 Time Scale – TimeMods widget

Time Scale modifies the speed of the simulation, allowing the user to speed it up, go backwards or to a specific date. The solution takes advantage of BP_SharedData by providing two variables necessary to manipulate simulation speed and current date: BP_CurrentTime and BP_TimeScale.

When calculating the motion of celestial bodies and Karvel, the current date and simulation speed are stored and updated in these two variables, which are shared across several functions and blueprints. Thus, by modifying these values, the simulation will use them in the next tick to update the solar system and Karvel. The values can be updated by the user via input boxes.

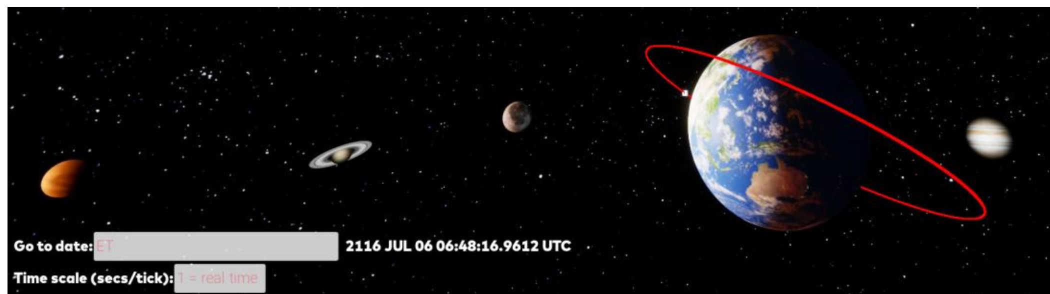


Figure 20 – Example of the position of several celestial bodies in the future

6.3.10 Lighthouse – Lighthouse widget

Adding Lighthouse was simple, as Unreal Engine already contained a web browser widget. The only thing required was to place a WebBrowser with the URL. In Figure 21 is possible to see Lighthouse open:

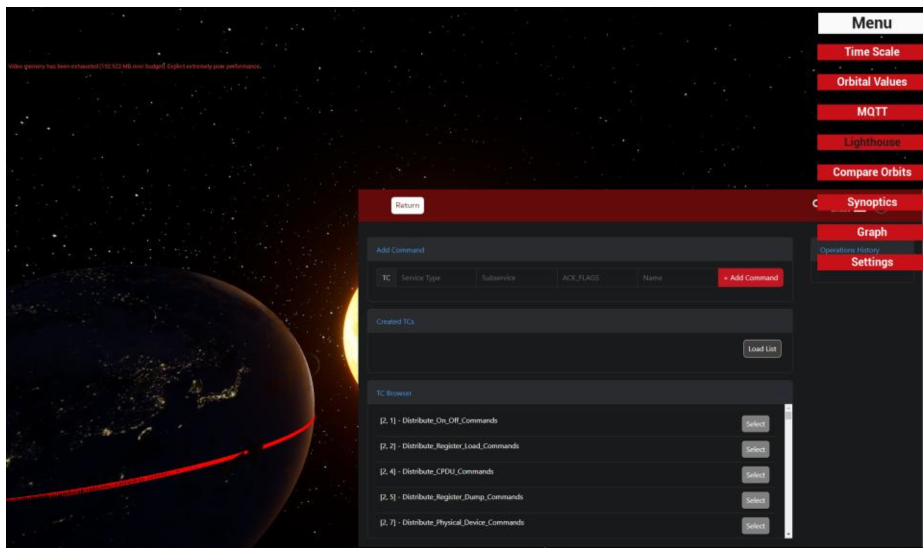


Figure 21 – Lighthouse open in digital twin

6.3.11 MQTT – MQTT widget

The MQTT connection is responsible for receiving telemetry data from Karvel and for sending telecommands to the satellite. This is implemented with the MQTT plugin for Unreal Engine, which provides several functions to handle MQTT connections with a broker.

Before establishing a connection, the user must configure the connection parameters, including the broker's host URL, port, client ID, and credentials. The **Create Mqtt Client** function initializes the connection and outputs a **Mqtt Client Interface** variable. The Event Loop Delta Ms parameter was set to -1 (disabled) in this implementation.

The plugin allows binding to several event handlers, such as OnPublishCallback, OnSubscribeCallback, OnUnsubscribeCallback, OnMessageCallback, and OnErrorCallback. These events are automatically triggered by the plugin in response to relevant broker actions, which allow the application to handle message delivery, subscription status, and error reporting.

Once the client is configured, the **Connect** function is used to establish a connection to the broker, providing login and password credentials. After a successful connection, the **Subscribe** function subscribes to the satelliteTM topic to receive telemetries with a specified QoS level. To send telecommands or other messages, the **Publish** function is used, requiring the message string, topic name, and QoS. The function also includes optional parameters for a message buffer and a retain flag, which were not used in this project. The **Publish** function also handles topic creation: if the specified topic does not exist, it is created automatically using the provided message configuration.

The widget itself contains two buttons to toggle the visibility of telemetry data and a telecommand input text. The widget is visible in Figure 22:

Figure 22 – MQTT widget

When a telemetry message arrives, it triggers two actions:

- The telemetry message is cast to a JSON object and stored in BP_TM within BP_SharedData, making the data accessible to other Blueprints.
- The **Get TM** event is triggered. Any Blueprints bound to this event will execute their code each time new telemetry is received.

6.3.12 Compare Orbit – Copy_Satellite widget

The Compare Orbit feature allows the user to test and visualize alternative orbital configurations before applying them to the Karvel satellite. It shares most of its logic with the Karvel orbit system, including the use of the Render Debug Orbit function and similar input fields to retrieve orbital parameters. However, Compare Orbit requires additional interactions and validations through telecommand publishing and confirmation.

When the user selects the Compare Orbit option, the widget spawns an actor named Copy_Satellite, which is an exact duplicate of the Karvel Blueprint. This actor is used to render and simulate the proposed orbit. Users can modify the orbital parameters via the input fields, and upon applying changes, a confirmation popup is displayed. If the user declines, the popup closes, allowing further edits. If confirmed, the widget constructs a JSON object containing key-value pairs for each orbital parameter and builds a telecommand message.

This telecommand is published to the satelliteTC topic using the MQTT Publish function. The system then listens for a response on the TCCompletion topic to confirm whether the command was successfully received and applied by Karvel. Once the telecommand is acknowledged and the new telemetry is received, the popup closes, and the updated orbital parameters are applied to the original Karvel orbit, keeping the synchronization between the simulation and Karvel.

In Figure 23 is possible to see where compare orbit and the popup are visible:

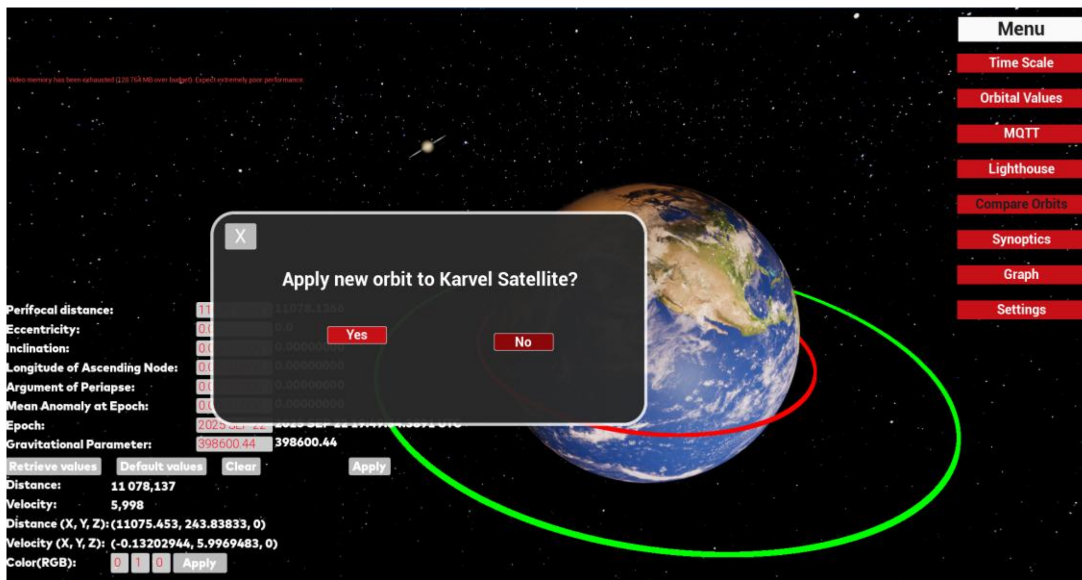


Figure 23 – Compare orbit popup

When the user confirms that they want to apply the orbit, the telecommand is published and awaits confirmation. The popup updates warning the user of success of failure, as seen in Figure 24:

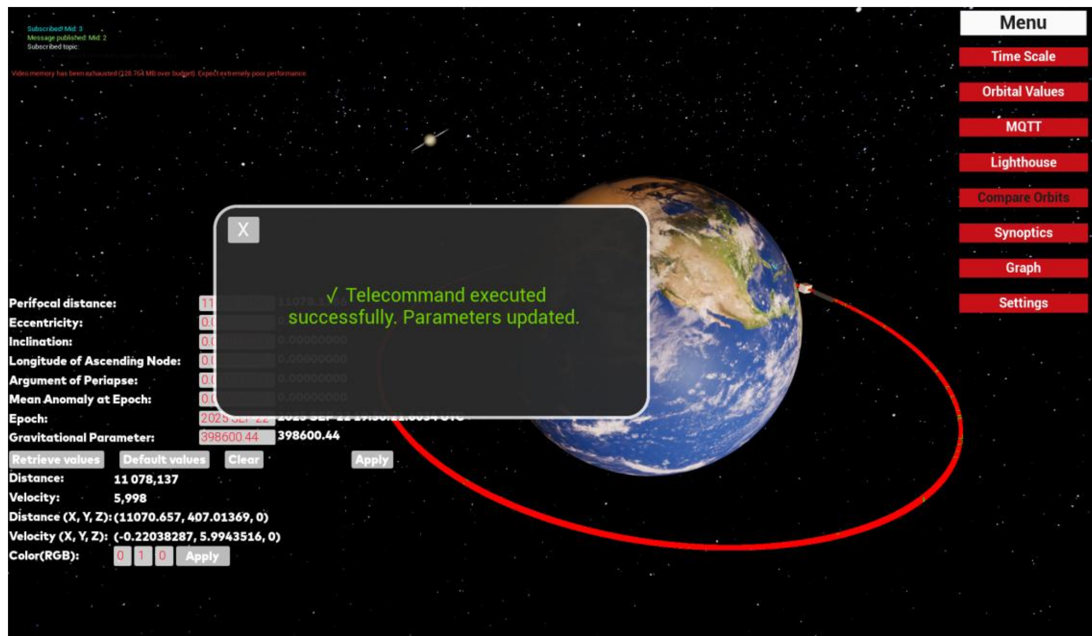


Figure 24 – Compare orbit completion

6.3.13 Synoptics – Synoptics widget

The Synoptics Widget provides users with an interface to view and manage telemetry data. It allows users to select which synoptic values they want to monitor, offering an overview of all values in Karvel. The widget is divided into four main sections, arranged from left to right:

- **History:** Displays a log of selected synoptics. Users can delete the contents of the history list with a dedicated button.
- **Services:** Lists all available services. Selecting a service automatically selects all associated synoptics.
- **Synoptics:** Shows all synoptics present in the telemetry. Users can select or deselect any synoptic.
- **Values:** Displays the current values of selected synoptics. A Reset button is available to clear all selections.

Figure 25 illustrates the Synoptics Widget, where selecting the "SVC 3 Orbit" service automatically selects all synoptics related to orbital parameters:

Figure 25 – Synoptics widget

When the widget is created, it initializes the telemetry database. If the database does not exist, it is created empty using the SQLite plugin with the SQLite command:

```
CREATE TABLE IF NOT EXISTS Synoptics (id INTEGER PRIMARY KEY
AUTOINCREMENT);
```

This table uses an auto-incrementing primary key.

The **Get_TM** event (section 6.3.11) is used here. When new telemetry is received, the Synoptics Widget analyses, displays, and stores the data in the database. Telemetry values are received in hexadecimal format and often vary in size. The **Hex to Value** function is used to convert these values to their types (integer, unsigned integer, float, Boolean, or charstring), based on the specified byte size.

After conversion, each key-value pair is stored in the **Synoptics map** variable, which is updated with every new telemetry message. Since the Synoptics database table is initially empty, new columns are added dynamically for each synoptic by iterating through the **Synoptics map** and executing the SQLite command:

```
ALTER TABLE Synoptics ADD COLUMN [synoptic_name] TEXT;
```

This ensures all current synoptics are represented as columns.

Telemetry values are then inserted into the database using:

```
INSERT INTO Synoptics ([columns]) VALUES ([values]);
```

This keeps the database up to date with the latest telemetry.

Finally, when a synoptic is selected, its key-value pair is stored in **BP_Synoptics** (a string map in **BP_SharedData**). This allows the Graph Widget to access and plot the selected synoptics and their values. A Graph widget event is triggered at the end, **TM_Updated**, which triggers the Graph widget to update the graph with new data.

6.3.14 Graph – Graph widget

The Graph widget allows the user to visualize synoptic data selected in the Synoptics Widget, tracking and displaying these values over time. The widget is divided into three main sections, arranged from left to right:

- History: Lists the selected synoptics and their historical values.
- **Graph**: A dynamic, real-time graph that updates with the selected synoptic(s). An input box allows users to set the update interval:
  - If set to 0, the graph updates immediately upon receiving new data.
  - If set to another value, updates occur only after the specified interval has elapsed.
- **Synoptics**: Displays buttons for all synoptics selected in the Synoptics Widget. Selecting a synoptic show all its received values in the graph. Multiple synoptics can be viewed simultaneously.

When a telemetry is received, it is compared to the previous telemetry. f the data has changed, the new values are stored and displayed; if not, the data is ignored to prevent cluttering the graph. Figure 26 demonstrates this process, showing the selection of the "SVC 3 Orbit" service and the update of the PERIFOCAL_PARAMETER value in the Compare Orbit widget:

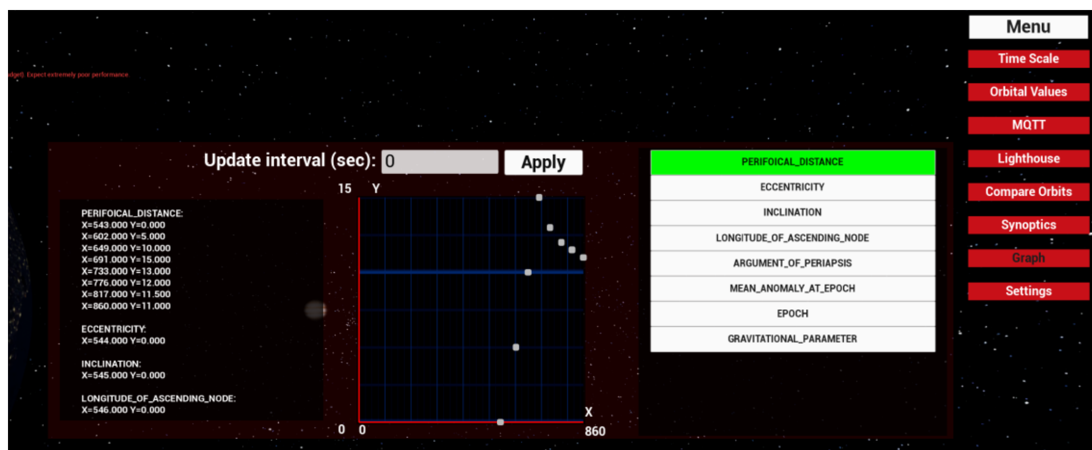


Figure 26 – Graph widget

In the graph:

- The Y-axis represents the maximum value of the selected synoptics.
- The X-axis represents the number of telemetry messages received.

If a user deselects a synoptic in the Synoptics Widget, it is automatically removed from the graph.

When telemetry is received, the **TM_Updated** event is triggered. This event checks which synoptics are currently selected and their values, to then decide if the graph requires resizing and new points placed.

All graph data is stored in an array of GraphPoint structs (6.3.3) within BP_SharedData, in the variable **BP_Points**. Each instance contains data for a selected synoptic and its points.

The events start by checking if the new telemetry differs from the last received telemetry. If they are different, it checks if any synoptics has been removed. When this happens, points need to be removed. If a new value is present, it is added as a new instance in the BP_Points.

Because the telemetry is different, we also need to recalculate all minimum and maximum values for X and Y axis, as to later update the dimensions of the graph. In the first part, each BP_point is analysed each individual X and Y values are stored in their respective index struct. Then all minimum and maximum values from all BP_Point are selected, as they will become the absolute dimensions of the graph.

With thew new dimensions, the graph must be recentred, and label values and point positions updated. The background of the graph with the grid is a Dynamic Material Instance named Base Graph. The graph can be resized by updating the RGBA values which represent the minimum and maximum values for X and Y axis.

With the graph with the correct dimensions, we can now cycle through BP_Points to check want points to add. When a synoptic is selected to view the points in the graph, the application deletes all current points and readds them to the graph. This is simpler than calculating the position of new points relative to the older position.

After displaying the new points, the history values on the left part are updated with the latest values.

6.3.15 Settings

Settings widget was not developed extensively. So far, only contains a slider that allows the user to modify the size and distance to Earth of all planets, Moon and Sun. This was done by adding a variable to BP_SharedData called **BP_Ratio**, which is used when calculating the dimensions and positions of celestial bodies.

6.3.16 Select Satellite

A feature requested was the ability to click on the satellite model, which would highlight it and display some housekeeping data. The material was done following a tutorial (134) and the widget with the information present in Synoptics widget.

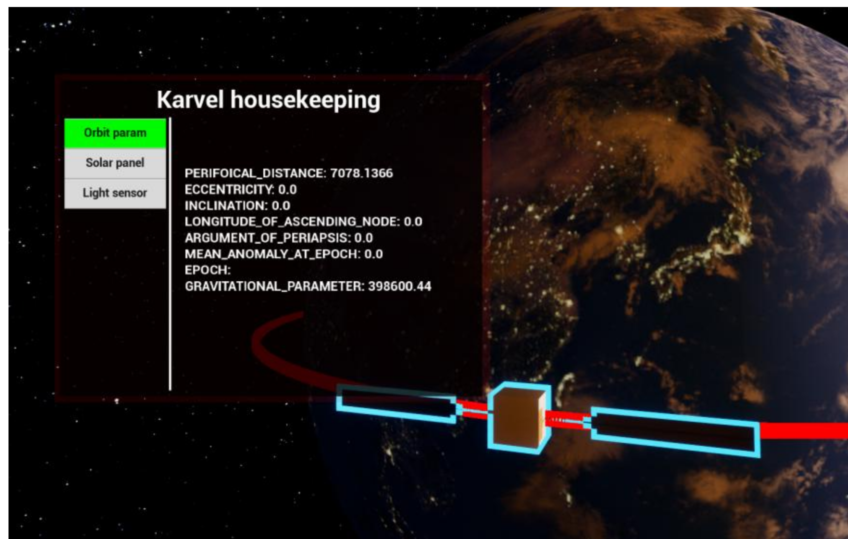


Figure 27 – Select satellite model

6.3.17 Main menu level

The last feature to discuss is the main menu level. This level is the start level when launching the application. It displays 4 buttons:

- **Launch**: Loads the actual digital twin simulator.
- **Setup MQTT**: Displays configuration options for host URL, port, client id, login, and password.
- **Settings**: Not implemented. Would contains options like language and units.
- **Quit**: Terminates the application.

The main menu contains a live position of the Sun, Earth, and Moon for a better UX. Shows an example of how the start menu may look like:

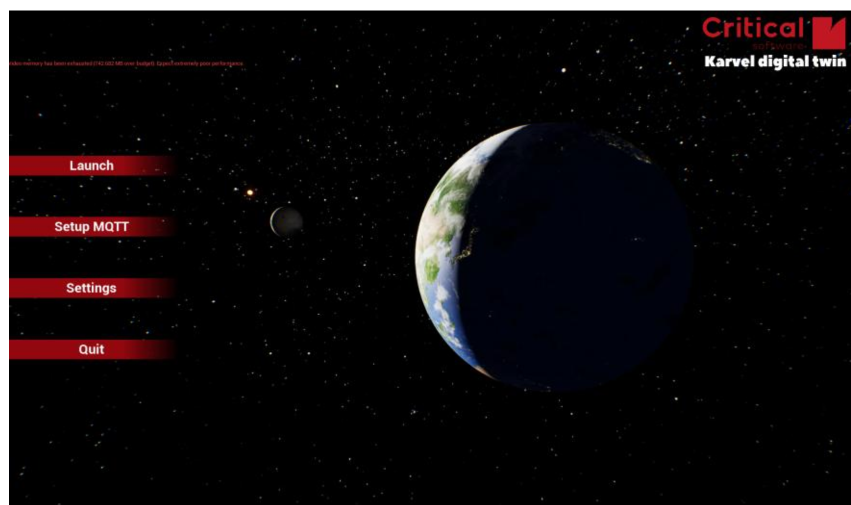


Figure 28 – Main menu

7 Conclusion

With the conclusion of the internship, this dissertation documents the development of a proof-of-concept application for a digital twin. The project addresses aspects such as communication with mission control and satellite platform, resolution of challenges related to orbital mechanics and software compatibility, and the implementation of features that support the overwatching and automation of satellite behaviours.

The work began with first contextualizing the challenges of building a digital twin within a 3D environment. A review of state-of-art technologies relevant to the theme of the project was presented, followed by an introduction to the fundamentals of orbital mechanics to highlight requirements and operations necessary for realistic orbit simulation.

Afterwards, a high-level overview of the architecture was provided, and requirements and diagrams were defined as a way to clarify the necessary components to be developed, constraints present, and data flow within the components of the digital twin.

To meet the defined architecture and requirements, several technologies were selected. Unreal Engine was chosen to contain the 3D environment, rendering, UI, and orbit calculations. Its built-in features that allow for quick deployment of a prototype, such as Blueprinting visual scripting language, plugins support, and UI development tools enabled rapid prototyping and implementation. The MaxQ plugin, which integrates NASA's SPICE toolkit into Unreal Engine, provided essential functions and 3D models of celestial bodies, facilitating the simulations of orbits as well a solar system with the main planets and the Sun.

For communications, MQTT was adopted due to accessibility in its implementations and integration via a suitable plugin. Telemetry data was serializable into JSON format, simplifying data exchange and manipulation. Regarding data storage, SQLite was selected along with a plugin to establish a connection to a database. Due to the lack of free time-series database plugins, SQLite proved sufficient in storing data, contributing the project's goals.

7.1 Requirement Fulfilment Overview

In this section, a summary of the implementation status for both functional and non-functional requirements is provided. Each requirement is classified as implemented, partially implemented, or not implemented. For each case, a brief justification on each status is provided as to explain the rational and clarify any implications on the project.

7.1.1 Functional requirements status

Functional Requirement	Priority	Status
DT-REQ-F-0010:NearRealTime	High	Implemented
DT-REQ-F-0020:AdaptableCommunication	Medium	Partially Implemented
DT-REQ-F-0030:OrbitViewer	Critical	Implemented
DT-REQ-F-0040:OrbitComparer	Critical	Implemented
DT-REQ-F-0050:TimeManipulation	High	Implemented
DT-REQ-F-0060:MissionControlIntegration	Medium	Implemented
DT-REQ-F-0070:TelemetryAndTelecommand	Critical	Implemented
DT-REQ-F-0080:DataVisualization	High	Implemented
DT-REQ-F-0090:TelemetrySelection	Medium	Implemented
DT-REQ-F-0100:DataStorage	Medium	Implemented
DT-REQ-F-0110:Customizability	Low	Partially Implemented

7.1.1.1 DT-REQ-F-0010:NearRealTime

- **Status**: Implemented
- **Rationale**: Changes made in Karvel are transmitted to Lighthouse for processing in the next telemetry cycle. Lighthouse then forwards the processed telemetry to the digital twin. The total transmission time depends only on the processing and transmission delays in Karvel and Lighthouse, and the time required for the digital twin to update the virtual satellite.

7.1.1.2 DT-REQ-F-0020:AdaptableCommunication

- **Status**: Partially Implemented
- **Rationale**: The main menu includes a section to manually edit MQTT connection parameters, which are variables stored in `BP_SharedData`. However, the submitted values are not yet applied. The current implementation uses hardcoded values.

7.1.1.3 DT-REQ-F-0030:OrbitViewer

- **Status**: Implemented
- **Rationale**: A satellite orbit visualizer around Earth is fully implemented using the six Keplerian orbital elements. In offline mode, users can manually modify the orbit without requiring a connection to Karvel or Lighthouse.

7.1.1.4 DT-REQ-F-0040:OrbitComparer

- **Status**: Implemented

- **Rationale**: A dedicated section allows users to compare the current Karvel orbit with a second, user-defined orbit. Upon submission, the application waits for confirmation and updates the telemetry accordingly.

7.1.1.5 DT-REQ-F-0050:TimeManipulation

- **Status**: Implemented
- **Rationale**: A simulation speed control interface was added, allowing users to fast-forward, backtrack, or jump to specific dates. Time manipulation affects both the virtual satellite and celestial bodies, which update their state accordingly.

7.1.1.6 DT-REQ-F-0060:MissionControlIntegration

- **Status**: Implemented
- **Rationale**: Unreal Engine's built-in web browser widget was configured to connect directly to the Lighthouse server.

7.1.1.7 DT-REQ-F-0070:TelemetryAndTelecommand

- **Status**: Implemented
- **Rationale**: Sections for telemetry visualization and telecommand transmission were added using the MQTT plugin.

7.1.1.8 DT-REQ-F-0080:DataVisualization

- **Status**: Implemented
- **Rationale**: Telemetry values can be visualized through graphs and synoptic panels, allowing users to monitor satellite parameters in across time.

7.1.1.9 DT-REQ-F-0090:TelemetrySelection

- **Status**: Implemented
- **Rationale**: Upon receiving telemetry, users can select which values to update and display in graphs. The application tracks received synoptics and automatically removes unused or missing ones.

7.1.1.10 DT-REQ-F-0100:DataStorage

- **Status**: Implemented
- **Rationale**: All telemetry is stored in an SQLite database. If the database file is missing, it is automatically created and configured.

7.1.1.11 DT-REQ-F-0110:Customizability

- **Status**: Partially Implemented
- **Rationale**: Some customization options are available, such as orbit colour and relative size of celestial bodies. Additional features would require further development.

7.1.2 Non-functional requirements status

ID	Priority	Status
DT-REQ-NF-0010:Usability	Medium	Partially Implemented
DT-REQ-NF-0020:SecureData	Low	Not implemented
DT-REQ-NF-0030:Replayability	Medium	Not implemented
DT-REQ-NF-0040:TelecommandCompletion	Critical	Implemented
DT-REQ-NF-0050:Performance	Medium	Not Implemented
DT-REQ-NF-0060:Portability	Medium	Implemented
DT-REQ-NF-0070:KnowledgeTransfer	Medium	Implemented

7.1.2.1 DT-REQ-NF-0010:Usability

- **Status**: Partially Implemented
- **Rationale**: During testing with Critical Software employees, basic controls (e.g., camera movement, orbital parameters) were intuitive. However, some features required explanation due to UI complexity.

7.1.2.2 DT-REQ-NF-0020:SecureData

- **Status**: Not Implemented
- **Rationale**: No security measures were implemented nor tested, as the project is a prototype focused on digital twin functionalities and is being tested in a closed environment with strict access. In a production environment, security would be a higher priority as to protect confidential data and satellite access.

7.1.2.3 DT-REQ-NF-0030:Replayability

- **Status**: Not Implemented
- **Rationale**: Although telemetry data is stored in the database, no feature currently retrieves or replays it.

7.1.2.4 DT-REQ-NF-0040:TelecommandCompletion

- **Status**: Implemented
- **Rationale**: Before deploying changes, the digital twin awaits for a telecommand acknowledgement (`TCCompletion`). If the deadline is missed, the previous state is retained.

7.1.2.5 DT-REQ-NF-0050:Performance

- **Status**: Not Implemented
- **Rationale**: Occasional slowdowns and crashes were observed during development and testing. While expected during prototyping, performance optimization is needed for production use. Issues also occurred when launching empty projects,

which means there’s no telling if it is the Unreal Engine application, digital twin project, or hardware/software incompatibilities.

7.1.2.6 DT-REQ-NF-0060:Portability

- **Status**: Implemented
- **Rationale**: Unreal Engine natively supports building the application for multiple OS.

7.1.2.7 DT-REQ-NF-0070:KnowledgeTransfer

- **Status**: Implemented
- **Rationale**: Code comments and documentation were provided. Additional notes and resources were shared via a dedicated Confluence page for the digital twin project.

7.2 Future work

By the end of the internship, the final project included a functional prototype showcasing the capabilities of the technology and highlighting useful features for implementing a digital twin visualizer. Despite this, several modifications can be made as to further improve what the application can do. These modifications were either not done because of time constraints or simply because they were not possible under the available resources. Additionally, some of the possible future features proposed are not related to the context of the internship and master’s degree’s field of study.

The features discussed in this section are not related to what is already implemented or partially implemented, but content that can bring more useful options and help the user integrating the digital twin in an existing project, or to simply offer more options to automatize the application and improve its usability.

7.2.1 Time-series database

One of the main features already mentioned in Section 5.6.2 is the replacement of the SQLite database by a proper time-series database. This type of database is specifically made to manage time-stamped events, is present in this project via telemetries, telecommands and other events. They can store and query data more efficiently and offer additional features that may support future enhancements, depending on the developers' goals.

SQLite is not a database developed for quick storage of time-stamped data or to quickly query it, but as a proper match of plugin and time-series database could not be found, the limited uses of SQLite were enough to simulate the requested features in a proof-of-concept project.

7.2.2 In-depth orbit and force calculations

Despite the great features of SPICE toolkit and MaxQ, there are several key components that affect real-life satellites and celestial bodies when in orbit, which would require manually developing complex code. Incorporating high-fidelity force models - such as atmospheric drag, detailed geopotential, and solar radiation pressure - would significantly enhance realism and orbital accuracy.

Furthermore, one could make further changes in developing custom functions that calculates not just the high-fidelity forces but also manages all the gravitational forces of the celestial bodies and artificial satellites that affect each other. In other words, an application that can calculate how a body with a certain gravity, mass, speed, rotation and atmospheres affect the movement of all the other bodies in the system. Such complex project would require careful planning, as to list all forces, variables and what sacrifices must be done to run it at real-time with minimal drops in performance, like only calculating those forces to a maximum distance, as outside of the max range they could be minimal and ignored.

The plugin MaxQ still bring great features by exposing SPICE toolkit functions to both C++ and Blueprints, so any useful feature present in them can still be used, lowering development time and resource usage.

7.2.3 AI

Due to the ever-increasing usage and efficiency of artificial intelligence (AI) in modern projects, one could also use it to overwatch satellite behaviour and automate the process. The type of AI would need to be discussed between developers, as different pros and cons exist.

Nonetheless, in relation to training of the AI, could be trained using historical mission data to recognize patterns and respond to anomalies. Additionally, custom data generated from simulated missions within the application could further improve its learning capabilities. Rendered images from the Unreal Engine could also be used to train the AI, using the actual visual position of the bodies in the system to predict behaviours. Both could be used as a way to modify the satellite with commands and view if the modifications are visible in the digital twin.

7.2.4 Satellite federation

The current digital twin project only tracks a single satellite as to demonstrate a connection is possible and what features can be provided using data provided by that satellite.

In a digital twin capable of managing a satellite federation, the application could track and overwatch several satellites at once in a single digital twin instance. Additionally, the application could serve as a communication bridge between satellites, either because they contain useful data worth sharing, are on the same mission, or simply to warn others of possible collisions and require changing paths.

7.2.5 Communication protocols

Right now, the project was only developed using MQTT, as it was a relatively simple and quick way to integrate into Unreal Engine via plugins and Eclipse Mosquitto application. Ideally, a digital twin application running in a game engine like Unreal Engine should be capable of connecting through various communication protocols, specially standardized protocols that connect directly to a satellite platform or the mission control.

7.2.6 Housekeeping and event warning

Another way to automate the project and enhance functionality is by proving an interface to see housekeeping data and payload status of the satellite. This data would be received by the digital twin and displayed in a specific section of the UI. Housekeeping data such as energy consumption, solar panel temperature, and signal strength would be displayed in a dedicated UI section. Threshold alerts could notify users of anomalies (e.g., overheating), prompting manual or automated corrective actions.

Event warnings are not limited to the satellite and the payload themselves, as the interaction of the satellite with other bodies orbiting the solar system may trigger collision alerts and unwanted orbit manoeuvres.

7.2.7 Customization and UI/UX

Finally, as this project was focused on implementing a digital twin and testing features, minor customizable options and visual aspects were left unpolished or missing. Some of the proposed options - such as unit selection (metric vs. imperial), multilingual support, customizable orbit colours and thickness, and improved UI/UX - would benefit from dedicated design and implementation.

These enhancements would significantly improve user experience and adaptability, making the digital twin more accessible and mission ready.

7.3 Reflections

This internship allowed me to improve my technical and problem-solving skills. The project focused on developing an environment to simulate orbit behaviours and test features for monitoring and automating artificial satellites, while also improving access to critical data for easier management. A key goal was to connect the digital twin to the Karvel OBSW and Lighthouse mission control application.

Having no prior experience with game engines, the initial weeks were dedicated to learning how Unreal Engine operates – such as adding 3D models, plugin management, creating variables and functions, and understanding how code is executed through Blueprints. In parallel, I studied Karvel and Lighthouse, running them locally and analysing their intercommunications. This was crucial to identify the types of interactions and features the digital twin would need. Fortunately, MaxQ provided example projects that helped understand how to implement specific logics and features.

Once the foundations were in place, development of the digital began by taking advantage of MaxQ to building the solar system. Starting with Earth and Sun positioned according to real-life data leveraged form kernel files, gradually the remaining planets and Moon were added, with accurate placement, rotation and movement. At this stage, plugins for MQTT and SQLite were researched, installed, and tested to prepare the digital twin for receiving telemetry data from Karvel and Lighthouse once properly configured.

With telemetry successfully received and applied to the virtual counterpart, additional features were developed to visualize and interact with the data - such as graph and synoptics panel. Storing telemetry data proved to be a more challenging aspect, as no suitable time-series database and plugins were available. The fallback solution using SQLite required more complex logic for query handling and data management.

I began this project fully aware that I lacked expertise in both orbital mechanics and game engines. However, I was committed to learning with a genuine interest in these areas. This project provided a unique opportunity to learn new skills and solve problems in unfamiliar environments.

Although not all requirements were fully implemented, I view this project as a learning journey, exploring subjects of my interest and achieving results within the time available. Through the digital twin project, I gained new skills that motivate me to pursue similar projects in the future, and to continue improving my skills in these technologies and learn new ones.

8 Bibliography

1. **Galeb, Basim, et al.** *Advancements in Satellite Communication Systems: Challenges and Opportunities.* Baghdad, Iraq : Journal of Mechanics of Continua and Mathematical Sciences, 2024.

2. **Shah, Syed, Nasir, Ammar and Ahmed, Hafeez.** *A Survey Paper on Security Issues in Satellite Communication Network infrastructure.* s.l. : International Journal of Engineering Research and General Science Volume 2, 2014.

3. **Kodheli, Oltjon, et al.** *Satellite Communications in the New Space Era: A Survey and Future Challenges.* s.l. : IEEE Communications Surveys & Tutorials, 2021.

4. **Trimble, Jay and Walton, Joan.** *Mission Control Technologies: A New Way of Designing and Evolving Mission Systems.* s.l. : SpaceOps 2006 Conference, 2006.

5. **Modenini, Andrea and Ripani, Barbara.** *A Tutorial on the Tracking, Telemetry, and Command (TT&C) for Space Missions.* 2022.

6. **Keesee, Col John E.** *Satellite Telemetry, Tracking and Control Subsystems.* s.l. : Massachusetts Institute of, 2003.

7. **European Space Agency (ESA).** TT&C and PDT Systems and Techniques section. [Online] [Cited: 22 September 2025.] https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Radio_Frequency_Systems/TT_C_and_PDT_Systems_and_Techniques_section.

8. **EUMETSAT.** Telemetry, Tracking and Control. *EUMETSAT.* [Online] 15 November 2022. [Cited: 22 September 2025.] https://www.eumetsat.int/telemetry-tracking-and-control.

9. **Sela, Alejandro and Mihalache, Nicolae.** *YAMCS - A Mission Control System.* s.l. : Space Applications Services.

10. **YAMCS.** YAMCS. [Online] [Cited: 22 September 2025.] https://yamcs.org/about.

11. *Fuzzing NASA Core Flight System Software.* **Brobert, R.** DEF CON 29 Aerospace Village : NASA, 2021.

12. *Introduction to the core Flight System (cFS).* **Dave.** 2020.

13. **Knutsen, Dan.** *The Core Flight System (cFS).* Pasadena, California, USA : NASA Flight Software Workshop, 2023.

14. **NASA Goddard Space Flight Center.** Core Flight System (cFS). *Core Flight System.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://www.coreflightsystem.net/.

15. **Bradbury, John W.** *Open Source Core Flight System (cFS) Flight Software (FSW) Verification & Validation (V&V) Final Summary.* Greenbelt, Maryland, USA : NASA Goddard Space Flight Center, 2020.

16. **NASA.** Core Flight System (cFS). *GitHub.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://github.com/nasa/cFS.

17. **Prokop, Lorraine.** *NASA's Core Flight Software - a Reusable Real-Time Framework.* Houston, Texas, USA : Johnson Space Center (JSC) - NASA, 2014.

18. **Kowalski, Richard and Huy, Frank.** Real-Time Operating Systems (RTOS) 101. *NASA.* [Online] [Cited: 22 September 2025.] https://www.nasa.gov/wp-content/uploads/2016/10/482489main_4100_-_rtos_101.pdf?emrc=0f3dd6.

19. **Taylor, John T. and Taylor, Wayne T.** *The Embedded Project Cookbook.* 2024. https://doi.org/10.1007/979-8-8688-0327-7.

20. **Cudmore, Alan.** Platform Layer Updates for the Caelum (7.0) Release of the Core Flight System. *NASA Technical Reports Server (NTRS).* [Online] 1 April 2021. [Cited: 22 September 2025.] https://ntrs.nasa.gov/api/citations/20210000909/downloads/cFS_Platform_Updates-v5-final.pptx.pdf.

21. **McComas, David.** *NASA/GSFC's Flight Software Core Flight System.* Southwest Research Institute San Antonio, Texas, USA : NASA Goddard Space Flight Center, 2012.

22. **NASA.** cFE. *GitHub.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://github.com/nasa/cfe.

23. —. cFE Application Developers Guide. *GitHub.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://github.com/nasa/cFE/blob/main/docs/cFE%20Application%20Developers%20Guide.md.

24. **Hodson, Daniel.** fuzzotron. *Github.* [Online] [Cited: 22 September 2025.] https://github.com/denandz/fuzzotron.

25. **NASA.** Core Flight Executive User's Guide. [Online] [Cited: 22 September 2025.] https://github.com/nasa/cFE/blob/gh-pages/cfe-usersguide.pdf.

26. **CCSDS.** *Overview of Space Communications Protocols - Green Book.* s.l. : CCSDS, 2023. CCSDS 130.0-G-4.

27. —. *Encapsulation Packet Protocol - Blue Book.* s.l. : CCSDS, 2020. CCSDS 133.1-B-3.

28. —. *TC Space Data Link Protocol - Blue book.* s.l. : CCSDS, 2017. CCSDS 232.0-B-2-E-1.

29. —. *Space Packet Protocols - Green Book.* s.l. : CCSDS, 2023. CCSDS 130.3-G-1.

30. —. *Space Packet Protocols - Blue Book.* s.l. : CCSDS, 2020. CCSDS 133.0-B-2.

31. **NASA Goddard Space Flight Center.** SCH: Core Flight System Scheduler Application. *GitHub.* [Online] 4 Octoer 2019. [Cited: 22 September 2025.] https://github.com/nasa/SCH.

32. **Park, Jongyeob, et al.** *Application of NASA core Flight System to Telescope Control Software for 2017 Total Solar Eclipse Observation.* s.l. : Publications of the Astronomical Society of the Pacific, 2022. DOI 10.1088/1538-3873/ac5848.

33. **Eleffendi, Mohammed, et al.** *NASA/GSFC's Flight Software Core Flight System Implementation For A Lunar Surface Imaging Mission.* 2022.

34. **Consultative Committee for Space Data Systems (CCSDS).** *CCSDS File Delivery Protocol (CFDP).* s.l. : CCSDS, 2020. CCSDS 727.0-B-5.

35. **Paul, Partha Sarathi, Goon, Surajit and Bhattacharya, Abhishek.** *History and Comparative Study of Modern Game Engines.* s.l. : International Journal of Advanced Computer and Mathematical Sciences, 2012. 2230-9624.

36. **Šmíd, Antonín.** *Comparison of Unity and Unreal Engine.* 2017.

37. **Lee, Joanna.** *Learning Unreal Engine Game Development.* Birmingham, UK : Packt Publishing Ltd, 2016. 978-1784398156.

38. **Winter, David.** Introduction. *Pong-Story.* [Online] [Cited: 22 September 2025.] https://www.pong-story.com/intro.htm.

39. **Campbell-Kelly, Martin.** *A Tutorial Guide to the EDSAC Simulator.* Milton Keynes, UK : The National Museum of Computing, 2001.

40. **Atari, Inc.** *Schematic Diagrams: Pong.* s.l. : Internet Archive, 1972.

41. **Kent, Steven.** *The Ultimate History of Video Games.* s.l. : Three Rivers Press, 2001.

42. **Perron, Bernard, et al.** *Fifty Key Video Games.* New York : s.n., 2022. 9781003199205.

43. **Cantrell, Topher.** Space Invaders Code Listing. *Computer Archeology.* [Online] [Cited: 22 September 2025.] https://computerarcheology.com/Arcade/SpaceInvaders/Code.html.

44. **Sanglard, Fabien.** *Game Engine Black Book: Wolfenstein 3D v2.1.* Scotts Valley, CA : CreateSpace Independent Publishing Platform, 2019. 978-1070515847.

45. **Lowood, Henry.** *Game Engines and Game History.* s.l. : History of Games International Conference Proceedings, 2014. 1916-985X.

46. **Wade, Bretton.** Binary Space Partitioning Trees FAQ. *AQs.org.* [Online] 1995. [Cited: 22 September 2025.] http://www.faqs.org/faqs/graphics/bsptree-faq/.

47. **Sanglard, Fabien.** *Game Engine Black Book: DOOM.* s.l. : Software Wizards, 2018. 978-0-9994843-0-9.

48. **Munro, James, Boldyreff, Cornelia and Capiluppi, Andrea.** *Architectural Studies of Games Engines – the Quake Series.* 2009.

49. **USDQC Project Contributors.** QuakeC Manual. *USDQC GitHub Pages.* [Online] [Cited: 22 September 2025.] https://usdqc.github.io/quakec-resources/qcmanual.html.

50. **Zimbinski, Bob.** *Getting Started with Quake.* s.l. : Association for Computing Machinery (ACM).

51. **VG Insights.** *The Big Game Engines Report of 2025.* s.l. : InvestGame / VG Insights, 2025.

52. **Cowan, Brent and Kapralos, Bill.** *A Survey of Frameworks and Game Engines for Serious Game Development.* s.l. : IEEE, 2014.

53. **Ciekanowska, Agata Malwina , Kiszczak , Adam Krzysztof and Dziedzic , Krzysztof .** *Comparative analysis of Unity and Unreal Engine efficiency in creating virtual exhibitions of 3D scanned models.* Lublin, Poland : Journal of Computer Sciences Institute, 2021. 20 (2021) 247-253.

54. **Haas, John.** *A History of the Unity Game Engine.*

55. **Unity Technologies.** Introduction to Scripting. *Unity.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://docs.unity3d.com/6000.0/Documentation/Manual/intro-to-scripting.html.

56. —. Programming Solutions. *Unity.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://unity.com/solutions/programming.

57. —. Unity Compare Plans. *Unity.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://unity.com/products/compare-plans.

58. —. Unity Products. *Unity.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://unity.com/products.

59. **Barczak, Andrzej and Woźniak, Hubert.** *Comparative Study on Game Engines.* s.l. : University of Siedlce, 2019.

60. **Epic Games.** Unreal Engine Programming and Scripting. [Online] 2025. [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-programming-and-scripting.

61. **Grieves, Michael.** *Digital Model, Digital Shadow, Digital Twin.* 2023.

62. **Singh, Maulshree, et al.** *Digital Twin: Origin to Future.* s.l. : Applied System innovation, 2021.

63. **Allen, B. Danette.** *Digital Twins and Living Models at NASA.* Langley Research Center Hampton, Virginia, United States : s.n., 2021.

64. **Schade, Aleksander.** *Digital Twin Technology for Aviation.* University of California, Santa Cruz Santa Cruz, California, United States : s.n., 2023.

65. **Kampkera, A., et al.** *Business Models for Industrial Smart Services – The Example of a Digital Twin for a Product-Service-System for Potato Harvesting.* s.l. : ScienceDirect, 2019.

66. **Chakshu, Neeraj Kavan, et al.** *A semi-active human digital twin model for detecting severity of carotid stenoses from head vibration—A coupled computational mechanics and computer vision method.* 2019.

67. **Ferguson, Stephen.** Apollo 13: The First Digital Twin. *Simcenter Blog – Siemens Digital Industries Software.* [Online] 2020. [Cited: 22 September 2025.] https://blogs.sw.siemens.com/simcenter/apollo-13-the-first-digital-twin/.

68. **Gamergenic.** MaxQ. *GitHub.* [Online] [Cited: 22 September 2025.] https://github.com/Gamergenic1/MaxQ.

69. **Epic Games.** Unreal Engine License. [Online] 2025. [Cited: 22 September 2025.] https://www.unrealengine.com/license.

70. **Azure.** Open Digital Twins Definition Language (DTDL) v3. *GitHub.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v3/DTDL.v3.md.

71. **Microsoft.** Digital Twins. *GitHub.* [Online] 28 April 2025. [Cited: 22 September 2025.] https://learn.microsoft.com/en-us/azure/digital-twins/.

72. **Amazon Web Services.** AWS IoT. *Amazon Web Services.* [Online] [Cited: 22 September 2025.] https://aws.amazon.com/iot/.

73. —. AWS IoT TwinMaker. *Amazon Web Services.* [Online] [Cited: 22 September 2025.] https://aws.amazon.com/iot-twinmaker/.

74. **Blackswan Space.** Mission Design Simulator. *Blackswan Space.* [Online] [Cited: 22 September 2025.] https://www.blackswan.ltd/mission-design-simulator/.

75. **Sedaro.** Sedaro | Mission Simulation at Scale. *Sedaro.* [Online] [Cited: 22 September 2025.] https://www.sedaro.com/.

76. **D-Orbit.** Aurora. *D-Orbit.* [Online] [Cited: 22 September 2025.] https://www.dorbit.space/aurora.

77. **SOS Orbital Mechanics.** Satellite Orbit Simulator. *SOS Orbital Mechanics.* [Online] [Cited: 22 September 2025.] https://sos-orbital-mechanics.com/.

78. **Burgess, Alan and Motes, Andrew.** *Space Mission Design Using Satellite Orbit Simulator (SOS) Version 28.3.* 2025.

79. **Banks, Andrew, et al.** MQTT Version 5.0 OASIS Standard. *OASIS Open.* [Online] 7 March 2019. [Cited: 22 September 2025.] https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

80. **Sengul, C. and Kirby, A.A.** RFC 9431: Message Queuing Telemetry Transport (MQTT) and Transport Layer Security (TLS) Profile of Authentication and Authorization for Constrained Environments (ACE) Framework. *RFC Editor.* [Online] 3 July 2023. [Cited: 22 September 2025.] https://www.rfc-editor.org/rfc/rfc9431.html.

81. **Toldinas, Jevgenijus, et al.** *MQTT Quality of Service versus Energy Consumption.* s.l. : IEEE, 2019. DOI: 10.1109/ACCESS.2019.2926410.

82. **Shelby, Z., Frank, B. and Sturek, D.** Constrained Application Protocol (CoAP). *IETF (Internet Engineering Task Force).* [Online] 27 September 2010. [Cited: 22 September 2025.] https://www.ietf.org/archive/id/draft-ietf-core-coap-02.html. draft-ietf-core-coap-02.

83. **Shelby, Z., Hartke, K. and Bormann, C.** The Constrained Application Protocol (CoAP). *RFC Editor.* [Online] 1 June 2014. [Cited: 22 September 2025.] https://www.rfc-editor.org/rfc/rfc7252. RFC 7252.

84. **Object Management Group (OMG).** *Data Distribution Service (DDS), Version 1.4.* s.l. : Object Management Group (OMG), 2015. OMG Document Number: formal/2015-04-10.

85. **DDS Foundation.** DDS Guidebook. *OMG Wiki.* [Online] 29 October 2021. [Cited: 22 September 2025.] omgwiki.org/ddsf/doku.php?id=ddsf:public:guidebook:guidebook.

86. **Object Management Group (OMG).** *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPS) Specification.* s.l. : Object Management Group (OMG), 2022. formal/2022-04-01.

87. **Guerra, André and Carvalho, Paulo Simeão.** *Orbital motions of astronomical bodies and their centre of mass from different reference frames: a conceptual step between the geocentric and heliocentric models.* Porto : IOP, 2016.

88. **Kuhn, Thomas S.** *The Copernican Revolution.* s.l. : Cambridge: Harvard University.

89. **NASA.** Orbits and Kepler's Laws. *NASA Science.* [Online] [Cited: 22 September 2025.] https://science.nasa.gov/resource/orbits-and-keplers-laws/.

90. **Nave, Carl Rod.** Kepler's Laws. *HyperPhysics.* [Online] [Cited: 22 September 2025.] http://hyperphysics.phy-astr.gsu.edu/hbase/kepler.html.

91. **Bourdon, James S.** Definition of an Ellipse. *James S. Bourdon's Math Lectures – Richland College.* [Online] [Cited: 22 September 2025.] https://people.richland.edu/james/lecture/m116/conics/elldef.html.

92. **Ridpath, Ian.** *A Dictionary of Astronomy (2 ed.).* s.l. : Oxford University Press, 2012. 9780191739439.

93. **Algebrica.** Ellipse. *Algebrica.* [Online] [Cited: 22 September 2025.] https://algebrica.org/ellipse/.

94. **Weisstein, Eric W.** Eccentricity. *MathWorld.* [Online] [Cited: 22 September 2025.] https://mathworld.wolfram.com/Eccentricity.html.

95. **Encyclopædia Britannica.** Eccentricity (astronomy). *Britannica.* [Online] 3 August 2011. [Cited: 22 September 2025.] https://www.britannica.com/science/eccentricity-astronomy.

96. **NASA Jet Propulsion Laboratory (JPL).** Chapter 3 – Coordinate and Reference Systems. *SPICE Toolkit Documentation – NASA NAIF.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://spsweb.fltops.jpl.nasa.gov/portaldataops/mpg/MPG_Docs/MPG%20Book/Release/Chapter3-Coordinate%20&%20Reference%20Systems.pdf.

97. **NAIF.** *Fundamental Concepts.* s.l. : NASA, 2023.

98. **Vallado, David A.** *Fundamentals of Astrodynamics and Applications.* s.l. : Springer Science & Business Media, 2001.

99. **Thompson, William.** *Reading STEREO ephemerides as SPICE kernels.* 2006.

100. **NASA Navigation and Ancillary Information Facility (NAIF).** Frames Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/frames.html.

101. **Dinis, Gustavo E. S.** *Sun-synchronous Satellite Simulator.* 2017.

102. **NASA Jet Propulsion Laboratory (JPL).** Chapter 7 - Fundamentals of Orbital Mechanics. *SPICE Toolkit Documentation – NASA NAIF.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://spsweb.fltops.jpl.nasa.gov/portaldataops/mpg/MPG_Docs/MPG%20Book/Release/Chapter7-OrbitalMechanics.pdf.

103. **Gamergenic.** MaxQ. *Gamergenic.* [Online] [Cited: 2025 September 22.] https://www.gamergenic.com/project/maxq/.

104. **NASA Navigation and Ancillary Information Facility (NAIF).** Kernel Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/kernel.html.

105. —. *Introduction to Kernels.* s.l. : NASA Navigation and Ancillary Information Facility (NAIF), 2023.

106. —. SPICE Concept. *NASA NAIF.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/naif/spiceconcept.html.

107. —. DAF Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/daf.html.

108. —. DAS Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/das.html.

109. —. CK Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/ck.html.

110. —. DSK Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/dsk.html.

111. —. *The SPICE Concept.* s.l. : NASA, 1998.

112. —. EK Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 2022 January 3. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/ek.html.

113. —. *Instrument Kernel.* s.l. : NASA, 2023.

114. —. *Leapseconds and Spacecraft Clock Kernels.* s.l. : NASA, 2023.

115. —. FURNSH Details: What is a Meta-Kernel. *NASA NAIF.* [Online] 3 january 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/naif/furnsh_details.html.

116. —. PCK Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/pck.html.

117. —. SPK Required Reading. *NAIF SPICE Toolkit Documentation.* [Online] [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/spk.html.

118. —. Kernel Selection. *NASA NAIF.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/naif/kernel_selection.html.

119. —. Generic Kernels. *NASA NAIF.* [Online] 3 January 2022. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/naif/data_generic.html.

120. —. Competing Kernels. *NASA NAIF.* [Online] 29 October 2013. [Cited: 22 September 2025.] https://naif.jpl.nasa.gov/naif/WGC_competing_kernels.html.

121. **Dix, Paul.** *Why Time Series Matters For Metrics, Real-Time Analytics And Sensor Data.* s.l. : InfluxData, 2023.

122. **Piacentini, Mauricio.** DB Browser for SQLite. *sqlitebrowser.org.* [Online] 19 August 2003. [Cited: 22 September 2025.] https://sqlitebrowser.org/.

123. **sitonmoon.** UE4SimpleSQLite. *UE4SimpleSQLite.* [Online] 9 October 2019. [Cited: 22 September 2025.] https://github.com/sitonmoon/UE4SimpleSQLite.

124. **Allen, Grant and Owens, Mike.** *The Definitive Guide to SQLite.* 2010.

125. **Nineva Studios.** mqtt-utilities-unreal. *GitHub.* [Online] 17 April 2024. [Cited: 22 September 2025.] https://github.com/NinevaStudios/mqtt-utilities-unreal.

126. **Epic Games.** Blueprint Class. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprint-class-assets-in-unreal-engine.

127. —. Blueprints Function Libraries. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprint-function-libraries-in-unreal-engine.

128. —. Levels. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/levels-in-unreal-engine.

129. —. Level Blueprint. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/level-blueprint-in-unreal-engine.

130. —. Structs. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/structs-in-unreal-engine.

131. —. Widget Blueprints. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/widget-components-in-unreal-engine.

132. —. UGameInstance. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/Engine/UGameInstance.

133. —. Units of Measurement. *Epic Games.* [Online] [Cited: 22 September 2025.] https://dev.epicgames.com/documentation/en-us/unreal-engine/units-of-measurement-in-unreal-engine.

134. **The Game Dev Channel.** Unreal Engine City Building Game - Mouse Cursor, Click Object In-World and Highlight Material - EP 3. *Youtube.* [Online] [Cited: 22 September 2025.] https://www.youtube.com/watch?v=KakLXqpwnjo.