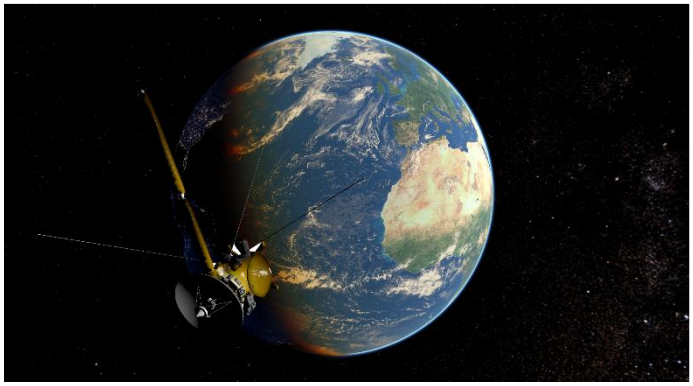
TÉCNICO LISBOA



A Software Package to Support Mission Analysis and Orbital Mechanics Calculations

Jorge Tiago Melo Barbosa da Silva e Vila

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor: Prof. Paulo Jorge Soares Gil

Examination Committee

Chairperson: Prof. João Manuel Lage de Miranda Lemos
Supervisor: Prof. Paulo Jorge Soares Gil
Members of the Committee: Prof. Bertinho Manuel D'Andrade da Costa

July of 2015

Dedicated to the loving memory of my grandfather
Joaquim Barbosa, whom I shall always remember.

Acknowledgements

Firstly, I would like to express my gratitude to my supervisor Prof. Paulo Gil for the ideas, remarks, comments, and many engaging conversations which not only saw me through the learning process of this thesis, but also helped me personally.

I want to thank Dr. Carlos del Burgo Díaz for the confidence shown in future applications of the software and the help provided with the star map alignment.

I want to acknowledge Patrice-Emmanuel Schmitz, from the Open Source Observatory and Repository, for his legal opinions regarding the EUPL licensing. Furthermore, I want to thank Pavel Holoborodko for the excellent MPFR C++ interface class and the professionalism, sympathy, and cooperation displayed while resolving licensing issues.

I would also like to thank my parents for the care and interest shown throughout the years in my education, and for the support given in pursuing the interests which led me here.

Lastly, I want to express my deepest gratitude to Ana Morais for the love, patience, and continuous support during the development of this thesis. Without you none of this would have been possible.

Abstract

The objective of this work was to develop a free and open-source software to support the tasks of space mission analysis and orbital mechanics calculations. It is comprised of two modules – a trajectory propagator and a three-dimensional Solar System simulator. The software was developed for Windows 7, and programmed in C++11. The distribution is made through a website, under the EUPL v.1.1.

The propagator uses the initial conditions provided by the user, either through the graphical interface, or XML input files, and propagates the trajectory aided by a Runge-Kutta-Fehlberg 7(8) numerical integrator. The propagator uses a list of force systems, based on several central bodies, and selects the most adequate during the computation. The force model includes the central body's gravity field and the perturbation introduced by its J_2 coefficient, third bodies' gravity fields, and the solar radiation pressure. The propagator generates plain-text and NASA SPK output files.

The simulator is a three-dimensional rendering of the Solar system and allows the user to visualize the trajectory and attitude of celestial bodies and spacecraft. The 3D models of the Sun and planets, with day and night textures, atmosphere, clouds, and planetary rings, are procedurally generated. It is also possible to load 3D models for more complex geometries. A manual camera controlled by the keyboard and mouse was implemented, along with a system of automatic cameras parametrized by the user. The simulator is also capable of exporting movies.

Keywords: Orbital mechanics, Software, Orbital propagator, Visualization tool

Resumo

O objectivo deste trabalho foi desenvolver um software gratuito e de código aberto para auxiliar as tarefas de análise de missões espaciais e cálculos de mecânica orbital. Este é composto por dois módulos – um propagador de trajectória e um simulador tridimensional do Sistema Solar. O software foi desenvolvido para Windows 7, e foi programado em C++11. A distribuição é feita através de um website, ao abrigo da EUPL v.1.1.

O propagador utiliza condições iniciais fornecidas pelo utilizador através do interface gráfico, ou de ficheiros de entrada XML, e propaga a trajectória com recurso ao integrador numérico Runge-Kutta-Fehlberg 7(8). O propagador utiliza uma lista de sistemas de força, baseado em vários corpos centrais, e selecciona automaticamente o adequado durante o cálculo. O modelo de forças inclui o campo gravítico do corpo central e perturbação introduzida pelo seu coeficiente J_2, o campo gravítico de terceiros corpos e a pressão de radiação solar. O propagador gera ficheiros de saída em texto simples e no formato SPK da NASA.

O simulador é uma representação tridimensional realística do Sistema Solar e permite ao utilizador visualizar a trajectória e atitude de corpos celestes e naves espaciais. Os modelos 3D do Sol e dos planetas, com texturas de dia e noite, atmosfera, nuvens e anéis planetários, são gerados automaticamente. É também possível carregar modelos 3D para geometrias mais complexas. Foi implementada uma câmara manual controlada pelo teclado e rato, e um sistema de câmaras automáticas parametrizadas pelo utilizador. O simulador é também capaz de exportar filmes.

Palavras-chave: Mecânica orbital, Software, Propagador orbital, Ferramenta de visualização

Contents

List of Tables

List of Figures

List of Abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
Assimp	Open Asset Import Library
BRDF	Bidirectional Reflectance Distribution Function
CBR	Constant Bit Rate
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DLL	Dynamic Link Library
DXUT	DirectX Utility Toolkit
ESA	European Space Agency
EULA	End-User License Agreement
EUPL	European Union Public Licence
FOSS	Free and Open Source Software
FPS	Frames per Second
GEO	Geosynchronous Equatorial Orbit
GMAT	General Mission Analysis Tool
GMP	GNU Multiple Precision Arithmetic Library
GPLv3	GNU Public License v3
GUI	Graphical User Interface
HIC++	High Integrity C++
HLSL	High Level Shader Language
HTML	HyperText Markup Language
IAU	International Astronomical Union
ICATT	International Conference on Astrodynamics Tools and Techniques
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
Inria	Institut National de Recherche en Informatique et en Automatique
ISO	International Organization for Standardization
IST	Instituto Superior Técnico
LEO	Low Earth Orbit
LIP	Laboratoire de l'Informatique du Parallélisme
Loria	Laboratoire Lorrain de Recherche en Informatique et ses Applications
MPIR	Multiple Precision Integers and Rationals
MSDN	Microsoft Developer Network
MSDNAA	MSDN Academic Alliance
NAIF	Navigation and Ancillary Information Facility
NASA	National Aeronautics and Space Administration
Glee	OpenGL Easy Extension Library
GLEW	OpenGL Extension Wrangler Library
OSI	Open Source Initiative
PDF	Portable Document Format
RFK7(8)	Runge-Kutta-Fehlberg 7(8)
SDK	Software Development Kit
SGI	Silicon Graphics, Inc.
SI	International System of Units
SOLib	SimOrbit Library
STA	Space Trajectory Analysis
STK	Systems Tool Kit
SVN	Apache Subversion
TDB	Barycentric Dynamical Time
TDT	Terrestrial Dynamical Time
TLE	Two-Line Element
UTC	Coordinated Universal Time
VBR	Variable Bit Rate
WMV	Windows Media Video

XML	Extensible Markup Language
XSD	XML Schema Definition

List of Symbols

a	Semi-major axis length (orbital element)
$\boldsymbol{a}$	Acceleration vector of an engine constant acceleration input
A	Cross sectional area of the spacecraft perpendicular to the Sun direction
C_R	Radiation pressure coefficient
e	Eccentricity (orbital element)
$E(t)$	Time-dependent matrix that transforms coordinates in the engine frame to the J2000 frame
E_0	Reference epoch
G	Gravitational constant
h	Numerical integrator time step
i	Inclination (orbital element)
J_2	Zonal harmonic coefficient of the gravity field of a central body
m	Mass of a spacecraft
M	Mass of a central body
M_s	Mass of a third body
M_0	Mean anomaly at reference epoch (orbital element)
n	Mean orbital velocity
$P_\odot$	Solar radiation pressure in the vicinity of the Earth
$\boldsymbol{r}$	Position of a spacecraft with reference to a central body
$\dot{\boldsymbol{r}}$	Velocity of a spacecraft with reference to a central body
$\ddot{\boldsymbol{r}}$	Acceleration of a spacecraft with reference to a central body
$\boldsymbol{r_s}$	Position of a third body with reference to a central body
$\boldsymbol{r}_\odot$	Position of the Sun relative to a central body
R	Coefficient of cross-correlation
R^2	Coefficient of determination
R_E	Equatorial radius of a central body
t_H	Time taken to perform a Hohmann transfer
T_P	Time of periapsis (orbital element)
tol_{pos}	Numerical integrator position local truncation error tolerance coefficient
tol_{vel}	Numerical integrator velocity local truncation error tolerance coefficient
$U(t)$	Time-dependent matrix that describes a planet's rotation in the J2000 frame
$\boldsymbol{v}$	Velocity vector of an engine delta-V
Δv	Magnitude of an engine delta-V impulse
$\Delta\omega$	Variation of the argument of periapsis
$\Delta\Omega$	Variation of the longitude of ascending node
ε	Reflectivity of a spacecraft
ε_i	Residual of a data point
ε_{pos}	Numerical integrator local position error factor
ε_{vel}	Numerical integrator local velocity error factor
ν	Fraction of sunlight received by a spacecraft
τ_{pos}	Numerical integrator position local truncation error
τ_{vel}	Numerical integrator velocity local truncation error
ω	Argument of periapsis (orbital element)
Ω	Longitude of ascending node (orbital element)

Chapter 1

INTRODUCTION

The objective of this work is to develop a free and open-source (FOSS) software package, comprising a three-dimensional visualization tool for enhanced data analysis and an orbital propagator for data generation.

1.1 Context

The simulation and analysis of trajectories is required throughout the entire lifecycle of every space mission. Designing a space mission trajectory requires precise ephemerides, reliable physical models, and robust propagation algorithms. Furthermore, the calculated trajectory needs to be optimized and analysed for viability. This analysis must take into account fuel consumption, illumination and radio-contact conditions, objects in the flight path, among many other factors.

Computers have aided space mission design since the beginning of the space era and their increasing capabilities allow for novel mission analysis and design methods [1]. Currently, it is possible to take into account many more factors when propagating trajectories and do so in any ordinary household computer, instead of a supercomputer. These advances in computer technology allowed for the development of general mission design software solutions.

The birth of three-dimensional computer graphics and its ensuing staggering improvement, largely driven by the videogame industry, contributed to the creation of brand new visualization tools. Long data tables and plots are no longer the only data analysis tools available. Orbital mechanics problems can now be viewed in fully realistic three-dimensional scenarios.

Combining accurate trajectory propagation and visualization capabilities within the same software suite provides mission designers a fully-featured solution, greatly supporting their work.

1.2 Modern Software for Mission Analysis and Design

There are two types of software for mission analysis and design – mission-specific and general. The mission specific software is usually developed by a space agency or a contractor with a single mission in mind. It is highly specific and difficult to adapt to other missions. On the other hand, the general software is a flexible solution that supports several mission types. Typically, it comes in the form of a costly commercial solution or a free and collaborative open-source project led by a space agency. In addition to these integrated solutions, there are several other tools that serve specific purposes, such as visualization of trajectories.

In the next sections the leading commercial off-the-shelf (COTS) software suites and open-source projects are reviewed, along with the most popular space visualization tool, to establish a standard of comparison.

1.2.1 Systems Tool Kit (STK)[1]

Developer: Analytical Graphics, Inc.

Version: 10.1.1 (February 2014)

Operating system: Windows XP; Windows Vista; Windows 7 (32-bit and 64-bit)

License: Proprietary

The Systems Tool Kit (STK) [2] is a COTS 2D and 3D modelling and visualization environment for complex mission systems on land, sea, air, and space, as demonstrated in Figure 1.1. STK is a mature product with over 40,000 installations worldwide. The STK basic version is free; however, the modules required for most applications are not. Hence, STK is widely regarded as a costly solution and is typically negotiated under a firm-fixed price contract in a per-client basis [3].

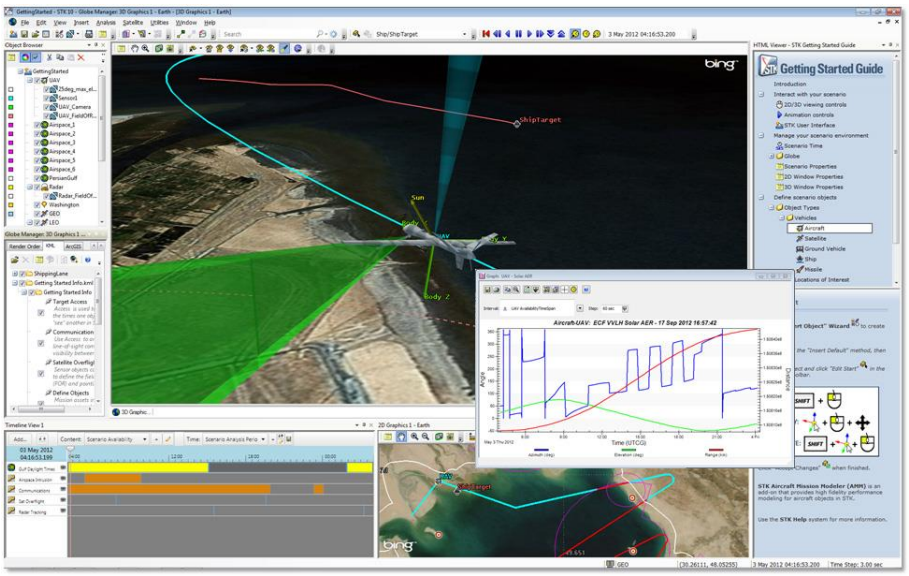


Figure 1.1 – The STK graphical user interface [2].

The free STK version is mostly an analysis and visualization tool. It is able to import data (such as vehicle position and orientation), evaluate the system performance, and convey results through custom reports, graphs, and mission simulation. Added functionality is provided by the 19 available paid modules, two of which relevant in the orbital mechanics context:

- The STK Pro module features high-precision, long-term and real-time satellite orbit propagators, the ability to simulate satellite attitude, and movie-making capabilities;
- The SatPro module adds specialized satellite propagators, attitude profiles, and satellite engineering tools.

[1] Formerly Satellite Tool Kit.

1.2.2 FreeFlyer

Developer: a.i. solutions, Inc.
Version: 6.10 (February 2014)
Operating system: Windows
License: Proprietary

FreeFlyer [4] is another COTS software application for use in satellite mission analysis, design, and operations. It is also a widely used and mature product having supported over 225 missions. Unlike STK, it is not modular-based and is provided as a single package at an initial network license cost of approximately USD 57,000 with annual maintenance costs of USD 10,000 [3].

FreeFlyer adopts a tiered approach with increasing functionality (each tier adding to the previous):

- Design tier is an analysis tool for mission concept development and preliminary design, performing tasks such as orbit design, ground station location and coverage times, and sensor modelling;
- Engineer tier offers comprehensive mission analysis and design functionality, with more complex modelling and analysis, and a scripting language which allows users to implement mission logic;
- Mission tier provides complete spacecraft mission design and operations functionality, with automation, orbit determination, and integration with any ground system or other third party software.

1.2.3 General Mission Analysis Tool (GMAT)

Developers: NASA; Others
Version: R2014a (July 2014)
Operating system: Windows; Linux; Mac OS
License: Apache License, Version 2.0

The General Mission Analysis Tool (GMAT) [5] is a platform independent, free and open-source, space trajectory optimization and mission analysis system. It is developed by NASA, private industry, academia, government agencies, and the open source community. GMAT is currently used for real-world engineering studies and as an educational and public engagement tool. It is also currently undergoing significant testing to prepare it for operational use.

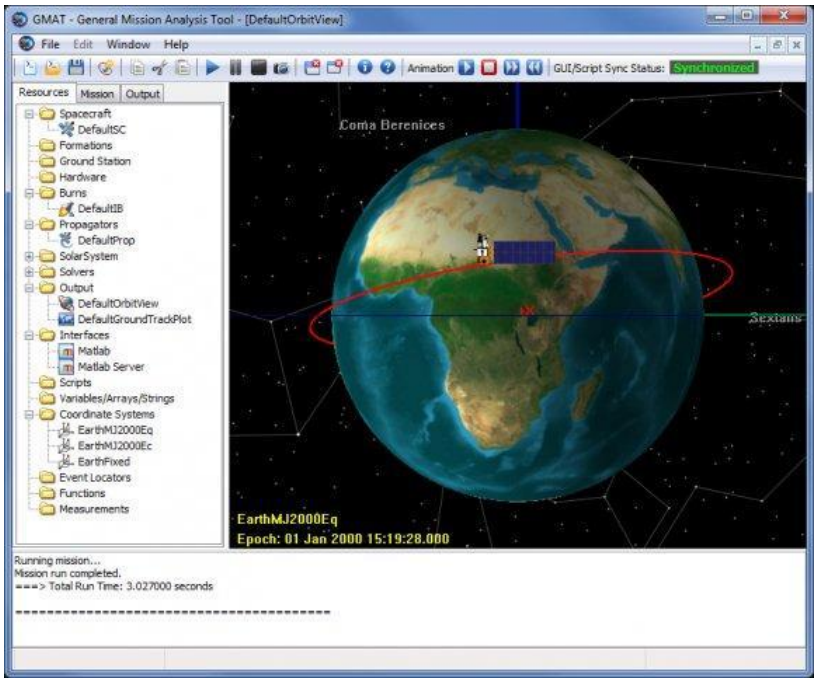


Figure 1.2 – The GMAT graphical user interface [6].

GMAT propagates and optimizes spacecraft trajectories in a variety of flight regimes ranging from low Earth orbit to deep space missions. The system is also able to produce fully interactive 3D graphics, customizable plots, and reports for data analysis.

Users create and configure resources, such as spacecraft, propagators, and optimizers, which are later used in the mission sequence to model the spacecraft motion. This can be accomplished by using either the graphical user interface (GUI) illustrated in Figure 1.2 or a scripting language. Scripts can be generated and viewed within the software or written by the user and loaded.

1.2.4 Space Trajectory Analysis (STA)

Developers: European Space Agency (ESA); Academia
Version: No longer available
Operating system: Windows; Linux; Mac OS
License: European Union Public Licence (EUPL) v.1.1

The Space Trajectory Analysis (STA) [7] was a project initiated in 2005 by the European Space Agency (ESA) in collaboration with universities and research institutions. Its aim was to develop a free and open-source astrodynamics software suite for space trajectory simulation and analysis. STA was developed in a modular fashion by undergraduate and postgraduate students with module integration and validation performed by ESA. The project was discontinued in the end of 2012.

STA adopted the same scenario-based interface used by STK (cf. section 1.2.1), as shown in Figure 1.3, and its three-dimensional rendering engine was imported from Celestia (cf. section 1.2.5).

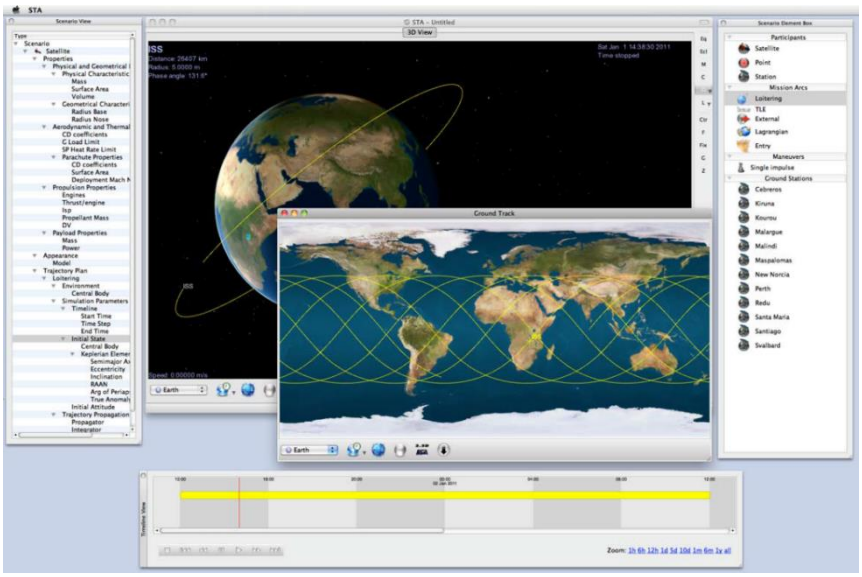


Figure 1.3 – The STA graphical user interface [7].

1.2.5 Celestia

Developers: Chris Laurel; Celestia Development Team

Version: 1.6.1 (10/06/2011)

Operating system: Windows; Linux; Mac OS

License: GNU General Public License version 2.0 (GPLv2)

Celestia [8] is a platform independent, free and open-source, three-dimensional visualization tool. It is mostly used by astronomy enthusiasts at home and educators in schools, museums, and planetariums. Lately, it has also been used in mission design for spaceflight visualization, with its rendering engine being integrated into STA (cf. section 1.2.4)

Celestia's primary goal is visual realism. It has the ability to load complex 3D models and simulate a variety of natural phenomena, namely eclipses, atmospheric scattering, and planetary surface reflectance. The trajectory and attitude of loaded objects can be defined in several ways, including via NASA's Navigation and Ancillary Information Facility (NAIF) SPICE kernels (cf. section 3.2.2). Despite its advanced rendering features, Celestia lacks data generation capabilities, severely limiting its functionality for the purpose of mission analysis and design.

1.3 Objectives

After reviewing the available software and identifying both their advantages and disadvantages, a series of objectives were defined. In order to have standalone mission analysis and design capabilities the software will include a trajectory propagator and a three-dimensional simulation module. The trajectory propagator module will be able to calculate the trajectory of a spacecraft within the Solar System and the simulation module will display it in a three-dimensional realistic environment. These modules are to be integrated, nevertheless, retaining the ability to be used on their own and in conjunction with existing industry tools. From the user perspective the software should be intuitive and easy to use.

Chapter 2

DEFINITION OF REQUIREMENTS

To achieve the objectives (cf. section 1.3) a series of requirements were defined. These take into account not only the mission analysis and design objectives but also usability, technical feasibility, and the available time for development.

2.1 General

The most basic requirement is that software should be usable by the widest possible audience. In programming terms, this means targeting as many operating systems and hardware configurations as feasible. The software package will therefore be designed for Microsoft Windows 7, since this is currently, and in the foreseeable future, the most used operation system worldwide [9]. Nevertheless, the core features should be compatible with older versions of the Windows operating system, ensuring basic functionality on legacy systems. The software will also be central processing unit (CPU) independent and delivered in binary form for 32 and 64 bit architectures. It is, however, desirable to include specific optimizations for the common modern CPUs, in order to enhance computational performance.

The spacecraft mission analysis and design process is extremely complex, and requires the usage of multiple software tools. To allow the integration of the software being developed in this process it is necessary to ensure its upstream and downstream compatibility, which hinges on many factors. Firstly, the definitions used in the software (e.g. systems of units and frames of reference) must match those used in the industry, and be well documented to avoid any possible confusion. Secondly, the software must use globally accepted constant values and ephemerides data. This implies using a widely used and unified data source – such as SPICE kernels (cf. section 3.2.2). Lastly, both the input and output of the software need either be industry-standard, or easily convertible to and from other formats.

A common problem found in general mission analysis and design software are their shallow learning curves, requiring months of experience to master features. Addressing this issue is one of the main design concerns, since the software is meant to be intuitive, allowing its use by anyone with minimal amount of training. To achieve this usability goal it is necessary to simplify the set up process of common tasks, minimize the number of controls without compromising functionality, validate all input, and document use cases. The simplification of the set up process can be accomplished by pre-filling dialog controls with common default values and options, and using simple understandable formats for input files, such as extensible markup language (XML) documents. Minimizing the number of controls requires dynamically altering the dialog's contents based on previous input, a technique commonly used in website forms, where selecting an option can show, or hide, a group of controls. Input validation should be done immediately, to prevent the propagation of errors, and an explanation must be provided

explaining what the problem is. Finally, the user manual must include step-by-step examples of how to perform common tasks, thus steepening the learning curve.

2.1.1 Software Development

Since this will be a large open-source software package, the source code must be standard-compliant, maintainable, and well documented, thus ensuring that anyone that needs or wants to modify the source code to suit their needs can understand it. The strategy to produce quality code is twofold. Firstly, the programming will strive to produce low-complexity functions and methods, which can be modified having little understanding of the codebase. Secondly, a detailed programming manual is to be auto-generated from the code structure and comments. Having an auto-generated programming manual is very effective since it can be compiled at the same time as the software, guaranteeing it is always up to date.

Possible future software development (i.e. the inclusion of new features) must also be taken into account while defining the architecture. The software needs to be able to accommodate completely new components that use the underlying base functionality. Therefore, the software will be designed in a modular interface-based fashion that allows a future developer to effortlessly integrate a new module in the package.

2.2 Trajectory Propagation

One of the components to be implemented is a trajectory propagator. This component, which has the sole purpose of data generation, should be able to calculate the trajectory of a single spacecraft within the Solar System.

In this context, trajectory propagation is usually done by integrating the equations that describe the motion of the spacecraft, starting from a set initial conditions. This approach requires the definition of the aforementioned equations, or force model, and the implementation of a numerical integrator. The force model must take into account the fundamental forces that influence the trajectory of a spacecraft travelling through the Solar system, and the effect of any engine the spacecraft may be equipped with. The numerical integrator needs to have adequate precision to keep the truncation error within acceptable limits, while having a reasonable time step.

Trajectory propagation is a computationally intensive task, meaning the propagator algorithm should be highly optimized and make use of all possible performance increase methods. To achieve this goal it is necessary to both carefully design the algorithms minimizing unnecessary computations and to implement them keeping in mind the CPU cost of the operations at play.

2.2.1 Force Model

The fundamental forces that influence the trajectory of a spacecraft travelling through the Solar System are very well known [10] and have been extensively studied. The challenge of the force model definition lies in selecting which of those forces are relevant for inclusion, and where, since it is infeasible to include them all. Several possible space mission scenarios, presented below, were studied.

Earth Satellite

This is by far the most common space mission scenario, consisting on launching a satellite that will orbit the Earth and perform trajectory corrections using short engine bursts. From the analysis of the various perturbations shown in Figure 2.1, it is concluded that:

- The gravity field of the Earth has the most significant contribution;
- The non-sphericity of the gravity field has a very important contribution, especially its zonal harmonic coefficient J_2;
- The Moon and the Sun have lesser contributions, though surpassing the J_2 perturbation for orbits higher than the Geosynchronous Equatorial Orbit (GEO);
- The atmospheric drag has a very high influence for very low orbits inside the thermosphere, and quickly decays with altitude.

The engine bursts that the spacecraft may possibly require for trajectory corrections can be reduced to instantaneous changes in its velocity vector, commonly known as delta-V.

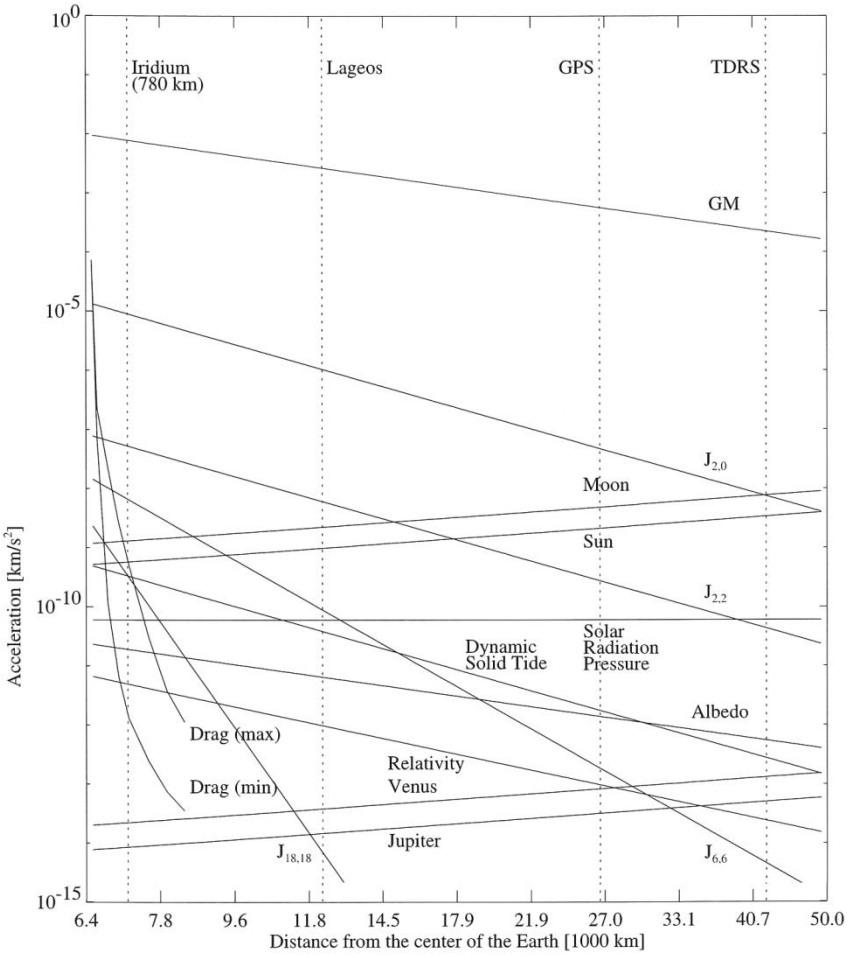


Figure 2.1 – Order of magnitude of various perturbations of a satellite orbiting Earth [10] (adapted).

Interplanetary Mission

Extrapolating Figure 2.1 for very high altitudes it can be inferred that the only forces of interest in interplanetary space are those related to the Sun – the gravity field and the solar radiation pressure. The solar radiation pressure is especially important for spacecraft with a very large area and a very low mass, such as solar sails. If, however, a spacecraft passes near a planetary system its gravity field is also worth taking into account. Interplanetary missions also feature a multitude of engines, namely constant acceleration engines, which are not ideally simulated by delta-V impulses.

Upon arrival at the destination planetary system it is necessary to start taking into account the planet and its moons, like in the Earth satellite case. The only difference to the Earth case is that the atmospheric density profile of other planets, with the exception of Mars, is not very well known.

Asteroid/Comet Rendezvous

Several past, current, and planned space missions (e.g. Deep Impact, Rosetta, Hayabusa) involve rendezvous with asteroids or comets. Although these mission have the same deep space profile as the interplanetary missions, they require taking into account their target's gravity field and ephemerides for the rendezvous. These are gravity fields which would be irrelevant, and never considered, in any other type of mission.

The immediate conclusion of studying these three mission scenarios is that different missions require different forces to be taken into account. Since this meant to be a general mission analysis and design software, a way of generalizing the problem needs to be developed. Firstly, due to only being relevant for low orbiting satellites, the atmospheric drag effect is not to be included at this stage. All the other forces described are to be included (i.e. central body's gravity field and J_2 perturbation, third bodies' gravity field, solar radiation pressure). The concept of force system, as a collection of bodies whose gravitational force is relevant in a region of space, needs to be implemented. The software needs to be capable of switching between the available force systems automatically, such as in the case of an interplanetary mission approaching its target planet. The force systems should be easy to edit in order to include another body, like an asteroid or comet. Lastly, an engine must be implemented with support for an unspecified number of delta-V and continuous acceleration inputs.

2.2.2 Numerical Integrator

The choice of the numerical integrator used for the integration of the equations of motion has to be made taking into account not only its precision, but also its computational performance. Additionally, due to the nature of the software, and the proposed force systems implementation, it is highly desirable that the numerical integrator has an adaptive step size. The adaptive step size will enable the propagator to accelerate when in interplanetary space where the forces remain fairly constant, and use a finer step when near celestial bodies, thus keeping the precision constant. Furthermore, an adaptive step size reduces the computation time and the size of the output files.

Following a brief analysis of commonly used numerical integrators for orbital mechanics problems [10], the Runge-Kutta-Fehlberg 7(8) (RFK7(8)) [11] was chosen. This integrator is adequate since it is mature, widely used, fast, and sufficiently accurate for the task. Furthermore, it has local truncation error evaluation and self-adjusts the step sized based on it. There are, however, instances where it is necessary to have added control of the step size, so the algorithm will be modified to include minimum, maximum, and fixed step size constraints.

2.2.3 Input and Output

The trajectory propagation requires the user to provide a set of initial conditions, and configure the force model and numerical integrator. To facilitate the task a GUI is to be developed, where the user can directly input values and set options. The GUI controls should also allow the user to input data in several different formats, such as providing either a state vector or the orbital elements for the spacecraft's initial conditions. Furthermore, the user should be able input values in different unit of the International System of Units (SI) and the International Astronomical Union (IAU) (2009) System of Astronomical Constants [12], where applicable. This prevents the user from having to perform unit conversions outside of the software, and possibly make a mistake.

Since a propagator setup can be modified as the mission definition changes, or may need to be shared between several people working on the project, a way of storing it for later usage is required. There is no pre-defined industry-standard for this task, so the XML format was chosen, as it is both human readable and machine friendly. Additionally, for Earth satellites it is common to use two-line element (TLE) sets to describe the initial conditions, so being able to use them as input is also desirable.

The propagator output needs to ensure maximum compatibility with third party tools. NAIF's SPK format (cf. section 3.2.2) is the de facto standard for planetary and spacecraft ephemerides, hence the software will use it for output. This format, however, is not easily convertible to others, so an additional plain-text output will also be generated.

2.3 Three-Dimensional Simulation

The second component to be implemented, the simulator module, is a three-dimensional realistic rendering of the Solar System. This component's primary purpose is spacecraft trajectory, position, and attitude visualization.

Most mission analysis and design tools that include 3D visualization features only provide very basic renderings of the Solar System bodies and spacecraft of interest. Although this is often enough for a preliminary trajectory analysis, it is insufficient to fully understand the scene. To ensure the success of a space mission it is necessary to know if a spacecraft's solar panels are illuminated, if its instruments and antennas are pointed correctly, if a feature being photographed is illuminated, if a lander is in view, if there are objects in its path that were not taken into account while propagating the trajectory, etc… For all these purposes it is necessary to have a rendering of the 3D environment which strives to match the reality.

2.3.1 Visual Realism

The main development goal for the simulator module is to maximize visual realism. In order to achieve this goal, it is necessary to:

- Generate or load the geometry for simple objects (i.e. ellipsoidal celestial bodies);
- Load 3D models for more complex objects, such as spacecraft, asteroids, and comets;
- Place the models on their correct positions and attitudes;
- Render the scene using physically-based shading techniques.

To generate the geometry for ellipsoidal celestial bodies a procedural generator will be developed. This generator must be able to generate the ellipsoidal shape of the body, including features such as atmosphere, clouds, and planetary rings. Since 3D models come in a variety of formats, and converting between them is not trivial and requires additional software, it is desirable to include a unified model loaded that supports a wide variety of formats.

In order to correctly place the generated or loaded objects it is necessary to have their ephemerides and orientation data. Similarly to the trajectory propagator, the data is provided by SPICE kernels. Additionally, the ability to load custom SPK kernels, such as the ones generated by the propagator model will also be implemented, thus ensuring full compatibility between the propagator output and the simulator input.

Physically-based shading techniques are algorithms that take into account the physical properties of the materials and the lighting in order to render images that resemble reality. The downside of these algorithms is that they are computationally intensive and cannot always be rendered in real time. It is necessary to study the algorithms, access their performance, and decide what needs to be simplified in order to provide smooth real-time rendering of the highest possible quality.

2.3.2 Auxiliary Features and Simulation Control

Besides rendering a realistic scene, it is also necessary to include auxiliary features for scientific purposes. Firstly, to analyse the evolution of the position of a spacecraft as it travels through the Solar System it is necessary to draw its trajectory. Secondly, to monitor its attitude it is necessary to render local frames of reference (cf. Appendix A). Lastly, to know the position of distant objects whose apparent size is too small to discern, markers need to be placed, along with labels with the object's name.

The module must implement methods to control the simulation. A user must be able to load kernels containing ephemerides and orientation information for objects of interest, and load 3D models as desired. Additionally, for each object, it is necessary to have the ability to configure and toggle the display of auxiliary features. Time control is also necessary for controlling the simulation, namely the ability to jump to a specific date, and to slow down or fast forward the simulation as desired. These features should be controlled via a non-intrusive GUI which can be hidden when not in use.

2.3.3 Cameras and Movie Making

Controlling the point-of-view of the simulation (i.e. the camera position and orientation) is essential to evaluate all aspects of a scene. Often it is necessary to have a camera near a spacecraft, in other occasions it is ideal to have it on the surface of a planet, or even the top or an orbital plane. To allow the user full control over the camera, two control modes are to be implemented – manual and automatic.

The manual control provides a first-person keyboard and mouse controlled camera, which allows the user to freely explore the scene. This camera is ideally suited to quickly navigate to between objects and evaluate their status.

The automatic mode is a “must have” for presentations and directing high-quality movies. In this mode the point-of-view is controlled automatically throughout the simulation, based on a list of parametrized cameras. In order to fully control the point-of-view it is necessary to specify at least the camera position and look at points. Additionally, if a user desires to rotate the camera, the direction of the camera “up” vector also needs to be specified. Since this control mode is intended to be fully automatic, it is also desirable to be able to specify the timescale at which the camera operated, a task that would normally be done via the GUI.

As with any other software tool, it is necessary to have a way of “saving” the work done. Since the simulator module only renders the scene on the screen, the only possible output is a movie. Hence, the module needs to be able to output movies captured from the rendering surface. The GUI is the only part that should be omitted from the movie as it will serve no purpose.

To avoid having to post-process the movie, for example to reduce its size for internet distribution, it is desirable to have the ability to control its resolution, and bitrate. Finally, the exported movies should be in a format which can be natively played on the operating systems supported by the software without requiring the user to install third-party codecs or players.

Chapter 3

SOFTWARE DESIGN

The software was designed using the waterfall model [13] which proposes a sequential development process as shown in Figure 3.1. The first step is the requirements specification; these describe the behaviour of the program and its user interactions. The design stage is where the architecture of the program is defined. The coding stage refers to the actual development of the software. The testing stage, found in Chapter 6, is where the code is visually inspected for errors and all its logical paths are tested. Additionally, product assurance techniques are used to ensure the requirements are met. Finally, after final software acceptance, the operations stage is reached, where maintenance and future updates are performed.

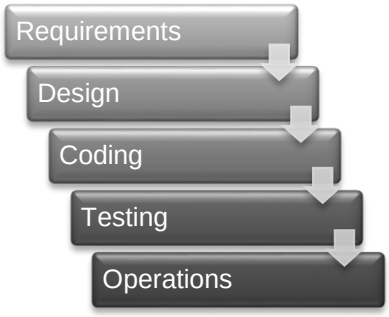


Figure 3.1 – Waterfall model. Implementation steps to develop the software.

Following the definition of the requirements in Chapter 2, this chapter is dedicated to the software design, including the choice of programing language and third party components, and the definition of the architecture. The documentation and the licensing and distribution strategy for the software are also defined.

The software was named SimOrbit, since it is, in its essence, an orbital simulator, and SimOrbit is not a registered trademark according to TMview. The adopted logo, shown in Figure 3.2, consists of the text “SimOrbit” written in Titilllium Web Bold Italic font in white over a black background.

Figure 3.2 – The SimOrbit logo.

3.1 Programming Language and Development Environment

The chosen programming language was C++ as defined in its most recent standard from the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), ISO/IEC 14882:2011 [14]. This choice took into account several factors, including the stability and maturity of the programming language, familiarity with the standard, and availability of third party components.

In order to facilitate the programming, deliver a more feature-rich product, and for compliance with industry standards the software needs to use several third party components. Most third party components do not limit the programming language choice, since they either have bindings for multiple languages or there are suitable alternatives which do. However, NAIF's SPICE Toolkit (cf. section 3.2.2), which is the source of ephemerides and planetary constants, is only available for FORTRAN, C, IDL, and MATLAB, thus limiting the programming language choice. Out of these variants the easiest to integrate with the remaining third party components was the C version, requiring only the development of a C++ interface library.

Microsoft Visual Studio 2013 was the selected integrated development environment (IDE). The selection was based on previous experience using this and past versions of the IDE and the fact that it is available free of charge to Instituto Superior Técnico (IST) faculty and students via the Microsoft DreamSpark program[2].

The software was developed under Apache Subversion (SVN) revision control. The SVN server is located in Roubaix, France, with an off-site backup.

3.2 Third Party Components

Several third party components were integrated in the software in order to increase its functionality and reduce the development time:

- A graphics application programming interface (API) was required to render the GUI and the three-dimensional environment;
- A source of spacecraft and planetary data was needed for accurate propagation of spacecraft trajectories and determination of the position and orientation of the objects to render;
- A high performance multi-precision library was useful to both accelerate and provide a finer control over time vs. precision in calculations;
- A three-dimensional model loader was necessary to greatly expand the variety formats the software can load without increasing the development time.

In the following sections the choices of third party components for the listed tasks are discussed along with a brief overview of the component.

[2] Formerly MSDN Academic Alliance (MSDNAA).

3.2.1 Graphics API (Direct3D)

Two graphics APIs were considered – Microsoft Direct3D and OpenGL – which are very similar in terms of features, performance, and ease of use.

Direct3D is designed for, and exclusively compatible, with the Windows operating system. The development headers and libraries are provided either as part of the Microsoft DirectX Software Development Toolkit (SDK) or more recently the Microsoft Windows SDK, along with several other media-related APIs. Also part of the DirectX SDK is the DirectX Utility Toolkit (DXUT), which allows for the fast development of a Direct3D application. DXUT is able to manage window and device creation and maintenance, and features callback driven event processing. It also implements two types of cameras – first-person and arc-ball – both controlled via user input. Finally, it provides a basic GUI implementation featuring a dialog class, and several types of controls. DXUT is open-source and free to modify.

OpenGL is a cross-platform graphics API, originally developed by Silicon Graphics, Inc. (SGI). The OpenGL 1.1 development headers and libraries are included in the Windows SDK. Using more recent features requires the manual retrieval of the function addresses from the dynamic link libraries (DLL) or the usage of an extension library such as the OpenGL Easy Extension library (Glee) or the OpenGL Extension Wrangler library (GLEW). OpenGL also has a utility toolkit – GLUT – however it is no longer maintained. There are alternatives, such as FreeGLUT, but none providing a GUI implementation similar to that of DXUT.

Direct3D was the selected graphics API, mostly due to the DXUT GUI implementation which greatly reduces development time, and also due to previous experience using it. Additionally, DXUT simplifies the development of an application supporting both Direct3D 9 and Direct3D 10, allowing the software to be backwards compatible with Windows XP. The choice has some drawbacks, however. Direct3D is not cross-platform, meaning a future port of the software to another operating system would require a complete rewrite of the rendering engine. Also, Direct3D uses a left-handed coordinate system, requiring a coordinate transformation before rendering.

3.2.2 SPICE Toolkit

NAIF built an information system named SPICE [15] to assist scientists and engineers from the mission concept development to the post-mission data analysis phase. The SPICE system includes the SPICE Toolkit which provides library of APIs to read/write data files and calculate observation geometry parameters. The toolkit was originally implemented in ANSI FORTRAN 77, as defined by the American National Standards Institute (ANSI), but is now also available in ANSI C, IDL and MATLAB.

The primary SPICE data sets are called "kernels", whose types are shown in Table 3.1. These are composed of navigation and other ancillary information, and are produced by the most knowledgeable sources of information. NAIF also serves as the "Navigation Node" of NASA Planetary Data System, archiving and providing public access to the SPICE data sets.

The SPICE Toolkit APIs are very helpful in developing the software, since spacecraft and planetary ephemerides, planetary constants, and orientation data are required to both propagate spacecraft trajectories and render a realistic simulation of the Solar System. Furthermore, the usage of SPICE kernels as the source of information provides an easy means to keep the software accurate and up-to-date. The ANSI C version N65 of the SPICE Toolkit (July 23, 2014) was integrated in the software, via a C++ interface.

Table 3.1 – SPICE kernel types and their respective contents (in red the origin of the SPICE acronym).

Type	Contents
SPK	Spacecraft and planet ephemeris.
PCK	Planetary constants for natural bodies.
IK	Instrument descriptive data.
CK	Pointing (“C-matrix”). Orientation angles for a spacecraft bus.
EK	Events summarizing mission activities.
FK	Frame of reference frame specifications.
SCLK	Spacecraft clock correlation data.
LSK	Leap seconds data.
DSK	Digital shape model (under development).
MK	Mechanism for aggregating and loading a collection of kernels.

3.2.3 Multiple-precision Libraries and Interface (MPFR, MPFRC++, and MPIR)

The trajectory propagator is computationally intensive, and its accuracy depends on the bit precision of the underlying floating-point number formats used. Therefore, it is desirable to incorporate reliable libraries to speed up the computations and allow control over the bit precision.

The MPFR library [16] is a C library for multiple-precision floating-point computations with correct rounding. It is developed mainly by researchers from the Laboratoire Lorrain de Recherche en Informatique et ses Applications (Loria) and the Laboratoire de l'Informatique du Parallélisme (LIP), and had been continuously supported by the Institut National de Recherche en Informatique et en Automatique (Inria). MPFR is based and depends on the GNU Multiple Precision Arithmetic Library (GMP), and extends the Institute of Electrical and Electronics Engineers (IEEE) 754 standard for floating-point arithmetic [17], providing correct rounding and exceptions. The library is portable, efficient, and has well defined semantics.

However, MPFR lacks a C++ interface, resulting in extensive code length, lengthy development, and error susceptibility, to program complex operations. MPFR C++ [18] is an interface class for MPFR, written by Pavel Holoborodko, which aims to solve this issue. Among other features, MPFR C++ allows usage of human-friendly notation for MPFR mathematical expressions, and implements MPFR conversion operators for built-in types.

The GMP library, which MPFR depends on, has no support for using native development tools in a Windows environment. The Multiple Precision Integers and Rationals (MPIR) library [19] is a Windows friendly fork from the GMP project, supporting the Visual Studio development environment. MPIR has a fully compatible interface with GMP, and can be used as a drop-in replacement for it. Like GMP, MPIR is portable, written in C, and provides arbitrary precision arithmetic on integers, rational, and floating-

point numbers. It aims to provide the fastest possible arithmetic, including optimized assembly code for a variety of CPUs.

The MPFR version 3.1.2 (March 13, 2013) and the MPIR version 2.6.0 (November 8, 2012) libraries are dynamically linked in the software and the MPFR C++ version 3.5.9 (July 15, 2014), modified for wide character support, was incorporated within the source code.

3.2.4 Open Asset Import Library (Assimp)

In order to improve the visual realism, the three-dimensional visualization tool relies on models for the spacecraft. These models are usually created by the designers and may come in a variety of 3D formats. Therefore, a 3D model loader was desired to ensure compatibility with multiple formats without lengthening the development process.

The Open Asset Import Library (Assimp) [20] is a portable open source library, written in C+, to import many well-known 3D model formats in a uniform method. Assimp loads the different 3D models into an identical data structure for further processing and rendering. Additionally, it includes a set of post-processing tools to perform a variety of tasks, such as mesh optimization, normal vector generation, and tangent space calculation.

Assimp version 3.1.1 (June 14, 2014) was dynamically linked in the software.

3.3 Architecture

The software is comprises a main application – SimOrbit – and two dynamic link libraries (DLL) – SimOrbit Library (SOLib) and SPICE C++. The complete high level layer diagram can be found on Appendix B, and a simplified version is shown in Figure 3.3.

SimOrbit is a modular application controlled by the Module Manager, and includes three modules – Main Menu, Propagator, and Simulator. The Main Menu allows the user to navigate through all available modules and view general information, such as general help and author information. The Propagator module is able to calculate the trajectory of a spacecraft. It uses a combination of XML and GUI input to set up the initial spacecraft conditions, available force systems, and engine parameters. A numerical integrator is then used in conjunction with a force model to generate the output. The propagator relies on the high performance multiple-precision libraries and interfaces, through the incorporation of the MPFR C++ interface and the dynamic linking of the MPFR and MPIR libraries (cf. section 3.2.3). The Simulator is three-dimensional Solar System renderer and trajectory visualizer, which uses procedural generators to create the Solar System and a three-dimensional model loader to load spacecraft models. The model loader is powered by Assimp, which is dynamically linked (cf. section 3.2.4). The simulator is able to load and display spacecraft trajectories using two types of cameras – first-person and scripted. The simulations can also be recorded in movies.

The SimOrbit Library is part of a framework which harmonizes and simplifies the development of modules. The framework is comprises a set of headers and the library itself. The headers define and

implement data types, validation policies, GUI controls, and callback methods. The library provides common DirectX interfaces for textures, materials, and effects, supporting both DirectX 9 and DirectX 10, and a Windows Media Video (WMV) movie maker.

The SPICE C++ is a C++ wrapper for the SPICE Toolkit (cf. section 3.2.2) which takes advantage of C++ features, such as function overloading, exceptions, vectors and strings to interface with the SPICE routines. Although it does not fully implement the entire SPICE library routines it includes the most used ones, namely, kernel management, object code translation, date and time conversion, constants and ephemerides retrieval, frame and conics transformation, coverage intervals determination, TLE interpretation, and SPK writing. The SPICE Toolkit is statically linked in the SPICE C++ library.

Lastly, both SimOrbit and the SimOrbit Library depend on the DXUT and DirectX, which are statically linked.

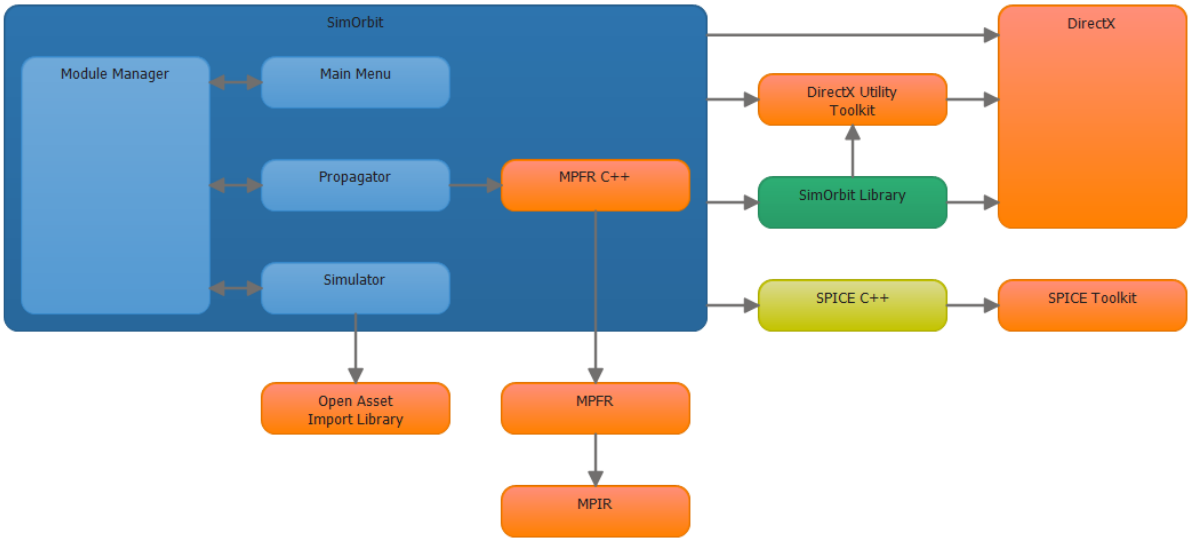


Figure 3.3 – Very high level layer diagram of the software architecture showing the relations between components (third party components displayed in orange).

3.4 Coding Standards

Producing high quality C++ code requires the use of coding standards which restrict the use of the ISO C++ in order to improve software reliability and maintenance. There are several C++ coding standards publically available, most of them being industry-specific. The High Integrity C++ (HIC++) Coding Standard Version 4.0 [21], as created by the PRQA, was chosen for this project since it is a mature standard, with over ten years of development, widely accepted, and not industry-specific. It defines a set of 155 rules and best practices for the production of high quality C++ code.

The changes and additions to third-party components (i.e. MPFR C++ and DirectX Utility Toolkit) follow the coding practices of the original source code.

Static code analysis was performed automatically using the Visual Studio 2013 C/C++ Code Analysis tool. The HIC++ rules compliance, however, was checked manually due to the very high cost of the automated tools supporting it.

3.5 Documentation

The C++ code was documented using XML documentation comments with the help of Atomineer Pro Documentation for Visual Studio [22]. The comments present in file, class, and method headers detail their purpose, input and output, possible exceptions thrown, and academic source where applicable. Additional comments are present throughout key algorithms explaining their flow.

The XML documentation comments served as input in the creation of the reference manual. Doxygen [23] was the selected tool for this task since it allows for the generation of both Portable Document Format (PDF) and HyperText Markup Language (HTML) versions of the reference manual, the latter to be integrated in the website (cf. section 3.6). Additionally, Doxygen is also responsible for the creation of the include dependency graphs and class inheritance diagrams.

The software uses several XML files for input and output, such as storage of propagator parameters, procedural planet generator instructions, and scripted camera actions. Each XML file structure is described in a XML Schema Definition (XSD) file, which is annotated for the purpose of documentation generation. The FiForms Solutions xs3p XSD documentation generator [24] was used since Doxygen does not provide support for XSD documentation.

Finally, a user manual was written, describing the general operation of the software. The user manual also includes the definitions used throughout the software, such as frames of reference and units of measurement.

3.6 Licensing and Distribution

3.6.1 Licensing

Since SimOrbit is a free and open-source software it was necessary to select a license that ensured the original authors retained full ownership, and that any derivative works would be distributed in a similar fashion. Furthermore it was necessary to review the third party component licenses in order to respect the upstream compatibility.

Several FOSS licenses were considered, namely the GNU Public License v3 (GPLv3) [25] and the European Union Public Licence (EUPL) v.1.1 [26]. Both licenses are very similar in all aspects, granting the same rights and obligations, barring a few exceptions which are not relevant in the context of this software. The EUPL was created by the European Commission, has 15 articles and 22 linguistic versions of identical value. The GPLv3 was created by the Free Software Foundation, has 17 articles and is only provided in English. The EUPL has advantages over the GPLv3. Firstly, it ensures interoperability with a downstream compatibility list, whereas the GPLv3 requires derivative work to also

be licensed under the GPLv3. Additionally, the EUPL explicitly designates the applicable law and court as the “law of the European Union country of the Licensor” and “where the Licensor resides”, respectively. The GPLv3 does not explicitly designate either applicable law or court.

Upstream Compatibility Analysis

The software must respect the licenses of the components in relies upon, hence, an upstream compatibility analysis must be performed. The European Commission provides a compatibility matrix between Open Source Initiative (OSI) approved licenses and the EUPL, the relevant part of which is shown Table 3.2.

Table 3.2 – Compatibility Matrix between component’s OSI-approved licenses and the EUPL [27] (adapted).

License of existing FOSS Component	FOSS Component(s)	Distribution of the larger application under the EUPL		
		Incorporation	Static link	Dynamic link
BSD license (all versions)	Assimp	OK	OK	OK
GNU GPL v3.0	MPFR C++	NO (exceptions exist)	NO	OK
GNU LGPL v3.0	MPIR MPFR	OK (object)	OK	OK

The MPFR C++ interface presents a licensing problem – since it is a header the only choice is incorporation, which is prohibited under the terms of the GNU GPL v3.0. However, it is still possible to incorporate the component if the original author includes the EUPL in a FOSS exception list. Therefore the original author was contacted, and an agreement was reached in order to allow the incorporation of MPFR C++ into SimOrbit under the terms of a FOSS exception list.

NAIF provides a set of “Rules Regarding Use of SPICE”, which were analysed for compatibility. Firstly, the acknowledgement of use of SPICE is encouraged and a reference is provided for doing so. The redistribution of the complete Toolkit is prohibited, but the inclusion of the library modules “is entirely appropriate”. The redistribution of unmodified SPICE kernels distributed by NAIF, and the creation of user kernels using appropriate methods, is permitted. Hence, there are no obstacles preventing the static linking of the SPICE Toolkit and redistribution of unmodified SPICE kernels with SimOrbit.

The DirectX SDK is licensed under the terms of an End-User License Agreement (EULA) which permits the distribution of the DirectX Runtime and the modification, copy, and distribution of the source and object code of the DXUT, among others.

3.6.2 Distribution

The software and source code will be distributed via a website under the domain “simorbit.eu”. The website also details the software’s features, along with screenshots, and provides a means to contact the authors. The server-side scripting language is PHP and the output markup is valid HTML5 and Cascading Style Sheets (CSS) 3, ensuring compatibility with all modern browsers.

Figure 3.4 – The SimOrbit website home page.

Chapter 4

SOFTWARE OVERVIEW

4.1 General

The software is designed in a modular architecture so that new features can be easily added. Initially, it ships with two modules – a propagator and a simulator. The propagator module is a spacecraft trajectory propagator based on a numerical integrator, and the simulator module is a 3D realistic rendering of the Solar System with support for the visualization and tracking of spacecraft trajectories.

The main menu, shown in Figure 4.1, is the entry point of the software. The dialog allows the users to access the loaded modules, the user guide, the settings dialog, and general software information.

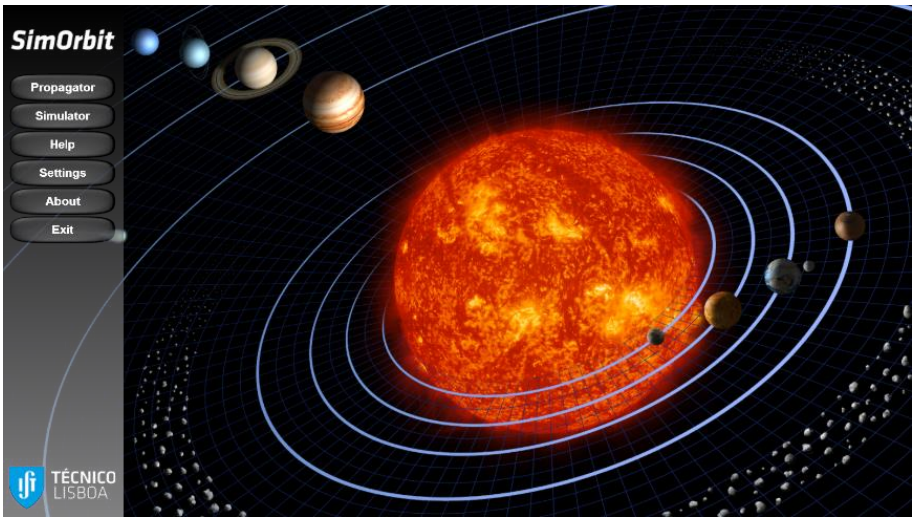


Figure 4.1 – Screenshot of the SimOrbit main menu.

One of the main design concerns regarding the software was making it very simple and intuitive, ensuring it can be used with minimum amount of training. For this purpose, care was given towards reducing the number of available user controls to the bare minimum, without loss of functionality, and also towards the logical placement of tabs and controls within dialogs.

Data input is made through a combination of XML files and GUI controls. The software validates all input against the set of policies, as defined in Appendix C. These policies detail the valid range and units of a value, their defaults, and the error messages to display should validation fail. Additionally the structure of the XML files used for input and output is formally described in their respective XSD files. The XSDs were written using the venetian blinds design [28], a design intended to maximize reuse, exceptionally well suited to the validation policies.

In order to provide backwards compatibility with older operating systems and graphics cards, the software has a dual rendering system supporting both Direct3D 9 and Direct3D 10.

SPICE kernels were chosen as the main data source for constants and ephemerides, and several kernels, listed on Table 4.1, are shipped with the software to ensure adequate functionality. Nevertheless, the user can load additional kernels via the kernels.furnsh file. This file is a meta-kernel whose structure is explained on the SPICE Toolkit documentation.

Table 4.1 – SPICE kernels that the software loads by default, listed in ascending priority, and their usage.

Type	Filename	Usage
LSK	naif0011.tls	Leap seconds information.
PCK	geophysical.ker	NORAD two-line elements constants.
	pck00010.tpc	Orientation data for natural bodies.
	Gravity.tpc	J_2 values.
	gm_de431.tpc	"GM" values.
SPK	de430.bsp	Planetary ephemerides.
	mar097.bsp	Mars system ephemerides.
	jup310.bsp	Jupiter system ephemerides.
	sat375.bsp	Saturn system ephemerides.
	ura111.bsp	Uranus system ephemerides.
	nep081.bsp	Neptune system ephemerides.
	plu043.bsp	Pluto system ephemerides.

4.2 Propagator Module

The propagator module is responsible for generating a spacecraft trajectory from its initial conditions. It takes into account the major forces at play in the Solar System and the spacecraft's engine events.

4.2.1 Graphical User Interface

The propagator GUI consists of a tabbed dialog, divided in general, force model, integrator, and output sections, and two pop-up dialogs for configuring force systems, and adding and editing engine inputs. Figure 4.2 shows the general tab of the propagator GUI, filled with valid values.

The general tab is where the initial conditions of the propagation are set. The user needs to input the initial state, either via state vector or classical orbital elements, and specify the propagation stop date. The force model tab is where the physical properties of the spacecraft (mass, area, and reflectivity) and its engine inputs are set. The engine inputs are added and edited via pop-up dialog and support both delta-V (i.e. instant velocity change) and constant acceleration events. The force model tab also lists the available force systems, and allows the user to configure them via pop-up dialog. The integrator tab is used to configure the integrator time step, its position and velocity truncation error tolerances, and the bit precision of the computation. Finally, the output tab allows the user to select and configure the output files the propagator generates.

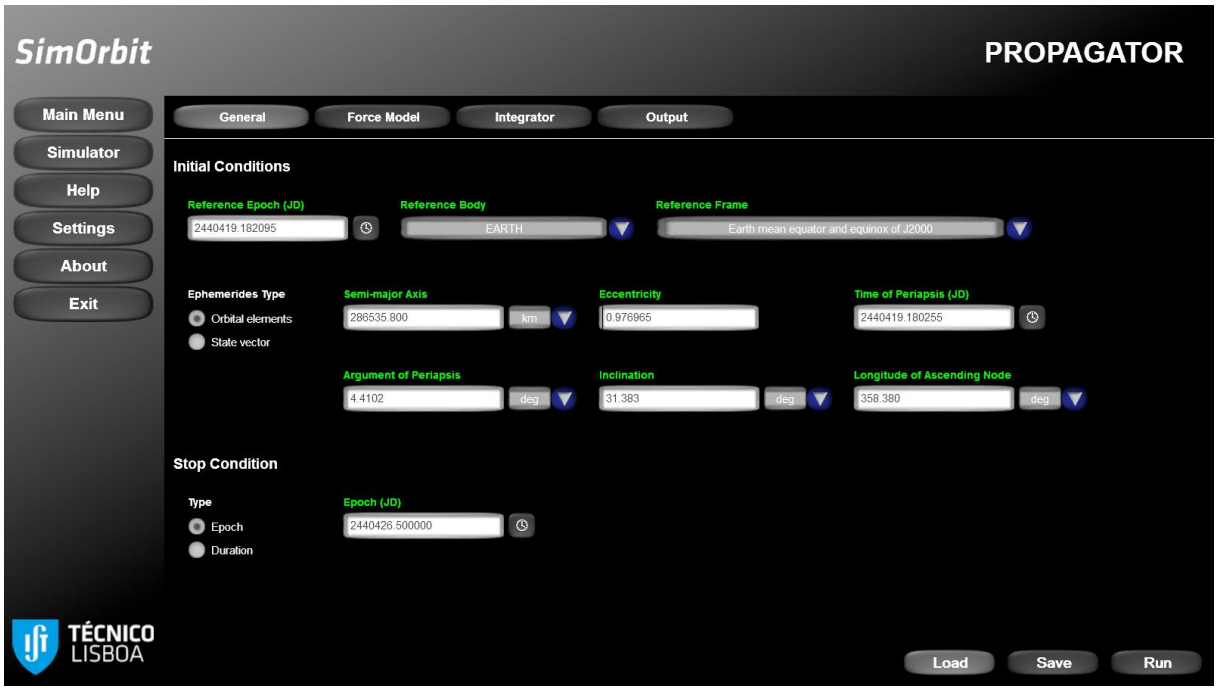


Figure 4.2 – Screenshot of the propagator module general tab filled with valid values.

The propagator GUI validates all input values and flags input errors as soon as the affected control loses focus. Hovering the control with the mouse displays a tooltip indicating what the error is, as shown in Figure 4.3.

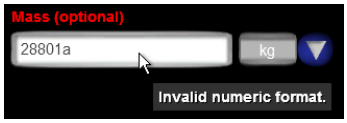


Figure 4.3 – Example of a tooltip indicating an error on the spacecraft mass control.

After the user fills all the required input and presses the “Run” button, the propagator progress dialog is exhibited. This dialog, shown in Figure 4.5, displays the current progress of the task, both in percentage, and via a progress bar, the abbreviated current information, and the event log. The current information lists the active force system, current Julian date of the propagator, and the integrator time step. The event log displays the messages generated by the propagator that may be of relevance to the user. Among other events, the messages inform the user of:

- Propagator start, pause, resume, stop, and end;
- Segment start and end;
- Force system switch;
- Application of an engine delta-V input, and start/stop of a constant acceleration input;
- Spacecraft crash.

The progress dialog also features buttons for the user to pause/resume or stop the operation of the propagator.

In addition to the progress dialog, the SimOrbit taskbar icon reflects the status of the operation, emulating a colour coded progress bar, as shown in Figure 4.4. This allows the user to work on other applications while still monitoring the status of the SimOrbit propagator.

Figure 4.4 – SimOrbit taskbar icon status during propagator operation. From left to right – running, paused, stopped.

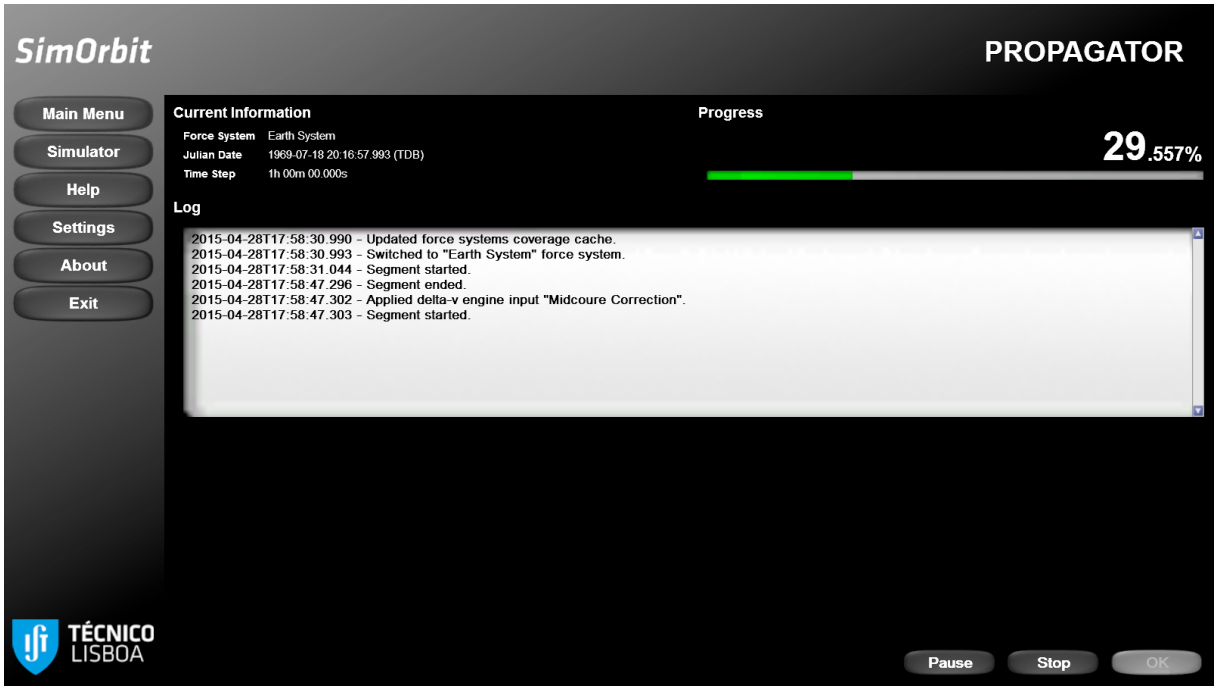


Figure 4.5 – Propagator progress dialog.

4.2.2 Force Model

The force model components were modelled in accordance with the conclusions drawn from the study presented in section 2.2.1. The mathematical formulae used for the implementation, for an arbitrary spacecraft under the main influence of an arbitrary central body, are presented in this section.

Assuming the position of a spacecraft relative to a central body in the J2000 frame is

$$\boldsymbol{r} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \tag{4.1}$$

then its acceleration $\ddot{\boldsymbol{r}}$ due to the presence of the central body point mass M is given by the gravitational law

$$\ddot{\boldsymbol{r}} = -\frac{GM \cdot \boldsymbol{r}}{|\boldsymbol{r}|^3}, \tag{4.2}$$

where G represents the gravitational constant.

In order to build a more accurate force model, it is necessary to take into account the actual mass distribution of the central body, which causes the gravity field to be non-spherical. Modelling this perturbation requires the expansion of the gravity potential in a series of Legendre polynomials. For the purpose of this software it is sufficient to only take into account the zonal harmonic coefficient J_2, resulting in a perturbation, in the body-fixed frame of the main body, given by

$$\ddot{\boldsymbol{r}}_{\boldsymbol{BODY}} = -\frac{3 \cdot GM \cdot J_2 \cdot {R_E}^2 \cdot \boldsymbol{r}_{\boldsymbol{BODY}}}{2 \cdot |\boldsymbol{r}_{\boldsymbol{BODY}}|^5} \circ \begin{bmatrix} 1 - \frac{5 \cdot z_{BODY}^2}{|\boldsymbol{r}_{\boldsymbol{BODY}}|^2} \\ 1 - \frac{5 \cdot z_{BODY}^2}{|\boldsymbol{r}_{\boldsymbol{BODY}}|^2} \\ 3 - \frac{5 \cdot z_{BODY}^2}{|\boldsymbol{r}_{\boldsymbol{BODY}}|^2} \end{bmatrix}, \tag{4.3}$$

where $\boldsymbol{r}_{\boldsymbol{BODY}}$ is the position of the spacecraft relative to the main body in the body-fixed frame, R_E is the equatorial radius of the main body, and $\circ$ denotes the Hadamard product. Converting the perturbation to the J2000 frame yields

$$\ddot{\boldsymbol{r}} = U^T(t) \cdot \ddot{\boldsymbol{r}}_{\boldsymbol{BODY}}, \tag{4.4}$$

with $U(t)$ being the time-dependent matrix that describes the planet's rotation in the J2000 frame.

The trajectory of a spacecraft is also influenced by third bodies such as moons, the sun, or other planets. The perturbation due to a third body, represented by the point mass M_s is given, in the central body frame of reference, by

$$\ddot{\boldsymbol{r}} = GM_s \cdot \left(\frac{\boldsymbol{r}_{\boldsymbol{s}} - \boldsymbol{r}}{|\boldsymbol{r}_{\boldsymbol{s}} - \boldsymbol{r}|^3} - \frac{\boldsymbol{r}_{\boldsymbol{s}}}{|\boldsymbol{r}_{\boldsymbol{s}}|^3} \right), \tag{4.5}$$

where $\boldsymbol{r}_{\boldsymbol{s}}$ is the position of M_s relative to the central body.

The trajectories of spacecraft travelling through the solar system are also influenced by the solar radiation pressure. The perturbation can be described by

$$\ddot{\boldsymbol{r}} = -\nu P_{\odot} \quad C_R \frac{A}{m} \frac{\boldsymbol{r} - \boldsymbol{r}_{\odot}}{\left| \boldsymbol{r} - \boldsymbol{r}_{\odot} \right|^3} AU^2, \tag{4.6}$$

where:

- $P_{\odot} = 4.560 \times 10^{-6} N \cdot m^{-1}$ is the solar radiation pressure in the vicinity of the Earth (i.e. at a distance of $1AU$ from the Sun);
- C_R is the radiation pressure coefficient;
- A is the cross-sectional area of the spacecraft perpendicular to $\boldsymbol{r} - \boldsymbol{r}_{\odot}$;
- m is the mass of the spacecraft;
- $\boldsymbol{r}_{\odot}$ is the position of the Sun relative to the central body;
- ν represents the fraction of sunlight received by the spacecraft.

The fraction of sunlight received by the spacecraft is determined using the conical shadow model illustrated on Figure 4.6, such that:

- $\nu = 0$ if the spacecraft is in umbra;
- $\nu = 1$ if the spacecraft is in sunlight;
- $0 < \nu < 1$ if the spacecraft is in penumbra.

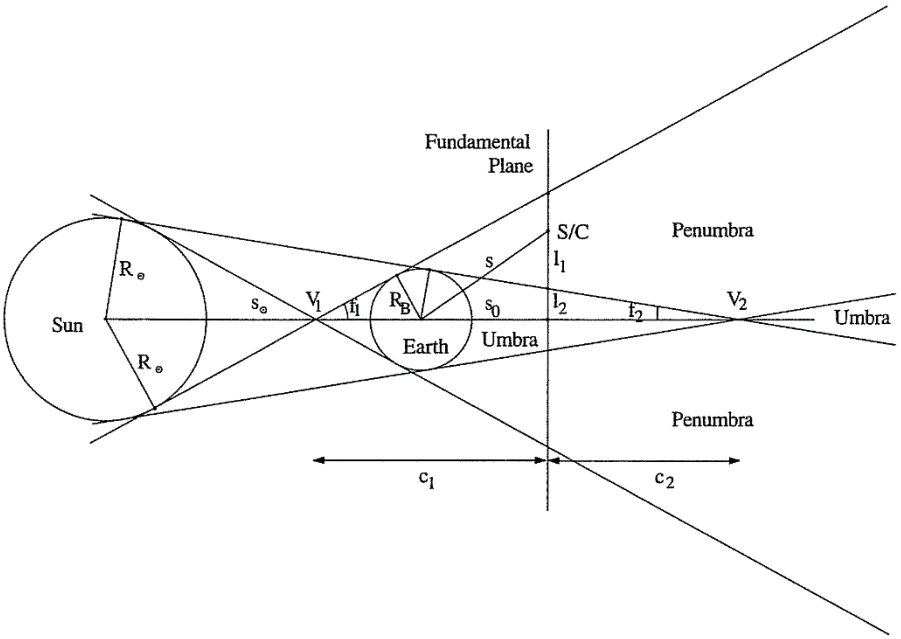


Figure 4.6 – Conical shadow model [10].

The umbra region, to the right of V_2, also called antumbra, was not considered since V_2 is outside the sphere of influence of any planet on the Solar System, thus whenever the spacecraft enters this region shadows are no longer being considered.

The radiation pressure coefficient is obtained from the reflectivity ε using

$$C_R = 1 + \varepsilon. \tag{4.7}$$

The acceleration experienced by a spacecraft due to the fundamental forces is the sum of Equations (4.2), (4.4), (4.5), and (4.6).

The propagator has a list of force system definitions which can be edited by the user, either via the GUI or a XML input file, that list which forces are to be considered on each system. Each system is valid within the sphere of influence of the central body and for the coverage interval of the ephemerides of all its components. A force system is comprises a central body, a list of third bodies, and flags to indicate if the perturbations due to the central body J_2 coefficient, the Sun as a third body, and the solar radiation pressure, should be included in the calculations. By default all options are enabled, with the exception of third bodies whose

$$M_S \leq 2 \times 10^{-9} M. \tag{4.8}$$

This limit was obtained by analysing all planetary systems to make sure the major moons were included, and the small moons and rocks excluded. The transition between force systems is done automatically using the algorithm found in section 5.1.1.

4.2.3 Engine

Two types of engine inputs are implemented – delta-V and constant acceleration over time. The delta-V engine input is an instantaneous change of the velocity of the spacecraft in a specific direction. It is represented in the equations of motion by

$$\dot{\boldsymbol{r}} = \begin{cases} E(t) \cdot \boldsymbol{v} & , t = T \\ 0 & , t \neq T \end{cases}, \tag{4.9}$$

where $\boldsymbol{v}$ is the engine input velocity vector, T is the time of the delta-V and $E(t)$ the time-dependent matrix that transforms coordinates in the engine frame to the J2000 frame. The constant acceleration simulates a thrust force applied over an interval of time in a time-dependent direction, and is represented by

$$\ddot{\boldsymbol{r}} = \begin{cases} 0 & , t < T_1 \\ E(t) \cdot \boldsymbol{a} & , T_1 \leq t < T_2 \\ 0 & , t \geq T_2 \end{cases}, \tag{4.10}$$

where $\boldsymbol{a}$ is engine input acceleration vector, T_1 is the thrust start time, and T_2 the end time. Since the software does not propagate the mass of the spacecraft and assumes it is constant, then the constant acceleration is equivalent to a constant thrust.

The direction of the engine inputs can be given as a 3-component direction vector or an azimuth and elevation, both relative to any implemented frame of reference (cf. Appendix A).

4.2.4 Numerical Integrator

The chosen numerical integration method is the Runge-Kutta-Fehlberg 7(8) [3], widely considered adequate for a general purpose trajectory propagator. It is, nevertheless, fairly simple to implement any other integrator in case it is required.

The readily available FORTRAN algorithm was ported to C++ and modified to provide full time step control. The self-adjusting time step algorithm differentiates the position and the velocity truncation error and compares it to the tolerance specified by the user. The error factors for the position and velocity are

$$\varepsilon_{pos_i} = \frac{\tau_{pos_i}}{tol_{pos}}, \quad \varepsilon_{vel_i} = \frac{\tau_{vel_i}}{tol_{vel}}, \tag{4.11}$$

where τ represents a truncation error, and tol the user-defined tolerance. The final error factor for the step i is set to the maximum

$$\varepsilon_i = \max\left(\varepsilon_{pos_i}, \varepsilon_{vel_i}\right). \tag{4.12}$$

The error factor is used to determine the next step size

$$h_{i+1} = 0.8 \times \left(\frac{1}{\max\left(\varepsilon_{pos_i}, \varepsilon_{vel_i}\right)} \right)^{\frac{1}{8}} \times h_i, \tag{4.13}$$

where h_i is the current step size. A safety factor of 0.8 was used as it is the most common used for Runge-Kutta-Fehlberg methods. If the truncation error is above the tolerance ($\varepsilon_i > 1$) the step is revaluated using the new step size until the error is tolerable. In order to make the method more feature-rich the time step control was also modified to include options for fixed time step, and minimum and maximum step boundaries.

The integrator makes use of the MPIR and MPFR libraries to both take advantage of CPU specific instruction sets leading to performance increases as well as give the user a fine-control of the bit-precision. This grants flexibility to the integrator, allowing for its use in both time-critical and precision-critical environments.

4.2.5 Input and Output Files

A save and load mechanism for the user input parameters was implemented. This ensures that a simulation state can be saved for later use or modification. The chosen file format for this purpose is XML. Due to the flexibility of this file format it is possible to have multiple XML files for different simulation parameters and load them independently and incrementally.

The XML input file schema is divided into four sections – <general>, <forceModel>, <integrator>, and <output> – each corresponding to the tabs found in the propagator GUI. An example of a propagator setup input XML file can be found on Appendix D.

It is also possible to load the initial conditions from a TLE set, which is commonly used for satellites orbiting the Earth.

The propagator is able to simultaneously generate two output files – a binary SPK (cf. section 3.2.2) and a raw plain text file. The binary SPK can have Hermite or Lagrange interpolation of a variable degree, specified by the user. The only raw plain text file configuration is its comment line start character. Table 4.2 shows a plain text output sample line.

Table 4.2 – Propagator plain text output sample (truncated values).

```
# Trajectory generated by SimOrbit v1.3.0.372M
#
# Legend:
# JED      Julian ephemeris date (days past 1 Jan 2000 12:00:00 (TDB))
# X        x component of position vector (km)
# Y        y component of position vector (km)
# Z        z component of position vector (km)
# VX       x component of velocity vector (km/s)
# VY       y component of velocity vector (km/s)
# VZ       z component of velocity vector (km/s)
# CENTRAL  Central body NAIF code
# FRAME    Frame of reference label
#
# JED         X           Y           Z           VX          VY          VZ         CENTRAL   FRAME
2440419.18209 +6.38508e+3 +1.71602e+3 +1.15646e+3 -1.98714e+0 +9.10852e+0 +5.51966e+0 399       J2000
```

4.3 Simulator Module

The simulator module provides a realistic rendering of the Solar System which allows the user to track the position, attitude, and trajectory of one or more spacecraft.

To enhance the realism of the simulation the module includes a procedural planet generator which can generate a planet's geometry, includes features such as atmosphere scattering, clouds, and rings. Additionally the Sun is also automatically generated, and the Tycho star map [29] is used as background.

For more complex geometries, such as spacecraft and some asteroids and comets, a 3D model loader is provided, supporting most common interchange formats. Having the ability to support multiple formats is very useful to a user, since 3D models can be found in a variety of major formats. The conversion process between 3D formats is not always a simple task, and requires additional software.

Additionally, the visualizer is able to render auxiliary feature such as trajectories, frames of reference, labels, and markers for distant objects.

To ensure compatibility with pre-existing data and readily available propagators the simulator module trajectory input format is a binary SPK file (cf. section 3.2.2). This method has the advantage of allowing the usage of the SPICE routines for both the planetary and the spacecraft ephemerides.

The camera can be controlled manually using the keyboard and mouse to move the camera or automatically by defining a list of parametrized cameras. The latter being ideal to use in conjunction with the movie-making feature in order to generate high-quality movies.

4.3.1 Graphical User Interface

The simulator GUI was designed to be as unobtrusive as possible, and consists of a semi-transparent, auto-hiding dialog on the bottom of the window, with tabs to control the scene, camera, and movie-making.

The scene tab allows the user to control all the objects being rendered. The user can load SPK files containing ephemerides of objects of interest, associate 3D models, and select which auxiliary features should be rendered and under what conditions. For example, it is possible to specify if a marker for a distant body should be rendered always, when the camera is in the central body system, or never.

The camera tab is used to toggle between the manual and automatic cameras. It also lists the loaded automatic cameras, and provides means to add, edit, and delete them. Additionally, it shows the diagram of the manual camera keyboard and mouse controls.

Finally, the movie-making tab allows the user to configure the movie writer output filename, quality, and resolution and initiate or finish the capture of a movie.

4.3.2 Procedural Planet Generator

The software features a procedural planet generator for planets and moons whose body radii are known. The generator creates an ellipsoid with the appropriate dimensions and sets the colours, textures, and atmosphere and ring parameters in accordance with the information present in the Planets.xml file, for which an example is presented on Appendix D. Figure 4.7 shows planet Earth rendered with the parameters present in the example file.

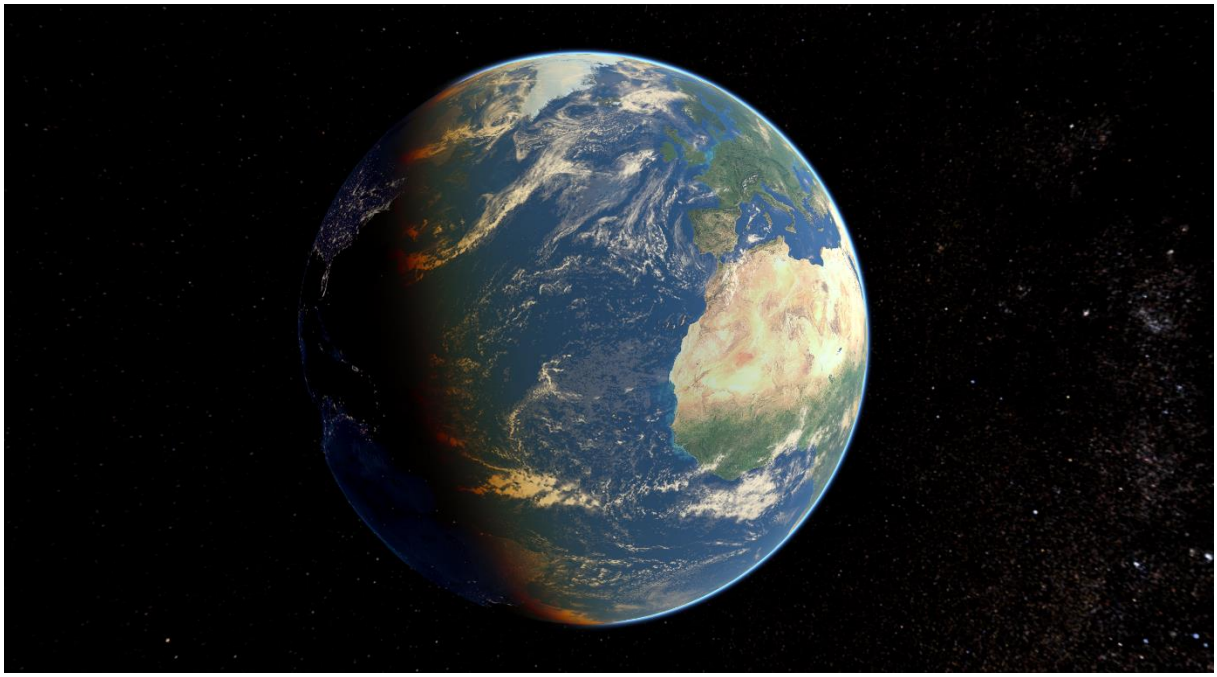


Figure 4.7 – Screenshot of the Earth as rendered by SimOrbit.

Surface

The colour of the surface of the planet is a combination of diffuse and emissive colours and textures. The diffuse texture stores the daylight representation of the planet's surface, whereas the emissive texture provides the night-time colours, such as city lights or auroras. The user can also provide a specular map, which controls the specular strength (i.e. the reflectivity of the surface), and a tangent-space normal map which stores the per-pixel normal in order to give the illusion of terrain height.

The specular lighting of planets is based on the Kelemen-Szirmay-Kalos bidirectional reflectance distribution function (BRDF) model [30]. This model closely approximates the Cook-Torrance model [31] but is significantly cheaper to evaluate. The classic Blinn-Phong model [32] was also tested, but proved too glossy and unrealistic.

Atmosphere and Clouds

There are numerous algorithms for atmospheric rendering. The simplest ones render a single colour semi-transparent sphere whose opacity varies with the angle between the surface normal and the direction of the incoming light. These algorithms, although very easy to implement, fail to take into account the atmosphere density variation with altitude, the atmospheric scattering, and its wavelength

dependency. The physically based algorithms, which take into account Rayleigh and Mie scattering, are computationally expensive, requiring the calculation of nested integrals for which no analytical solution exists. The simple algorithms were quickly discarded since they do not provide a realistic representation of the atmosphere and just create a coloured halo around the planet.

Upon review of the available physically based algorithms [33], the method proposed by Sean O'Neal [34] was implemented. This model is very popular and widely used and has the advantage of not requiring the pre-computation of any integrals. Its only significant drawback is that it relies on a scale function which is dependent on the scale height and ratio between the atmosphere's thickness and the planet's radius. Since this scale function is obtained via polynomial curve fitting it is not trivial to recalculate for arbitrary atmosphere parameters, hence the atmosphere's thickness of 2.5% of the planet's radius and the scale height of 25% of the atmosphere's thickness proposed by O'Neal are used in the software.

It is also worth noting that the atmospheric scattering model was developed for Earth's atmosphere, and requires coefficient adjustments in order to be used in other planets. These coefficients can be set in the Planets.xml file for each planet for which an atmosphere is rendered. The Martian atmosphere coefficients [35] have already been included. Figure 4.8 illustrates the Earth and Mars atmospheres as rendered by the software.

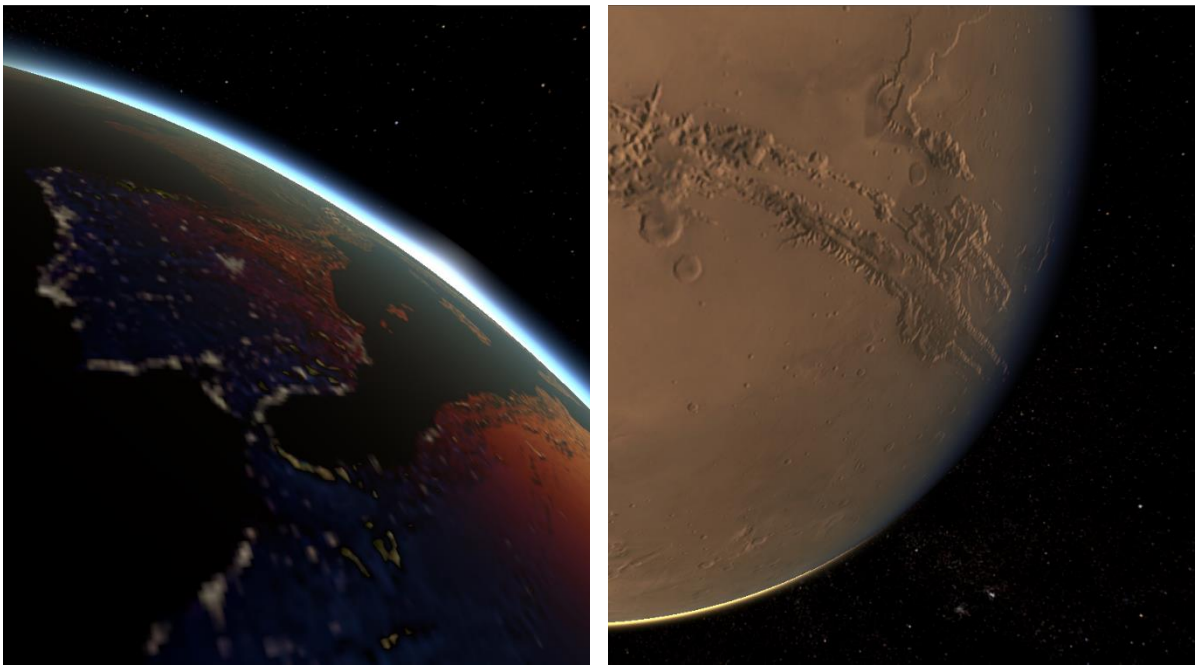


Figure 4.8 – Atmospheric scattering examples. Sunrise over the Iberian Peninsula on Earth (left); sunset over Valles Marineris on Mars (right).

The software also supports loading and rendering clouds texture maps. The clouds texture map is drawn over the planet at the altitude set in the Planets.xml file. For simplicity, and since it didn't affect visual realism when seen from a moderate altitude, no cloud shadows are casted on the planet's surface.

Rings

Finally, the procedural planet generator supports the generation and rendering of planetary rings. The rings are a flat dual-sided anullus centred on the planet, laying on its equatorial plane. The inner and outer ring radius are set in the Planets.xml file, along with the rings texture, which stores both colour and transparency.

The rings' illumination presents a very complex problem, since these are illuminated by backscattered light from the planet and the neighbouring ring particles, and forward scattered light. A simple model was devised (cf. section 5.2.3) attempting to match Saturn's rings renderings to the photos from the Voyager and Cassini missions. The model is not physically accurate but provides acceptable results.

4.3.3 3D Models

Spacecraft and other complex geometry models, such as some asteroids and comets, cannot be procedurally generated. For these it was necessary to implement a 3D model loader. Since there is a wide variety of common interchange formats, and it was desirable for the software to be compatible with as many as possible, the Assimp library (cf. section 3.2.4) was used as a 3D model loader.

Assimp loads the models into one unified data structure, which the software processes to render the model. The model's geometry is pretty straightforward to process but the material list is not. Materials range from just one colour to complex structures, which specify which shading model to use and have multiple textures, each with its own mapping and blending modes. For simplicity not all material properties are processed, and the software uses the Blinn-Phong shading model [32] to render all models. The supported material properties are:

- Ambient, diffuse, specular, and emissive colours;
- Shininess and specular colour scaling factor;
- One texture per type (ambient, diffuse, specular, and emissive), with UV coordinates mapping;
- Tangent-space normal maps.

Figure 4.9 shows NASA's Cassini spacecraft 3D model [29], as rendered by SimOrbit.

Two ways were implemented for associating a 3D model with an object – static and dynamic. The static associations are listed in the Models.xml file, for which an example is presented on Appendix D. The user can also load a 3D model dynamically via the GUI. Additionally, since there is no guarantee the loaded models are correctly scaled, the software requires a scaling factor to be entered by the user (e.g. on an asteroid model units may represent kilometres, whereas on a spacecraft model they can represent inches). This factor is defined as the constant that converts model units to kilometres, and by omission it has the value of 1.

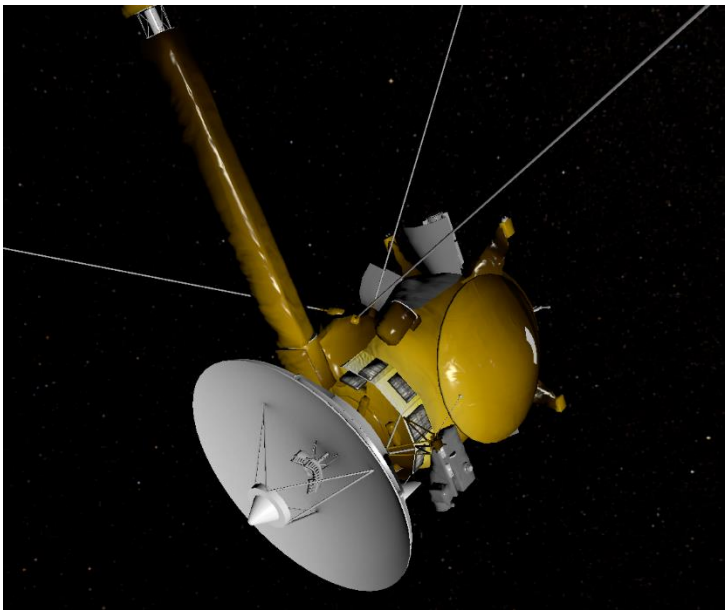


Figure 4.9 – Screenshot of the Cassini spacecraft model as rendered by SimOrbit.

4.3.4 Auxiliary Features (Frames, Markers, Labels, and Trajectories)

To analyse the 3D environment from a scientific point of view it is necessary to draw auxiliary features, such as local frames of reference, markers, labels, and trajectories. These features are all controllable on an object basis and can be turned on and off via GUI. Additionally, they are rendered with the light positioned at the camera, thus ensuring they are always lit, and in chartreuse colour since it is widely regarded as the most visible colour.

Frames of Reference

The frames of reference are invaluable to assess the orientation of an object, for example to determine its relative orientation to another, or to have a general idea of flight path and lighting angles.

They are composed by three perpendicular vectors, labelled X, Y, and Z, according the orientation of the primary axis of the desired frame (cf. Appendix A). The frames of reference are scaled with the camera distance and have a minimum axis length of two times their associated object size. This ensures they are always visible regardless of the distance, and do not shrink inside the object when viewed from a close distance.

The user can select which frame to draw, taking into account that the BODY frame needs SPICE body orientation data, and the spacecraft LVLH and TRAJECTORY frames require the user to specify the central body, since this information is not directly deducible from the ephemeris.

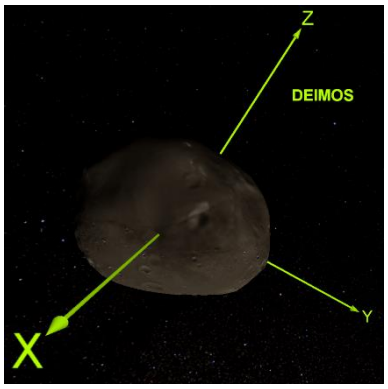


Figure 4.10 – Deimos BODY frame of reference and label.

Markers

Since the size of the bodies in the Solar System is extremely small when compared to the distance between bodies, it becomes necessary to place markers in the position of distant bodies in order to have an idea of their position. The implemented markers are four-point stars [36], and have a label with the name of their associated body, both with constant screen size. The markers are rendered for spacecraft that have no 3D model associated and procedurally generated bodies whose on-screen size is smaller than 10 pixels. An example of the implemented markers is show on Figure 4.11.

Figure 4.11 – Sun, Mercury, and Venus markers and labels.

Trajectories

The most complex auxiliary feature is a trajectory, and several problems arise while trying to generate and render them. Firstly, neither Direct3D, nor any other graphics API, has the ability to draw curves, meaning all trajectories must be reduced to a set of points and line segments in between them. This means the trajectory needs to be sampled in order to determine a set of points that adequately represent it, and that can provide the illusion of a curve. Ideally the positions should match the points present in the input files, however not all SPK trajectories are defined via a collection of states, and even on those that are the API does not provide a way to retrieve them.

The set of points that are used to define a trajectory are determined using an iterative method that divides each coverage window (i.e. contiguous time span for which ephemeris are known) in segments until a tolerance condition is met, or the maximum number of trajectory points is reached. The algorithm begins by retrieving the positions corresponding to the first, middle and last epoch times in the coverage window. Then it calculates the distance from the middle point to the line segment between the first and last points, if it is greater than the tolerance it adds the first-middle and middle-last line segments to an evaluation queue for further division. The algorithm keeps dividing the segments in the queue on their

middle time point until their tolerance condition is met, or it reaches the maximum trajectory points, set to 65537. The algorithm operation is represented graphically on Figure 4.12, for the first four iterations on the Mars Science Laboratory mission. This method is far from ideal, and is a point where future improvement is desirable.

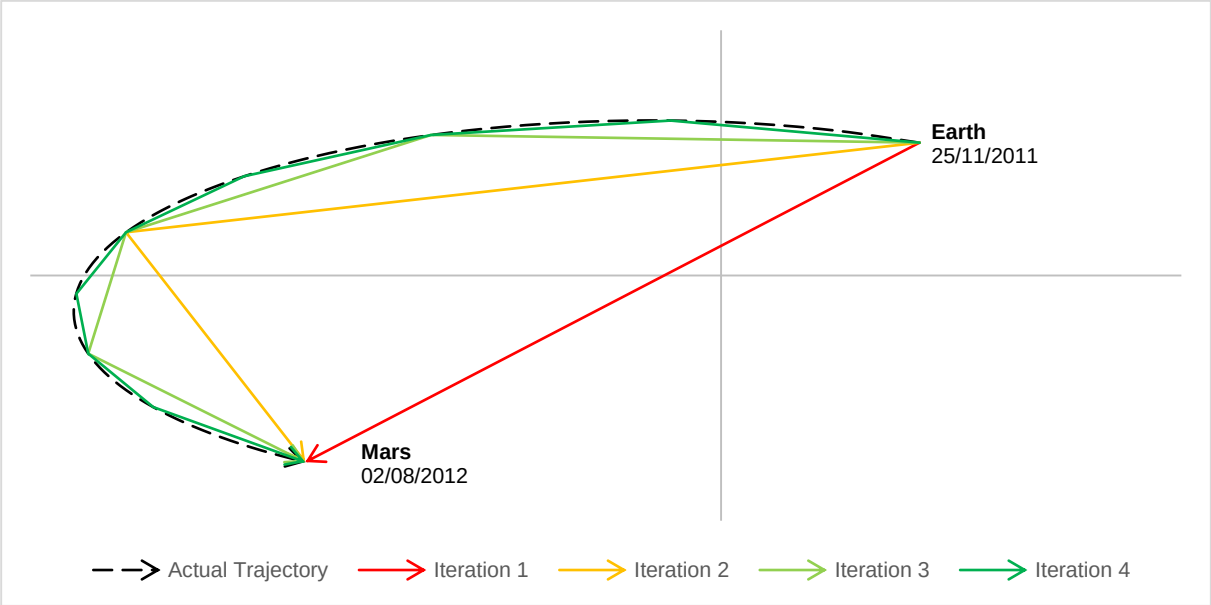


Figure 4.12 – Example of the first four iterations in the determination of the trajectory points of the Mars Science Laboratory mission.

Since the trajectory retrieval is a costly process involving tens of thousands of ephemeris queries, it is not possible to do it in real time while rendering. To mitigate the issue, the trajectories are only generated for spacecraft, since this is the focus of the software. Furthermore, the loaded trajectory cannot be adjusted when changing reference body, meaning the trajectory is always drawn with reference to the Sun. For example, the trajectory of a satellite orbiting the Earth will not be rendered as a series of orbits around the Earth but rather as a perturbed orbit around the Sun. It is arguable if this is ideal or not, since it has the advantage of representing the actual trajectory of the satellite within the Solar System, but is inadequate for the determination of its relative position to the Earth.

4.3.5 Cameras

A first person camera, which attempts to replicate what a user would see if he was in space, was implemented. In order to provide maximum of flexibility two types of camera controls were developed, a manual control via direct keyboard and mouse input, and an automatic control with parametrized cameras.

Manual Camera

The manual camera is a five degrees of freedom (three-dimensional translation, pitch, and yaw) camera controlled in real time by the user using the keyboard for translation and the mouse for rotation as shown in Figure 4.13 (alternatively it is also possible to use the arrow keys instead of W, A, S, and D). The roll

movement was restricted since it would require either a third degree of freedom on the mouse, or adding rotation controls to the keyboard. The camera vertical direction was defined as the vertical direction of the J2000 frame of reference.

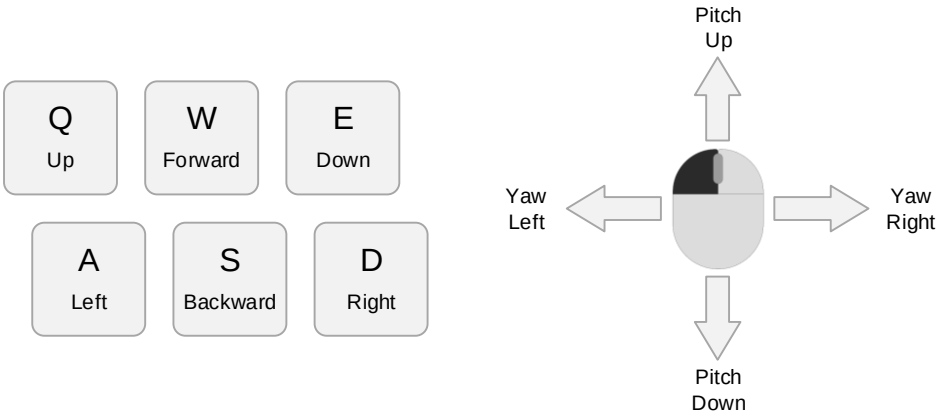


Figure 4.13 – Manual camera keyboard and mouse controls.

Implementing the manual camera presented three major problems. Firstly, it was necessary to prevent the user from driving the camera inside solid bodies, namely planets. To resolve this issue, upon loading or generating a 3D model the software computes its bounding sphere, which is used by the camera to determine where to stop. The bounding sphere solution is very well suited for planets and other quasi-spherical bodies, but has the disadvantage of preventing the camera getting close to non-spherical objects such as most spacecraft and some asteroids and comets. Other types of bounding volumes were considered but were ultimately disregarded due to either the computational cost, the complexity of the implementation, or causing erratic camera movement (as is the case of multiple bounding boxes, where the camera could easily become stuck on corners).

Secondly, it was desirable to devise a way of quickly travelling to distant objects (i.e. celestial bodies and spacecraft) without overshooting them. From the tested methods the most promising were those that scaled the camera velocity with the distance to the nearest objects. Both linear and exponential velocity scaling were tested, but the latter was found to be uncontrollable. Additionally, in order to avoid abrupt stops when colliding with a bounding sphere the method was modified to scale the camera velocity linearly with the distance to the nearest object's bounding sphere.

Lastly, a method of assigning the camera reference object (i.e. the object whose relative position to the camera is constant) was required. The most natural method tried was determining the Hill sphere radius of all the objects being rendered, testing if the camera was inside, and selecting the object with the smallest Hill sphere for reference. This method, however, does not work for spacecraft which have negligible GM. For spacecraft, if their screen size is larger than 10 pixels the closest one is selected as the camera reference object.

Automatic Camera

The automatic camera is actually a collection of parametrized cameras which are selected based on their valid time span. The cameras can be added, edited, and deleted via the GUI or loaded from one or more XML files. An example of an automatic camera XML input file can be found in Appendix D.

An automatic camera is defined by:

- Time span – The time interval in which the camera should be active, defined by a start epoch and optionally a stop epoch or duration;
- Time scale – The time that passes within the simulation for every second;
- Eye position – The position of the camera;
- Look at position –The position the camera is looking at;
- Up direction – The direction of the camera vertical (optional, by default the vertical direction of the J2000 frame of reference).

To facilitate the parametrization of the camera the positions (eye and look at) are defined by an offset from a specific reference body in one of the implemented frames of reference (cf. Appendix A). The offsets can either be given in Cartesian or spherical coordinates. The up direction is defined in a similar manner except a direction vector or azimuth/elevation is required, instead of an offset.

The software does not require the cameras time span to be contiguous or non-intersecting. Instead it selects the camera based on which cameras are valid for the current epoch and gives precedence to the one that has a later start epoch. If no camera is defined it stops updating the camera parameters until another valid camera is found. An example of this behaviour is shown in Table 4.3.

Table 4.3 – Graphical representation of an example cameras timeline with the time spans highlighted and final usage sequence.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Ago	Set	Oct	Nov	Dec
Camera 1												
Camera 2												
Camera 3												
Camera 4												
Sequence												

4.3.6 Movie Making

The software exports WMV movies. This format was selected since it is guaranteed to be compatible with the operating systems the software is compatible with. Using a different format would require shipping additional third party libraries and might require the user to install a codec or media player to be able to watch the rendered movie. The WMV3 codec, commercially known as Windows Media Video 9, was chosen for video compression since it provides support for a wide range of bit rates, allowing the user to create high definition quality video, or low-bitrate internet video [37].

The video encoding bitrate presented two possibilities – constant or variable. The constant bit rate encoding (CBR) strives to maintain the bitrate, whereas the variable bit rate (VBR) encoding strives to achieve the best possible quality. Since improving the quality is always desirable, the VBR encoding was chosen. There are several types of VBR encoding which determine how the instantaneous bitrate is determined. Quality-based VBR encoding seemed the most adequate since it is the easiest for the user to comprehend – a level of quality is selected, and then the codec encodes the content ensuring all frames are of comparable quality. The only disadvantage to this approach is that there is no way of determining the size of the movie prior to encoding.

In order to capture a movie the user needs to first configure it, by selecting the output filename, the resolution, and the quality level. The resolution can be set to one of three options – Native (the current window resolution), HD720 (1280x720), or HD1080 (1920x1080). The quality level is a discrete value 1 to 100, where 1 represents the lowest possible quality and 100 the highest. After the output is configured, capturing can be started and stopped by pressing the appropriate button on the GUI, or the C key on the keyboard.

While rendering a movie at a non-native resolution it is likely that the image displayed in the program window is just a part of the one being captured, or vice-versa. This cropping is a consequence of the necessity of not having the ability to scale Direct3D texture resources while copying them.

Finally, it is possible for the user to interact with the GUI while capturing a movie, for example to turn auxiliary features on and off, since the capturing occurs before the GUI is rendered to prevent it from being shown in the movie. The movie maker does, however, capture the on-screen text, namely the simulation epoch time.

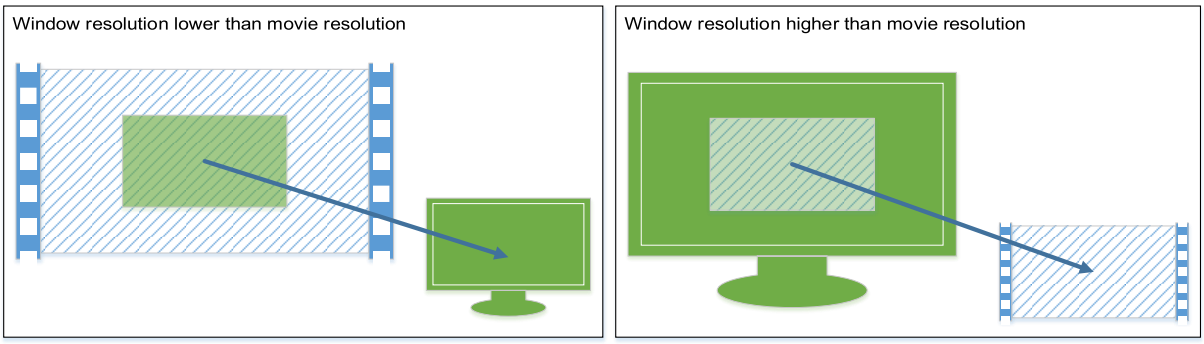


Figure 4.14 – Movie making cropping scenarios when capturing at fixed resolutions.

Chapter 5

IMPLEMENTATION DETAILS

SimOrbit was developed as single Visual Studio solution encompassing three projects – SimOrbit, SimOrbit Library and Framework (cf. section 5.3), and the SPICE C++ Wrapper Library (cf. section 5.4). All three projects were developed using modern and effective coding techniques, such as templates, containers, and safe pointers [38] [39].

In order to have a better insight into the code, and to ensure the low cyclomatic complexity and maintainability objectives were reached, it was necessary to determine its metrics. The C++ code metrics were generated using the SourceMonitor software [40] since Visual Studio 2013 does not provide code metrics functionality for native C++ code. A summary of the metrics for the main project – SimOrbit – is shown in Table 5.1. These include the module manager and both the propagator and simulator modules. The copyright notice header that is present on all files was excluded to avoid interference with the "lines" and "percent of lines with comments" values.

Table 5.1 – Metrics summary for the SimOrbit project.

Parameter	Value
Files	180
Lines	42,664
Statements	15,146
Percent Branch Statements	12.7
Percent Lines with Comments	30.6
Classes Defined	129
Methods Implemented per Class	11.61
Average Statements per Method	7.3
Maximum Complexity	36
Maximum Block Depth	9+
Average Block Depth	1.28
Average Complexity	2.25
Functions	252

Although the metrics summary provides a quick overview of the dimension of the project, it is inadequate to assess if the objectives were met, hence the Kiviat metrics graph (Figure 5.1) was generated. This graphic includes a highlighted region indicating the ideal value range for the different metrics. All average metrics were found to be within the ideal range; however the "percent of lines with comments", "maximum complexity" and "maximum block depth" are too high. Since XML comments are present throughout the code in order to be able to automatically generate the reference manual an inflated value of lines with comments was to be expected. The exaggerated maximum cyclomatic complexity and block depth were traced to the trajectory propagator worker thread function. The algorithm behind this function is indeed very complex, and thus is explained in depth on section 5.1.1. Additionally, its flowcharts have been included in Appendix E, and also the software reference manual.

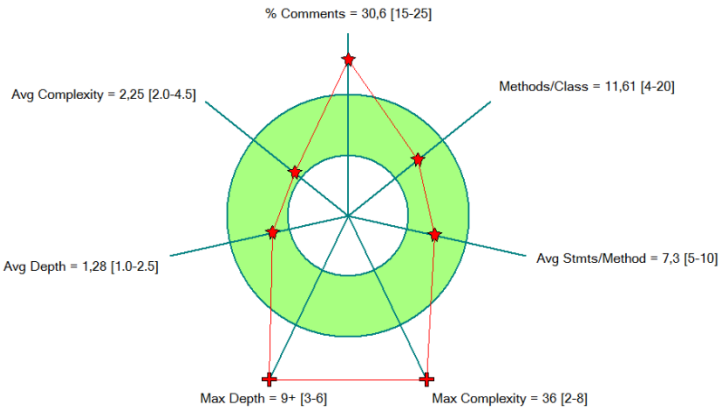


Figure 5.1 – Kiviat metrics graph for the SimOrbit project.

5.1 Propagator Module

The propagator module is the only multi-threaded component of the software. It consists of two main dialogs – setup and progress. The setup dialog is the module's entry point and allows the user to fully configure every aspect of the trajectory propagator. When the trajectory propagator is active the progress dialog is shown instead, allowing the user to monitor and control its operation.

The thread communication and information flow is shown in Figure 5.2. When the user issues the command to start the computation a worker thread is created. This thread is self-contained, creating instances of all the components it uses, and its outside communication is through a shared progress information class. This solution has the advantage of enabling a future extension of the propagator module where multiple input parameters can be used to simultaneously generate several trajectory alternatives, by creating several worker threads.

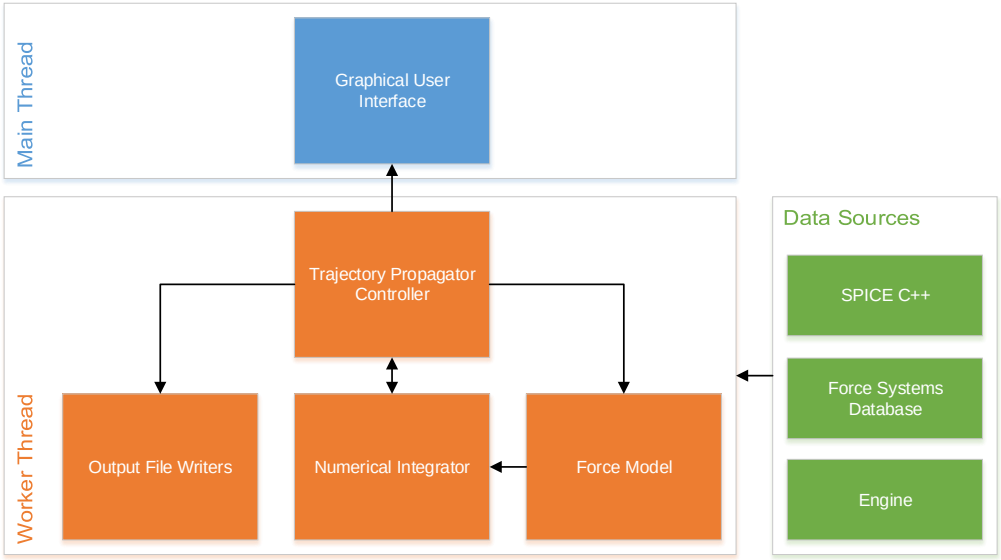


Figure 5.2 – Thread communication and information flow diagram during the operation of the propagator module.

5.1.1 Trajectory Propagator Operation

The trajectory propagator controller is the most complex algorithm present in the software (cf. Appendix E). This controller runs on a separate thread, and is responsible for configuring and operating the numerical integrator, selecting the appropriate for systems and configuring the force model accordingly, pushing the computed spacecraft states to the output file writers, and reporting progress to the GUI on the main thread.

The first task performed by the propagator controller is configuring all the components it operates – numerical integrator, force model and output file writers (SPK and plain text) – using the information previously provided by the user via the GUI or the propagator setup XML files.

Once all the components are configured, the controller determines the best available force system for the initial spacecraft state and enters its main loop, which runs until the numerical integrator has reached the specified stop date, or duration.

Inside the main loop the controller manages the segments, which are defined as a collection of states in which the central body does not change and the velocity is continuous. Firstly, the current segment stop time is set to the next delta-V engine input or the propagation stop date, whichever is the earliest. Secondly, any delta-V inputs coinciding with the segment start time are applied. Afterwards, the output file writers are instructed to initiate a new segment and the initial spacecraft state is pushed (i.e. written to the output). Finally, the controller enters its secondary loop, called the segment loop.

The segment loop is where the actual trajectory propagation occurs. It starts by performing a numerical integration step, and pushing the updated state to the output file writers. Afterwards, it tests if the spacecraft has crashed inside the central body, by comparing its distance to the equatorial radius of the body. If it crashed, the segment and the trajectory propagation are immediately terminated and an error message is sent to the event log. In case a crash did not occur it checks if a manual stop command was issues by the user, and if so terminates the segment and the trajectory propagation. This is also where a user pause/resume command is processed. If no crash was detected and no stop command was issued the controller communicates its updated progress to the GUI. The last step in the segment loop consists in evaluating if the force system is still the most adequate, and in case it is not, the segment is terminated, and a new one is initiated using the best force system available.

Force System Selection

Since no Solar System planet is inside the sphere of influence of any other, the only force system changes that are necessary to test are between the planetary systems and the default force system – Sun. Hence, the force system selection algorithm is divided in two major paths. If the current force system is a planetary system, it evaluates if the spacecraft is still within 101% of its central body's sphere of influence, switching to the Sun system if not. If the current force system is the Sun system, the algorithm iterates over all other available systems to determine is the spacecraft is inside 99% of their respective central body's sphere of influence. If the spacecraft is found to be inside, the loop is terminated and that planetary system is set as current.

Numerical Integrator Step

The numerical integrator is responsible for integrating the equations of motion of the spacecraft (i.e. the derivatives of the position and velocity). Firstly, it determines the new state using the RKF7(8) methods and the derivatives provided by the force model implementation. If the time step is fixed, no more operations are performed. If, however, the time step is variable it proceeds to estimate the truncation error and compare it with the user-defined tolerance. If the truncation error is within tolerance the step is considered valid, the time step is adjusted for the next iteration (taking into account any limits set by the user). In case of a failed tolerance check the time step is adjusted if possible, and the step calculation is repeated. If the time step is already at a user defined minimum it is not adjusted, and instead a "truncation error over tolerance" warning is issued.

Calculate Derivatives

The force model algorithm, i.e. the computation of the position and velocity derivatives, is fairly simple and follows the equation sequence discussed in section 4.2.2. The algorithm begins by calculating the acceleration due to the central body's spherical gravity field (GM). If the J_2 perturbation is enabled and the coefficient is known, its perturbation is added. This is done by converting the position to the body-fixed frame of the central body, calculating the perturbation, and converting it back to the J2000 frame. Afterwards, the list of active third bodies present in the force system is iterated, and their perturbations are calculated and added. The Sun is external to the list of third bodies list as it shares code with the solar radiation pressure implementation, these contributions are calculated and added if enabled. Finally, the list of constant acceleration engine inputs is iterated and their contributions are also added. The derivative of the position is set to the state's velocity and the derivative of the velocity to the calculated acceleration.

5.2 Simulator Module

The simulator module is a 3D rendering environment which uses a collection of High Level Shader Language (HLSL) files to program the vertex, geometry (Direct3D 10 only), and pixel shaders. These control how the vertices of the models are transformed into screen coordinates, and how each individual pixel colour is calculated. These files are embedded in the software executable as resource files, and are compiled at run-time. Since the software has a dual Direct3D 9 and 10 rendering engine, HLSL files are provided for both versions. The only exception is the Perlin noise implementation (cf. section 5.2.2) that is only available in Direct3D 10 mode, due to shader model limitations.

5.2.1 Depth Buffering

The rendering process converts the object's 3D coordinates into 2D screen coordinates, thus eliminating the depth information. There are two techniques that can be used to determine which objects should be drawn, and which ones are hidden from view – the painter's algorithm, and depth buffering. The painter's algorithm consists in rendering the scene starting with the farthest object from the camera, and overlapping every other object in depth decreasing order. This algorithm, although simple, is inefficient

since the pixels are constantly being redrawn, and can also fail completely for objects which intersect others. Depth buffering, also known as z-buffering, stores the depth of the objects being rendered in a surface with the same dimensions as the render target. The depth values are then used to determine if a pixel should be drawn, or immediately discarded due to being hidden from view. Depth buffering is the technique used in most modern 3D computer graphics applications.

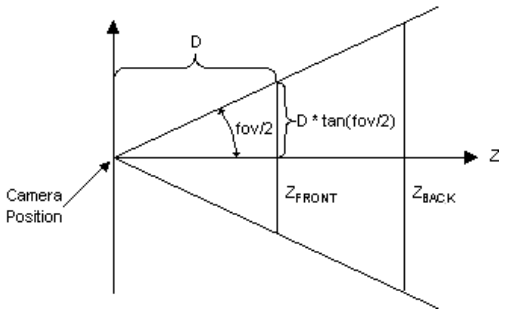


Figure 5.3 – Illustration of the Direct3D viewing frustum.

The depth buffer hardware implementation does have a major problem, since it uses

$$depth(z) \propto \frac{1}{z}, \tag{5.1}$$

where z is the object's distance from the camera (cf. Figure 5.3). This causes the depth buffer to use half its resolution just in the distance from the near plane to twice that distance. Rendering the Solar System becomes very problematic. To be able to view spacecraft the near plane needs to be no farther than 1 meter from the camera, and to be able to view far away planets the far plane needs to be hundreds of millions of kilometres away from it. Using these viewing frustum limits makes the scene unrenderable.

Brano Kemen addressed this issue [41] and proposed the usage of a logarithmic depth buffer to increase the precision (i.e. decrease the resolution) and bring it to a usable region, as shown in Figure 5.4.

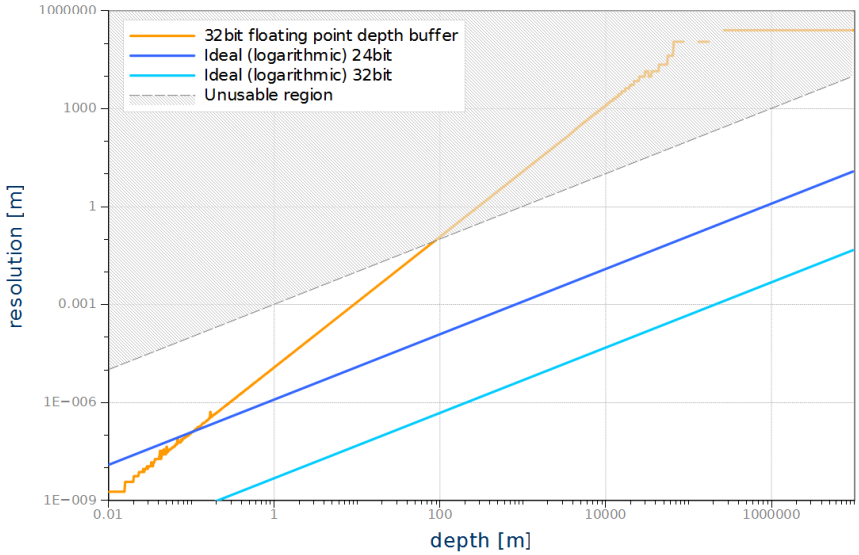


Figure 5.4 – Comparison of the precision of normal and logarithmic depth buffers for depths ranging from of 0.01m to 10,000,000m [42].

The proposed modification suggests storing depth values as

$$depth(z) = K\frac{\ln(C \cdot z + 1)}{\ln(C \cdot z_f + 1)}, \tag{5.2}$$

where K is the maximum value in the depth buffer, and C is a tuning parameter controlling the width of the linear part of the function. Thatcher Ulrich made a slight variation to the formula, removing the dependency on the tuning parameter

$$depth(z) = (2^k - 1)\frac{\ln(z/z_n)}{\ln(z_f/z_n)}, \tag{5.3}$$

where k is the number of bits of the depth buffer, assuming an integer depth buffer. This method had the advantage of providing constant relative precision. Direct3D, however, has no support for integer depth buffers, which results requires the formula to be modified to

$$depth(z) = \frac{\ln(z/z_n)}{\ln(z_f/z_n)}, \tag{5.4}$$

resulting in a bit of instability in the precision due to the way floating-point numbers are stored in binary format. Nevertheless, the method performs far better than the standard Direct3D depth buffering implementation, and allows the rendering of very close objects with high level of detail, such as a spacecraft, and very far objects, such as the Sun seen from millions of kilometres away.

Even using the modified depth buffering, it is still not possible to have a depth buffer than encompasses the entire Solar System and still allows a user to “zoom in” on a small spacecraft. To solve the issue the objects that are so far away that a marker has to be rendered to make them visible are rendered using the painter’s algorithm, instead of depth buffering.

Hence, to render the scene:

1. The star background is drawn;
2. The far object markers are rendered from the farthest to the nearest with depth testing off;
3. The near objects’ 3D models are rendered from the closest to the farthest with depth testing on.

5.2.2 Noise Generator and Sun Shader

In order to have elements like the Sun’s corona appear more natural it was necessary to implement a noise generator. Upon reviewing the state-of-the art [43], and the available implementations, the widely used improved Perlin noise [44] algorithm was chosen.

Perlin noise “determines noise at a point in space by computing a pseudo-random gradient at each of the eight nearest vertices on the integer cubic lattice and then doing a splined interpolation” [43]. The noise results are usually scaled and summed together. One of the ways of doing this is via fractional Brownian motion (fBm).

Fractional Brownian motion was implemented as

$$fBm(P) = \frac{\sum_{octave} noise\big(P \times f(octave)\big) \times A(octave)}{\sum_{octave} A(octave)}, \tag{5.5}$$

where P is the point at which the noise is being calculated.

Octaves are how many noise layers are being putting together, the higher the number is, the higher the detail. The frequency, f, determines the width of the noise features, with each octave the frequency increases according to

$$f(octave) = \lambda^{octave-1}|_{\lambda>1}, \tag{5.6}$$

where λ is the lacunarity, i.e. the frequency multiplier between octaves. The amplitude, A, controls the height of the noise features, and changes with each octave through the relation

$$A(octave) = G^{octave-1}|_{0<G<1}, \tag{5.7}$$

where G represents the gain, also called persistence. A graphical representation of the fractional Brownian algorithm, applied to the Perlin noise, is shown in Figure 5.5.

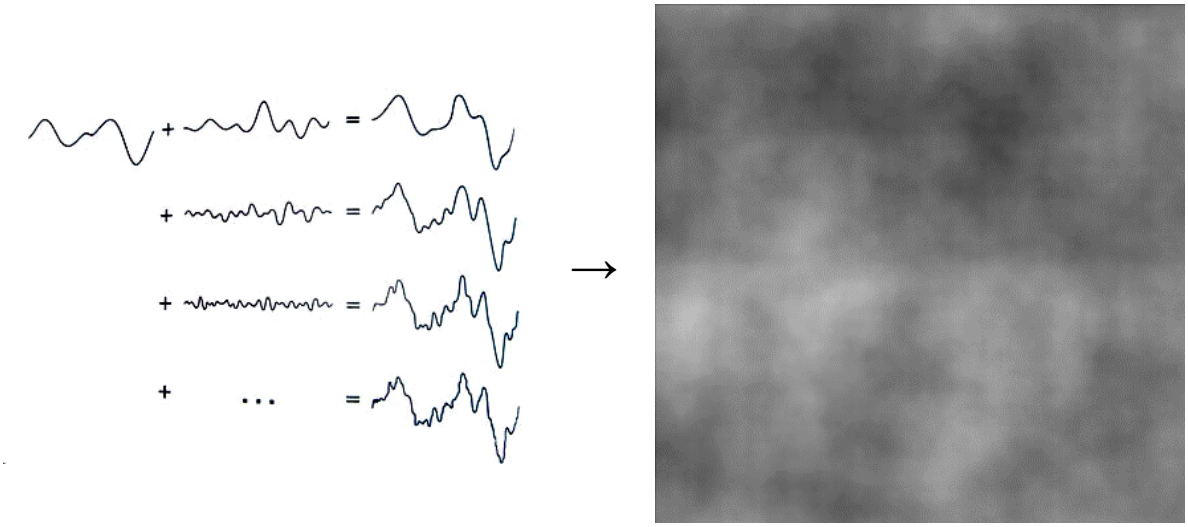


Figure 5.5 - Graphical representation of the fractional Brownian motion algorithm applied to Perlin noise.

The software uses the Perlin noise implementation proposed by NVIDIA [34], which computes noise directly in the pixel shader, differing from the typical methods that require the use of precomputed 3D textures. This has several advantages, namely it allows four-dimensional noise, enabling the usage of time as a fourth dimension to create three-dimensional effects with animations. The Perlin noise and fractional Brownian motion algorithms were placed in a separate HLSL include file to facilitate their usage across multiple effect files.

Sun Shader

Currently the only application of the noise generator within the software is the rendering of the Sun's corona. The developed algorithm is not physically based and is only meant to add some realism to the rendering of the Sun, as shown in Figure 5.6.

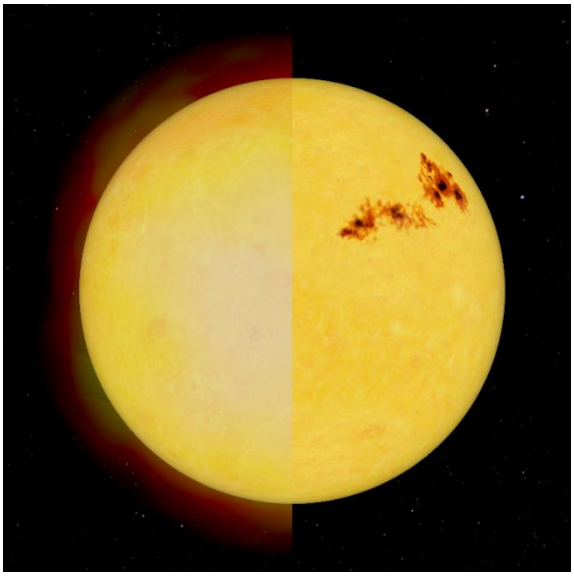


Figure 5.6 – Comparison of the Sun with corona rendering enabled (left) and disabled (right).

The corona is rendered has a double-sided, semi-transparent sphere, centred at the Sun, with a radius of $R_{\odot} + 150{,}000km$ ($R_{\odot}$ being the radius of the Sun). For visible point $\boldsymbol{p}$ on the the corona, the angle

$$\theta = \angle NV, \tag{5.8}$$

between its surface normal, N, and the direction to the camera position, V, is calculated. The angle is then used to determine the base intensity of the corona

$$I_{base} = 504 \times \cos(\theta) + 84 \times \cos(3\theta) - 36 \times \cos(5\theta) + 9 \times \cos(7\theta) - \cos(9\theta), \tag{5.9}$$

to which the fBm noise at the normalized point $\hat{\boldsymbol{p}}$ is added, obtaining the final intensity

$$I = I_{base} + \mathrm{fBm}(\hat{\boldsymbol{p}}). \tag{5.10}$$

The fBm is calculated with 3 octaves, $\lambda = 2.0$ and $G = 0.5$, making it a pink noise.

The intensity is then converted to a RGB (red, green, blue) colour using the process described on equation (5.11), which produces the gradient shown in Figure 5.7 for intensities ranging from 0 to 1.

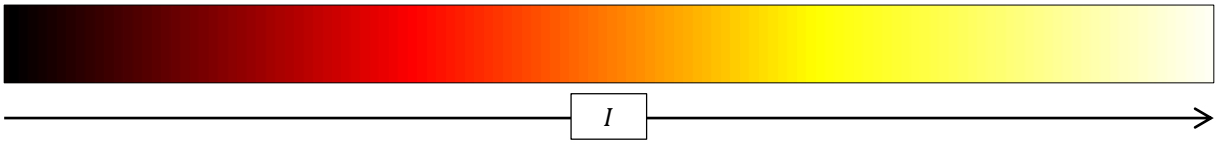


Figure 5.7 – Sun intensity colour gradient.

$$
\begin{aligned}
&\boxed{red = 3 \times \min(I,\ 3^{-1})} \\
&I' = \max(I - 3^{-1},\ 0) \\
&\boxed{green = 3 \times \min(I',\ 3^{-1})} \\
&I'' = \max(I' - 3^{-1},\ 0) \\
&\boxed{blue = 3 \times \min(I'',\ 3^{-1})}
\end{aligned}
\tag{5.11}
$$

5.2.3 Planetary and Ring Shaders

The planetary and ring shaders are responsible for performing the lighting calculations to determine the colour of every pixel in the visible geometry.

Planetary Shader

Rendering a photo-realistic representation of a planet requires understanding what happens to the light rays as they enter the planet's atmosphere, are reflected on the surface, and travel back to the camera. The planet shader process, illustrated in Figure 5.8, splits this process into three stages.

Firstly, the lighting reflection coefficients are calculated, assuming a flat homogeneous surface. The diffuse reflection coefficient

$$
C_d = saturate(\hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{L}}),
\tag{5.12}
$$

where $\hat{\boldsymbol{L}}$ is the direction to the light source and $\hat{\boldsymbol{N}}$ the surface normal, represents the light that is reflected in all angles. The saturate function clamps the values to the $[0,1]$ range. The specular reflection coefficient

$$
C_s = k_s \cdot \hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{L}} \cdot saturate\left(\frac{(1+n)}{8\pi}\frac{(\hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{H}})^n}{(\hat{\boldsymbol{L}} \cdot \hat{\boldsymbol{H}})^3}\right),
\tag{5.13}
$$

where $\hat{\boldsymbol{H}}$ is the halfway versor between camera and light directions, k_s is the material shininess strength, and n its shininess exponent, represents the light that is perfectly reflected by the surface. If a tangent space normal map is provided, the geometric surface normal is disregarded in favour of

$$
\begin{cases}
N_x = 2 \cdot (red \cdot alpha) - 1 \\
N_y = 2 \cdot green - 1 \\
N_z = \sqrt{1 - {N_x}^2 + {N_y}^2}
\end{cases}.
\tag{5.14}
$$

This creates the illusion of three-dimensional terrain on a per-pixel basis. The shader also supports specular mapping. Specular mapping stores the material specular colour, meaning the shininess strength k_s can be controlled on a per-pixel basis to make features such as bodies of water and ice reflective while keeping terrain non-reflective.

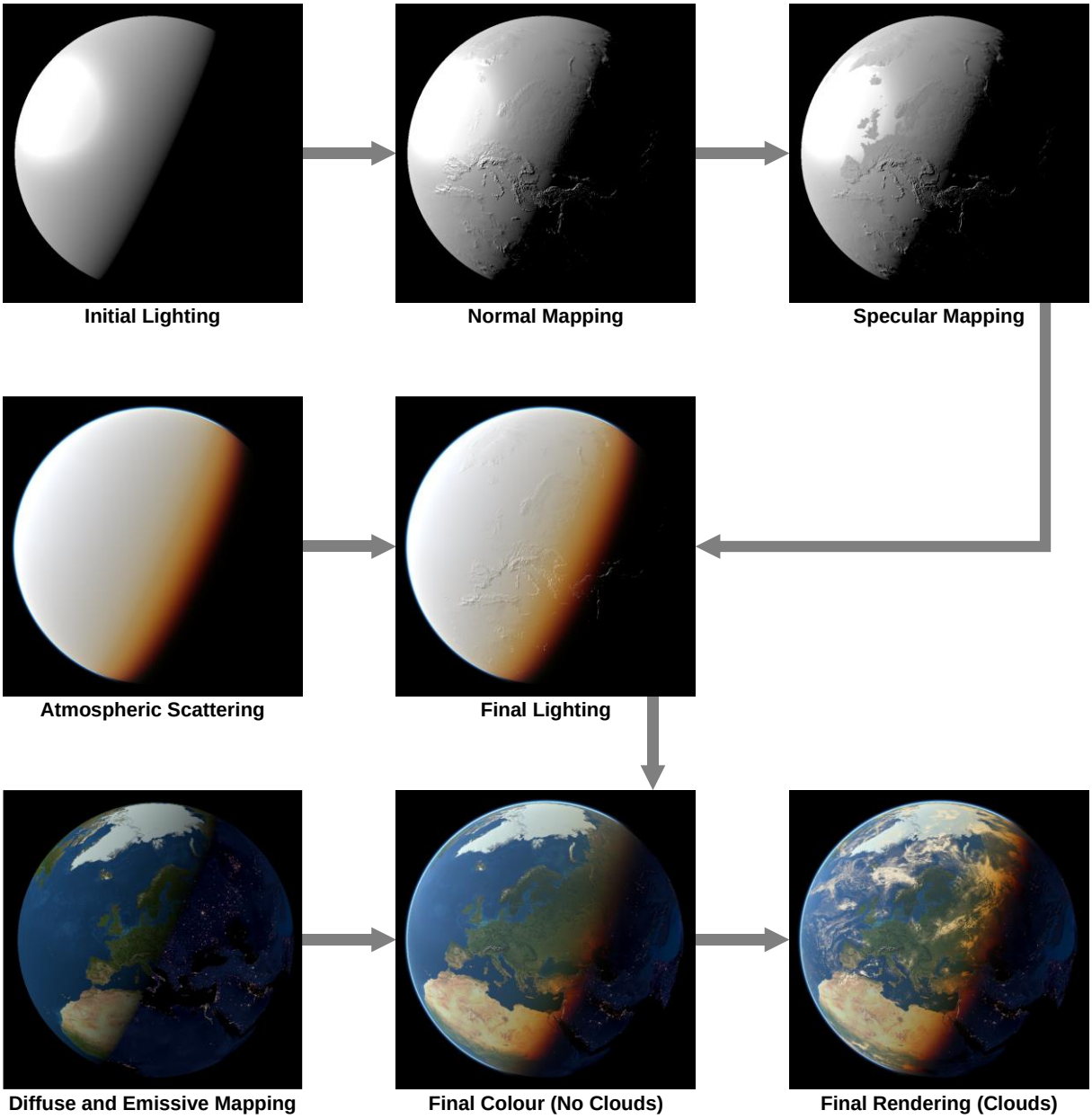


Figure 5.8 – Schematic representation of the planetary shader using planet Earth as an example.

The second stage of the planetary shader is calculating the atmospheric light scattering, which is done in two passes. The first pass extrudes the planet's geometry by a factor of 2.5% and traces the path of a light particle from the Sun to the camera passing through the atmosphere without hitting the planet's surface. The second pass traces the light from the Sun to a position on the surface where it is reflected back to the camera, thus determining how the surface is perceived by the observer. The atmospheric scattering results are combined with the final lighting, resulting in the surface colour

$$\boldsymbol{colour} = C_d + (1 + C_s) \cdot \boldsymbol{colour}_{\boldsymbol{Rayleigh}} + (1 - I_{Mie}) \cdot \boldsymbol{colour}_{\boldsymbol{Mie}}, \tag{5.15}$$

where $\boldsymbol{colour}_{\boldsymbol{Rayleigh}}$ and $\boldsymbol{colour}_{\boldsymbol{Mie}}$ correspond to the RGB colour vectors obtained for Rayleigh and Mie scattering using O'Neal's implementation [34], and the Mie intensity is

$$I_{Mie} = \frac{\|\boldsymbol{colour_{Mie}}\|}{\sqrt{3}}. \tag{5.16}$$

The third stage consists in modulating the surface colour maps with the computed lighting intensity and colour. The diffuse map stores the daytime representation of the planetary surface under isotropic lighting conditions, whereas the emissive map stores the colour and intensity of the light emitted by the planet (e.g. city lights) in the absence of sunlight. The diffuse map is modulated by the diffuse lighting coefficient C_d, and the emissive map by

$$C_e = \frac{saturate^4(1 - 20 \cdot C_d)}{2}, \tag{5.17}$$

when no atmosphere is present, or by

$$C_e = \begin{cases} 0 & , I_{Mie} < 0.05 \\ -\hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{L}} - I_{Mie} & , I_{Mie} \geq 0.05 \end{cases}, \tag{5.18}$$

when an atmosphere is present. Both these coefficients were determined empirically by comparison of the obtained renderings to Earth's photographs.

Finally, the cloud cover is rendered by extruding the geometry to match the user-defined cloud height and rendering $\boldsymbol{colour_{Mie}}$ with the transparency values provided by the alpha channel of the clouds texture map.

Rings Shader

Illuminating planetary rings is a very complex optics problem. Rings are composed by small particles which are illuminated by direct light from the Sun (forward scattering), and light reflected by the planet and the neighbouring ring particles (backscattering). Accurately calculating all the contributions from all the light sources is possible, though technically infeasible. An approximate model was developed to simulate both the forward scattered and the backscattered light. The model provides satisfactory results for a wide range of phase angles, as shown in Figure 5.9, for the rings of Saturn. The forward scattered lighting coefficient is modelled by

$$C_{fs} = 0.1 \cdot \frac{\left(1 + \hat{\boldsymbol{V}} \cdot \hat{\boldsymbol{L}}\right)}{2}, \tag{5.19}$$

where $\hat{\boldsymbol{V}}$ is the versor from the ring position being calculated to the camera position. The backscattered lighting coefficient is

$$C_{bs} = 0.75 \cdot \frac{\left(1 + \boldsymbol{r_R} \cdot \hat{\boldsymbol{L}}\right)}{2 \cdot \|\boldsymbol{R}\|} \frac{\left(1 + \hat{\boldsymbol{V}} \cdot \hat{\boldsymbol{L}}_{\boldsymbol{P}}\right)}{2} \frac{\|\boldsymbol{r_R}\| - R_P}{R_{OR}}, \tag{5.20}$$

where:

- $\boldsymbol{r_R}$ is the ring position;
- $\hat{\boldsymbol{L}}_{\boldsymbol{P}} = R_P \cdot \hat{\boldsymbol{L}} - \boldsymbol{r_R}$ is the direction, from the ring position to planet surface position where the light hits at a normal angle;

- R_P is the equatorial radius of the planet;
- R_{OR} is the outer ring radius.

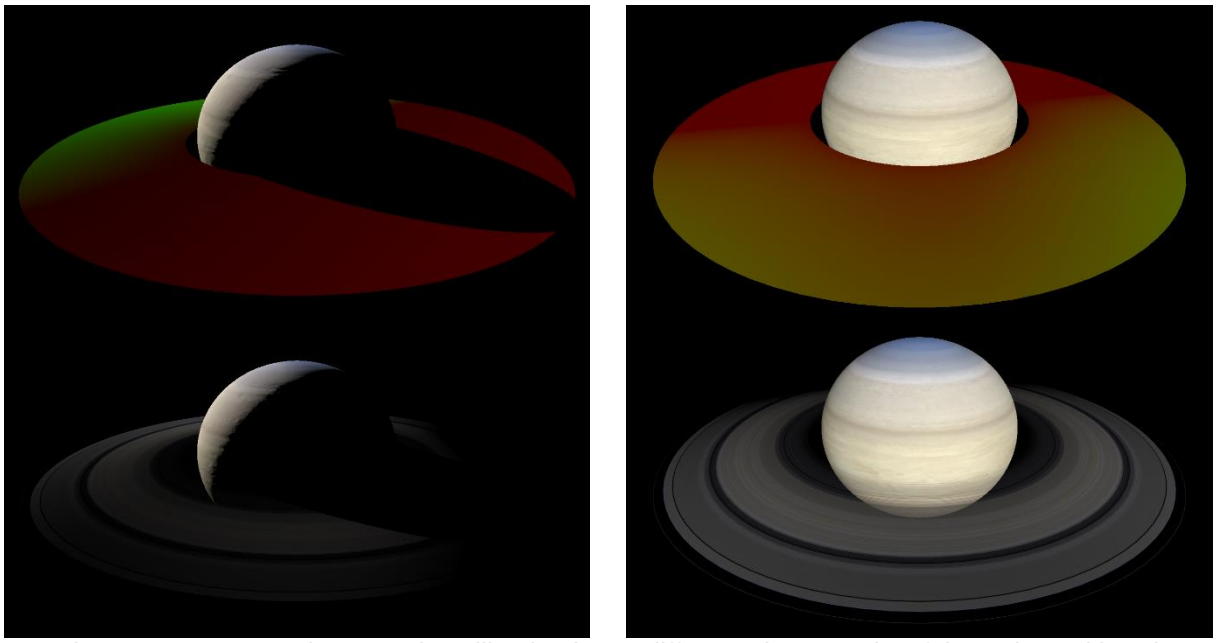


Figure 5.9 – Saturn planetary rings illumination at different phase angles; false colour with red indicating forward scattering and green indicating backscattering (top); real colour (bottom)

5.3 SimOrbit Library (SOLib) and Framework

The SimOrbit Library (SOLib) and Framework is the foundation of the modules. It is a collection of template classes and interfaces that can, and should, be used as building blocks for the creation of GUI dialogs and controls, storage and validation of values, and interfacing with a skeleton graphics engine. Similarly to the SimOrbit project, the metrics were calculated (Table 5.2) and the Kiviat graph was generated (Figure 5.10). SOLib is also a large project, with a high percentage of lines with comments, again due to the XML documentation, and maintainable average metrics. There are a few local maximum cyclomatic complexity and depth issues but the values still ensure maintainability. The Framework is divided in Callbacks, Datatypes, Policies, GUI, DirectX, and Helpers sections.

Table 5.2 – Metrics summary for the SimOrbit Library (SOLib) project.

Parameter	Value
Files	150
Lines	29,797
Statements	8,271
Percent Branch Statements	14.5
Percent Lines with Comments	35.6
Classes Defined	211
Methods Implemented per Class	5.40
Average Statements per Method	4.2
Maximum Complexity	23
Maximum Block Depth	7
Average Block Depth	1.29
Average Complexity	2,27
Functions	109

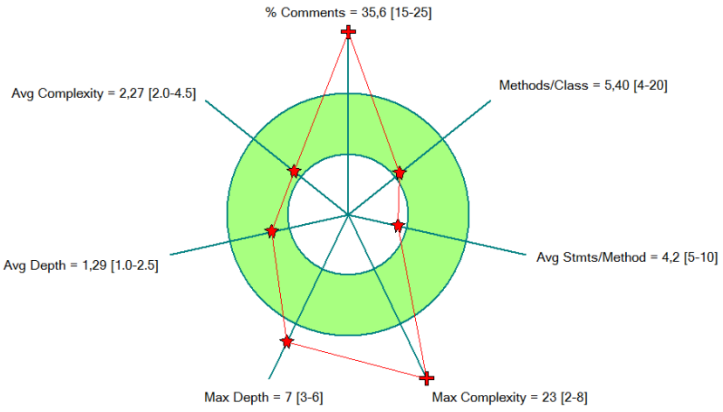


Figure 5.10 – Kiviat metrics graph for the SimOrbit Library (SOLib) project.

5.4 SPICE C++ Wrapper Library

A C++ thread-safe wrapper library was developed for selected SPICE Toolkit routines (cf. section 3.2.2), namely those pertaining to kernel management, NAIF codes, conics, constants and ephemerides retrieval, frame transformations, and SPK file generation. The library allows the SPICE Toolkit to be used in a thread-safe environment by ensuring mutual exclusion on critical sections. It also implements an exception error system that is fully compatible with C++ exception specifications. Furthermore, it adds support for safe function calls by using template classes like *vector* and *valarray* to input and output data from functions, instead of the raw arrays the SPICE Toolkit uses.

The code metrics were calculated (Table 5.3), and the Kiviat graph was generated (Figure 5.11). The SPICE C++ Wrapper Library is much smaller than the other two developed components, as it only encapsulates the functionality already present in the SPICE Toolkit, which is further confirmed by the low number of average statements per method. Nevertheless, it exports 149 functions, which correspond to the most used functions in the SPICE API.

Table 5.3 – Metrics summary for the SPICE C++ wrapper project.

Parameter	Value
Files	37
Lines	5,464
Statements	1,278
Percent Branch Statements	7.7
Percent Lines with Comments	38.4
Classes Defined	10
Methods Implemented per Class	4.43
Average Statements per Method	2.9
Maximum Complexity	22
Maximum Block Depth	6
Average Block Depth	1.32
Average Complexity	1.49
Functions	149

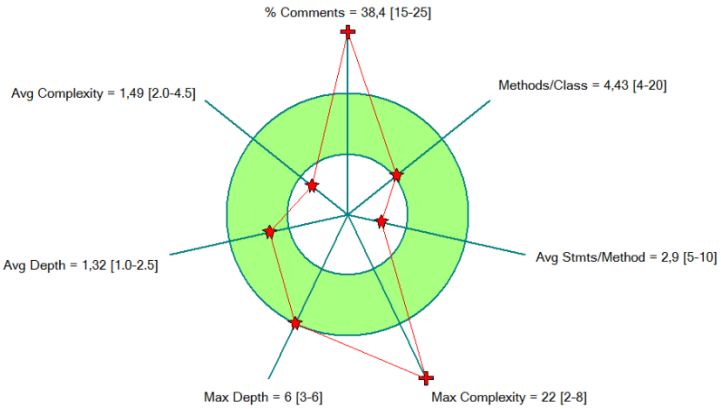


Figure 5.11 – Kiviat metrics graph for the SPICE C++ wrapper project.

Chapter 6

VALIDATION AND TESTING

6.1 Trajectory Propagator Validation

In order to be able to use the trajectory propagator in an operational environment it is necessary to perform an extensive validation process to ensure there are no errors which can compromise a space mission. The validation presented in this section is preliminary, and its purpose is only to confirm that the force model is capable of accurately calculating simple trajectories for which there are analytical solutions. The precision of the floating-point significant used throughout the validation process was 53 bits, conforming to the IEEE 754 double-precision floating point format [17].

6.1.1 Keplerian Orbit

The most basic test of the trajectory propagator consists in verifying the orbital stability of a Keplerian orbit. To achieve this all perturbations were disabled, leaving only the spherical gravity field of the central body. A set of orbital elements, for an Earth orbit, was chosen:

- Semi-major axis $a = 30000km$
- Eccentricity $e = 0.05$
- Inclination $i = 15°$
- Longitude of ascending node $\Omega = 60°$
- Argument of periapsis $\omega = 30°$
- Mean anomaly at epoch $M_0 = 0°$

The trajectory was propagated for a duration of 100 days with a fixed time step of 10 minutes, and the force system was configured to disable all perturbations. The orbital elements residuals were calculated for each data point using

$$\varepsilon_i = X_0 - X_i, \tag{6.1}$$

where X represents an arbitrary orbital element, and i is the index of the trajectory data point.

Table 6.1 – Statistical analysis of the residuals of the classical orbital elements during a 100 days propagation of a Keplerian orbit.

Element	Mean (μ)	Standard Deviation (σ)
Semi-major axis a	-3.0337e-03	7.2842e-03
Eccentricity e	-4.5622e-08	1.4529e-07
Inclination i	2.5361e-07	1.3024e-06
Longitude of ascending node Ω	2.2247e-06	4.7369e-06
Argument of periapsis ω	9.1730e-07	1.6502e-04
Mean anomaly at epoch M_0	4.6052e-03	2.6692e-03

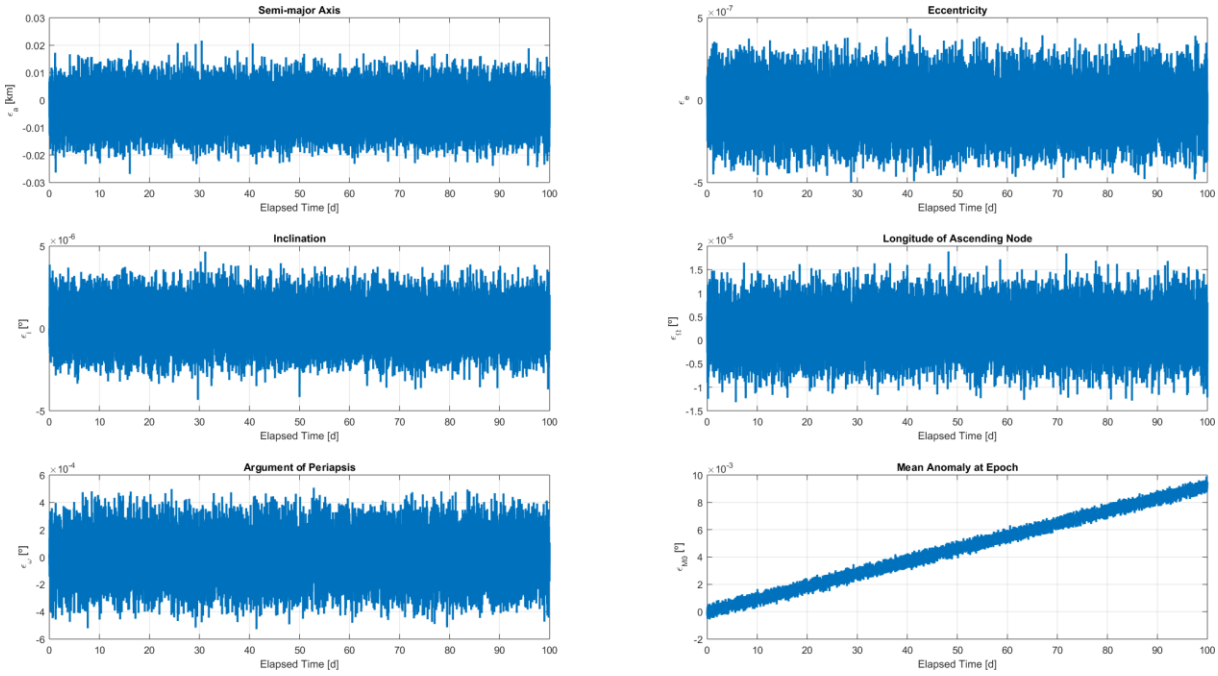


Figure 6.1 – Residuals of the classical orbital elements during a 100 days propagation of a Keplerian orbit.

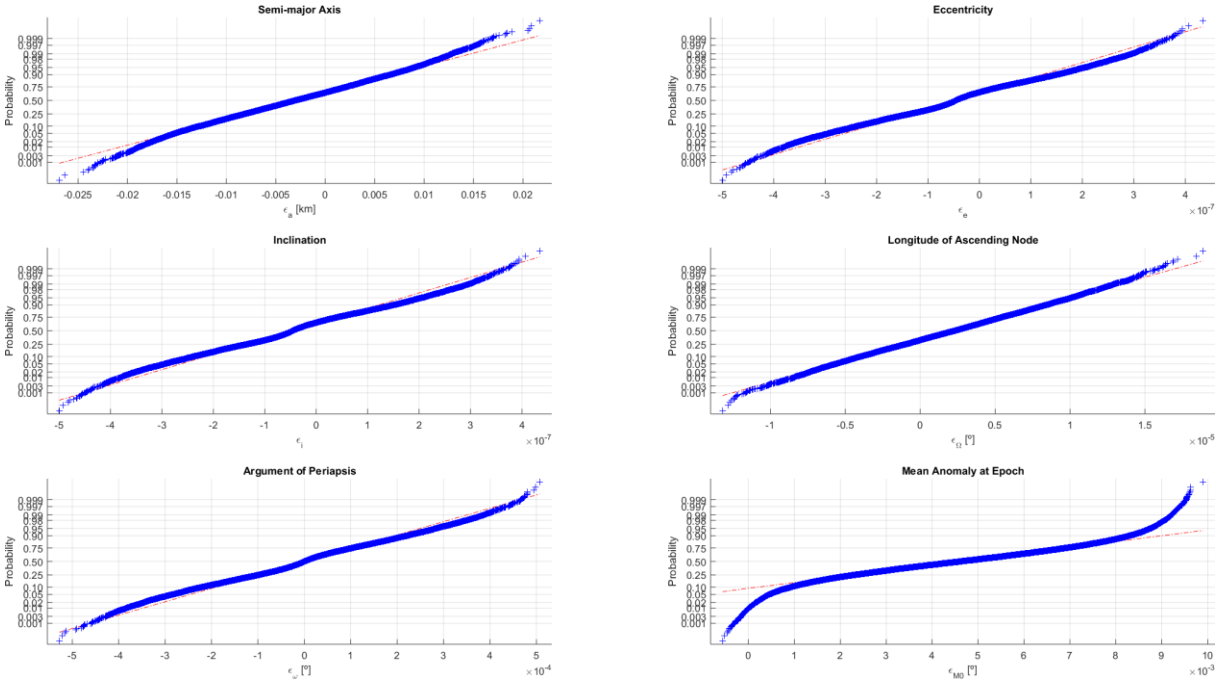


Figure 6.2 – Normal probability plots for the residuals of the classical orbital elements during a 100 days propagation of a Keplerian orbit.

The residuals were plotted, as illustrated on Figure 6.1, their mean and standard deviation were calculated (Table 6.1), and their normal probability plots were generated (Figure 6.2). Their analysis reveals minor, well-behaved, normally distributed residuals for all elements with the exception of the mean anomaly at epoch (M_0). This suggests the orbit is stable with near-constant orbital parameters,

as expected, but the mean motion of the spacecraft is higher than that of the analytical solution. Nevertheless, the accumulated error of the mean anomaly at epoch over the course of the 100 days is of only $9.2606 \times 10^{-3}°$, well within an acceptable tolerance.

6.1.2 J_2 Perturbation

For a satellite orbiting a central body, the J_2 perturbation causes a variation of the longitude of ascending node

$$\Delta\Omega = -\frac{3}{2}\frac{nJ_2R_E^2}{a^2(1-e^2)^2}cos\, i, \tag{6.2}$$

and a variation of the argument of periapsis

$$\Delta\omega = \frac{3}{2}\frac{nJ_2R_E^2}{a^2(1-e^2)^2}\left(2-\frac{5}{2}\sin^2 i\right), \tag{6.3}$$

where the mean orbital velocity is

$$n = \sqrt{\frac{GM}{a^3}}. \tag{6.4}$$

Two types of orbits were used to validate the J_2 contribution – a sun-synchronous orbit, and a Molniya orbit. Sun-synchronous satellites orbit the Earth in a way that they appear to always orbit the same position from the perspective of the Sun, i.e. the orbital plane rotates at the rate of $0.9856° \cdot \mathrm{day}^{-1}$. The Molniya satellites are placed in highly elliptical orbits with an inclination near $63.4°$. At this inclination the argument of periapsis is not perturbed by the J_2 coefficient of the Earth's gravitational field, thus remaining constant.

The sun-synchronous satellite AQUA was selected to test the variation of the longitude of ascending node. The TLE set describing the initial conditions (Table 6.2) was obtained from CelesTrak [45].

Table 6.2 – AQUA satellite TLE set used for validating the propagator.

```
AQUA
1 27424U 02022A   15130.82927265  .00000847  00000-0  19781-3 0  9994
2 27424  98.1932  72.3717 0001448  53.4004  32.2452 14.57133010692384
```

The TLE set was converted to the orbital elements:

- Epoch $E_0 = 2457153.330050\, JD$ (10-05-2015 19:55:16 UTC)
- Semi-major axis $a = 7068.6376km$
- Eccentricity $e = 0.001448$
- Inclination $i = 98.1932°$
- Longitude of ascending node $\Omega = 72.3717°$
- Argument of periapsis $\omega = 53.4004°$
- Mean anomaly at epoch $M_0 = 32.2452°$

The trajectory was propagated for a duration of 100 days with a fixed time step of 10 minutes, which is sufficient to guarantee any significant errors are introduced by the force model, and not the numerical integrator (i.e. decreasing the time step does not change the output). The force system was configured to only include the Earth's gravity field. Figure 6.3 shows the comparison of the solution given by the software to the analytical one. As expected the numerical solution diverges from the analytical one as it integrates errors over time, however these were considered to be within an acceptable tolerance. A linear regression of the numerical solution reveals a $\Delta\Omega_{SimOrbit} = 0.9681°day^{-1}$ (coefficient of determination $R^2 = 1$), whereas the analytical solution has a $\Delta\Omega = 0.9915°day^{-1}$. The coefficient of correlation between the solutions is $R = 1.0000$.

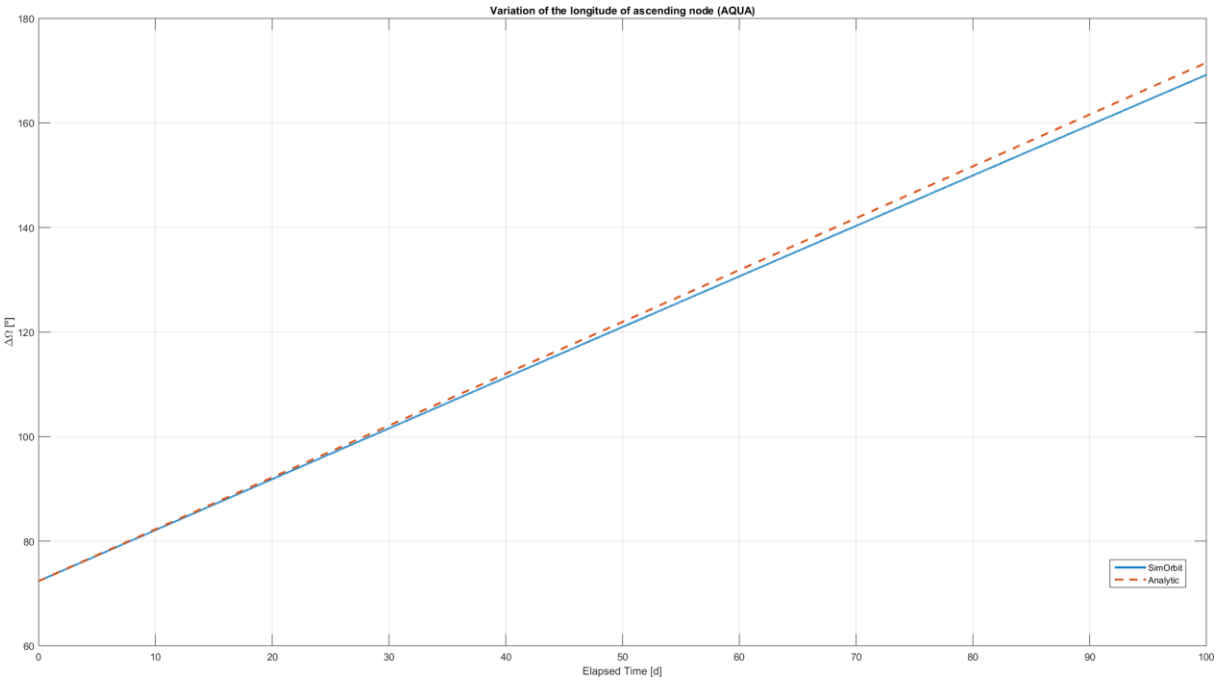


Figure 6.3 – Comparison of the variation of the longitude of ascending node of the AQUA satellite orbit during a 100 days trajectory propagation and the analytic solution.

The MOLNIYA 1-81 satellite was selected to test the variation of the argument of periapsis, since from all the operational Molniya satellites, its inclination was the one closest to $63.4°$. Table 6.3 lists the TLE set used for the initial conditions of the propagator, also obtained from CelesTrak.

Table 6.3 – MOLNIYA 1-81 satellite TLE set used for validating the propagator.

```
MOLNIYA 1-81
1 21426U 91043A   15129.48198403 -.00000182  00000-0 -58611-2 0  9993
2 21426  63.2998 287.3923 7218024 283.8640  13.1449  2.00622014175072
```

The associated classical orbital elements are:

- Epoch $E_0 = 2457151.982762\ JD$ (09-05-2015 11:35:11 UTC)
- Semi-major axis $a = 26607.835km$
- Eccentricity $e = 0.7218024$
- Inclination $i = 63.2998°$

- Longitude of ascending node $\Omega = 287.3923°$
- Argument of periapsis $\omega = 283.8640°$
- Mean anomaly at epoch $M_0 = 13.1449°$

The trajectory was propagated exactly as before, and Figure 6.4 shows the comparison of the solution given by the software with the analytical one. Here the error increases linearly over time, and the linear regression of the numerical solution reveals a $\Delta\omega_{SimOrbit} = 0.0007° day^{-1}$ $(R^2 = 0.9)$, whereas the analytical solution has a $\Delta\omega = 0.0013° day^{-1}$. The coefficient of correlation between the two solutions is $R = 0.9487$.

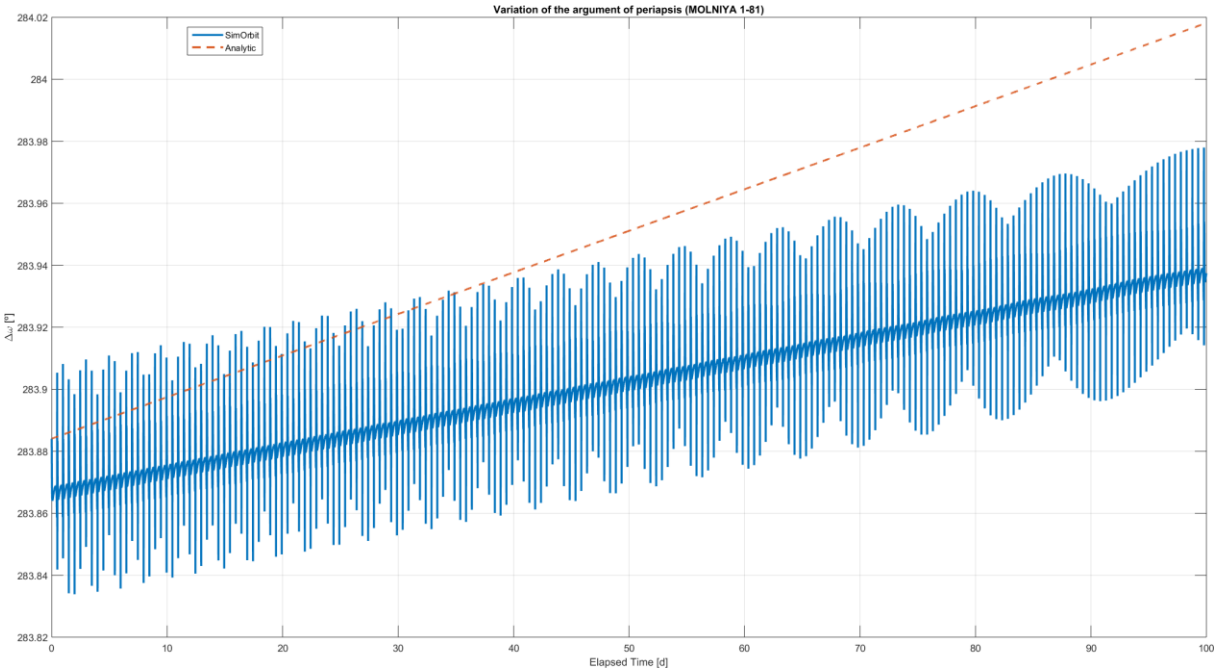


Figure 6.4 – Comparison of the variation of the argument of periapsis of the MOLNIYA 1-81 satellite orbit during a 100 days trajectory propagation and the analytic solution.

Taking into account the results obtained for both the AQUA and the MOLNIYA 1-81 satellites, the J_2 perturbation implementation was considered valid.

6.1.3 Other Perturbations (Third Bodies and Solar Radiation Pressure)

Testing the trajectory perturbations introduced by third bodies and the solar radiation pressure is a more complex problem, for which no analytical solution exists. Ideally, the software should be tested against previously validated software, such as those presented in Section 1.2. Unfortunately, due to time constraints that was not possible. Nevertheless, it was necessary to verify that at least the influence of third-bodies was being taken into account and reasonably calculated.

To verify the influence of third-bodies a trans-lunar trajectory was propagated. This is a type of free return trajectory which takes a spacecraft from a circular parking orbit around the Earth, around the far side of the Moon, and back to Earth without requiring any propulsion, except that required to initially set the trajectory. This trajectory was widely used in the Apollo missions, as it allowed the astronauts to

return to Earth in case of a propulsion failure. The Apollo 11 was one of the missions where this type of trajectory was used, and is one for which an abundance of information is available, making it ideal for testing.

The Apollo 11 trans-lunar injection (TLI) parameters [46] were converted to the orbital elements:

- Epoch $E_0 = 2440419.182095\,JD$ (16-07-1969 16:22:13 UTC)
- Semi-major axis $a = 286535.800km$
- Eccentricity $e = 0.976965$
- Inclination $i = 31.383°$
- Longitude of ascending node $\Omega = 358.380°$
- Argument of periapsis $\omega = 4.410°$
- Time of periapsis $T_p = 2440419.180255\,JD$ (16-07-1969 16:19:34 UTC)

The spacecraft was considered to have a mass of $m = 43866kg$, corresponding to the mass of the Command and Service Module (CSM) and the mass of the Lunar Module (LM) combined, a projected area of $A = 27m^2$ and a reflectivity of $\varepsilon = 0.5$. The numerical integrator was set up with an initial time step of 1 minute with lower and upper bounds of 1 second and 1 hour, respectively, and truncation error tolerances of $tol_{pos} = 1 \times 10^{-6}m$ and $tol_{vel} = 1 \times 10^{-6}m \cdot s^{-1}$. The trajectory was propagated for a duration of duration of 5.8 days, with all Earth's force system perturbations enabled.

The propagated trajectory, shown in Figure 6.5, is as expected. The spacecraft leaves Earth, performs a swing-by at the Moon, and returns. Despite the lack of numerical validation of the trajectory, it is enough to verify that the third-body perturbation due to the presence of the Moon is being accurately taken into account.

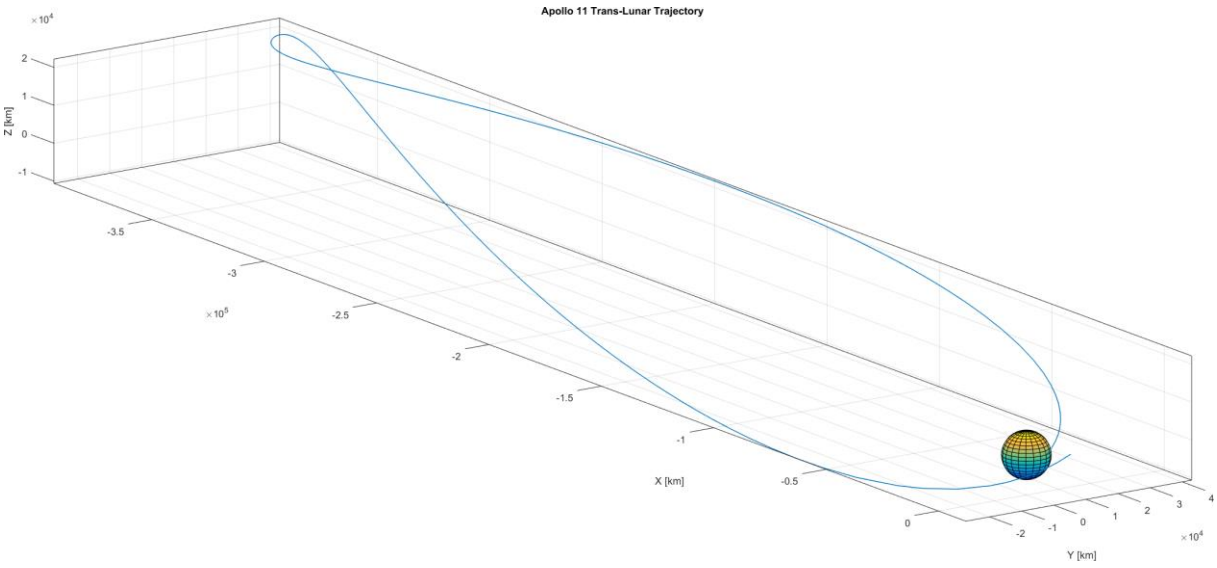


Figure 6.5 – Apollo 11 trans-lunar trajectory as propagated by SimOrbit.

6.1.4 Engine

Lastly, it was necessary to test the engine implementation. Ideally, the two modes – delta-V and constant accelerations would have to be tested, however, there is no analytic method for testing constant acceleration inputs. Hence, only the delta-V inputs were tested by propagating a Hohmann transfer, which is a manoeuvre that takes a spacecraft from one circular orbit to another, on the same plane, using two delta-V impulses.

The delta-V required to leave the initial circular orbit of radius r_1 and enter the elliptical transfer orbit is

$$\Delta v_1 = \sqrt{\frac{GM}{r_1}}\left(\sqrt{\frac{2r_2}{r_1 + r_2}} - 1\right), \tag{6.5}$$

and the one required to leave the elliptical transfer orbit and enter the final circular orbit of radius r_2 is

$$\Delta v_2 = \sqrt{\frac{GM}{r_2}}\left(1 - \sqrt{\frac{2r_1}{r_1 + r_2}}\right). \tag{6.6}$$

The time taken to perform the transfer is

$$t_H = \pi\sqrt{\frac{(r_1 + r_2)^3}{8GM}}. \tag{6.7}$$

From an initial circular Low Earth Orbit (LEO) described by:

- Semi-major axis $a = 7378.1366km$ (altitude of $1000km$ above the equator)
- Eccentricity $e = 0$
- Inclination $i = 0°$
- Longitude of ascending node $\Omega = 0°$
- Argument of periapsis $\omega = 90°$

It was decided to use the engine to transfer to a Geosynchronous Equatorial Orbit (GEO), which has an altitude of $35{,}786\text{km}$ above the equator. This results in $r_1 = 7378.1366km$ and $r_2 = 42164.1366km$. Using Equations (6.5), (6.6), and (6.7) the $\Delta v_1 = 2.23932km \cdot s^{-1}$, $\Delta v_2 = 1.3966km \cdot s^{-1}$, and $t_H = 19399.9178s$ were obtained.

The engine was configured with the two delta-V, using the TRAJECTORY frame of reference, and the trajectory was propagated for the duration of 3 days, starting 1 day before the first delta-V. The time step was fixed to 10 minutes. The resulting trajectory is represented on Figure 6.6, and is exactly what was expected.

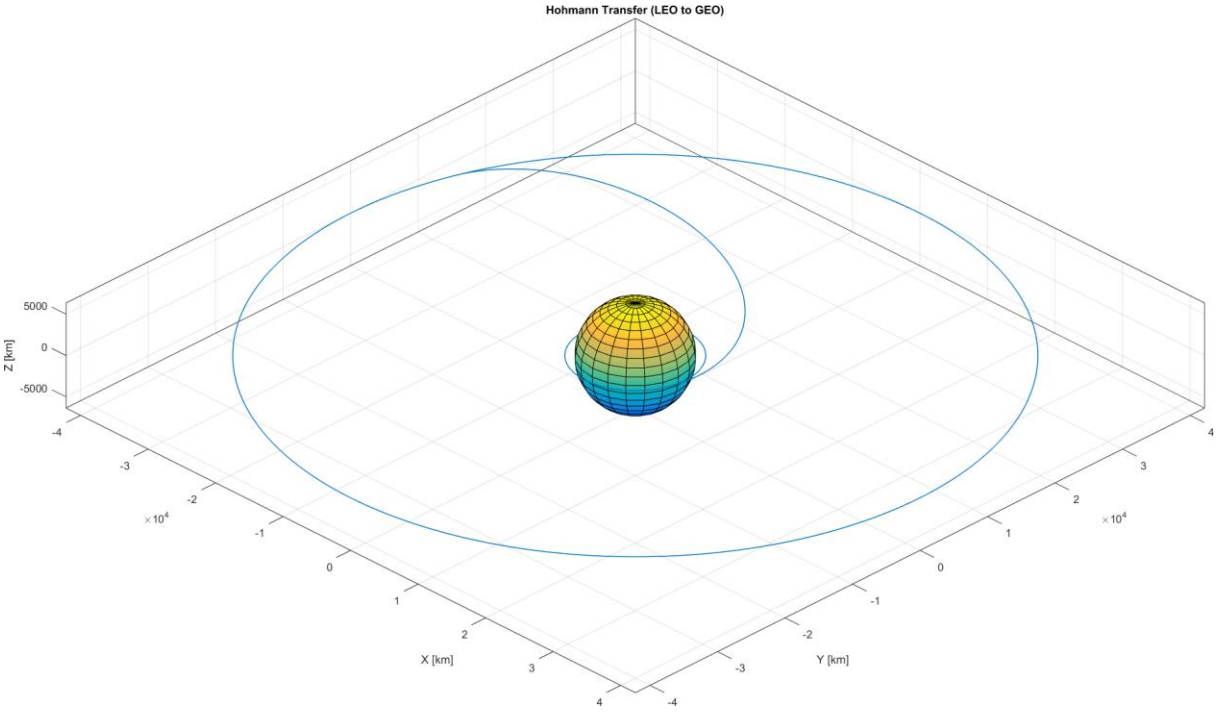


Figure 6.6 – Hohmann transfer orbit from Low Earth Orbit (LEO) to Geosynchronous Earth Orbit (GEO).

The final orbit has the parameters:

- Semi-major axis $a = 42164.1365km$ (altitude of $35785{,}9999km$ above the equator)
- Eccentricity $e = 1.4330 \times 10^{-7}$
- Inclination $i = 0°$
- Longitude of ascending node $\Omega = 0°$
- Argument of periapsis $\omega = 97.9235°$

The semi-major axis error is $\varepsilon_a = 1 \times 10^{-4}km$, and the eccentricity error is $\varepsilon_e = -1.4330 \times 10^{-7}$, were both considered to be well within an acceptable tolerance. This verifies the engine's delta-V implementation is valid.

6.2 Performance

The SimOrbit performance was assessed on an Intel Core i7-5600U CPU @ 2.60GHz, with 8GB RAM, and an integrated Intel[HD Graphics 5500 graphics processor, running the Windows 7 Professional 64-bit operating system. The display resolution was 2560x1440 pixel and the desktop scaling factor, which controls the scaling of the GUI, was set to 125%. The executable used for testing was built with full program optimization and for a 64-bit system.

The software was profiled and optimized using the Visual Studio Performance and Diagnostics tool. The CPU sampling method was used to discover "hot" paths and lines of code that reduced performance, and the resource contention data for concurrency analysis.

6.2.1 Propagator

To assess the propagator module performance, a trajectory propagation of an Earth satellite with a duration of one sidereal year, was used. The force system was configured to account for all perturbations (i.e. Earth J_2 coefficient, Moon and Sun as third bodies, and solar radiation pressure). Additionally, a very low-thrust constant acceleration engine input was added to be able to measure the performance of the engine. Delta-V inputs were not added since they have a negligible performance impact due to only being calculated at one specific time instance. The initial time step was set to 10 minutes and was left unbounded.

Firstly, the results of the CPU sampling were analysed – one "hot path" was found. The propagator worker thread function was responsible for 96.9% of the CPU usage, with 73.0% corresponding to the RKF7(8) integrator and 22.6% to the dynamic force system change algorithm. Looking inside the integrator it was found that 64.3% of the total CPU usage came from that force model implementation (i.e. the code responsible for calculating the derivatives the numerical integrator uses), and that the *asin* and *acos* operations were extremely intensive on their own.

To gain a better understanding of what force model components were more expensive to compute, several performance counters, based on the High Precision Event Timer (HPET), were added to the force model code. These are hardware-implemented timers present in most modern computers. The implemented counters measured how long it took to execute the code that calculates each component of the acceleration (central body gravity field and J_2 perturbation, third bodies' perturbations, solar radiation pressure perturbation, and engine).

Both a generic and the Sandy Bridge microarchitecture optimized DLLs for the MPIR and MPFR libraries were used. The output of the performance counters is shown in Table 6.4. Through the analysis of the performance counter values it is obvious the most time consuming computations are the central body J_2 and solar radiation pressure perturbations. A possible reason behind this are the mathematical operations involved – the J_2 perturbation requires two 3x3 matrix multiplications and five *pow* operations, and the solar radiation pressure perturbation as many as two *asin* and three *acos* operations. The usage of the microarchitecture optimized DLLs did not increase the performance significantly, and only revealed itself useful in the solar radiation pressure computation. This conclusion cannot however be extrapolated to other CPU microarchitectures where the performance gain may be greater.

Table 6.4 – Force model performance counter results for the generic and Sandy Bridge DLLs and their respective improvement.

Performance Counter	Generic DLLs [s]	Optimized DLLs [s]	Difference [%]
Central body GM	4.5101	4.3952	-2.5476
Central body J_2	38.6601	38.9222	+0.6780
Third bodies GM	14.5795	14.6385	+0.4030
Sun GM	14.1066	13.9925	-0.8088
Solar radiation pressure	52.8765	49.4690	-6.4443
Engine	6.7374	6.7632	+0.3829
Total	**133.7283**	**130.6471**	**-2.3041**

6.2.2 Simulator

The simulator module testing focused on measuring the frames per second (FPS) under different conditions to try and assess the individual performance of the shaders, and the influence movie capturing had on graphics performance. The Direct3D 10 mode was used in the tests, since it was supported by the graphics processor, and the FPS was measured using the Visual Studio Graphics Analyzer.

Windowed Mode (1600x900 pixel)

The first test was run on windowed mode at 1600x900 pixel, which is SimOrbit's default window size under a desktop scaling factor of 125%. A sequence of six automated cameras that focused on objects being rendered by different types of shaders were defined. Each camera was set to be active for a total of 30 seconds. The sequence was ran twice, with native resolution movie capture off and on. Table 6.5 lists the measured average FPS values and the influence of movie capturing.

Table 6.5 – Average FPS measured for different types of shaders on a 1600x900 pixel window, with native resolution movie capturing off and on.

Shader	Normal Rendering [FPS]	Movie Capturing [FPS]	Difference [FPS]
None (deep space)	319.4	125.4	-194.0
Sun (Sun)	97.7	**59.6**	-38.1
Planet (Venus)	247.7	95.7	-152.0
Planet with atmosphere (Earth)	109.4	**55.8**	-53.6
Rings (Saturn)	226.4	106.7	-119.7
Model (Cassini)	265.7	100.4	-163.3

The worst performance was found on the "Sun" (i.e. the Perlin noise implementation) and the "Planet with atmosphere" shaders, which is not surprising since both are computationally expensive algorithms. Nevertheless, the performance of all shaders is very good. Capturing a movie, even at the native resolution, causes a significant drop in FPS. In order to determine exactly how much of an influence the movie capture has, the average frame time (i.e. the time required to render the frame)

$$t_{frame} = \frac{1}{FPS}, \tag{6.8}$$

was calculated. Averaging the frame time difference between both cases, a value of $6.29ms$ was found for the time it takes to capture a single 1600x900 pixel frame to a movie, corresponding to $4.37ms \cdot MP^{-1}$ (millisecond per megapixel).

Full Screen Mode (2560x1440 pixel)

Subsequently, full screen mode was tested at the 2560x1440 pixel, commonly known as Quad HD (QHD). This test was meant to put as much strain as possible on the graphics processor and determine the lower FPS limits at which a simulation would run. The camera sequence and methodology from the previous test were used, and the results are shown in Table 6.6.

As expected, due to the 2.56 times higher resolution, the FPS are lower than those of the previous test. The relative shader performance remains constant. Movie capture at this resolution was found to reduce the FPS to very low values, where camera movement causes noticeable flicker. The average increase in frame time due to movie capturing was of $20.29ms$, corresponding to $5.56ms \cdot MP^{-1}$, a performance slightly lower than observed before, which suggests the encoding time varies non-linearly with the resolution.

Table 6.6 – Average FPS measured for different types of shaders on a 2560x1440 pixel screen, with native resolution movie capturing off and on.

Shader	Normal Rendering [FPS]	Movie Capturing [FPS]	Difference [FPS]
None (deep space)	208.7	**39.3**	-169.4
Sun (Sun)	**56.2**	**25.4**	-30.8
Planet (Venus)	167.9	**36.9**	-131.0
Planet with atmosphere (Earth)	60.4	**31.1**	-26.6
Rings (Saturn)	153.4	**35.6**	-117.8
Model (Cassini)	175.6	**37.2**	-138.4

Chapter 7

CONCLUSIONS

This thesis was aimed at developing a free and open-source software package to support mission analysis and orbital mechanics calculations. In order to provide an integrated solution, enabling a user to generate and analyse data, two components were proposed – a trajectory propagator for data generation, and a 3D visualization component. The trajectory propagator was meant to be able to generate accurate spacecraft trajectory data based on a set of initial conditions describing its state (i.e. position and velocity) at a reference epoch. The 3D visualization component was envisioned as a visually realistic 3D rendering of the Solar System for spacecraft motion analysis.

A review of modern software for mission analysis and design was performed, encompassing the major general tools available in the industry – STK, FreeFlyer, GMAT, and STA. Celestia, one of the most recognized software tools whose primary goal is a visually realistic rendering of the Solar System, was also reviewed. This review was extremely useful towards gaining an understanding of the current approach used by the industry in the development of this type of tool. Additionally, common problems that hinder mission analysis and design software usability were also identified. On a lesser extent, the review also facilitated the familiarization with the common terminology, standards, and file formats used in the industry.

With the knowledge gained from the software review, and the initial goals in mind, a set of requirements for the software were defined. These outlined what was necessary to implement in order to provide adequate functionality, and what was desirable to have to further assist the user. Additionally, given the fact this was a large open source software project, the requirements also took into account the needs of third-party developers – low-complexity, maintainable, and well documented source code. For the definition of the trajectory propagator requirements, a brief study space missions mission scenarios (i.e. use cases) was conducted. This determined what fundamental forces and engine types needed to be included for accurate propagation of a trajectory. Additionally, the study also highlighted the necessity of defining a collection of different force systems, each valid within a region of space, and having the propagator switch between them automatically. The requirements for the visualization component were defined based on its primary goal of visual realism, and the necessity of also displaying auxiliary features to aid mission analysis. The need for controlling the point-of-view was also established, leading to the definition of two types of camera control modes – manual and automatic. Furthermore, a method of exporting the visualization component output in the form of a movie, was also proposed.

The software was designed using the waterfall model, which proposes a sequential development process in which the software is designed (based on the requirements), coded, and tested before release. On the design stage the software's name, programming language and development environment, third party components, architecture, coding standards, documentation, and licensing and

distribution method were discussed and chosen. It was decided to name the software SimOrbit, and release it under the European Union Public License (EUPL) v.1.1, via a website.

Three software components were developed in C++ programming language – the SimOrbit application, the SimOrbit Library and Framework, and the SPICE C++ wrapper. The SimOrbit application is an executable that encompasses the trajectory propagator and 3D visualization modules. It is the implementation of the user-end requirements. The SimOrbit Library and Framework is the realization of the developer-requirements, providing an interface framework that allows future developer to effortlessly integrate new functions (i.e. modules) within SimOrbit. The SPICE C++ wrapper is a library that improves the NAIF's SPICE Toolkit compatibility with the C++ programming language, and provides thread-safety.

Finally, the trajectory propagator was validated and the software performance was tested. The trajectory propagator validation focused comparing its results with known analytic solutions for simple orbits. The algorithms for the central body's gravity field, including the J_2 perturbation, and the engine delta-V inputs were tested and found to be within acceptable limits. A tentative test of the influence of third bodies and the solar radiation pressure was also conducted. The test consisted in propagating the Apollo 11 trans-lunar trajectory, and visually comparing it to the known profile. The propagator was considered valid, having successfully passed all tests. However, the validation process was not extensive, which precludes the trajectory propagator from being used operationally.

In order to assess software performance, the propagator was instrumented with performance counters that measured the computational time taken calculating the different components of the force model. The solar radiation pressure and the J_2 perturbation were found to be the most computationally expensive components, consuming 39.54% and 28.91% of the total computation time, respectively. The performance gain using optimized MPIR and MPFR DLLs was also measured and found to be negligible for the test environment. The performance of the simulator was also evaluated by measuring the FPS of the individual shaders under different rendering conditions. It was found that all shaders perform adequately, although the Perlin noise and atmospheric scattering perform considerably worse than any other shader. Movie making was found to reduce the FPS since it requires the render target data to be copied to the movie frame buffer and encoded.

Although the developed software conforms to the specifications, performs adequately, and has been preliminary validated, it was found to have some flaws. On the propagator module, a very short constant acceleration engine input (i.e. with a time span with an order of magnitude similar or lower than that of the time step) can be leaped over and ignored by the numerical integrator. In order to mitigate this problem the software resets the initial time step at the start of the engine input, but this was found to be inadequate to solve the issue. A recommendation was included in the user manual to use delta-V inputs, instead of constant acceleration, in these cases. The propagator should also be modified to allow for the propagation of spacecraft attitude and mass in addition to the position and velocity. This would greatly improve the accuracy of the solar radiation pressure component, and allow for propellant calculations. On the simulator module, loading a spacecraft trajectory is a time consuming operation, and for very long or complex trajectories, limited by the maximum number of points, it does not always

produce an accurate representation of the reality. This issue needs further studying, possibly creating an algorithm that refines the trajectory depending on the camera position. Furthermore, also on the simulator module, the Sun's corona shader has the lowest performance of all implemented shaders and is not physically-based. A study of the corona's lighting properties needs to be performed in order to create a fast physically-based algorithm that improves both the visual realism and performance of this shader.

In addition to correcting the flaws currently present in the implemented modules it is also desirable to add new functionality, in the form of new modules. Two modules were already considered for future development – trajectory optimization and ground track visualization. Trajectory optimization is a feature of most modern mission analysis and design tools, and is of the utmost necessity for the design of space missions. It determines the ideal launch window, and minimizes the engine thrust necessary to reach the mission objective, under a set of constraints. Ground track visualization is especially useful for satellites as it facilitates the determination of communication windows between the satellite and the surface tracking stations.

The design and development of the software was a challenging endeavour, as it draws knowledge from several distinct areas – orbital mechanics, programming, computer graphics, optics, and media generation. The work also greatly contributed towards understanding the difficulties associated with large software projects, and their planning and time management requirements. Extensive computer programming knowledge, namely in the C++11 and HLSL programming languages, was also acquired. Furthermore, the importance of defining requirements, establishing standards, and producing appropriate documentation was learnt.

The main challenge now lies in fully validating SimOrbit's computations, approving it for operational usage, and gaining industry acceptance. An early version of SimOrbit was presented at the 4th International Conference on Astrodynamics Tools and Techniques (ICATT) [47], and since then its overall functionality and visual realism of the simulator module has greatly increased.

Overall, the software is considered to successfully fulfil the proposed objective, allowing a user to accurately propagate and visualize the trajectory of spacecraft in a visually realistic environment.

References

[1] James R. Wertz and Wiley J. Larson, Eds., *Space Mission Analysis and Design*, 3rd ed.: Springer, 1999, Space Technology Library.

[2] Analytical Graphics, Inc. (2013, June) Analytical Graphics, Inc. [Online]. http://www.agi.com

[3] Charles Deveas et al., "Flight Dynamics System Design," in *2012 IEEE Systems and Information Engineering Design Symposium*, Charlottesville, VA, USA, 2012, pp. 56-61. [Online]. http://dx.doi.org/10.1109/SIEDS.2012.6215136

[4] a.i. solutions. (2014, July) a.i. solutions. [Online]. http://www.ai-solutions.com

[5] Steven P. Hughes, "General Mission Analysis Tool (GMAT)," NASA Goddard Space Flight Center, Greenbelt, MD, USA, 2007. [Online]. http://dx.doi.org/2060/20080045879

[6] NASA Goddard Space Flight Center. (2013, June) GMAT Wiki. [Online]. http://gmatcentral.org/

[7] Guillermo Ortega et al., "STA, The Space Trajectory Analysis Project," in *4th International Conference on Astrodynamics Tools and Techniques*, Madrid, Spain, 2010.

[8] Chris Laurel and Jason Perry, "Spaceflight Visualization with Celestia," in *4th International Conference on Astrodynamics Tools and Techniques*, Madrid, Spain, 2010. [Online]. http://www.periapsisvisual.com/1864747_laurel.pdf

[9] Awio Web Services LLC. (2013, June) W3Counter. [Online]. http://www.w3counter.com/globalstats.php

[10] Oliver Montenbruck and Eberhard Gill, *Satellite Orbits - Models, Methods, and Applications*, 1st ed. Heidelberg, Germany: Springer-Verlag Berlin, 2000.

[11] Erwin Fehlberg, "Classical Fifth., Sixth-, Seventh-, and Eight-Order Runge-Kutta Formulas with Stepsize Control," NASA George C. Marshall Space Flight Center, Huntsville, AL, USA, 1968. [Online]. http://dx.doi.org/2060/19680027281

[12] Brian Luzum et al., "The IAU 2009 system of astronomical constants: the report of the IAU working group on numerical standards for Fundamental Astronomy," *Celestial Mechanics and Dynamical Astronomy*, vol. 110, no. 4, pp. 293-304, August 2011. [Online]. http://dx.doi.org/10.1007/s10569-011-9352-4

[13] Winston W. Royce, "Managing the Development of Large Software Systems," in *IEEE WESCON*, vol. 26, 1970, pp. 1-9. [Online]. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[14] "Information technology – Programming languages – C++," ISO/IEC Standard 14882:2011, 2011. [Online]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50372

[15] Charles H. Acton, "Ancillary data services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, vol. 44, no. 1, pp. 65-70, January 1996. [Online]. http://dx.doi.org/10.1016/0032-0633(95)00107-7

[16] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, p. Article 13, June 2007. [Online]. http://dx.doi.org/10.1145/1236463.1236468

[17] "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, 1985. [Online]. http://dx.doi.org/10.1109/IEEESTD.1985.82928

[18] Pavel Holoborodko. (2013) MPFR C++. [Online]. http://www.holoborodko.com/pavel/mpfr/

[19] MPIR Team. (2014, July) MPIR. [Online]. http://www.mpir.org

[20] Assimp Development Team. (2013) Open Asset Import Library. [Online]. http://assimp.sourceforge.net

[21] "High Integrity C++ Coding Standard Version 4.0," Programming Research Ltd., 2013.

[22] Atomineer. (2014, July) Atomineer Pro Documentation for Visual Studio. [Online]. http://www.atomineerutils.com

[23] Dimitri van Heesch. (2014, July) Doxygen. [Online]. http://www.doxygen.org

[24] FiForms Solutions. (2014, July) xs3p. [Online]. http://xml.fiforms.org/xs3p/

[25] Free Software Foundation, Inc. (2007, June) GNU General Public License v3. [Online]. http://www.gnu.org/copyleft/gpl.html

[26] European Commission. (2009, November) European Union Public Licence - EUPL v.1.1. [Online]. http://ec.europa.eu/idabc/eupl.html

[27] Joinup. (2014, July) EUPL compatible open source licences. [Online]. https://joinup.ec.europa.eu/software/page/eupl/eupl-compatible-open-source-licences

[28] Casey D. Jordan and Dale Waldt. (2010, September) Schema scope: Primer and best practices. [Online]. http://www.ibm.com/developerworks/library/x-schemascope/x-schemascope-pdf.pdf

[29] National Aeronautics and Space Administration. (2015, May) NASA 3D Resources. [Online]. http://nasa3d.arc.nasa.gov

[30] Csaba Kelemen and László Szirmay-Kalos, "A Microfacet Based Coupled Specular-Matte BRDF Model with Importance Sampling," in *EUROGRAPHICS 2001*, 2001. [Online]. http://www.fsz.bme.hu/~szirmay/scook.pdf

[31] Robert L. Cook and Kenneth E. Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics*, vol. 1, no. 1, pp. 7-24, January 1982. [Online]. http://dx.doi.org/10.1145/357290.357293

[32] James F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, pp. 192-198, July 1977. [Online]. http://dx.doi.org/10.1145/965141.563893

[33] Diogo A. R. Lopes and António Ramires Fernandes, "Atmospheric Scattering - State of the Art," in *EPCG 2014 – 21º Encontro Português de Computação Gráfica*, Leiria, Portugal, 2014.

[34] NVIDIA Corporation, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Matt Pharr and Randima Fernando, Eds. Boston, MA, USA: Addison-Wesley Professional, 2005. [Online]. http://http.developer.nvidia.com/GPUGems2/gpugems2_frontmatter.html

[35] Peter Collienne, Robin Wolff, Andreas Gerndt, and Torsten Kuhlen, "Physically Based Rendering of the Martian Atmosphere," in *X. Workshop der GI-Fachgruppe VR/AR*, Würzburg, 2013, pp. 97-108. [Online]. http://elib.dlr.de/86477

[36] Nobuo Yamamoto. (2014, October) Equation Expressing Star or Flower Graphics. [Online]. http://www.geocities.jp/nyjp07/asteroid/index_asteroid_E.html

[37] Microsoft. (2015, May) About the Windows Media Codecs. [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/gg153556(v=vs.85).aspx

[38] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, 1st ed. Boston, MA, USA: Addison-Wesley Professional, 2001.

[39] Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed. Boston, MA, USA: Addison-Wesley Professional, 2005.

[40] Campwood Software. (2015, May) SourceMonitor. [Online]. http://www.campwoodsw.com/sourcemonitor.html

[41] Patrick Cozzi and Kevin Ring, *3D Engine Design for Virtual Globes*, 1st ed. Natick, MA, Massachusetts, United States of America: A. K. Peters/CRC Press, 2011.

[42] Brano Kemen. (2014, September) Outerra: Maximizing Depth Buffer Range and Precision. [Online]. http://outerra.blogspot.pt/2012/11/maximizing-depth-buffer-range-and.html

[43] A. Lagae et al., "State of the Art in Procedural Noise Functions," in *Eurographics 2010 - State of the Art Reports*, Norrköping, Sweden, 2010.

[44] Ken Perlin, "Improving Noise," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 681-682, July 2002. [Online]. http://dx.doi.org/10.1145/566654.566636

[45] T. S. Kelso. (2015, May) CelesTrak. [Online]. http://www.celestrak.com/

[46] Richard W. Orloff, *Apollo by the Numbers: A Statistical Reference*. Washington, DC, United States of America: NASA, 2000. [Online]. http://ntrs.nasa.gov/search.jsp?R=20010008244

[47] Jorge Tiago Vila and Paulo J. S. Gil, "A Tool with Enhanced Graphical Capabilities for Spacecraft Trajectory Visualization," in *4th International Conference on Astrodynamics Tools and Techniques*, Madrid, Spain, 2010. [Online]. https://sotis.tecnico.ulisboa.pt/sotis-ui/#record/261993118661

Appendix A

FRAMES OF REFERENCE

A.1 Definitions

$\mathbf{r}_{\mathbf{central}}$ – Central body position in the inertial frame J2000;

$\mathbf{v}_{\mathbf{central}}$ – Central body velocity in the inertial frame J2000;

$\mathbf{r}_{\mathbf{spacecraft}}$ – Spacecraft position in the inertial frame J2000;

$\mathbf{v}_{\mathbf{spacecraft}}$ – Spacecraft velocity in the inertial frame J2000.

Relative orbital position vector – position of the spacecraft with reference to the central body:

$$\mathbf{r} = \mathbf{r}_{\mathbf{spacecraft}} - \mathbf{r}_{\mathbf{central}}$$

Relative orbital velocity vector – velocity of the spacecraft with reference to the central body:

$$\mathbf{v} = \mathbf{v}_{\mathbf{spacecraft}} - \mathbf{v}_{\mathbf{central}}$$

Orbital plane – plane perpendicular to the orbital momentum vector:

$$\Pi_{\text{orbit}} \perp \mathbf{h} = \mathbf{r} \times \mathbf{v}$$

Or, in case of a straight line trajectory, perpendicular to:

$$\Pi_{\text{orbit}} \perp \mathbf{r} \times \left(\mathbf{e}_{\boldsymbol{z}}^{\boldsymbol{J2000}} \times \mathbf{r}\right)$$

Or, in case of a straight line trajectory parallel to the J2000 frame z-axis, perpendicular to:

$$\Pi_{\text{orbit}} \perp \mathbf{e}_{\boldsymbol{y}}^{\boldsymbol{J2000}}$$

A.2 J2000 – Earth mean equator and equinox of J2000

- X-axis is pointed towards the mean vernal equinox on January 1, 2000 at 12:00 TDT;
- Z-axis is parallel to the Earth's mean rotational axis pointed towards its North pole;
- Y-axis completes the right-handed coordinate frame.

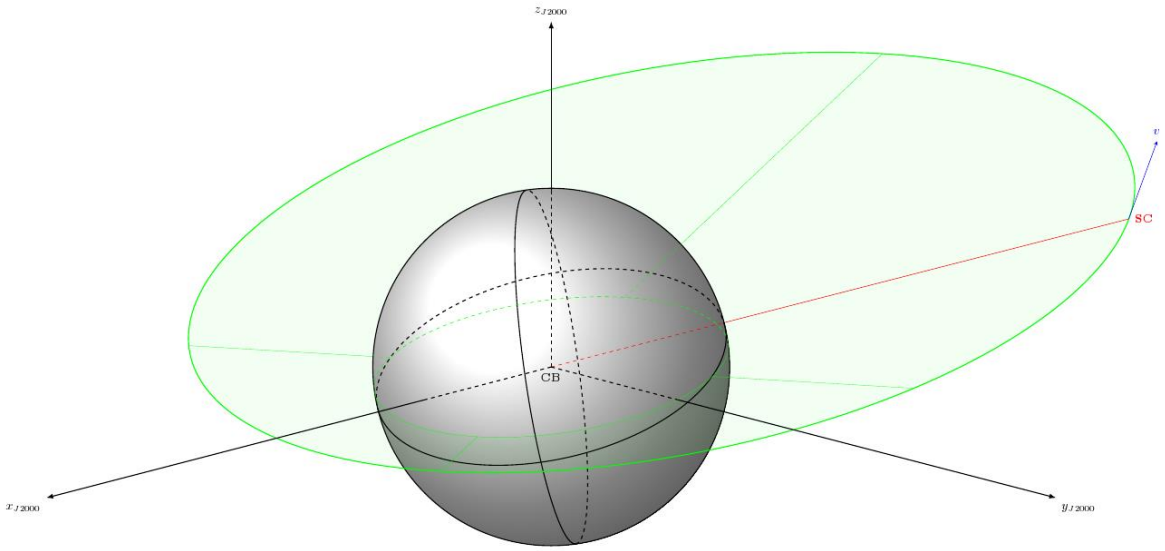


Figure A.1 – The local J2000 reference frame on the central body (CB).

A.3 ECLIPJ2000 – Ecliptic and equinox of J2000

- X-axis is pointed towards the mean vernal equinox on January 1, 2000 at 12:00 TDT;
- Z-axis is perpendicular to the Earth's orbital plane orientated with its angular momentum vector;
- Y-axis completes the right-handed coordinate frame.

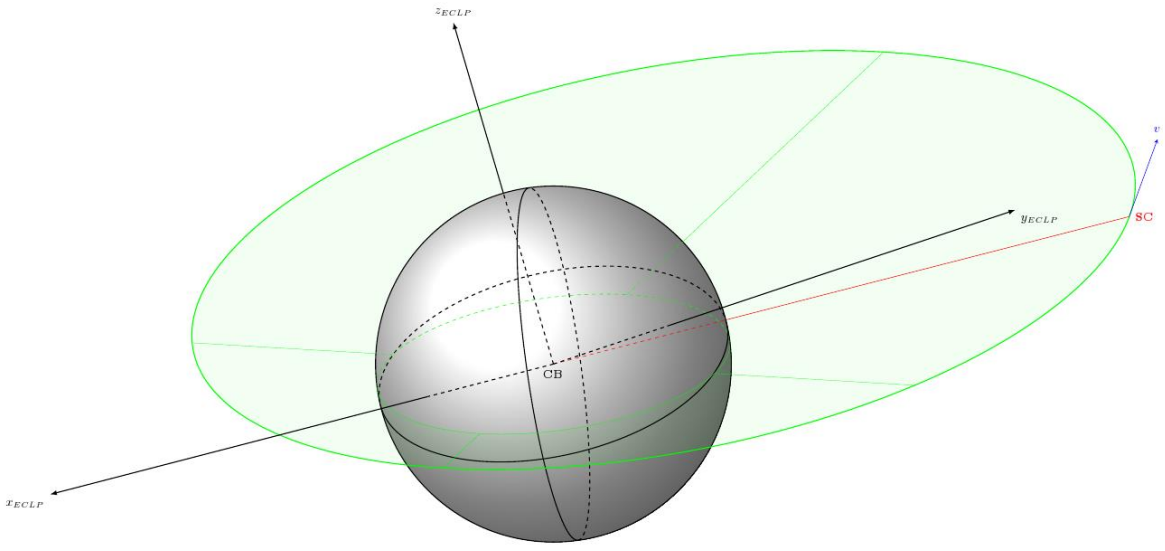


Figure A.2 – The local ECLIPTJ2000 reference frame on the central body (CB).

A.4 BODY – Body-fixed frame

On a planet:

- X axis is pointed towards the body's prime meridian;
- Z axis is parallel to the body's rotational axis and pointed towards its North pole;
- Y axis completes the right-handed coordinate frame.

On a spacecraft:

- X, Y, and Z axes are parallel to the body's principal axis of inertia;
- Other definitions are possible, according to the loaded spacecraft orientation data.

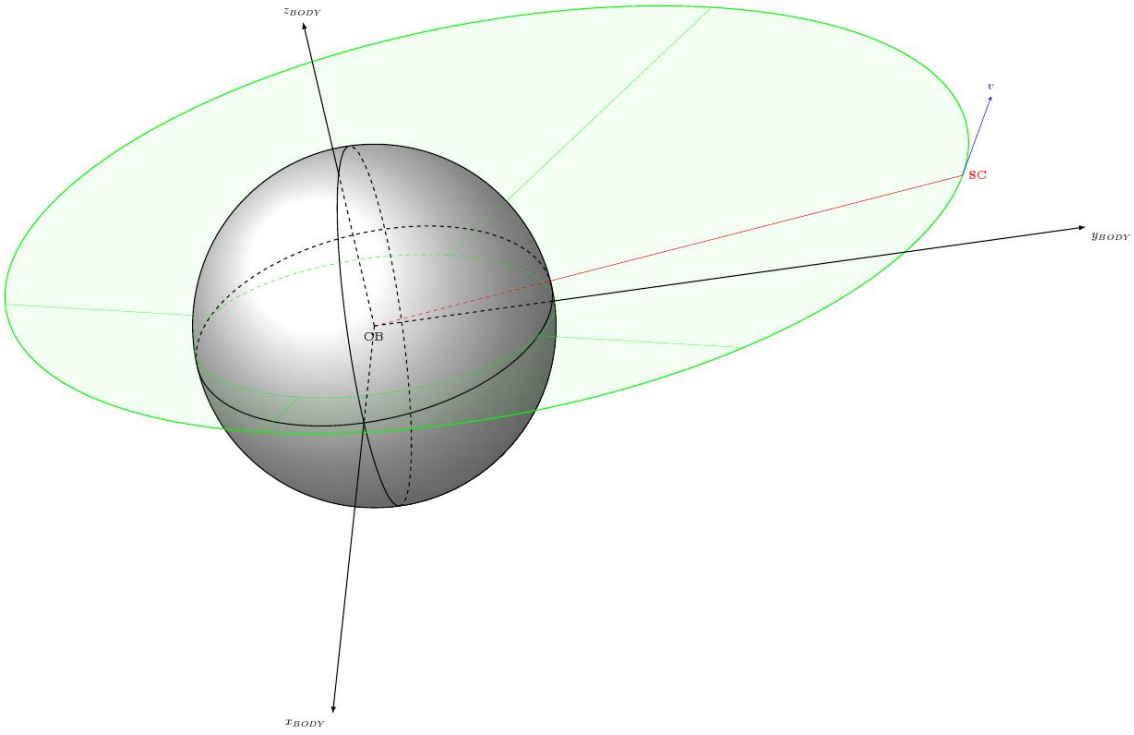


Figure A.3 – The local BODY reference frame on the central body (CB).

A.5 LVLH – Local Vertical Local Horizontal

- Z axis lies in the local vertical direction and points towards the planet (nadir);
- Y axis is perpendicular to the orbital plane oriented opposite the angular momentum vector;
- X axis is the projection of the velocity in the horizontal plane and completes the right-handed frame.

$$\mathbf{e_z} = -\frac{\mathbf{r}}{\|\mathbf{r}\|} \quad \mathbf{e_y} = \frac{\mathbf{v}\times\mathbf{r}}{\|\mathbf{v}\times\mathbf{r}\|} \quad \mathbf{e_x} = \mathbf{e_y}\times\mathbf{e_z}$$

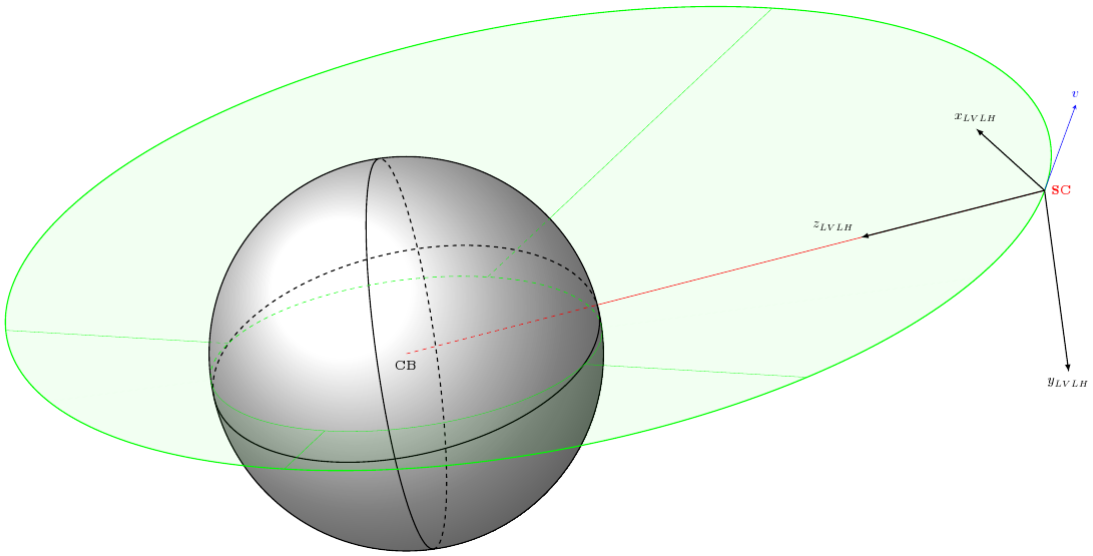


Figure A.4 – The local LVLH reference frame on the spacecraft (SC).

A.6 TRAJECTORY – X parallel to velocity and orbital plane

- X axis is parallel to the velocity vector and is positive in the direction of motion;
- Y axis is perpendicular to the orbital plane oriented opposite the angular momentum vector;
- Z axis is the projection of the vertical in the orbital plane and completes the right-handed frame.

$$\mathbf{e_x} = \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad \mathbf{e_y} = \frac{\mathbf{v} \times \mathbf{r}}{\|\mathbf{v} \times \mathbf{r}\|} \quad \mathbf{e_z} = \mathbf{e_x} \times \mathbf{e_y}$$

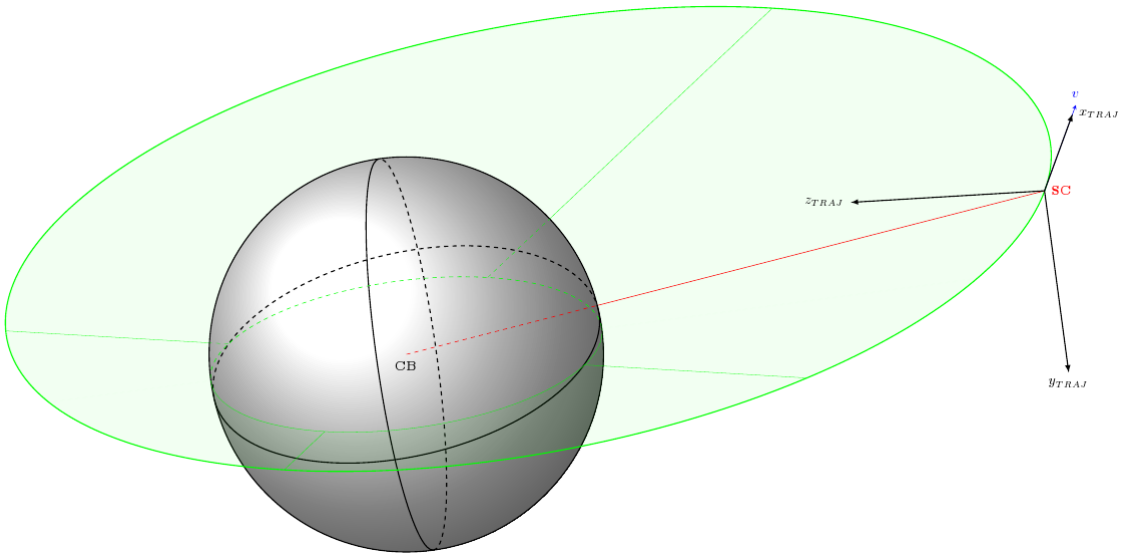


Figure A.5 – The local TRAJECTORY reference frame on the spacecraft (SC).

Appendix B

COMPLETE LAYER DIAGRAM OF THE SOFTWARE ARCHITECTURE

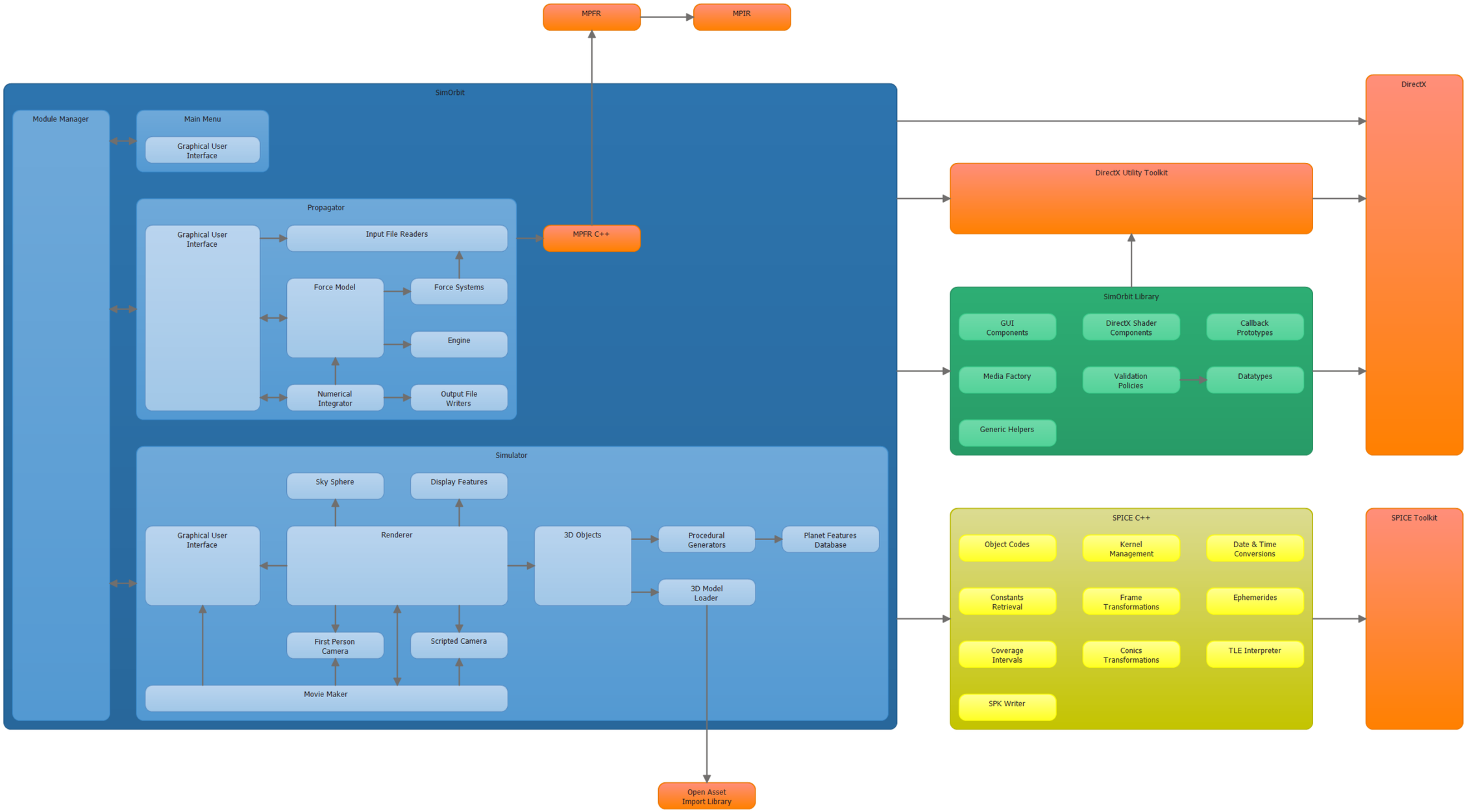


Figure B.1 – Complete layer diagram of the software architecture the showing relations between the various components (third party components displayed in orange) and their sub-architectures.

Appendix C

UNITS AND VALUE POLICIES

To ensure reliable and consistent data input the software uses a single set of units and conversion factors, found on Table C.1, and a collection of policies to validate the input.

Table C.1 – Units used in the software and respective conversion factors to the default unit.

Type	Unit	Conversion Factor
Time	s	1.0
	min	60.0
	h	3600.0
	d	86400.0
Length	m	1.0×10^{-3}
	km	1.0
	AU	149597870.7
Velocity	m/s	1.0×10^{-3}
	m/min	$1.0 \times 10^{-3}/60.0$
	m/h	$1.0 \times 10^{-3}/3600.0$
	km/s	1.0
	km/min	$1.0/60.0$
	km/h	$1.0/3600.0$
Acceleration	m/s2	1.0×10^{-3}
	m/min2	$1.0 \times 10^{-3}/60.0^2$
	m/h2	$1.0 \times 10^{-3}/3600.0^2$
	km/s2	1.0
	km/min2	$1.0/60.0^2$
	km/h2	$1.0/3600.0^2$
Mass	g	1.0×10^{-3}
	kg	1.0
	t	$1.0 \times 10^{+3}$
Area	m2	1.0
	km2	$1.0 \times 10^{+6}$
Angular	deg	$\pi/180.0$
	rad	1.0

The tables below detail the policies used for data input validation throughout the software. The type from which the policies derive is listed under the policy name (┗) and any values in bold are defaults.

Table C.2 – Date & Time policies.

Policy	Valid Range	Valid Units
JulianDate ┗NumericValue	$[2305813.5,\ \ 5373484.5]$	*NIL*
Duration ┗NumericValue	$[0,\ \ \infty[$	**s**, min, h, d

Table C.3 – Motion-related (position, velocity, and acceleration) policies.

Policy	Valid Range	Valid Units
Position └NumericValue	$\mathbb{R}$	m, **km**, AU
PositionError └ NumericValue	$\mathbb{R}_{>0}$	
PositionVector └ NumericVector(3)	$\mathbb{R}^3$	
Velocity └NumericValue	$\mathbb{R}$	m/s, m/min, m/h, **km/s**, km/min, km/h
VelocityError └ NumericValue	$\mathbb{R}_{>0}$	
VelocityVector └ NumericVector(3)	$\mathbb{R}^3$	
Acceleration └NumericValue	$\mathbb{R}$	m/s2, m/min2, m/h2, **km/s2**, km/min2, km/h2
AccelerationError └ NumericValue	$\mathbb{R}_{>0}$	
AccelerationVector └ NumericVector(3)	$\mathbb{R}^3$	

Table C.4 – Orbital elements policies.

Policy	Valid Range	Valid Units
SemimajorAxis └NumericValue	$\mathbb{R}_{>0}$	m, km, AU
Eccentricity └NumericValue	$\mathbb{R}_{>0}$	*NIL*
LongitudeOfAscendingNode └NumericValue	$[0,\ 360[\ deg$	**deg**, rad
Inclination └NumericValue	$[0,\ 180]\ deg$	**deg**, rad
ArgumentOfPeriapsis └NumericValue	$[0,\ 360[\ deg$	**deg**, rad

Table C.5 – Physical constant policies.

Policy	Valid Range	Valid Units
Mass └NumericValue	$\mathbb{R}_{>0}$	g, kg, t
Area └NumericValue	$\mathbb{R}_{>0}$	**m2**, km2
Reflectivity └NumericValue	$[0,\ 1]$	*NIL*

Table C.6 – Direction policies (Euclidean vector and Azimuth/Elevation).

Policy	Valid Range	Valid Units
DirectionVector └NumericVector(3)	$\mathbb{R}^3 \backslash (0,\ 0,\ 0)$	NIL
DirectionAzEl └NumericVector(2)	$\left\{ \begin{matrix} (az, el): \\ az \in [0,\ 360[\ deg, \\ el \in [-90,\ 90]\ deg \end{matrix} \right\}$	**deg**, rad

Table C.7 – Frames of reference policies (cf. Appendix A).

Policy	Validity Condition
StandardFrame └TextValue	J2000, ECLIPJ2000, BODY
CustomFrame └TextValue	LVLH, TRAJECTORY
Frame └TextValue	J2000, ECLIPJ2000, BODY, LVLH, TRAJECTORY

Table C.8 – NAIF object name and code policies.

Policy	Validity Condition	
NAIFBodyName └TextValue	As defined in the NAIF Integer ID Codes: http://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/naif_ids.html	
Policy	**Valid Range**	**Valid Units**
NAIFSpacecraftCode └NumericValue	$\mathbb{Z}_{<0}$	*NIL*

Table C.9 – Interpolation polynomials policies.

Policy	Validity Condition	
InterpolationPolynomial └TextValue	Lagrange, Hermite	
Policy	**Valid Range**	**Valid Units**
InterpolationPolynomialDegree └NumericValue	$\{1 \ldots 15\}$	*NIL*

Table C.10 – Bit precision policy.

Policy	Valid Range	Valid Units
BitPrecision └NumericValue	$\{24 \ldots \mathbf{53} \ldots 113\}$	*NIL*

Table C.11 – Filename policy.

Policy	Validity Condition
ReadableFile └TextValue	Filename must correspond to a readable file.

Table C.12 – Color policy (red, green, blue).

Policy	Valid Range	Valid Units
Color └ NumericVector	$\{(r,g,b): r,g,b \in [0,\ 1]\}$	*NIL*

Appendix D

XML INPUT FILES EXAMPLES

The software input files are in XML, with examples of each type presented below.

D.1 Trajectory Propagator Module

```
<?xml version="1.0" encoding="UTF-8"?>
<propagatorSetup xmlns="urn:simorbit">
  <general>
    <referenceEpoch>2440419.182095</referenceEpoch>
    <referenceBody>EARTH</referenceBody>
    <referenceFrame>J2000</referenceFrame>
    <orbitalElements>
      <semimajorAxis unit="km">286535.800</semimajorAxis>
      <eccentricity>0.976965</eccentricity>
      <periapsisTime>2440419.180255</periapsisTime>
      <argumentOfPeriapsis unit="deg">4.4102</argumentOfPeriapsis>
      <inclination unit="deg">31.383</inclination>
      <longitudeOfAscendingNode unit="deg">358.380</longitudeOfAscendingNode>
    </orbitalElements>
    <stopEpoch>2440426.500000</stopEpoch>
  </general>
  <forceModel>
    <mass unit="kg">28801</mass>
    <area unit="m2">11.95</area>
    <reflectivity>0.5</reflectivity>
    <engineInputs>
      <deltaV name="Midcourse Correction">
        <epoch>2440420.178449</epoch>
        <magnitude unit="km/s">6.73</magnitude>
        <referenceBody>EARTH</referenceBody>
        <referenceFrame>LVLH</referenceFrame>
        <directionVector>
          <x>0</x>
          <y>0</y>
          <z>1</z>
        </directionVector>
      </deltaV>
    </engineInputs>
  </forceModel>
  <integrator>
    <timeStep>
      <initial unit="s">30</initial>
      <minimum unit="s">1</minimum>
      <maximum unit="h">1</maximum>
    </timeStep>
    <truncationErrorTolerance>
      <position unit="m">100</position>
      <velocity unit="m/h">10</velocity>
    </truncationErrorTolerance>
  </integrator>
  <output>
    <spk>
      <objectCode>-911</objectCode>
      <interpolationPolynomial degree="9">LAGRANGE</interpolationPolynomial>
    </spk>
    <raw>
      <commentLineStart>#</commentLineStart>
    </raw>
  </output>
</propagatorSetup>
```

Listing D.1 – Procedural planet generator input XML file example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<forceSystems xmlns="urn:simorbit">
  <forceSystem name="Solar System">
    <barycenter>SOLAR SYSTEM BARYCENTER</barycenter>
    <centralBody>SUN</centralBody>
    <thirdBodies>
      <thirdBody>MERCURY BARYCENTER</thirdBody>
      <thirdBody>VENUS BARYCENTER</thirdBody>
      <thirdBody>EARTH BARYCENTER</thirdBody>
      <thirdBody>MARS BARYCENTER</thirdBody>
      <thirdBody>JUPITER BARYCENTER</thirdBody>
      <thirdBody>SATURN BARYCENTER</thirdBody>
      <thirdBody>URANUS BARYCENTER</thirdBody>
      <thirdBody>NEPTUNE BARYCENTER</thirdBody>
      <thirdBody>PLUTO BARYCENTER</thirdBody>
    </thirdBodies>
  </forceSystem>
  <forceSystem name="Mercury System">
    <barycenter>MERCURY BARYCENTER</barycenter>
    <thirdBodies>
      <thirdBody>VENUS</thirdBody>
    </thirdBodies>
    <centralBody>MERCURY</centralBody>
  </forceSystem>
  <forceSystem name="Venus System">
    <barycenter>VENUS BARYCENTER</barycenter>
    <centralBody>VENUS</centralBody>
  </forceSystem>
  <forceSystem name="Earth System">
    <barycenter>EARTH BARYCENTER</barycenter>
    <centralBody>EARTH</centralBody>
    <thirdBodies>
      <thirdBody>MOON</thirdBody>
    </thirdBodies>
  </forceSystem>
  <!--TRUNCATED-->
</forceSystems>
```

Listing D.2 – Procedural planet generator input XML file example.

D.2 Simulator Module

```xml
<?xml version="1.0" encoding="UTF-8"?>
<models xmlns="urn:simorbit">
  <model>
    <object>-82</object>
    <filename>Models\Cassini\Cassini.obj</filename>
    <scale>0.001</scale>
  </model>
  <model>
    <object>401</object>
    <filename>Models\Cassini\Phobos.3ds</filename>
    <scale>1.0</scale>
  </model>
  <model>
    <object>402</object>
    <filename>Models\Deimos\Deimos.3ds</filename>
    <scale>1.0</scale>
  </model>
</models>
```

Listing D.3 – Models information input XML file example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<planets xmlns="urn:simorbit">
  <planet name="EARTH">
    <material>
      <ambientColor>
        <r>0.00</r>
        <g>0.00</g>
        <b>0.00</b>
      </ambientColor>
      <diffuseColor>
        <r>1.00</r>
        <g>1.00</g>
        <b>1.00</b>
      </diffuseColor>
      <specularColor>
        <r>1.00</r>
        <g>1.00</g>
        <b>1.00</b>
      </specularColor>
      <emissiveColor>
        <r>1.00</r>
        <g>1.00</g>
        <b>1.00</b>
      </emissiveColor>
      <shininess>20</shininess>
      <shininessStrength>1.0</shininessStrength>
      <diffuseMap>Models\Textures\Earth\diffuse.dds</diffuseMap>
      <specularMap>Models\Textures\Earth\specular.dds</specularMap>
      <emissiveMap>Models\Textures\Earth\emissive.dds</emissiveMap>
      <normalsMap>Models\Textures\Earth\normals.dds</normalsMap>
    </material>
    <atmosphere>
      <wavelength>
        <r>0.680</r>
        <g>0.550</g>
        <b>0.440</b>
      </wavelength>
      <rayleighConstant>0.0022</rayleighConstant>
      <mieConstant>0.0015</mieConstant>
      <asymmetryFactor>0.99</asymmetryFactor>
    </atmosphere>  <planet name="SATURN">
    <!--TRUNCATED-->
    <rings>
      <innerRadius unit="km">66900.0</innerRadius>
      <outerRadius unit="km">140225.0</outerRadius>
      <colorMap>Models\Textures\Saturn\rings.dds</colorMap>
    </rings>
  </planet>
</planets>
```

Listing D.4 – Procedural planet generator input XML file example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cameras xmlns="urn:simorbit">
  <camera name="Apollo 11 (Moon Orbit)">
    <startEpoch>2440422.0145833</startEpoch>
    <timeScale unit="min">1</timeScale>
    <eyePosition>
      <referenceBody>-911</referenceBody>
      <offset>
        <referenceFrame>LVLH</referenceFrame>
        <centralBody>MOON</centralBody>
        <directionVector>
          <x>-1</x>
          <y>0</y>
          <z>-1</z>
        </directionVector>
        <range unit="m">300.0</range>
      </offset>
    </eyePosition>
    <lookAtPosition>
      <referenceBody>-911</referenceBody>
    </lookAtPosition>
    <upDirection>
      <referenceBody>-911</referenceBody>
      <referenceFrame>TRAJECTORY</referenceFrame>
      <centralBody>MOON</centralBody>
      <directionVector>
        <x>0</x>
        <y>0</y>
        <z>1</z>
      </directionVector>
    </upDirection>
  </camera>
</cameras>
```

Listing D.5 – Cameras input XML file example.

Appendix E

TRAJECTORY PROPAGATOR FLOWCHARTS

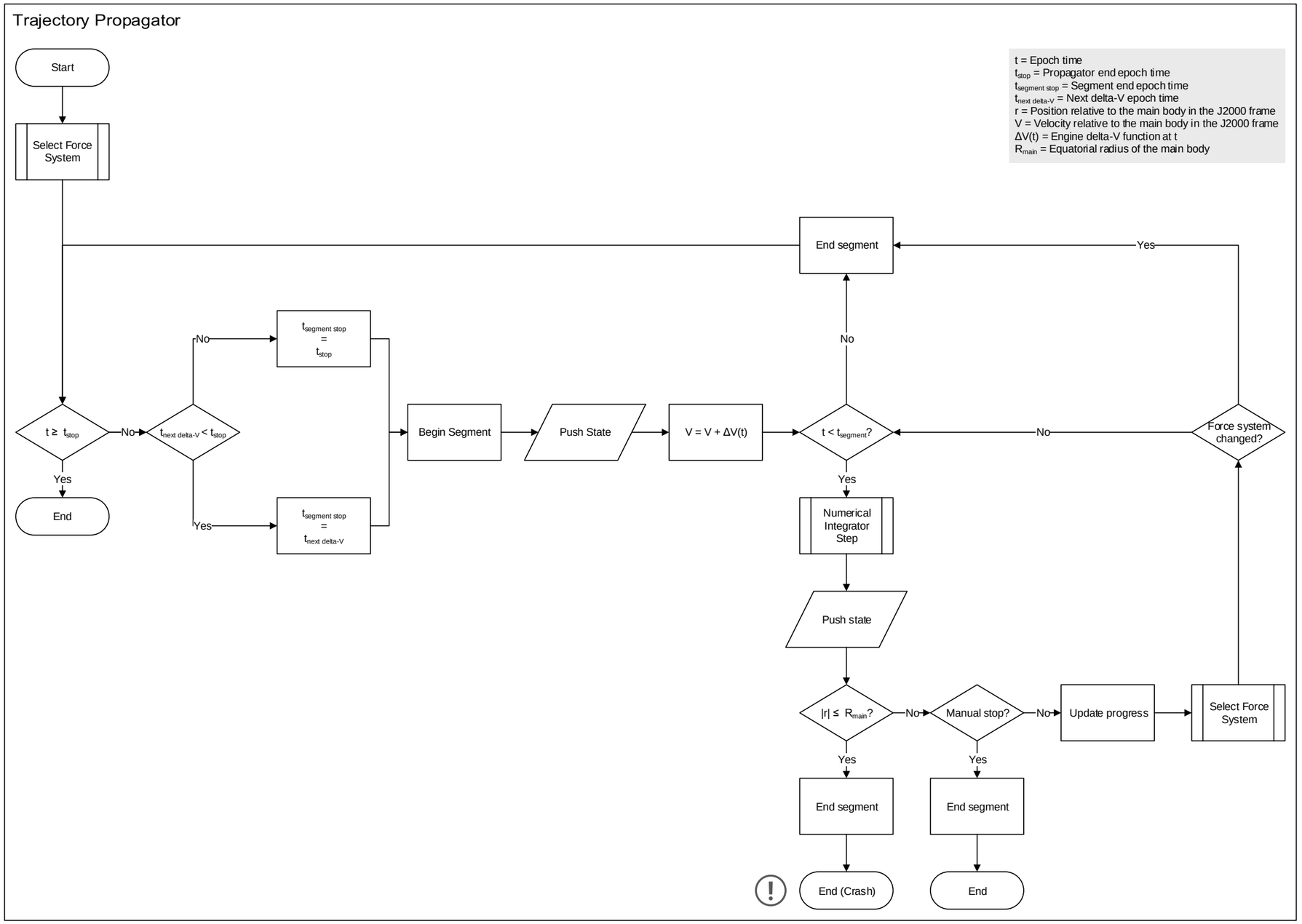


Figure E.1 – Trajectory propagator flowchart.

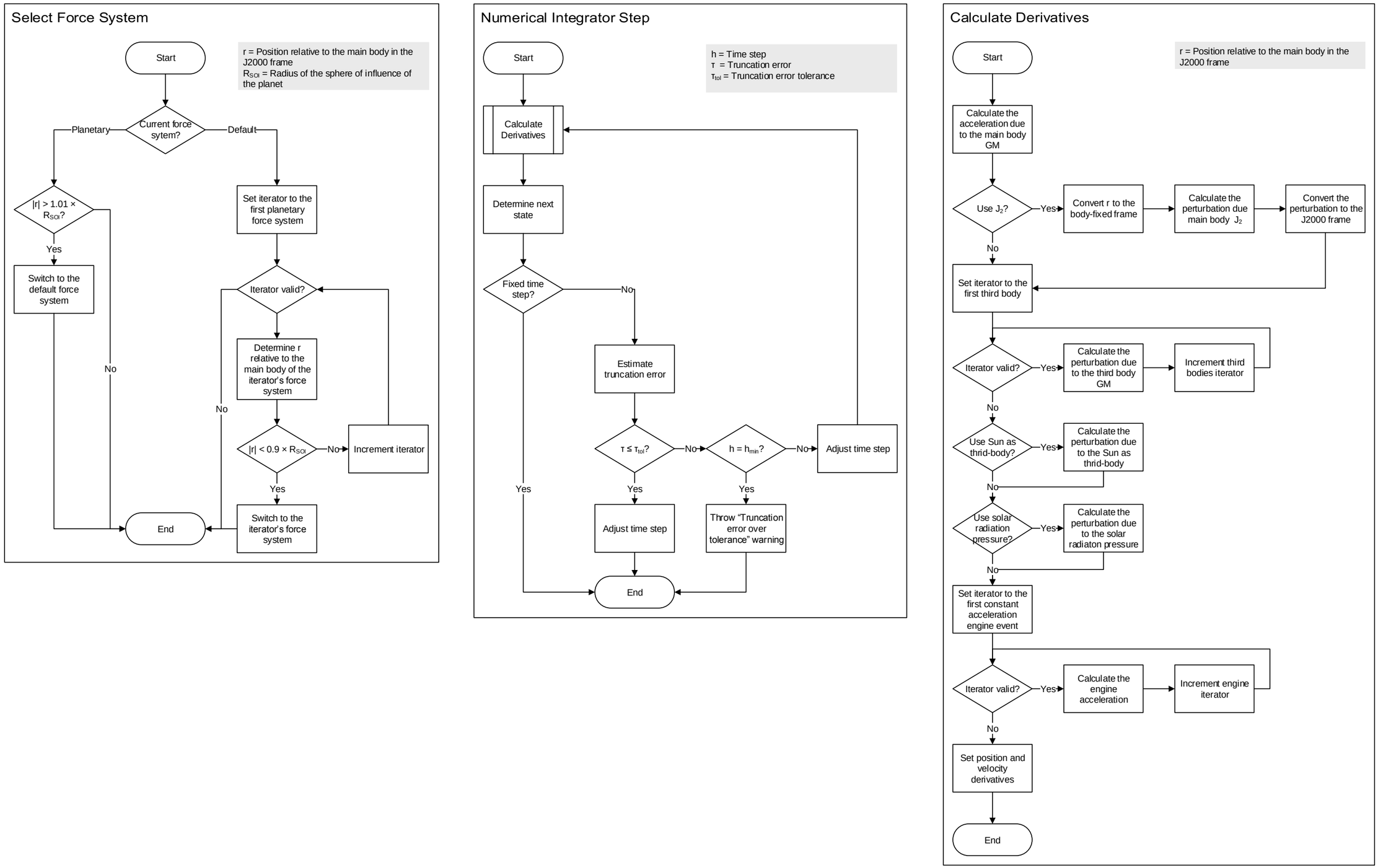


Figure E.2 – Trajectory propagator subroutine flowcharts (Select Force System, Numerical Integrator Step, Calculate Derivatives).