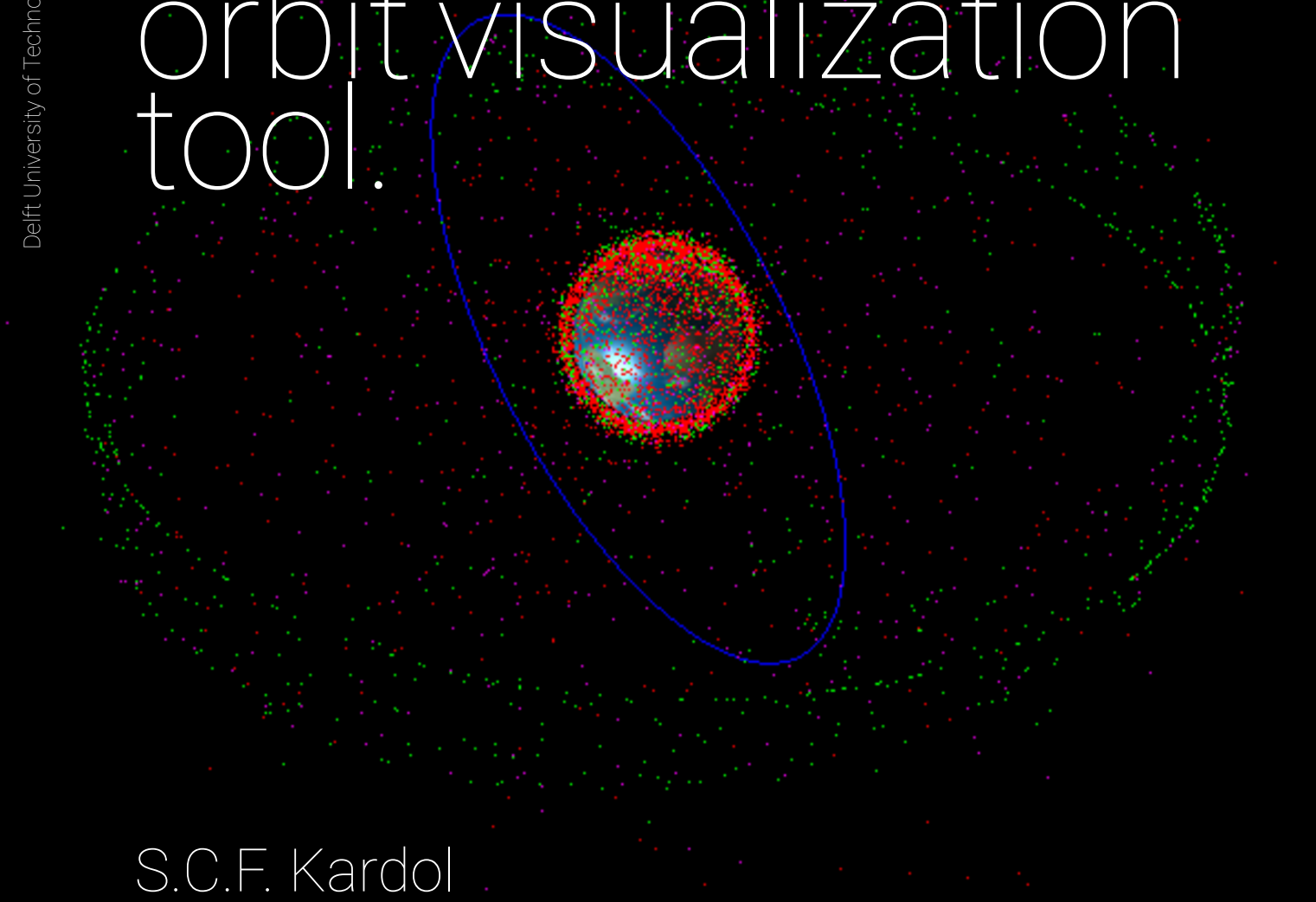
An interactive, web-based, near-Earth orbit visualization tool.



S.C.F. Kardol

Delft University of Technology

TU Delft

An interactive, web-based, near-Earth orbit visualization tool.

by

S.C.F. Kardol

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday October 2, 2018 at 14:00

Student number: 4502655
Project duration: May 1, 2017 – October 2, 2018
Thesis committee:

Prof. dr. ir. P.N.A.M. Visser,	TU Delft, Chair
Dr. ir. E.N. Doornbos,	TU Delft, Supervisor
Dr. A. Menicucci,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Even though several orbit visualization tools exist, the ability to compare multiple satellite orbits or manipulate time are not readily available with the current tools. Furthermore, most other visualization tools require additional effort or installations to work properly. To address these issues with other tools an interactive, web-based, near-Earth orbit visualization tool was developed. This was done to answer the question: *What are the uses of another interactive, web-based, near-Earth orbit visualization tool?*

After the development of the tool was completed, several use cases were examined. These use cases included visualizing constellations and space debris, reproducing the Iridium 33 and Cosmos 2251 collision, visualizing special orbits for educational purposes, and the reproduction of an aurora event which involved SWARM and EPOP satellites.

The tool provides its users with a wide range of functionality, while keeping it easy to use. The tool can be accessed on orbits.tudelft.nl and the source code can be found on https://gitlab.com/SvenKardol/Orbit-Visualization-Tool.

Preface

Before you lies the thesis "An interactive, web-based, near-Earth orbit visualization tool". It has been written to fulfill the graduation requirements of the Master of Science in Aerospace Engineering program at the Delft University of Technology. I was engaged in researching and writing this thesis from May 2017 to October 2018.

Even though the project has been challenging at times, the end result has surpassed my own expectations. I hope that this project could form the basis for many great thesis projects in the future as well as provide a visual support to whomever is working with satellites, space debris, or other objects. Furthermore, I hope that the tool could be used by students and space enthusiast alike to deepen their understanding of orbits.

A special thanks goes out to my supervisor Dr. ir. E.N. Doornbos. Without his excellent guidance and support during this process I would have not been able to succeed. Furthermore I'd like the other committee members Prof. dr. ir. P.N.A.M. Visser and Dr. A. Menicucci for taking time out of their schedule to help me finish my degree.

Lastly, I would like to thank my family and friends for support and patience during the entire process.

I hope you enjoy your reading.

S.C.F. Kardol
Delft, September 2018

Contents

List of Figures

1 Introduction

This thesis project examines the development of an interactive, web-based, near-Earth orbit visualization tool. The project will focus on the development and the use of this tool. This is done to answer the research question: *What are the uses of another interactive, web-based, near-Earth orbit visualization tool?*

The question specifically states that the newly developed tool is another tool at the disposal of researchers, students, and anyone else interested in using the tool. This followed directly from the literature study conducted previously [4]. In the literature study, several existing tools from different backgrounds were examined in order to find any missing elements from these existing tools. The literature study answered the question: *Is another orbit visualization tool useful?* The conclusion of the literature study was that another orbit visualization tool could be useful, if it met specific criteria. The criteria are based on the functionality and capabilities of the tools examined.

In the literature study, it was concluded that current tools have a wide variety of functionalities. However, it seemed that comparing multiple satellite orbits at certain times, past, present, or near future, is not an option that is readily available. In the past, it might have been sufficient for a tool to focus on a single satellite, but since the rise of small satellites and the higher use of constellations, nowadays, more features that can combine multiple satellites are desirable.

Another feature that is missing from a few of these tools is the ability to navigate through time easily without having to perform a lot of complicated steps, like manually inserting historical data sets. This feature would allow a user to see the movements of a satellite over a certain period of time or even visualize the growth of the number of satellites over the years.

Most of the tools examined were desktop application that require a good understanding of the program in order to use it properly. For some of the applications, the user is required to learn a programming language that is specifically written for this one application and is not used anywhere else. The user has to learn how to use the program, which requires time and effort. Desktop applications can also provide a significant barrier to usability. This is especially the case when the application requires additional installations, for example a Java runtime environment.

The web-based tools that were looked at were Stuff-in-Space, Swarm-Aurora, and VirES for Swarm. Both Swarm-Aurora and VirES for swarm have great functionality, but were specifically tailored to their mission, rendering it useless for other missions. Stuff-in-Space is a generic tool that visualizes all objects, but lacks basic functionality, since it only shows real-time orbits.

The conclusion of the literature study was that an orbit visualization tool should meet the following criteria:

- The tool should be a web-based application.
- The user should be able to easily manipulate time.
- Comparison of multiple satellites and their orbits should be included.
- The tool has to be fast and responsive.
- The tool should be easy to use.

Another feature that has not been seen in the tools that were looked at is the ability to experience the visualization from the user's point of view. That is, select a place on Earth and have a view that is looking up at the sky. This feature is not necessary for a successful tool, but could be implemented to enhance the user experience.

Another conclusion from the literature study was that the tool should focus on near-Earth objects. The focus on objects that are in solar orbits, or even outside the solar system would be unnecessary, since Cosmographia, another orbit visualization tool, is such a sophisticated tool visualizing these objects. Furthermore, the functionality of the tools seems out of the reach for a graduate thesis project.

In order to find the answer the research question of this thesis project, an interactive, web-based, near-Earth orbit visualization tool has been developed. The development consisted of three parts: the back-end, server, front-end of the tool. The back-end of the tool retrieves and processes data that is needed for the tool to be accurate. The back-end has set up the environment to store data, which helps to reduce the time needed to use the same data.

The front-end of the tool is what the user will experience. The front-end obtains the data sent by the server and uses the data to render the three dimensional orbits of objects. Furthermore, the front-end allows the user to communicate with the server and retrieve specific data for their needs. It also allows the user to change certain settings and manipulate the visualization by, for example, changing the time.

The server functions as a bridge between the front-end and the back-end. Commands taken from the front-end are converted to commands. These commands are send to the back-end. The server then receives the data from the back-end, which is then send to the front-end.

When the tool was completed, several use cases have been examined to determine the uses of the tool. This resulted in the limitations of the tool. These cases and limitations lead to recommendation for future improvements to either extend the tools functionality or enhance the user experience.

2

Background Information

In the introduction, the structure of the tool was briefly mentioned. In this chapter, a closer look is taken at what resources are needed to develop the tool and why those resources were selected. The chapter is divided into three sections. These sections are the data management, the data communication, and the visualization of data. These parts will correspond to the 3 main parts of the tool, namely the back-end, server, and front-end, respectively. In the literature study it was determined that the back-end would be written in Python, the front-end in Javascript, and the server would use Apache. To lead to a better integration of the Python back-end a different server was used, namely a Flask server.

2.1. Data Management

As mentioned in the introduction, the data is managed in the back-end of the tool. There are several types of data to be dealt with by the back-end:

- Two line elements of objects in near-Earth orbits.
- The satellite catalog.
- Sub-solar point coordinates.
- Earth-orientation parameters.

2.1.1. Two line elements of objects in near-Earth orbits

In order to model the motion of a satellite, we need data provided by a satellite tracking system. A great source of tracking data for satellites in Earth's orbit are two line elements (TLE). Two line elements are available for any object greater than 10 centimeters. In order to easily obtain the TLE data, one should create an account on the website: `www.space-track.org`. The organization behind the website allows users to download the TLE of any object at any point in time. Their database currently[1] holds over 118 million two line elements and is steadily growing every day. The organization collects the data from the data provider, Joint Space Operations Center (JSpOC). The JSpOC is the organization responsible for performing all of the orbit determination activities necessary to maintain the US space catalog[1]. They provide information of every object that they track, which also includes objects from other countries, since these objects are a potential collision hazard.

To automatically download the TLE data, a protocol has been written. In order to be able to access the information, a login is required and a protocol that specifies which information is required. To avoid multiple login sessions to obtain the data the *requests*-library can be used. This python library can open an active session. A session would log in to the website and then remembers that the user has the authority to access files behind the authentication API. Using this session, multiple data downloads can happen before another authentication is required. The library also allows the user to download the data by using the *.get()*-function. This function requires the url of the data and uses the login information automatically. This library simplifies obtaining data greatly.

[1] April 2018

2.1.2. The satellite catalog

The next type of data is the satellite catalog. The satellite catalog is a document, which contains information of all objects that have ever been tracked. The catalog includes information about the objects, which is not available in the TLE. The satellite catalog includes various pieces of information that are specifically related to the objects, like the satellite name, type of object, launch and decay dates. This information will be used to provide the user with the information about the object, as well as give colors to various types of objects in the visualization. Currently[2] the satellite catalog includes more than 43,000 objects.

There are two sources available for the satellite catalog. The first source is the space-track organization, which also maintain the TLE database. The second source is Dr. T.S. Kelso, the man behind `www.celestrak.com`. Dr. Kelso works as a Senior Research Astrodynamicist for Analytical Graphics, Inc.'s Center for Space Standards and Innovation (CSSI). This website is part of his job at the CSSI. This website allows users to download the satellite catalog without the use of an authorization protocol. The difference between these sources is how to obtain the data and the long term stability of the website. Even though the information is readily available at celestrak, the website has not seen an update since January 26, 2017. This means that even though the website has all the information, the stability of the website could experience problems in the future. Space-track on the other hand is updated regularly and does not seem to depend on one person controlling the website, making it a seemingly more stable choice looking towards the future. The one downside of the satellite catalog of space-track is that the request protocol is more difficult to implement than the protocol for downloading from celestrak. Ultimately the decision was made to use the space-track satellite catalog, because it is more secure and is better maintained than the celestrak website. Furthermore, the data from space-track is the primary source of the data, which celestrak regularly copies to keep up-to-date.

2.1.3. Sub-solar point coordinates

The third type of information are the coordinates of the sub-solar point. This information is not critical information for the functioning of the tool, but gives extra information to the user. The information can tell the user if an object is located on the sunlit or dark side of the Earth. Furthermore, the information can be used to set the direction from which the light is coming in the visualization. In order to obtain the information about the sub-solar point, a Python library can be used. The library is the *Pyephem*-library. This library is capable of determine the position of the sub-solar point at any given time. The library computes the position by determining where the sun is located in the sky for an observer on the ground. Using this information, the coordinates of the sub-solar point can be calculated.

By setting an observer on the Greenwich meridian and at the equator, the sub-solar coordinates are most easily calculated for a given time. This is done by calculating the position of the observer at the Greenwich meridian with respect of the reference epoch used by the library. This epoch is January 1st, 2000. Due to the orientation of Earth the position of the observer changes over time. The reason for these changes were described in the literature study. Similarly the position of the Sun is calculated from this epoch. The position of the Sun and the observer are expressed in angles, both longitudinal and latitudinal. The difference between the angles of the Sun and observer gives the sub-solar position relative to the observer.

$$\begin{aligned} \Delta\lambda &= \lambda_{Sun} - \lambda_{Observer} \\ \Delta\phi &= \phi_{Sun} - \phi_{Observer} \end{aligned}$$

In the formula λ and ϕ represent the latitudinal and longitudinal change of object with respect to the epoch, respectively. Since the observer is located at the Greenwich meridian and the equator, the latitude and longitude of the observer are both 0^o, making the difference the sub-solar point coordinates. Choosing a different observer point results in the same sub-solar point, but would require one more computation to get the exact coordinates on the Earth's surface. Hence, choosing the observer position at the Greenwich meridian and equator results in the easiest solution.

2.1.4. Earth-orientation parameters

Related to the sub-solar coordinates are the Earth-orientation parameters. To be precise, the sub-solar coordinates come from these parameters. The Earth's rotation varies over time due to the effect of precession and nutation. Since the Earth is not an isolated perfect sphere, we have to consider the effects of both the Sun and Moon. When we consider the motion of a spherical Moon orbiting a non-spherical Earth, the Earth will experience an equal and opposite force to the effect of Earth's gravitational potential on the Moon. This

[2] April 2018

results in a gravitational torque on the Earth. The Moon is not the only significant contributor of the motion of the Earth. The Sun is another significant contributors. These effects have been described in the literature more extensively.

The parameters are needed to propagate the satellite orbits correctly. However, the orbits are propagated in the front-end of the tool using a library that will be discussed in more detail in subsection 2.3.2. Because the library uses a set of parameters and models to determine the parameters as time progresses, actually storing Earth-orientation parameters in the back-end currently has no use. The code has been written to gather the information, but it is not in use.

2.2. Data Communication

The management of communication between the back-end and front-end is one of the most important tasks for a data server in order to have a stably running website. The server handles the input from users as well as send commands to the back-end to obtain the correct data for the visualization. In order to achieve this the Python-library *flask* is used. Flask is a Python library that functions as a micro framework that allows for similar functionality as Apache, but with easier integration of Python. Furthermore, this library contains all the necessary tools to build and maintain a server that communicates with a Python back-end. Flask is not commonly used to run a website, usually Apache is used. The main advantage Flask has over Apache is that the programming language is the same as that of the back-end. This means that the integration of the back-end and the server is achieved easily. The connection with the front-end is achieved by sending HTML-templates with the Javascript coding linked to these templates. Apache would have required more effort to accomplish this. Thus, Flask was selected over Apache to function as a server. This simplifies the project and produces more cohesion.

2.3. Visualization of Data

Most of the calculations for the visualization happen in the front-end. There are several reasons to do this. The main reason to use the front-end to calculate the orbits and motion of objects is to avoid unnecessary strain on the back-end. Unnecessary strain happens when the same calculations are performed multiple times for multiple users. Furthermore, performing the orbit calculations in the back-end means that instead of sending a single TLE for an object, the data of every point that is used to determine the orbit will be sends to the user. This means that the amount of data send between the back-end through the server to the front-end will go up significantly. More data send would also mean that the user has to wait longer to use the visualization, thus making the experience of the tool slower and the performance suffer.

Thus the calculations for the motion and orbits happen in the front-end, where the computing power of the user is used instead of the server. The front-end will receive three types of data. These types are the same as mentioned in the data management section, namely the TLE data, the satellite catalog, and the sub-solar point coordinates.

In order to produce a visualization in Javascript the *Three.js*-library can be used. Three.js is a lightweight Javascript library used to create, display, and animate three dimensional objects in a web browser. The Three.js scripts can be used with the HTML5 canvas element, Scalable Vector Graphics (SVG) or WebGL. Using the canvas, elements can be displayed in a web-browser separately from the rest of the website, as if it was a text box which does not influence the workings of the website. To use the library and visualize the data, a coordinate system, points in an orbit, and the sub-solar point need to set up or calculated.

2.3.1. Coordinate Systems

The coordinate system forms the basis of the visualization. This means that if the coordinate system changes, the calculations have to be performed again. In the literature study, several coordinate systems have been discussed for the visualization. From the literature study, the Earth-Centered Earth-Fixed (ECEF) and the Earth-Centered Inertial (ECI) systems were chosen to be accessible to the user.

ECEF is a geocentric Cartesian coordinate system. It represents positions as x, y, and z coordinates. The origin is defined at the Earth's mass center. The z-axis is defined as being parallel to the Earth's rotational axis, pointing towards north. The x-axis intersects the sphere of the earth at 0^o latitude and 0^o longitude. This means the ECEF rotates with the Earth around its z-axis and that every point on the Earth's surface has fixed coordinates. The y-axis is orthogonal to both the x- and z-axes, completing this right handed coordinate system. The system is shown on the left in Figure 2.1.

ECI is also a geocentric Cartesian coordinate system. Different from the ECEF system, the ECI system is

fixed with respect to the stars and does not rotate with the Earth. The xy-plane coincides with the Earth's equatorial plane. The x-axis is permanently fixed in a direction relative to the celestial sphere. The z axis lies perpendicular to the equatorial plane and extends through the North Pole. When the Earth rotates, the ECI coordinate system does not. This means that, unlike the ECEF system, places on the surface of the Earth have constantly changing coordinates. This system is shown on the right in Figure 2.1.

Both of these coordinate systems are used by the visualization. The user has the option to change the system to view the orbits in both systems. Depending on the coordinate system, calculation of motion and orbits of objects differs.

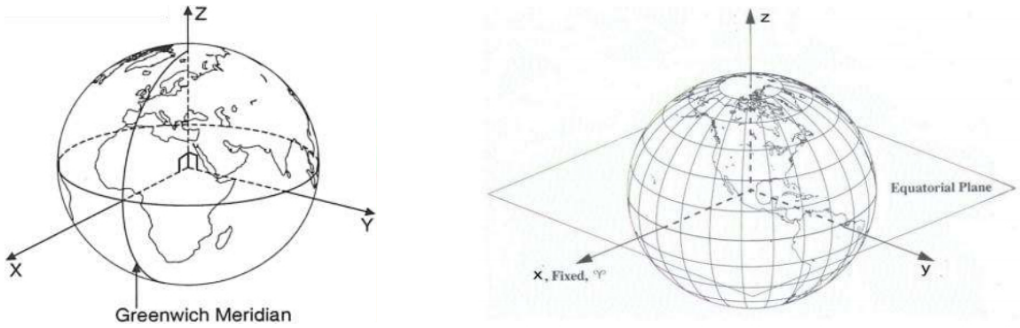


Figure 2.1: Left: Earth-Centered Earth-Fixed Coordinate System. Right: Earth-Centered Inertial Coordinate System. Images are taken from Olexiy Pogurelskiy as part of his Lecture 3: Coordinate Systems[8].

2.3.2. Orbit Calculation

To start the visualization, the first step is to produce a list of points (x, y, z, and t) for each object. To produce the list, the TLE data of an object has to be used. As described in the literature study, TLE data contains the information necessary to produce these points using the SGP4 and SDP4 algorithms. These algorithms are combined in a Javascript library called *satellite.js*. The library makes satellite propagation possible using TLE information in the front-end of a website. The library provides the functions necessary for SGP4/SDP4 calculations, as well as functions to perform coordinate transformations. The algorithms in the library are the Javascript adaptation of a the *sgp4*-library created by Brandon Rhodes using Python[9]. The Python library created by Rhodes passed automated tests that compare his algorithms with the reference algorithms of SGP4/SDP4 by Vallado et al.[12], who originally published their revision in 2006.

The *satellite.js*-library simplifies the algorithms used in the python library to only the functionality needed to track satellites and propagate their paths. The library can be used to propagate the paths of objects, by first converting the TLE into a satellite record. This satellite record is a simple conversion of the TLE into its individual components. The individual components can then be used to calculate points along the orbit of the satellite.

From this point there are two options to obtain points along the orbit. The first one is to use the *.sgp4()*-function. This function allows the user to insert the satellite record and a time in minutes past the TLE epoch point. This method is very useful to determine where a satellite would be after a certain time past the TLE epoch. It however is not very practical to use in a simulation where multiple objects are shown at the same time. Since each satellite has their own TLE epoch time, the program would have to track the time in minutes that has past since the specific epoch time for each satellite. This would result in an extra added complexity to the program. Fortunately, there is a second method to determine the position and velocity of an object: The *.propagate()*-function. This function requires the satellite record and a specific time. The specific time has to be a date around the epoch to not deviate from the actual orbit too much. Propagation away from the TLE epoch leads to errors the further away the propagation is. The *.propagate()*-function has a big advantage over the *sgp4()*-function. This advantage is that multiple objects can be calculated at the same time. This is because it requires a universal simulation time to propagate the satellites position and velocity, instead of a time that is related to the epoch of the satellite.

The accuracy of the propagation can be tested. This can be demonstrated comparing precise orbits with the propagated orbits. To verify the accuracy of the *.propagate()*-function, the precise orbit of CHAMP on January 1st, 2008, was compared with the orbit that was propagated from the TLE from the same day. The precise orbit was supplied to me by my supervisor, Dr. ir. E.N. Doornbos.

The precise orbit was provided in the International Terrestrial Reference Frame (ITRF). The ITRF is a set

of points with their 3-dimensional Cartesian coordinates which realize an ideal reference system, the International Terrestrial Reference System (ITRS). The coordinates in this system are expressed using the ECEF cartesian coordinates. This means that we can compare the coordinates from the ITRF orbit with the propagation of the orbit in the ECEF reference frame from the tool. The radial distance can be compared by simply taking the difference between them. The radial distance for both the precise orbit and the propagated orbit can be calculating using the Pythagorean theorem in three dimensions:

$$r = \sqrt{x^2 + y^2 + z^2}$$

Then taking the difference over a period of 24 hours results in the graph in Figure 2.2.

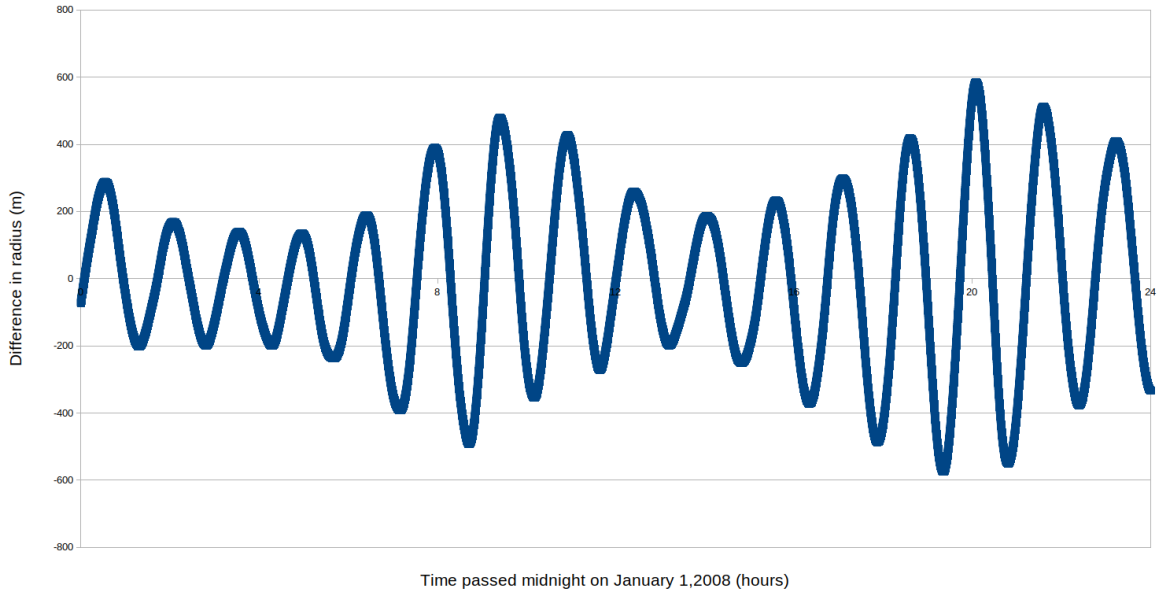


Figure 2.2: Difference between radius of the precise orbit and the propagated orbit.

The velocity comparison is slightly different. Later it is explained how the positional coordinates are converted from the TEME reference frame, which uses the ECI coordinates, to the ECEF system. The same conversion can not be done for the velocity, since it misses the effect of the rotation of the Earth. To include this factor, the cross product of Earth's rotation and the position in the ECEF system has to be included to the conversion:

$$v_{ECEF} = U^{ECI}_{ECEF} v_{ECI} + \frac{dU^{ECI}_{ECEF}}{dt} r_{ECI}$$

In this equation U^{ECI}_{ECEF} represents the conversion matrix between the ECI and ECEF at a certain time. The conversion includes the effects of nutation, procession and Earth's rotation. The equation for this conversion comes equation 5.90 from "Satellite Orbits" by Oliver Montenbruck and Eberhard Gill[7].

After the proper conversion of the velocity, the difference can be calculated in a similar fashion as the radial distance. This results in the graph shown in Figure 2.3

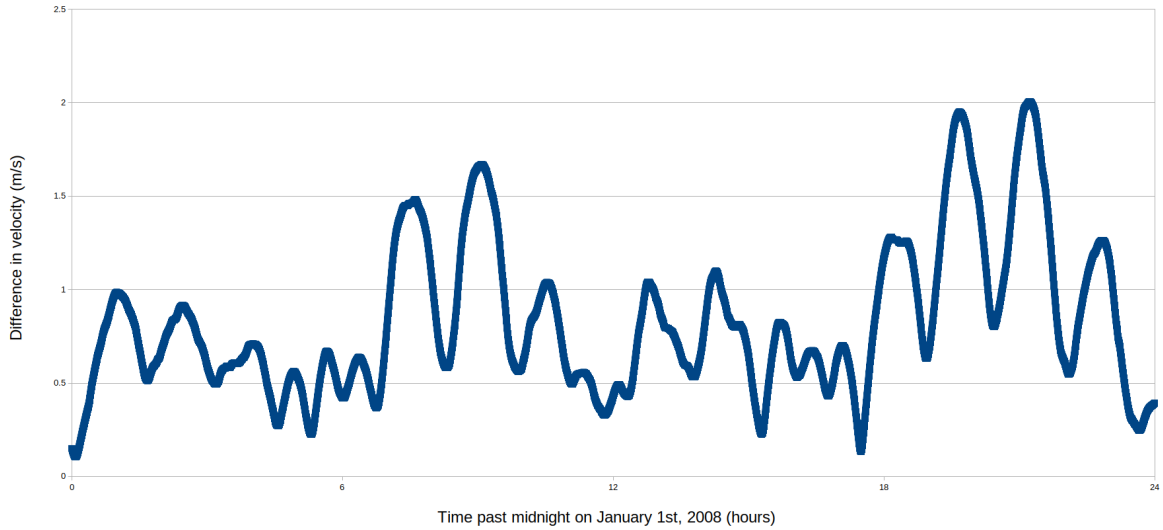


Figure 2.3: Difference between velocity of the precise orbit and the propagated orbit.

For both differences the mean and standard deviation have been calculated. These results are shown in Table 2.1. In this table it can be seen that radial distance and velocity are close. A difference of a few hundred meters is not noticeable by the user, since this difference would result in about a pixel difference on the screen. When zooming out the difference becomes even less noticeable. Another analysis was done for June 1, 2004. This date resulted in similar differences and standard deviations.

	Δ radial distance	Δ velocity
mean	-5.40 m	0.8251 m/s
standard deviation	253.82 m	0.4077 m/s

Table 2.1: The mean and standard deviation for the difference in radial distance and velocity.

The output of the propagation using either *propagate()* or *sgp4()*, is the position and velocity of the object. The reference frame of the ECI coordinates produced by the SGP4/SDP4 orbital models is true equator, mean equinox (TEME) of epoch. When a different coordinate system is required, the position and velocity have to converted to the desired coordinate system. This can also be done using the same *satellite.js*-library. The ECI to ECEF conversion requires Earth-orientation data to be converted properly. These parameters were discussed during the literature study as well.

The *satellite.js*-library has build-in function to convert from ECI to ECEF. To do this, the library requires the vector that needs to be converted and an additional term. This additional term that needs to be calculated for the conversion to ECEF is the Greenwich mean sidereal time (GMST). The GMST is linked to the Earth's orientation parameters. The GMST takes the precession of the Earth into account as well. The *satellite.js*-library has a function that calculates this factor based on the equations in chapter 3 of the Explanatory Supplement to the Astronomical Almanac from 1992 by P. Kenneth Seidelmann[11]. In 2003, the definitions of sidereal time were tweaked slightly to compensate for more accurate determination of astrometry. Using new techniques, like pulsar timing, the more accurate determinations were achieved. The new measurements lead to the introduction of UT1, the mean solar time at 0^o longitude. It measures the rotation of Earth, or Earth Rotation Angle (ERA), and replaced the old definitions of sidereal time. The ERA is defined as

$$\theta(t_U) = 2\pi(0.7790572732640 + 1.00273781191135448\,t_U)$$

In this equation θ is the ERA with t_U being the Julian UT1 date - 2451545.0.

From this equation the GMST can be calculated.

$$GMST(t_U, t) = \theta(t_U) - E_{\text{PREC}}(t)$$

where θ is the ERA and E_{PREC} is the accumulated precession at time t after t_U in hours. This E_{PREC} is calculated using the equation by Seidelmann.

After the position and velocity of the satellite have been determined at the simulation time using the *.propagate()*-function, the next step is to visualize motion. To do this, multiple propagations of the object are taken to represent the position of the satellite over time. To minimize the number of propagations needed to represent an orbit, several points are taken along the orbits. The position of the satellite between the point will be approximated using the an interpolation. The interpolation chosen was a polynomial interpolation, namely the Lagrange polynomials. For a given set of (t_j, x_j) without duplication of any t_j, the Lagrange polynomial is the polynomial of lowest degree that hits every coordinate (t_j, x_j). In this explanation t_j will assume a temporal coordinate and x_j a spacial coordinate, either x, y, or z component of an object. Since the polynomial only works in a two dimensional form, the calculations will be done for each dimension of the object. Given a set of $k+1$ data points, the interpolation polynomial in the Lagrange form is a linear combination:

$$L(t) = \sum_{j=0}^{k} x_j l_j(t)$$

where

$$l_j(t) = \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{t - t_m}{t_j - t_m}$$

where $0 \le j \le k$. To calculate a position at a certain time, simply use that $x_p = L(t_p)$, where (t_p, x_p) is that position at a certain time. In order to avoid having the calculate the $l_j(t)$ each time the interpolation has to done, the equations can be rewritten to the barycentric form. This form has the advantage that the $l_j(t)$ needs no evaluation for each evaluation of $L(t)$. Using the $l(t)$ and its derivative

$$l'(t) = \frac{dl(t)}{dt}\Big|_{t-t_j} = \prod_{\substack{0 \le i \le k \\ i \ne j}} (t_j - t_i) \tag{2.1}$$

the Lagrange basis polynomials can be rewritten as

$$l_j(t) = \frac{l(t)}{l'(t_j)(t - t_j)}.$$

Then using the definition of barycentric weights

$$w_j = \frac{1}{l'(t_j)}, \tag{2.2}$$

the equation becomes

$$l_j(t) = l(t)\frac{w_j}{t - t_j}.$$

This means that the polynomial interpolation in the Lagrange form can be written as

$$L(t) = l(t) \sum_{j=0}^{k} \frac{w_j}{t - t_j} x_j$$

By barycentric interpolation, we can consider the function $g(t) \equiv 1$. This function in a barycentric interpolation is the same as

$$g(t) = l(t) \sum_{j=0}^{k} \frac{w_j}{t - t_j}.$$

Then dividing $L(t)$ by $g(t)$ results in

$$\frac{L(t)}{g(t)} = L(t) = \frac{\sum_{j=0}^{k} \frac{w_j}{t - t_j} x_j}{\sum_{j=0}^{k} \frac{w_j}{t - t_j}}. \tag{2.3}$$

These equations are taken from "Barycentric Lagrange Interpolation" by Jean-Paul Berrut and Lloyd N. Trefethen[2]. The last function indeed does not require the $l(t)$ evaluation to calculate the position. Therefore, it is optimized for speed and can be used quickly in the code. This will enhance the speed of the tool and decrease the load on the users computer while running the tool.

2.3.3. The satellite catalog

The easiest of the three data types to implement into the front-end is the satellite catalog. The satellite catalog contains detailed information about the satellite that is not available from the TLE. Besides the full name, international designator, launch date, and launch site, the catalog provides information about the type of object, various orbital values, dates of decay (if already occured) and the radar cross section of the object. The user will be able to see this information if it searched the satellite or selected it from the visualization. The type of object will determine the color of the object in the visualization. These colors can be changed in a global variables files. The various orbital values are the apogee, perigee, period, and inclination. These will all be shown if the user selects the object.

The satellite catalog will also be used in a search algorithm the user can use to select certain satellites and show their orbits. The search algorithm will compare a search from the user with the name and the catalog id to find matches with the satellite catalog and the items that are displayed. This means that the user will be able to search everything that was originally loaded onto the screen, select a satellite to focus on just this satellite. The satellite catalog is used as the database that needs to be searched, since a user is more likely to know the name of a satellite rather than the international designator or catalog id.

3

Methodology

This chapter focuses on the methods used to create the tool. In order to properly understand the tool, the chapter will be broken up into the three sections: back-end, server, and front-end. The global set-up of the tool is displayed in Figure 3.1.

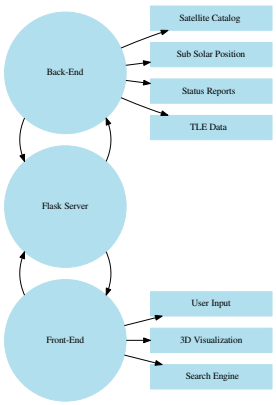


Figure 3.1: Global overview of tool workings.

In this figure the ellipses display the main part of the code. The rectangles show the main parts within each part of the code.

3.1. Flask Server

The server functions as the communication between the back-end and the front-end. The server is necessary to run optimal in order to prevent slow experience for the user. It also organizes the functions of the back-end and send the front-end the information it needs to show the right page with the right information. The set-up for the server is shown in Figure 3.2.

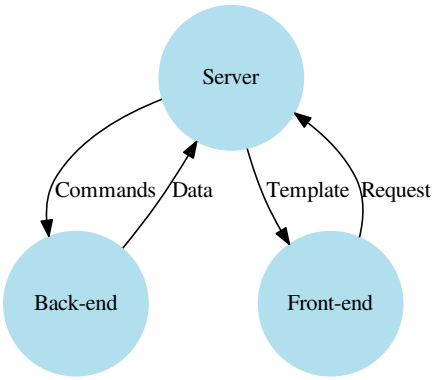


Figure 3.2: Overview of the server

The server takes a request from the front-end. It then analyzes what the request means and send the appropriate commands to the back-end. Then based on the same request, a template is send to the front-end along with the data from the back-end. This will determine the view for the user and the data that is to be used in the visualization. The request from the front-end can be one of several:

1. First Load
2. Reload Today's date
3. Search Date
4. About

Each of these requests has their own end point on the website. The end point on a website is defined as whatever comes after the '/' in the URL. For this tool, the end points are *'/'*, *'/home'*, *'/date/<yyyy-mm-dd>'*, and *'/about'*. To understand how the server reacts to each of these requests, we have to take a closer look.

3.1.1. First Load

The first load request happens when the user first contacts the web-application. The first end point the user uses is the *'/'* end point. Upon accessing this end point, the server receives a request for a template from the front-end. This template will be the loading template.The first thing the server does is send back a template to the user that indicates that a page is being loaded.This return of a template without sending commands to the back-end means that the user will see this template nearly instantly. The server then waits for the next request to come in. The load template is a very basic HTML page that shows the menu of the visualization. Furthermore the template shows a loading message and a small waiting animation. Lastly this template redirects the user to a new end point. This is the */home*-page. This page triggers the server to send todays data for the visualization.

It seems that this second trigger could happen in the first request effectively eliminating a request on the server. However, loading todays data can take a while. This would mean that the user would look at an empty white screen for the same period wondering if the website is even working. Since the template for the website cannot be send before the data is send, sending the template without the data would result in numerous errors: The visualization would start without any available data, the search engine would miss its data, etc. Therefore the extra step needs to be taken to send a loading screen to the user.

To understand the first load better, Figure 3.3 was created. In this figure, the green lines indicate the process of the first load. The red lines shows the requests from the *'/home'* end point. And the blue lines indicate the return to the front-end. The numbers also indicate the order of execution.

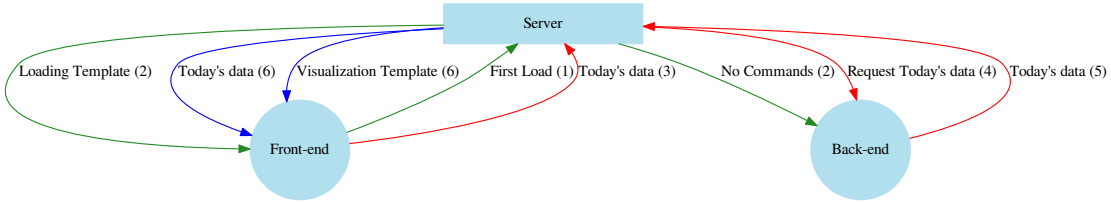


Figure 3.3: First load protocol

When the server is requested to send today's data, the same commands are used to reload today's date. The second part of the first load is to load the most recent data. This is explained in more detail in the next section, subsection 3.1.2.

3.1.2. Today's date

Once the server receives the request for today's data from the *'/home'* end point, the server will send specific commands to the back-end. The server will also send a template to the front-end that will use the data send by the back-end to create a visualization. This interaction between the front-end, server, and back-end is demonstrated by the red lines in Figure 3.3. What is unclear is which commands are send by the server to the back-end. Figure 3.4 demonstrates the commands that are send between the server and the back-end and the possible responses.

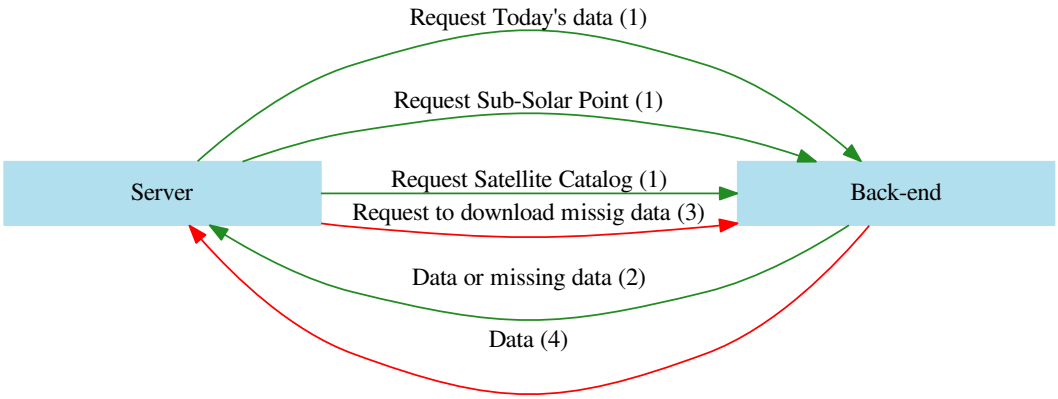


Figure 3.4: Server and back-end interaction for obtaining today's date. Order is indicated by the numbers.

After receiving the requests for today's data, the server will send three commands to the back-end. These commands are to obtain today's TLE data, the sub-solar point, and the satellite catalog. The back-end will take these commands, execute the algorithms that are required to fulfill the commands and send one of two responses back to the server: the data or a list of missing data. This response completes the green cycle in Figure 3.4. When the response is the data, it means that all data is available and ready to be send to the front-end. At this point no more commands will be send to the back-end, eliminating the red cycle in the figure. When the response is a list of missing data, the server will send a new command that requests the back-end to download the missing data and then send it back. This list of missing data triggers the red cycle of Figure 3.4. The server will either receive the data, or it will receive a response indicating that the download failed. If this happens, there will be no data send to the front-end. Furthermore, a specific message about the error will be saved to a log file, so it can be examined later. This is useful for maintenance of the tool.

3.1.3. Search Date

Using the tool it is also possible to select a specific date. This means that historical data can be obtained to be visualized by the tool. In order for this to happen, the user will have to enter a date and click a submit button. The button will redirect the user to a different end point, namely */data/<yyyy-mm-dd>*. Where *<yyyy-mm-dd>* represents the date the user has entered in the format year-month-day. This redirect to the page triggers the server. The server will perform very similar commands as described in subsection 3.1.2. The main difference is that the date used is not today's date.

The user is allows to enter any date after 1950. The database of the tool starts out being incomplete, since not every historical dataset was downloaded. When the server sends the first commands to the back-end, the response from the back-end will most likely be that the TLE data is missing. If this is the case, the red cycle in Figure 3.4 will be triggered to download the missing TLE data for that specific date. The TLE data from this specific date will be stored in the database. This means that as the tool is used more, the database becomes more complete and the response will become faster.

Once the new historical data is downloaded, the user will be presented the data with the same template as before. This means that the website provides the same tool but with different data to the user.

3.1.4. About

The last official end point that is currently available to the user is the *'/about'* end point. This end point is accessed when a user presses the about section of the website. The server will respond with a template that displays a menu to get back to the visualization and a text. The text contains information about the project and a short tutorial to use the tool. The server does not require to send any commands to the back-end, since there is no data required for this page.

3.2. Back-end

For the tool to work properly, data has to be stored and be accessible quickly. The back-end of the tool is mainly used to manage the data sources and to provide the front-end with the correct data sets to use for the visualization. The back-end of the tool is able to quickly show stored data. This is needed to reduce the time a user has to wait for the visualization to shows up. To function properly, the back-end is split up into four different files, each having a different purpose for the code. The files are the split up by their functions: login details, data downloading, data reading, and the support functions. The next few sections these files will be explained. The functions within each file are shown in Figure 3.5.

3.2.1. Connections

The *connections.py* file is the only file from this tool that is not available in the git repository, other than a template given in the *README*. This is due to the nature of the information that is contained within the file. The file contains functions that return the user name and password of accounts that are used to access information or data needed for the tool. The file currently has only one function. The sole purpose is to return the user name and password to my personal account at space-track.org. This is the reason I have not made this information available, since I do not want others to have access to my personal account.

The code in this file is limited to these lines of code. To ensure privacy the user name and password are changed.

```
#connections.py
def space_track():
    space_track_username = "actual_user_name"
    space_track_password = "actual_pass_word"
    return7space_track_username, space_track_password
```

These lines of code are straightforward. Whenever the function *s*pace_track() is called, it will return the user name and password to my personal account at space-track.org. This function can be called from another python file if it is imported into the file.

In the future more functions can be added that contain the login information for other sources that require an account to be used. This could for example be a website that has instrument data from a specific satellite

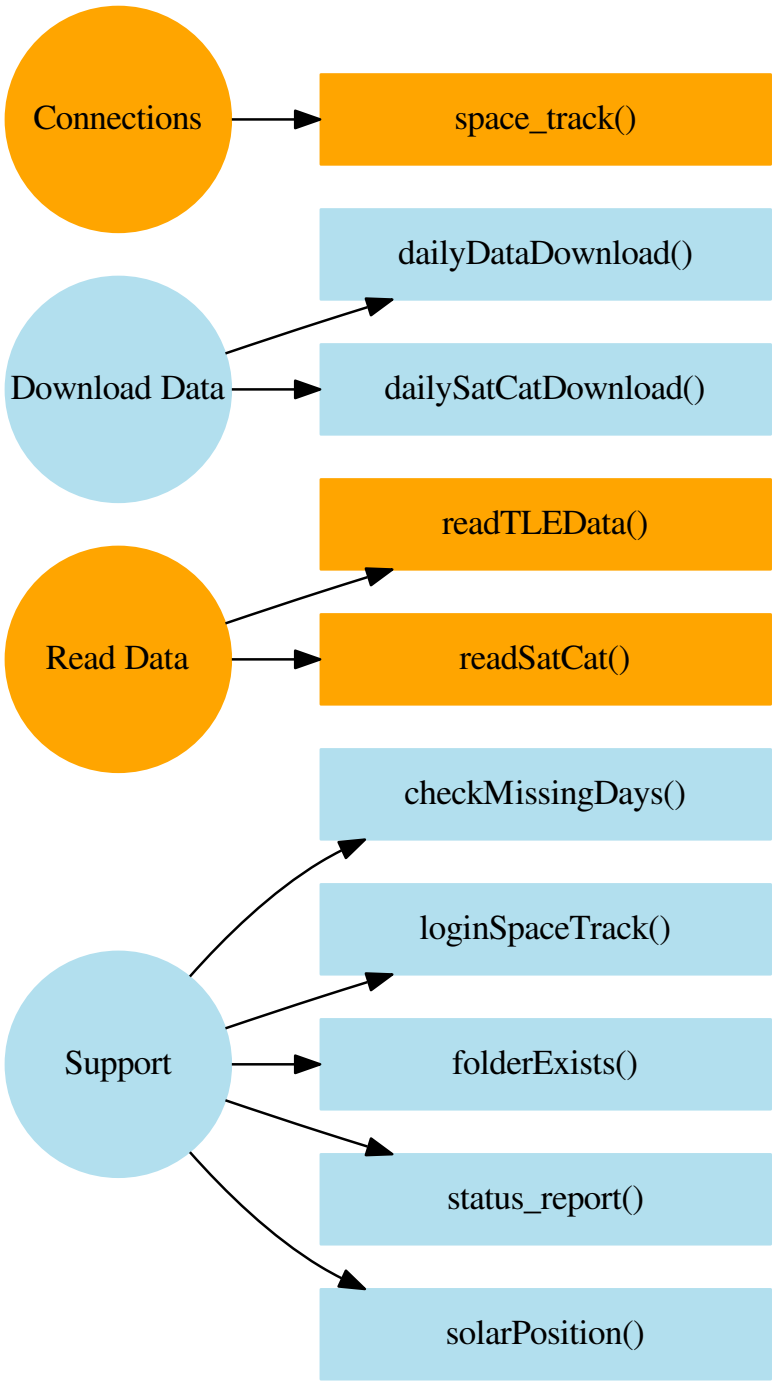


Figure 3.5: The global overview of the back-end with its files and containing functions.

that could be visualized in a new way.

3.2.2. Download Data

Another file is the download data file, *downloadData.py*. This file is bigger than the connections file and handles all the incoming data. The data is then written to a storage location, where it will be stored indefinitely. From this storage location the data can be accessed by the data reading function.

Downloading and storing data means that there should be some form of data management. The data management is used to ensure that there are no duplicate files, that folders exist to store the data, and that the data that is stored is not corrupt. The data downloading file contains two active functions: *dailyDataDownload()* and *dailySatCatDownload()*.

dailyDataDownload()

The *dailyDataDownload()*-function is a function that downloads TLE data. The function downloads all TLE data from a specific date. The function is called upon once a day to retrieve the most recent TLE data and store in the database. Furthermore, the function is used to download the TLE information from historical dates. From the *space-track.org*-website it is possible to set a range of dates for the TLE information. This function sets this range to one day and obtains the information. This way the database will fill up over time and prevent downloading information that is already in the database.

The function requires two inputs. The first one is the date or dates from which data is missing. When the function is run for the most recent data, it calls upon the *checkMissingDays()* function from within the support file. This function can be found in subsection 3.2.4. This function returns a list that contains every date in the five days leading up the requested date. Another condition to get on the list is that the data is not in the database yet. This means that if the function runs everyday for the most recent data, there will only be one date that needs to be downloaded, since the other dates are already in the database.

For historical dates the function returns a similar list. The data that is not in the database will then be downloaded by the *dailyDataDownload()* function. The tool will download the requested date and the five days leading up to this date. The reason to download the five days leading up to the requested date is that not every object gets a TLE every day. Downloading the leading five days has proven sufficiently to download TLE's from all satellites.

The second input the function required is an actively logged in session to the *space-track.org*-website. The actively logged in session is needed to be able to download the information from *space-track.org*.

In order to find the correct file, a URL has to be build. Before creating the URL the *folderExists()*-function is called to check if there exists a folder with the correct path and naming for the data to be stored in. If there is not, the function creates the folder. The URL is build up from two dates: the date of the missing day and the day after. The URL uses both dates at midnight. The URL is then build with a core, which is independent from the date, and these two dates. This means that the data will span exactly one day. The following lines of code show how the next day is determined, the *folderExists()*-function is called, and how the URL is build.

```
# ------------------------------------------------------------------------------#
# Download file from missing day -----------------------------------------------#
#-------------------------------------------------------------------------------#
def dailyDataDownload(log_in_connection, missing_date):
    next_day = missing_date + dt.timedelta(days=1)
    missing_date = missing_date.date()
    next_day = next_day.date()
    print(missing_date, next_day)
    folderExists(missing_date.year, missing_date.month)
    url = "https://www.space-track.org/basicspacedata/query/class/tle/EPOCH/{}%2000:00:
                                                00--{}%2000:00:00/orderby/TLE_LINE1/
                                                format/tle".format(missing_date,
                                                next_day)
```

Now that the URL is available, the next step is to attempt a download of the data. Using the active logged in session to space-track.org that was included in the variables of the function, the URL can be called using a *get* call. A *get* call is a call that retrieves information from the chosen URL. The *get* call will only work when a user is logged in and the session is used. If someone tries to access the data without being logged in, a authentication error is prompted.

The *get* call has a timeout of 5 seconds. This means that if there is no response, or no connection found with the website, the connection will be terminated automatically after 5 seconds. When the connection is

terminated a message will be logged that the connection failed. In theory this will only happen if the tool is used in an offline mode, or if there is maintenance on the side of *space-track.org*.

When there is no connection or timeout error, the next check on the data is the response code. The response code is a code that shows the status of the information returned. For example, the status 404 is linked to a page that does not exist. The code that is paired with a good response is 200. When this code is send with the information, the response was good and the data that is send should not be corrupted. If the response code is 200, the file will be stored in the folder that corresponds with the data. The *folderExists()*-function checked if that folder was indeed available.

Errors during the process of writing the data to this folder result in the deletion of the data in the database. This is done to prevent files that are only partially written to be in the database. It ensures that the database does not fill up with corrupt data files. It also ensures that the data accesses is free of corruption due to the writing of the file in the database.

satCatDownload()

The *satCatDownload()*-function is used to download the satellite catalog information. The method of obtaining the information is very similarly to the *dailyDataDownload()*-function, since the information originates from the same source. Since the source is *space-track.org*, the function needs to have an active connection with space-track.org. Unlike the *dailyDataDownload()*, the *satCatDownload()* is not based on dates. The satellite catalog is a single file that contains the information of every satellite and object that has ever been tracked. The file is updated as new objects are tracked and the information about the objects are updated when needed. This eliminates the need for storing every version of the catalog. Hence, the catalog only has to be in the database once. To avoid potential problems with new versions, the *satCatDownload()*-function has redundancy build in though. The redundancy is build in by keeping an older version of the satellite catalog in the database as well. When a catalog file is one day or older in age, it will be replaced by a newly downloaded version. The original version will then be renamed to be available as a back-up.

In the next snippet, the code for checking the existence and age of the file, and renaming of the file is shown.

```
satcat_file = Path('SatelliteCatalog/SatCat.txt')
if satcat_file.is_file():
  # obtain the file modification time
  satcat_stats = os.stat('SatelliteCatalog/SatCat.txt')
  file_age = satcat_stats.st_mtime
  # check if the file is a day or older (86400 seconds)
  time_delta = dt.datetime.now().timestamp() - file_age
  if time_delta < 86400: return
  else:
    # try to rename the old file to old
    try:
      os.rename('SatelliteCatalog/SatCat.txt', 'SatelliteCatalog/SatCat_old.txt')
    except OSError as e:
      support.status_report('dailySatCatDownload', 'Renaming the satellite catalog was
                                          unsuccessful. Due to {}.'.format(e))
```

Downloading of the satellite catalog uses a static URL. The difference between the satellite catalog download and TLE download is the fact that the URL is static, meaning that the URL does not change. The URL is not dependent on any variables, like dates. The URL that is used is

https://www.space-track.org/basicspacedata/query/class/satcat/orderby/NORAD_CAT_ID%20asc/metadata/false.

This URL does require the user to be logged in, which is why the function requires the active logged in session to space-track.org to function properly. The function does not return anything, since the data will be stored in the database and does not require an output. Reading the satellite catalog and preparing the data for the front-end will be done in the *readSatCat()*-function.

3.2.3. Read Data

The third file in the back-end is the read data file. This file takes data from the database and formats it in such a way that the front-end can use it for the visualization or to display information to the user. In order to be utilized properly by the front-end, the data should be supplied in a JSON format. An small example of JSON format is : *{item1: data1, item2:data2}*. JSON format also allows for nested JSON dictionaries. This means that instead of having $data1$ being a single value, it could be another JSON dictionary.

There are two types of data that are needed for the visualization to properly work. The first one is actual TLE data. The second one is the satellite catalog. The TLE data is obviously used to determine positions and orbits in the visualization. The satellite catalog is used to display more information to the user. From the satellite catalog, the visualization determines the color of the objects. The colors depend on the type of object, for example green is payload, whereas debris is red.

readTLEData()

The core of the visualization is based around the information from the two-line elements of the objects. This makes them the most important data to be send to the front-end. The orbital parameters within the TLE data are stored in the database, but need to be retrieved and formatted properly to be processed by the front-end. The *readTLEData()*-function supplies the front-end with the data correctly formatted. In order to do this, the function has three inputs. The function requires the year, month, and day of the data that needs to be supplied.

The function carefully starts by checking if there is a database available. If it is missing, the function logs in to the *space-track.org*-website and calls the *dataDownload()*-function to download the missing data from the date entered as an input. At this point, the database is available.

Next, the function enters a loop that runs five times. As mentioned earlier, the TLE data from one day might miss information from certain objects that did not get a TLE for that day. To compensate for this, the *readTLEData()*-function reads data from the five days that lead up to the date. Hence the loop has to run five times. Starting with the date closest to the chosen day, the function checks for the existence of the file in the database. When the file is missing from the database, the *dailyDataDownload()*-function is called upon again to download the missing day. The next snippet of code demonstrates the start of the loop, with checking the existence of the file and calling the download function if needed.

```
for i in range(0, 5):
  use_date -= dt.timedelta(days=1)
    file_location = 'Data/{}/{}/DailyTLE_{}-{}-{}'.format(use_date.year, int(use_date.
                                                month), use_date.year, str(use_date.
                                                month).zfill(2), str(use_date.day).
                                                zfill(2))
    orbit_file = Path(file_location)

    # check if file exists
    if not orbit_file.is_file():
    # not a folder, means the data is not available. Therefore start downloading data
                                                again.
       login = support.loginSpaceTrack()
       date = dt.datetime(year=use_date.year, month=use_date.month, day=use_date.day)
       dailyDataDownload(login, date)
```

The next step in the loop is to add the TLE in a JSON dictionary. The function loops through the lines of the data file two lines at the time. Since we are looking at a file of two-line elements it is obvious that two lines are read at the same time. Then the lines are checked on being the first and second line of the same satellite. If the order is wrong, or the satellite numbers do not match, the function sends a status update. Since it should not happen that the order is wrong or that the satellites do not match, it means that the file is somehow not correct. If the numbers and satellites do match with the expected, the TLE will be added to the dictionary. To make sure that there is only one TLE for each satellite, the dictionary is checked on the existence of this satellite. If the satellite is already in the dictionary, the TLE will not be added. Here the order of reading the files is important. The files are read in order from new to old, which means that the newest date will have the TLE in the dictionary first. Thus the TLE closest to the desired date is send to the front-end.

The test to see if the TLE is in the dictionary and adding the TLE in the correct formatting is shown in the following snippet.

```
try:
  test_orbit = orbit_dict[sat1]
except KeyError:
  orbit_dict[sat1] = {'tle1': line1.strip('\n'), 'tle2': line2.strip('\n')}
```

In this example, *sat1* is the variable that contains the number under which the satellite is known in the satellite catalog and the *orbit_dict* is the JSON dictionary that will be send to the front-end once the five days are checked and the loops have finished.

readSatCat()

The second type of data that the front-end needs for an optimal performance is the satellite catalog. The satellite catalog contains detailed information about every object that is flying or has ever flown near Earth. In the catalog information like launch date and perigee are available, but also the decay date of older objects. Another piece of information that is important to determine the color of each object in the visualization is the type of object.

The *readSatCat()*-function starts by checking if the file is available in the database and if the file is less than one day old. This is done by invoking the *loginSpaceTrack()*-function followed by the *downloadSatCat()*-function. The connection made with *space-track.org* is passed along as a variable in the *downloadSatCat()*-function. These functions are called in the read function, so that the server does not have to send a specific command to the back-end to check the catalog. It could have been included in the server part of the tool, but the choice was made to do it strictly in the back-end. The gained benefit was that testing the functionality was easier.

Next the function will try to open the satellite catalog in the database. The satellite catalog is already formatted in such a way that no more conversions are needed to be able to send it to the front-end. The function tests if the file is available. When the file is missing, the back-up file will be used and another attempt is made to download the catalog. The next snippet shows this.

```
# check if file exists
if satcat_file.is_file():
    file = open(file_location, 'r')
    return file.read()
else:
    # file doesn't exist, thus need to download the catalog.
    dailySatCatDownload(connection)
    support.status_report('readSatCat()', 'The file did not exist')

    file_location = 'SatelliteCatalog/SatCat_old.txt'
    satcat_file = Path(file_location)

    if satcat_file.is_file():
        file = open(file_location, 'r')
        return file.read()
```

The back-up file is the last file that was opened properly before it was older than one day. This file is used, because it could happen that the newest file is corrupt, which would cause problems.

If the function completely fails to load any satellite data, the front-end will not have a satellite catalog. This would mean that the user has no information about the orbits and objects that are shown in the visualization. It also prevents the colors in the visualization to occur. The error which caused the lack of satellite catalog is documented.

The *readSatCat()*-function does not require an input, but produces a dictonary with the information from the satellite catalog.

3.2.4. Support

The last file in the back-end, the *support.py* file, contains all the functions that support the other files. It also provides a function that can document errors or any other messages that occurred during any function. The file contains five functions as shown in Figure 3.5. Each function plays its own part in the back-end.

checkMissingDays()

The first function in the support file is the *checkMissingDays()*-function. This function takes an integer number that represents how many days the function should check. Then the function checks the number of days prior to today for missing data. It checks if there is TLE data for each day available. If the data is missing, that date will be added to a list. This list is returned.

The first thing the function does is obtain the current date using the *datetime*-library. Then, the function loops through the number of days that are prior to that date. In the loop, the day that is checked is the *test_day*. The date will decrease by one day every iteration of the loop. This day is in the *datetime*-format. Python has several types of variables, like string, integer, and arrays, but it also supports *datetimes* from the *datetime*-library. These are dates formatted in a way that the date, time, and timezone information is stored within the variable. The function then checks if there exists a data path to a daily TLE file of that date. The

location of this file should be in the *Data/year/month/* folder. The path to this file is defined in the code like this

```
p = Path("Data/{}/{}/DailyTLE_{}".format(test_day.year, test_day.month, test_day))
```

Next, the function checks if the path exists. If the file does not exist, the day is added to the missing days list. This is done for every iteration of the loop until the desired number of days is been checked. After the loop the function will return a list of the *datetimes* that need data.

loginSpaceTrack()

The second function, the *loginSpaceTrack()*-function, is a function to opens a connection between the tool and the space-track.org-website. In order to do this the functions calls the *space_track()*-function from the *connections.py*-file. It obtains the user name and password from the output of that function and use it to open an active session with *space-track*. To log in to *space-track.org* the function sends a JSON file with the user name and passport to an authentication api. This is shown in the following snippet.

```
# open a session
with requests.Session() as session:
  url_login = "https://www.space-track.org/auth/login"

  try:
    # test the connection
    session.get(url_login, timeout=5)

    # use login credentials to login and obtain active session.
    session.post(url_login, data=payload_login)
  except (TimeoutError, ConnectionError, requests.exceptions.ConnectionError, requests.
                                          exceptions.ReadTimeout)  as e:
    status_report('loginSpaceTrack', e)
    return 0

return session
```

In the case of a failed connection or response, a status report is logged and the function will return the value 0. This response is caught by the other functions utilizing the *loginSpaceTrack()*-function. The session will be terminated. The connection can fail for multiple reasons. The most common reasons are a connection error or a time out error. These errors occur when the connection with the internet is either not working or too slow to establish a stable connection. The function will either return the active session or a zero, indicating that the connection failed. If the connection failed, the tool will attempt to download data at a later time.

folderExists()

The third function in the Download Data file is the *folderExists()*-function. This function is used to determine if a certain folder containing the TLE data exists. The function requires two variables to work. The year and the month of the folder that needs to be checked. As shown earlier, daily data is stored under *Data/year/month/* folder. This function will check each layer of this folder. If it finds that the folder does not exist the function will create the folder and the underlying folders as needed.

First, it checks for the existence of the *Data/* folder. The only time when this folder does not exist is when the tool is migrated to a new location and no data has been copied to the new location. Next it will check the *Data/year/*-folder. This folder will be created whenever data is downloaded from a year that does not have data yet. This could be when someone requests historical data from a year without any data yet. Similarly to the *Data/year/*-folder the *Data/year/month/*-folder will be checked.

This function could potentially fail when either the system has a corrupt hard disk, when the disk is full, or when the application has insufficient rights to write on the disk. In the case of a fail, a message is logged detailing the error. The function has no specific output and will just return upon completion.

status_report()

The *status_report()*-function is an important maintenance function for the back-end. The function logs messages from every function to specific files for a day. These logs are stored in the database as well. Whenever a function fails or an error occurs this function is called. The call on this function includes two strings. The first one is the name of the function. The name of the function is important to be able to track where the error occurred. The second string is a message containing information on the error that occurred.

The function saves the error message in a file that is linked to the date on which it occurred. The file is opened and the message is added to the file. This means that every error from that day will be stored in the same file. The next snippet shows how the functions checks if there exists a file for the error of the day and how it writes the error to the file.

```
# check if a status file exists for today
p = Path('Status/Status_{}'.format(date))
if not p.is_file():
    f = open('Status/Status_{}'.format(date), "w+")
    f.write('[{}]: {} with the message: {}\n'.format(time, function_name, message))
    f.close()
else:
    f = open('Status/Status_{}'.format(date), "a+")
    f.write('[{}]: {} with the message: {}\n'.format(time, function_name, message))
    f.close()
```

In this snippet, *time* is the time at which the error happened.

solarPosition()

The last function in the support file is the *solarPosition()*-function. This function determines the position of the sub-solar point. This point is used in the visualization to determine where the light has to shine upon. It also is used to add an icon on the Earth showing where it is located. The function uses a python library called *ephem*. In subsection 2.1.3, it is explained how the sub solar coordinates are calculated using this library. The code to actually perform this calculation is shown below.

```
# greenwich position
greenwich = ephem.Observer()
greenwich.lat = "0"
greenwich.lon = "0"
greenwich.date = time
sun = ephem.Sun(greenwich)
sun.compute(greenwich.date)
sun_lon = math.degrees(sun.ra - greenwich.sidereal_time())
# limit the longitude between -180 and 180 degrees.
if sun_lon < -180.0:
    sun_lon = 360.0 + sun_lon
elif sun_lon > 180.0:
    sun_lon = sun_lon - 360.0
sun_lat = math.degrees(sun.dec)
print(time, sun_lon, sun_lat)
return [time, sun_lon, sun_lat]
```

The function starts by defining the observers position, which is called *greenwich*. Then it sets the time of the observer to the time that is passed as a variable in the function or to right now if there is no time passed. Then the function computes the latitude and longitude of the sub-solar point and using the method described earlier, the function calculates the new longitude of the sub-solar point. The last thing the function does is return an array with the time, latitude, and longitude of the sub-solar point.

3.2.5. Developed Unimplemented Code

The tool went through several iterations before landing at its current position. This means that there have also been some part of the code that are currently not used, but are developed. One of these functions is a way to download data for a specific time range and specific satellites. Another function is the functions that download Earth Orientation parameters. This function is still active, but the data is not used. Because these functions could be useful for future development, a description is added.

specificDataDownload()

Similarly to the *dailyDataDownload()*-function, the *specificDataDownload()*-function downloads TLE data. The difference between the functions is that the latter function downloads data over a specific period of time of specific satellites rather than a daily file with all the information. Another difference is that this function will not store the data in the database. Instead, the data is directly send to the user in the front-end. This is done to reduce the amount of data that is stored in the database, because specific searches could lead to duplication of data. In the future it might be useful to store these searches if users can get there own account.

The account could hold the information from specific searches and could have the ability to add your own data. More about this in the recommendations later.

One more thing to note about this function is the function is not available to the user at the moment. However, there is a working end point at the server that could be accessed through the URL. This means that only if you know that the end point is there, you could have your personal search visualized. The function is fully working and has been tested.

In the following code snippet, it shows how the URL is build for this specific search.

```
# ------------------------------------------------------------------------------------#
# Download file for missing day or specific satellite -------------------------------#
#-------------------------------------------------------------------------------------#
def specificDataDownload(log_in_connection, satellite, start_date, end_date):
    if satellite == 'all':
        satellite = '1--' + support.lastCatalogEntry()

    url = 'https://www.space-track.org/basicspacedata/query/class/tle/EPOCH/{}--{}/
                                                NORAD_CAT_ID/{}' \
          '/orderby/TLE_LINE1%20ASC/format/tle'.format(start_date, end_date, satellite)
```

The function first checks if all satellites are requested or just specific ones. If all satellites are selected, the URL requires a string that looks like this: *1–last available entry*. To do this a support function is called that returns the last entry in the satellite catalog. This is another function that is developed but not implemented yet. This basic function opens the satellite catalog and reads the number from the last available satellite. This number is then returned to be used.

The next step in the *specificDataDownload*-fuction is to the start time and end time are added to the URL. The satellites are added in a filter of the URL. The same method as the *dailyDataDownload()* is used to retrieve the data. The last part of the function is to rewrite the data in a format that can be interpreted by the front-end.

The data that is retrieved from *space-track.org* can contain multiple TLE from a single satellite. Currently the front-end solves this by overwriting the previously know TLE for this object. This means that it is not possible for the front-end to use multiple TLE over a time period to optimize the accuracy of the orbits.

dailyEOPDownload()

The *dailyEOPDownload()* is one of the only functions that has no input or outputs. This means that the function is self contained and is not dependent on other functions or variables. This function updates the Earth Orientation Parameters, EOP, daily. The function checks if there exists an EOP file in the database. If there is a file it determines the age of the file. When the file is older than one day, a new file will be downloaded. The original file will be renamed to a back-up file. This can be done because the EOP data is one file that contains the EOP information since recording started. It is updated with one line every day, showing the new parameters.

The parameters are currently just stored, since they are not required for the calculation of the orbits. This is because of a library that is used in the front-end, where these parameters are created using a model. Thus, the function that calculates the motion of satellites has these parameters build in. Furthermore that function propagates the EOP within the function, therefore eliminating the need to update the EOP every day.

Because of the library that is used to determine positions has the EOP build in, this data is currently not used in the visualization, but just stored. In the future there might be use for the EOP, which is why this function is mentioned.

3.3. Front-end

Now that the back-end and server set up is explained, it is time to dive into the front-end. The front-end is what the user will see. It also performs calculations for the orbits on the user's side. This means that instead of having a heavy back-end where everything is calculated, the user uses his/her own computer to automatically perform these calculations. It also limits the amount of information that has to be transmitted from the back-end to the front-end. For example, instead of sending 100 points along the orbit the back-end will only send the TLE information. This would ultimately save computing power on the back-end side and result in a faster working tool, if the user's computer is fast enough.

The front-end is build in multiple coding languages that work together to form a webpage that can be viewed by the user. The back bone of a webpage are the HTML templates. There are four templates available: The first load, visualization, error, and about templates. Out of these four templates, only the visualization template contains the tool. This template is also that template that is used the most often.

In order to keep a uniform styling between the templates, CSS is used. This styling language style every object that has a certain class the same way. This means that for example every button in the menu, will be styled the same. It also makes the webpage look more professional and eliminates the need to write styling in the HTML. It speeds up the process of writing new templates and allows for future templates to be added in the same style. For the CSS there is a clear split between the webdesign and the menu. These are the two main components of the tool that are influenced by styling.

The menu is the ultimate way for the user to interact with the visualization. In the menu, the user is able to adjust several parameters of the visualization. The main adjustments the user can make are time manipulation, visualization manipulation, reference frame changes, and searching for specific objects. These options are all available in the menu. The visualization manipulation deals with changing the look of the visualization, but also the groups that are selected.

The last part of the front-end is the actual visualization. The visualization of all data is done using a specific Javascript library, namely *three.js*. This library is capable of drawing a canvas with various objects in it. Furthermore, the library allows for updating of just the visualization without having to update the entire webpage. This makes the application lighter for the computer and eliminates lag due to loading times.

A global overview of the front-end is shown in Figure 3.7.

3.3.1. Templates

The templates of a webpage establish the back bone. This means that the template sets every element in the order that is desired as well as command Javascripts to run in a certain order. The order of these scripts is important, because some scripts depend on functions from other scripts to work properly.

First load template

The first load template is the template that is first send by the server as soon as the user enters the domain of the tool. The template build the menu and shows a loading animation. Once the template is build, it redirects the user to the visualization template.

The first load template is in place to show the user that the calculations are performed in the back-end to obtain the data of today. Without this first template, the user would see just a white screen, while the page is loading. Now the user sees the menu, a message, and the loading animation.

The template starts with a header that opens the relevant style sheets and a small script. The style sheets that are used are the house style of the website, which will be discussed later, a *jquery* style sheet[1], and a font style sheet[2]. The last two style sheets are style sheets developed by other webdesigners. These designers made their code public so that other people can enjoy the designs as well. The first style sheet contains the style for the waiting animation. The style sheet has a little script in it that shows nine block that grow and shrink to look like a waiting animation. The second style sheet contains the search icon for the search bar.

After loading the styling sheets, the header continues with the following script.

```
1  <!-- Scripts used in the code -->
2  <script>
3    function load(){
4      window.location.href = "/home";
5    };
6  </script>
```

[1] http://code.jquery.com/ui/1.9.2/themes/base/jquery-ui.css
[2] https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css

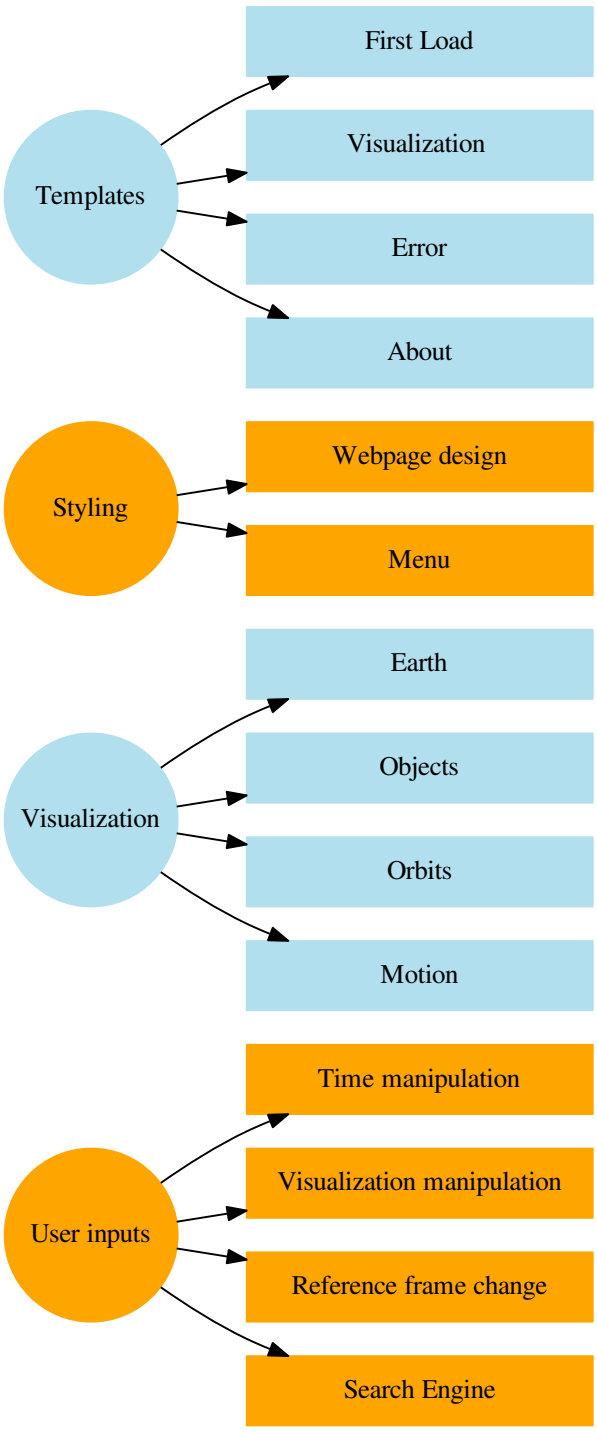


Figure 3.6: The global overview of the front-end.

This script defines one function. This function changes the location of the url to the *'/home'* end point. This is the end point where the server sends the data for today to the front-end. In the header this function is just defined and not actually used yet. The function is invoked as soon as the body of the HTML file is finished loading. This done using the following line of code:

```
1  <body onload="load()">
```

Within the body of the template, the areas for the menu and visualization are determined. Inside the area of the menu several buttons are added. Lastly the nine blocks that form the loading animation are placed in the template. Once the template has created all the elements, the page is redirected to the page that will actually show the visualization.

Visualization template

After the first load template has been loaded, the server is called and provides the user with a new template and data for the visualization. This template is more elaborate then the previous template, since this template has to call every function for the visualization as well. It calls several scripts and several style sheets to function and look the way it does.

In the header the template calls for four style sheet. The first three are the same style sheets as during the first load. The fourth style sheet is another freely available style sheet. This sheet is used to display a timepicker. The timepicker is an option in the user menu where the user picks a time and the visualization jumps to that time of the day it is currently visualizing.

Next, the header loads the Javascript files that only contain functions:

```
1  <!-- Scripts used in the code -->
2  <script type="text/javascript" src="{{ url_for('static', filename = 'js/
       jquery.min.js') }}"></script>
3  <script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.9.2/jquery-ui.min.js">
       </script>
4  <script type="text/javascript" src="{{ url_for('static', filename = 'js/
       satellite.min.js') }}"></script>
5  <script type="text/javascript" src="{{ url_for('static', filename = 'js/
       three.min.js') }}"></script>
6  <script type="text/javascript" src="{{ url_for('static', filename = 'js/main.js')
       }}"></script>
```

From these five scripts, the only script that is written as part of this project is the last script, the *main.js*.The *main.js*-script is a script that contains all the functions needed to support the visualization. The first two scripts are *jquery*-scripts. *Jquery* is a library that acts like a framework. Referencing objects within the HTML file can be done by simply adding $ in front of an element, instead of the usual $document.getElementById()$ function that is build in into javascript to access HTML elements. The *satellites.min.js*-script is a script that is used to propagate TLE data and the *three.min.js*-script is used to create a canvas on which three-dimensional objects can be drawn and visualized in a simple manner. This last library eliminates the tedious writing of code to visualize simple shapes like sphere. Without the library this has to be done one vertex at the time, with the library this is one line of code.

Once the header is done loading, the body is loaded. The lowest layer of the body is the visualization canvas. This element spans the entire screen and lies under the menu. This is done to center the visualization in the center of the screen. This proved useful when trying to determine which satellite is currently underneath the cursor. There will be no offset in the center of the screen and the center of the visualization.

Next, the menu and the loading animation are added to the screen. The menu bar is shown in Figure 3.8. The loading animation will be shown until the loading of the data is ready. Once the data is ready, the loading animation is set to invisible.

Figure 3.7: The menu bar

There are more elements that are invisible. These elements are loaded at the start of the template, with their display style set to none. This makes the element available for the code, but invisible to the user. One of the invisible elements are the elements that contain the information that is send by the server. The elements contain the information so that if the set up is changed slightly the information is still stored locally. It is also

possible to do this using a cookie, but I have no experience with working with cookies. Hence, the option to add the data as invisible items was chosen. The variables that are stored are

- Time of simulation
- Sub-solar latitude
- Sub-solar longitude
- TLE data from the chosen day
- Satellite catalog

Other items that are invisible are the menu to select a specific date, the satellite information box, and the satellite name box. These are invisible because they will only appear when certain actions are performed. In the case of the menu item, the user has to click on the search data button to access the panel. The satellite information and name boxes will show once the visualization is loaded. At this point, the user will be able to see the name and information of the satellite that the user is hovering over their mouse.

The last thing the template does is run a few scripts in a specific order. As mentioned before, the order of the scripts matter, because some scripts use functions from earlier loaded scripts. Some of these scripts are libraries that are used to make the visualization function better. The scripts that are loaded are displayed in the table below.

Order	Script name	Brief description
1	*global_variables*	Contains all the variables that can be used across the visualization.
2	*Detector*	Checks compatibility of the browser
3	*OrbitControls*	Allows users to move the visualization around.
4	*earth_visualization*	Renders a visualization of Earth
5	*satellite_object*	Create satellite object
6	*data_handler*	Reads data and creates satellite objects
7	*lagrange_interpolation*	Lagrange interpolation class
8	*timepicker*	Combined with a CSS creates a time picker
9	*visualization*	Main visualization script that runs every frame
10	*user_input*	Creates the user option in the menu

Table 3.1: Overview of scripts in order of being loaded.

The first script that is loaded is the script containing the global variables. These variables that are initiated in this file will also be available for the next functions. Doing this allows for variables to be accessed and manipulated from outside the visualization render loop.

The second and third script are scripts that have been written by third parties. The *Detector.js*-library is a library which contains a single function. This library checks if the browser is capable of running the *three.js*-library. This script is instantly run and produces an error message that will be displayed if the browser is not capable of running the tool. The third script is the *OrbitControls.js*-library. This library makes it able for the user to manipulate the view in the three-dimensional visualization. It translate the 2D motion of the mouse cursor, when being dragged in the browser window by the user, into a 3D orientation of the visualization of the globe, while keeping North towards the top of the screen.

The following four scripts are scripts that will be called from the visualization script. In order to keep order in the code, these scripts were each created with a different purpose. The first of four scripts is used to make a three-dimensional rendering of Earth. It create an object which will be added to the canvas. The second script is used to make a satellite object. Each satellite object contains the orbital parameters and propagates those to obtain a buffer with points of one orbit. The third of four scripts is a script that reads in the data and then calls the satellite object script to create a satellite object. The last of these scripts is a script that creates a class of the Lagrange basis polynomials, described in subsection 2.3.2 to determine the motion of objects between buffer points. All these scripts were specifically developed for this tool and will be discussed later in the report in more detail.

After these four scripts, a script called *timepicker.js* is loaded. This script is used in combination with the *timepicked* style sheet to allow the user to pick a time to jump to in the visualization.

The last two scripts are the visualization and user input scripts. The visualization script uses the power of the *three.js*-library to visualize the data. Then using the *lagrange*-script the visualization includes motion. The user input script is used to create the options the users have within the menu and allows for changing the global variables that are used to display certain views of the visualization. The developed scripts will all be explained in more detail in subsection 3.3.3

Error and About templates

The last two templates that are used by the front-end are very simple templates. Both templates create a small menu bar, with a limited amount of option, and show no visualization. These templates are similarly build as the first load template, but vary just slightly. Instead of loading data these templates show a message. The about template shows general information about the tool and the error template shows that an error has occurred and that the user should try to load the data again. These templates do have the same style as the other two templates; making the tool look uniform throughout.

3.3.2. Styling

To make the tool look uniform throughout a style for the website is centralized in a CSS file. The CSS file focuses on the parts of the tool that always look the same, like the menu.

In the tool one of the classes that is used is the *.message*-class. This class provided a styling for every message that pops up in the tool. This could be the name of a hovered object, or the information on the side of the screen, but also an error message or loading message is defined by this class. This class defines the message by giving the messages a solid green border and a graphite color background and white lettering.

The user menu is style with the same background color as the message box. Instead of using the green from the border of the message, the links in the menu turn red when they are hovered by the cursor.

This is also the case for the drop-down menus. Currently there are two drop-down. One for the user inputs and another one for the selection of certain groups. An example of the drop-down is shown in Figure 3.9.

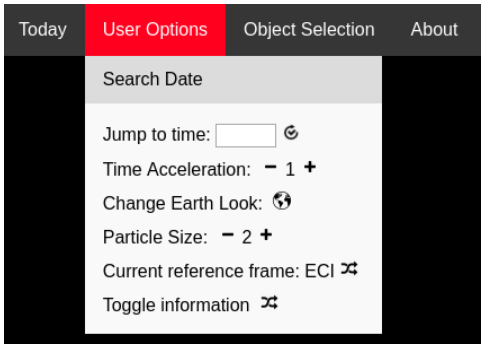


Figure 3.8: The menu bar with one of the drop-down menus shown.

In this image the user inputs are hovered over as well as the search date option is hovered by the mouse. The drop-down menu is created by adding HTML elements to an element that is classified as *drop-down*. The elements have the same padding as the links in the menu bar. Furthermore, links that are hovered within the drop-down menu are shown with a light gray background instead of the standard red.

The items within the drop-down menu are classified as *.user_inputs*. The user input is inherited by the drop-down menu as much as possible. Some elements however did not have any style yet, because they were not standard in the drop-down menu. The images that form buttons for example. The images are all sized to be 16 by 16 pixels. This makes the images fall in line with the text. The images also have a transparent background, making the background color of the drop-down menu shine through. This avoids having slightly different colors between the image background and the background of the menu. Furthermore, the buttons, which contain the images, do not show any lines, not even when clicked. This looks nicer and avoids weird squares around the images.In the other drop-down menu, there are several check boxes that are used to determine which groups should be shown by the visualization.

The last item to be styled in the menu bar is the search bar and its results. An example of this is shown in Figure 3.10

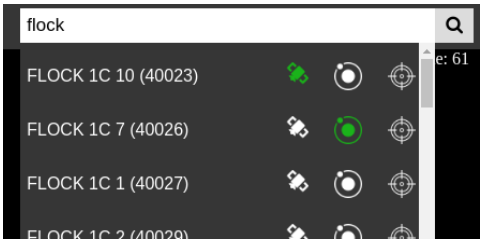


Figure 3.9: An search example with a few results.

This example illustrates the style of the actual search bar and icon, as well as potential results of the search. The results of the search are all elements with the same class, namely *.search_result*. These search results are build up of the name of the satellites and three buttons.

The first button has the shape of a satellite shows the satellite in the visualization and will delete any object that is not selected as well. This means that the visualization of other items disappear. The second button adds the orbit of the object and the last button focuses the camera on that object. When the satellite or orbit is shown the image of these buttons change colors to the color green. This is used for the first two satellites, but not the last, because the camera focuses the satellite once and then stays in that position.

If the search delivers a few items, the scroll bar will disappear and show all items at once. If there are around 12 or more items, the scroll bar is shown.

The last item that is styled by the style sheet is the slider bar on the bottom of the screen, which is shown in Figure 3.11. This item is not coupled to the menu bar, but is an individual component. The sliders allows the user to slide time in the simulation. The slider is very responsive for a few items, but as the number of items increases, loses the responsiveness of the visualization. This is causes by build in buffers in the visualization, which will be discussed in the next section. The slider is along the entire bottom line of the screen, laying atop of the visualization.

Figure 3.10: The slider

3.3.3. Visualization

Now that the tool is completely set up, it is time to explain the visualization and which elements are shown. As said in the beginning of this chapter, the visualization javascript file is the second to last file that is loaded by the visualization template. The last one is the user input, which needs the visualization to be ready. The *visualization.js* file starts out by obtaining the HTML element that will contain the visualization. This element is called *webgl*, which references the underlying web components used by the *three.js*-library.

Then the function from the *detector.js*-library is used. The function checks if the element can support the use of webGL properly. If the check fails, the browser does not support the visuals for the visualization. The function will be stopped completely and a message is shown to the user. The message tells the user that the current setup is insufficient to run the visualization.

After the check, some local variables are introduced. These variables are the height and width of the screen, the camera angle and max zoom, a raycaster object, and a vector that indicates the position of the mouse. The raycaster is used to determine if and which satellites are hovered over by the mouse.

There are three variables set for time. The first one, *GMT_time*, is a global variable, which is created in the *global_variables.js*-script. This variable takes the time of simulation value from the HTML element that has this information stored. This time was sent by the back-end and stored. The second time variable, *start_time*, records the start time of the simulation. This start time is the local time of the computer. The last time variable, *solar_time*, will have the same value as the *GMT_time* at the start of the simulation. But unlike the *GMT_time*, this variable will be constant. This is because this is the time from which the solar position is defined. The solar coordinates at this time are the coordinates that were sent by the back-end. Keeping the solar position up to date means that a delta needs to be calculated from this point.

The next step is to create the canvas on which the visualization will be drawn. This canvas, called *scene*, contains every element for the visualization and will be displayed in the HTML element designated for the visualization. With a scene, a camera has to be added, since this makes the user see the scene. The camera is added and adjusted to take the camera variables of angle and zoom into account. The up direction is added

to the camera and the camera is placed at a position that falls within the minimum and maximum zoom. Both the *scene* and *camera* are global variables initiated in the *global_variables.js*.

The next element is the *renderer*. This element is the element that converts the *scene* and all its objects to webGL components. The rendered is then added to the HTML element, making it visible to the user.

The *scene* is a *three.js* object that is usually completely dark. To lighten the *scene* an ambient light is added to the *scene*. The ambient light is created as a *three.js* object with a specific color and then the element is added. This is demonstrated in the next snippet.

```
1 // add ambient light to the scene
2 ambient = new THREE.AmbientLight(0x333333);
3 ambient.name = 'ambient light'
4 scene.add(ambient);
```

All elements that are added to the *scene* are added using the build-in *add()* function. Nested elements can also be created in this way. A nested element would be an element that is build up from other elements and then added to another object, like *scene*.

After the ambient light, a directional light is added to the scene. The directional light is the beam of light that comes from the sun. This means that in order to get the initial direction of the light correct, we use the solar coordinates that were saved after they were sent by the back-end. The coordinates are first converted to radians, then a vector is determined using these angles and set at a distance that equals the maximum zoom of the camera. By doing this, the user will never see the light source. As the visualization progresses through time, the light source will also move.

Earth

The next element to be created is the Earth. To be exact, two versions of the Earth. One version is a simple, fast loading version of the Earth, the other version is more cinematic and takes longer to load. The Earths are created by two functions within that were defined in the *earth_visualization.js*-script.

The script starts with declaring two variables that are used by both representations of the Earth. The first one is the radius of the Earth in Mm (Megameters). This unit was chosen, because the *three.js*-library has problems with large numbers that come from using kilometers. All other values, are scaled accordingly as well. This variable, *radius*, is defined to be 6.371. The second variable, *segments*, is defined to be 64. This variable determines from how many segments are used to create *three.js*-objects. 64 was chosen, since it looks like a sphere and not a polygon.

The first function, *createSimpleEarth.js*, creates an Earth that is fast loading, without much aesthetics. An example of this Earth is shown in Figure 3.12. The function that creates this Earth uses several functions from the *three.js*-library. First the function declares a variable, *earth_total*, that is defined to be of type THREE.OBJECT3D(). This ensures that the element can add other elements from the *three.js*-library.

The only element created for the simple Earth is a mesh. This mesh has the geometry of a sphere and have the texture of a low resolution Earth. The texture is called the *MeshBasicMaterial* and imports the image, which will then appear on the surface of the sphere. The code to create the mesh is shown below.

```
1 // create the Mesh for Earth
2 var earth = new THREE.Mesh(
3                 new THREE.SphereGeometry(radius, segments, segments),
4                 new THREE.MeshBasicMaterial({
5                     map:    THREE.ImageUtils.loadTexture('../static/img/
                          earth_simple.jpg')
6                 })
7     );
```

Next the mesh is added to *earth_total*. The position of *earth_total* is set to be at the center of the visualization, the name is set to *earth_simple*, the type is set to *simple*, and the Earth is rotated along the x-axis, to make the Earth orientated correctly. Without this rotation, the poles of the Earth would be along the equator.

Figure 3.11: Fast loading Earth

Figure 3.12: Cinematic Earth

The second version of the Earth is more cinematic and shows an impression of the Earth that looks better than the simple Earth. This rendering uses higher quality images and has some extra layers added for the enhanced experience. Similarly to the simple Earth, this rendering uses a mesh to visualize the Earth. The mesh has the same geometry, but instead of using *MeshBasicMaterial* to create the image, the function uses a more elaborate function in the *three.js*-library, namely the *MeshPhongMaterial*. This function creates a material of several layers, but more importantly it interacts with the light source in the *scene*. This means that the visualization shows a lit and dark side of the Earth.

The first layer of this material is the actual map that is used. The image used is a high quality, 4k-resolution image. On top of this layer, a different image is used to create an emissive map. The emissive map is an image that shows where the cities on Earth are and light up on the dark side of the Earth. The map is transparent, meaning that the underlying map shines through, unless the cities are shown. The emissive map is then set to a specific color to show clearly on the dark side, but seems to disappear on the light side. The third layer that is added is a bump map. A bump map creates texture to the Earth. The third layer reshapes the sphere slightly to show some roughness on the surface like mountains. These bumps also interact with the light, casting shadows on parts that can't be reached by the light source in that orientation.

The last layer that is added in this material is a specular map, or reflectivity map. The map is used to make the water reflective when light falls on water areas, like the ocean, but also lakes and rivers. The color and strength of the reflection can be tweaked to look more realistic, but the current version looks like it a good way to do visualize the Earth. This reflectivity also shows rivers more clearly and adds another visual effect.

The code to create this mesh is shown below.

```
1  // create the Mesh for Earth
2  var earth = new THREE.Mesh(
3                  new THREE.SphereGeometry(radius, segments, segments),
4                  new THREE.MeshPhongMaterial({
5                      map:          THREE.ImageUtils.loadTexture('../static/img/
                           earth_4k.jpg'),
6                      emissiveMap: THREE.ImageUtils.loadTexture('../static/img/
                           earth_night_transp.png'),
7                      emissive:     new THREE.Color(0x444444),
8                      bumpMap:      THREE.ImageUtils.loadTexture('../static/img/
                           elev_bump_4k.jpg'),
9                      bumpScale:    0.05,
10                     specularMap: THREE.ImageUtils.loadTexture('../static/img/
                           water_4k.png'),
11                     specular:     new THREE.Color(0x444444)
12                 })
13             )
```

This mesh is added to *earth_total*. The only layer that appears in Figure 3.13 that has not been described is the cloud layer. This is because the cloud layer is its own mesh. The mesh uses a slightly bigger radius than the Earth. The radius is increased by 150 kilometer, or 0.15 in the code, since it is scaled down by a factor of

1000. The reason the clouds are at 150 kilometers rather than closer, is to increase the effect it gives to the visualization. The layer is created using the *MeshPhongMaterial*-function again. This time, the material only creates a map, but sets the background to transparent. Hence, the Earth is visible under the clouds. This mesh is also added to *earth_total*.

The last step is to rotate the object once again. This is done to align the images and the object with the coordinate system that is used throughout the visualization.

The first time the visualization is loaded, the simple Earth is added to the *scene*, since this Earth loads the fastest. A user can change this in the menu. Now that the *scene* has its first object, the controls can be added. The controls allow the user to rotate and orientate the camera from different angles. To do this, the *OrbitControls.js*-library is used. This library allows the user to rotate the camera and view the objects from different angles, but prevents the user from flipping the Earth upside down. It is limited to a 180 deg rotation in the latitudinal direction, whereas the longitudinal direction knows no limit for rotation angel. The controls are created and limitation of the controls are included before adding them to the *scene*. The limitations are used to prevent the user to zoom in or out too far. When the user zooms in to far, the visualization of the Earth breaks and the user deletes the Earth until he zooms out again. Zooming out too far results in the visualization turning into a single dot and eventually disappearing as well. Another setting that is adjusted for the controls is how fast the controls respond to mouse movements. Originally the controls move very quickly, so this is slowed down to a more pleasant handling. The last thing that is changed in the controls is the ability to translate the camera. Translating would mean changing the point of view of the camera. When rotating the camera after a translation, the Earth looks does not rotate around its own axis but around the center view of the camera, resulting in a strange looking visualization.

The last object that is added before moving to the satellites and objects that are orbiting the Earth is the sub-solar point. The sub-solar point is depicted by a little sun icon that floats just above the surface of the Earth. The object is placed at the coordinates of the sub-solar point. And then added to the $scene$.

Objects and Orbits

In order to add the satellites and other tracked objects to the $scene$ the *addSatellites()*-function is called. This function can be found in *main.js* and starts out by removing any elements that might have been there from a previous search. The search engine can change which satellites are shown, but it will invoke the same function as when the visualization is started. This is why the shown satellites and orbits will be removed before adding new satellites.

Once the satellites and displayed orbits are removed, the function calls the *readData()*-function. This function is located within the *data_handler.js*-file and requires a time as an input. The time that is passed through to the function is the GMT_time. The *readData()*-function is the only function ins the *data_handler.js*-script. This function starts by extracting the TLE data from the HTML element that contains it, the element has the id 'data'. It also extracts the satellite catalog from the HTML element 'satcat'. The TLE data is stored in a local variable called raw_data en the satellite catalog is stored in the local variable $satcat\_temp$.

The second step in the *readData()*-function is to replace every single quotation mark by a double. This makes the data able to be parsed as a JSON format. The satellite catalog is parsed to see if the satellite catalog is indeed valid and the data is checked on containing data. There is the possibility that a data does not have any data, likely in very early days of the space age. When there is no data, the function returns that there is no data. When there is data, the function continues with the creation of a geometry that holds every object and satellite. In order to do this, a material is used: *PointsMaterial*. This material has the ability to draw vertices and show these vertices as a points cloud. This material is used because it reduces the amount of computing power needed to visualize large amount of points. Instead of having individual objects for every satellite, this material combines them. The material uses the colors that are set when a new item is added with a color property. Another feature from this material is that the size of the vertices depend on the distance to the camera. This means that objects close to the camera will appear bigger than the ones far away. This has the added advantage of clarity in the visualization as well as a feeling of depth in the view. The creation of this material is shown below.

```
1 var particle_system_material = new THREE.PointsMaterial({
2     vertexColors : true,
3     transparent: true,
4     size: particle_size/10,
5     sizeAttenuation: true,
6 });
```

The size of the particle, denoted as $particle\_size$, is a global variable that can be changed by the user.

The next part of the data reading process is computationally heavy, meaning that it takes a while. The user will experience this in the beginning when the data is loading. At this point the user does not see any visualizations yet, just the loading animation with a message that the data is being prepared. This start by parsing the raw_data as a JSON into the variable $data$. Parsing a JSON results in a dictionary that has keys. The keys from $data$ are all satellites that were available. From the back-end the data was organized by satellite number, and thus the data in the front-end is organized this way.

To create a single satellite object, the data passes through a *for*-loop that passes through every key, and thus every satellite, in the $data$. During the loop, the data that belongs to the key is extracted and the first and second line of the TLE are stored in local variables. These variables are redefined every iteration of the loop to prevent access use of memory. The TLE lines and the GMT_time are passed as variables into the *createSatelliteOrbit()*-function. This function is defined in the *satellite_object.js*-script. This script deals with everything that concerns the satellites and objects and their orbits.

The *createSatelliteOrbit()*-function starts by determining the buffer of the orbits. The buffer of the orbit are the amount of points that are calculated at once to determine the line segments that define the orbit and the motion of the satellites later in the code. Currently, the buffer is determined to be one orbit of 100 points. There the buffer is a global variable, but there is no way for a user to change this yet. The code is ready for this implementation, but I focused on a fast and reliable tool and determined that this was not a high priority feature to implement yet. After this declaration of the buffer, the TLE lines and GMT_time are passed in a function from the *satellite.js*-script. This third party script takes the TLE lines and time to determine the orbital parameters. This information is stored in a satellite record: $satrec$. An object is created that holds the data and is named after the satellite number that is found in the TLE lines.

Now all the information to determine all the points in the buffer are calculated. This means that the information is passed into another function: *satelliteBuffer()*. This function start by calculating the period of the satellite's orbit from the mean motion. Then the time difference between two points in the buffer is calculated using the period of the satellite. Next an empty array is created that will hold the points that are calculated for the buffer. One of the features of the tool is to be able to go back in time as well as forward in time. To do this, the buffer will extend half an orbit back and forward in time. This means that there are 50 points calculated before the GMT_time and 50 points after. To do this, there is another *for*-loop implemented that goes through each of these points in the buffer and calculates the position of the satellite at this point in time. This is done by using the *.propagate()*-function from the *satellite.js*-library. The function requires the $satrec$ and the time to propagate the position to. This function returns a vector that contains the position and velocity of the satellite. These values are calculated in the Earth Centered Inertial reference frame and in km or km/s. This means that, depending on the setting of the user, the reference frame has to be changed. If the chosen reference frame is ECI, then the only conversion is the conversion from km to Mm, since those are the chosen units for the visualization. If a conversion of reference frame is required, this step has to be added. In section subsection 2.3.2, a detailed theory on how the propagate and conversion function work are given. The conversion starts by calculating the Greenwich Mean Sidereal Time (GMST). Then the position and the GMST are passed in the conversion function: *eciToEcf()*. In the snippet below, the conversion is shown.

```
1  try{
2      if(reference_frame == 'ECEF'){
3          var gmst = satellite.gstimeFromDate(t)
4          var p = satellite.eciToEcf(po['position'], gmst);
5      }else{
6          var p = po['position'];
7      }
8  }catch(ex){
9      return [0, 0, 0];
10 }
11
12 try {
13     points[i] = new THREE.Vector3(p.x/1000, p.y/1000, p.z/1000);
14 } catch (ex) {
15     points[i] = new THREE.Vector3(0, 0, 0);
16 }
```

In this snippet several *try-catch* construction are added for security. Sometimes the algorithms from the

satellite.js-library fail. When one of them fails, the failure is caught by this construction without interrupting the visualization. This makes the visualization continue even though the algorithm did not complete its task properly. The points, along with the original time and positions are returned to the *createSatelliteOrbit()*-function. The buffer for this satellite is created and returned.

After the buffer is created, the *createSatelliteOrbit()*-function continues. The functions checks if there are any points in the buffer to make sure that the object can actually be visualized. When there are no points, the function returns zero. This is caught later, preventing the satellite object from being created. If there are points available, the *createSatelliteOrbit()*-function creates a new *three.js*-object that contains a line that connects the points, making the orbit of the satellite. This is done using the *three.CatmullRomCurve3()*-function. The orbit is added to the satellite object. This makes it available for the visualization if it is called upon.

The next item in a satellite object are the Lagrange polynomials equations for the buffer points. Using the equations from subsection 2.3.2, the *lagrange_interpolation.js*-script is created.The script creates an object with several build-in functions. This construction means that objects of this type can call the functions by just adding the function after the variable. For example, $polynomial$ is a Lagrange object, then $polynomial.position()$ is a function that can be called. The object contains two functions. The first function is a function that calculates the barycentric weights for the polynomial and the second function calculates the position of an object at a given time.

To create a Lagrange object, the function requires a list of times and positions as an input. The length of these lists should be greater or equal to two, since a minimum of two points are needed to determine the polynomial. Furthermore, the lists should have the same length, since every position has to be linked to a specific time. When the lists have different length, the function does not create an object, but returns an error. The function then continues and saves the time and position lists to an attribute of the object. Lastly the function adds an attribute of weights and then calls the *.updateWeights()*-function to fill this list of weights. The next snippet shows this.

```
1  var Lagrange = function(time, pos){
2    if (time.length < 2 || pos.length < 2 || time.length !=pos.length){
3      return 'error';
4      }
5
6    this.time = time;
7    this.pos = pos;
8    this.weights = [];
9    this.updateWeights();
10 }
```

The *.updateWeights()*-function calculates the weights that are used by the polynomial to determine the position. The function start by determining the length of the list of points (time and position). Then it loops through each point and calculates the product that is described in Equation 2.1. To calculate this product, another for-loop of all points is started. The terms for the product are the difference in time between the two points that are currently selected by both for-loops. The exception is that a point can not be compared to itself. The total product is needed to determine the weight for this one point. The weight is calculated using Equation 2.2. The *.updateWeights()*-function is shown below.

```
1  Lagrange.prototype.updateWeights = function(){
2      var weight;
3      var k = this.time.length
4
5    for (var j = 0; j < k; j++) {
6      weight = 1;
7      for (var i = 0; i < k; ++i) {
8        if (i != j) {
9          weight *= this.time[j] - this.time[i];
10       }
11     }
12     this.weights[j] = 1/weight;
13   }
14 }
```

The other function that Lagrange objects have access to is the *.position(x)*-function. This function takes a time and determines the position of the object at this time. To calculate the time, Equation 2.3 is used. The algorithm calculates the top and bottom sum independently and then divides them to obtain the correct

position. To calculate the sums, the algorithm uses a for-loop to cycle through all the times and weights that are in the object. For the sums, the factor is calculated first:

$$\sum_{j=0}^{k} \frac{w_j}{t - t_j}$$

This factor is added to the bottom sum. The factor has to be multiplied by the position for that specific item in the lists before being added to the top sum. The for-loop that calculates the sums is shown in the following snippet.

```
for (var j = 0; j < this.time.length; ++j) {
  if (x != this.time[j]) {
    factor = this.weights[j] / (x - this.time[j]);
    top_sum += factor * this.pos[j];
    bottom_sum += factor;
  } else {
    return this.pos[j];
  }
}
```

In this snippet, the function checks if the time entered for interpolation is the same as one of the points. If this is the case, the function will automatically return that point and stop calculation of the sums.

After Lagrange objects are created for the x, y, and z direction for the satellite object, the satellite object adds them as attributes. This concludes the creation of the satellite object, which means the object created in the *createSatelliteOrbit()*-function from the *satellite_object.js*-script is returned to the *readData()*-function from the *data_handler.js*-script. Here the function continues and creates a vertex for the points cloud that will show the objects. The positions are divided by 1000 to convert kilometers to megameters.

From the information that is stored in the satellite object, the color of the vertex can be determined. The type of object is stored in the object. This type can then be entered as a key a dictionary that is defined in the *global_variables*-script. The key in the dictionary is linked to the color of this specific item.

The vertex and color are pushed into the points cloud geometry. The orbit that is created is pushed to a list of orbits to be accessed to show the orbit in the visualization when an object is hovered. After every object in the data has been created and added to the geometry, the *three.js*-object is created that contains the geometry and material. The *readData()*-function ends with the return of the *three.js*-object, a list of all the catalog numbers in the object, and a list of orbit object. When something failed in creating the object, the function returns an error code, which leads to no visualization of objects.

After this function completes, the object containing the initial position of the all the objects is added to *scene*. Before moving to the motion part of the visualization a few empty objects are added to the scene as well. These are the highlighted orbit that is hovered, the searched satellite orbits, and the searched satellite positions.

Lastly, the HTML elements that show the user extra information are added as variables as well.

Hovering Satellites

The user is able to hover satellites. This hovering will provide the user with the orbit of the hovered object, and extra information from the satellite object. To be able to select the correct item when the user moves the mouse to an object, mouse tracking event is added to the visualization. This is shown in the next snippet.

```
// mouse movement eventlistener.
window.addEventListener( 'mousemove', onMouseMove, false );

function onMouseMove( event ) {
    mouse.x = ( event.clientX / width ) * 2 - 1;
    mouse.y = - ( event.clientY / height ) * 2 + 1;
    rayCasterFunction();
};
```

The tracker, or listener, detects mouse movement and then obtains the new coordinates from the mouse. There is also a function called. This function is the *rayCasterFunction()*-function.

The *rayCasterFunction()*-function determines which satellite is the closest one to the mouse. The function utilizes the raycaster object located within the *three.js*-library. This raycaster points a line from the camera to

the coordinates of the mouse. Then from this line it determines which objects are located in close proximity of the line. This results in a list of objects. To determine which item should be selected, a for-loop is created that determines the index of the closest object to the ray.

This index is then used to set several values. First the highlighted display object is emptied and set to the orbit from the newly highlighted object. The name of the satellite is shown in a box that appears close to the mouse and the information from the satellite catalog is displayed in a box in the left upper corner.

3.3.4. Motion

After creating all the objects, the next step is to introduce motion into the visualization. In order to create this motion, the function *render()* is called at the end of the *visualization.js*-script. The *render()*-function runs without end, because every time the function end, it calls itself. This means that the function will run over and over again until either the application is closed or the user changes the endpoint. A counter variable is defined and will be increased each time the function is run. Every tenth time that the function is run, the frame rate is calculated. Using dividing 10 iterations by the time it took to run the 10 iterations, results in a frame rate. This frame rate is shown to the user.

Next the function determines what the time difference between frame rates should be. This is done by using

$$\Delta t = factor * (t_{now} - t_{previous})/1000 + offset; \tag{3.1}$$

In this equation, $factor$ is the factor at which the visualization is played. This factor is used to influence at what speed the visualization is played. The t_{now} and $t_{previous}$ are times in milliseconds and these are divided by 1000 to convert them to seconds. The $offset$ is a value that tells if there was a time jump. The $offset$ is defined in seconds. The $offset$ is reset once it is used, otherwise the time would continue to jump by the $offset$ that was selected.

Next the variable that determines the difference in time from the epoch and the simulation time for the solar position is updated by adding the calculated Δt. The simulation time and the $t_{previous}$ are also updated. The simulation time is then shown updated for the user as well.

Next the solar position and Earth orientation are updated in the visualization depending on which reference frame is selected. For the ECEF reference frame, the solar position is calculated using the *sunPosition()*-function located within the *main.js*-script. This function takes the solar delta and the latitude and longitude of the sub-solar point at the epoch that was send by the back-end. The function starts by defining the Earth's rotation speed to be $7.2921159 \cdot 10^{-5}\ rad/s$. Then it updates the latitude and longitude of the sub-solar point. The latitude is set to be constant in the visualization, which means that the angle of the Earth is not visualized. The error that is caused by not adjusting the latitude grows over time, but is not large for a visualization within a few days around the epoch. At longer intervals from the epoch, the accuracy of the objects goes down as well, since the TLE data is extrapolated. The longitude is calculated using

$$lon = solarLong - \omega_{Earth} \cdot solar\_delta$$

The latitude and longitude are then converted to x, y, and z coordinates in the ECEF reference frame. These coordinates are the return of the function. The function is shown in the snippet below.

```
1  // function to determine the x,y,z of subsolar point at t after the initial
       coordinates
2  function sunPosition(t, solarLong, solarLat){
3      omega_earth = 7.2921159e-5;// rad/s
4
5    // this does not take the Earth's angle to the Sun into account. Thus this is not
         accurate over long periods of time.
6      lat = solarLat;
7      lon = solarLong - omega_earth*t;
8
9      lightX=Math.cos(lon)*Math.cos(lat);
10   lightY=Math.sin(lon)*Math.cos(lat);
11     lightZ=Math.sin(lat);
12
13     return [lightX, lightY, lightZ];
14 };
```

Once these coordinates are returned the light is set to come from this direction and the images depicting the sub-solar point is moved to the correct position. The rotation of Earth in ECEF is constant at zero. This means that there is no rotation of the Earth, which is consistent with the ECEF reference frame.

In the ECI reference frame, the calculations are converted to be in the correct reference frame. For the sub-solar point, the coordinates that are returned by the *sunPosition()*-function are converted using the conversion function from the *satellite.js*-script. For this function the *GMST* time is needed. To obtain this *GMST* time the *.gstimeFromDate()*-function from the *satellite.js*-script is used. This function takes in a date and returns the *GMST* time that corresponds to this date.

The next section of the visualization determines the motion of objects. When creating the satellite objects, a list with the ids of the objects in the visualization was returned. To add motion to the objects, all objects in this list are handled one by one. A condition that checks if all satellites are used or just a selection is checked. If the object is in the selection or all satellites are used, the object will be updated. If not, the object will be positioned at the center of the visualization, in the center of Earth, out of view of the user. The reason to place the object at the center, rather than deleting, is that at the center the object is still available for later use. The information of the object is still stored, which means that there is no need for recalculating the buffer of this object.

When the object receives updates, the algorithm creates several variables that copy all information about the object temporarily. These variables can than be used to calculate the update of the object. Whenever an object needs to be updated these temporarily variables will be used. This makes the visualization slightly faster than calling the information when needed from the object that has to be updated. Temporarily storing the information in separate variables instead of reading from avoids unnecessary calls on the object. It also improved legibility of the code. This information is used to determine the new position of the object. Using the Lagrange class function for position, the new position for one of the directions (x, y, z) is calculated. The position function returns two values, namely the new position and a boolean that tells if the buffer needs to be updated. In the *.position()*-function, it is checked whether or not the position is near the end of the buffer. To prevent extrapolation instead of interpolation, the buffer has to be updated. Using the new coordinates, the vertex within the satellite geometry is updated, moving the vertex associated with this satellite to the new position. Updating the position is shown in the next snippet.

```
try {
    [new_x,status_x] = Lx.position(time_delta);
    [new_y,status_y] = Ly.position(time_delta);
    [new_z,status_z] = Lz.position(time_delta);
}catch(ex){
    console.log(object, object_userData.Lx, Lx);
    console.log(ex);
}
var new_position = new THREE.Vector3(new_x,new_y,new_z);
satellites.geometry.vertices[id] = new_position;
```

The status of the coordinates determine if the buffer needs to be updated. If one of these statuses tell that the buffer needs to be updated, the function continues by calling the *satelliteUpdate()*-function from the *satellite_object.js*-script. This function requires the information from the satellite along with the simulation time as variables.

The first thing the function does, after setting some local variables with the satellite information, is finding the point in the buffer which is the closed to the simulation time. This point will be used as the new center for the buffer. When this point is within the first or last 5 points of the buffer, a new buffer will be calculated. Otherwise, the function returns zero, since no update has to happen. This is a second check if the orbit needs to be updated. When the orbit indeed needs updating, the function calls another function from the *satellite_object.js*-script, the *updateOrbitCalculations()*-function. This function works very similar to the *createSatellite()*-function. The difference is the simulation that is used as an input. For the *createSatellite()*-function the start time of the simulation is used, whereas in the *updateOrbitCalculations()*-function the current simulation time is used. Furthermore the difference between the function can be found in the manner the data is handled. In the *createSatellite()*-function the data is read from the TLE of that object, in the *updateOrbitCalculations()*-function the data is already stored in the satellite object, which means that there is no need for the conversion of TLE to orbital elements anymore. The satellite record can be used by the *satelliteBuffer()*-function to create a new buffer. This buffer is then used to create the new orbit object showing the new orbit when the item is hovered or selected.

Calculation of the buffer is computational heavy for the computer, this means that when a lot of objects are updated at the same visualization time, the visualization starts to slow down until all the buffers are calculated. This happens when the time acceleration is set to a high factor or when a time jump occurs. At either one, the visualization seems to lag a bit. This problem could be addressed in multiple ways. The first way is to

limit the time acceleration for larger number of objects. Another way to do this is, is to use multiple threads. This spreads the load of the calculations across the system.

The new buffer is then used to create new Lagrange objects that will be used to determine the position of the object. The new information is stored in the satellite object, overwriting the existing information. This updated object is then returned to the *render()*-function to be processed. In the *render()*-function, the new satellite object replaces the old object.

The function continues by checking if the orbit was either highlighted or selected. If either is the case, the orbit is updated in the $display\_orbit$ or $satellite\_orbit$. This is done to prevent the orbit to appear out of sync with the object that is moving in the visualization.

The last thing that happens in the *render()*-function is to update the vertices of the satellite geometry, update the controls, request the animation, and lastly call the *render()*-function again. The next snippet shows the last four lines of the *render()*-function depicting the last instructions.

```
satellites.geometry.verticesNeedUpdate = true;
controls.update();
requestAnimationFrame(render);
renderer.render(scene, camera);
```

3.3.5. User interaction

The last main part in the visualization are the tools that the user has to his or her disposal while using the tool. The user can manipulate time, change various visual aspects, change the reference frame, and search for specific satellites. The user functions are defined in the *user_input.js*-script. This script is able to create the options the user has access to as depicted in Figure 3.9.

User Options

The first option the user has is to search a specific date. Clicking this link shows a hidden menu that allows the user to select a date. When the user confirms this date, the user is redirected to a different end point of the tool, where the date is passed to the back-end. The back-end sends the data from this date, which is then visualized by the front-end.

The second option is for the user to jump to a specific time. When the user clicks the input field, a little screen pops down, making the user able to select and hour and minutes. The reason the date and the time are separate functions, rather than once function, is that selecting a date requires the server to retrieve data from that specific date. The time jumping function uses the same data that is already loaded into the visualization. Thus, separate functions were the more logical option.

Once the user has selected the desired time, the button next to the input is clicked. This triggers the *setTime()*-function. This function determines the offset in time that the user wants. To do this, the hours and minutes of the input are split. Then the hours and minutes are taken from the simulation time. Both times are then converted into seconds after midnight. The difference between the two is then calculated and set to be the global variable $time\_offset$. This offset can then be used in the visualization using Equation 3.1. The *setTime()*-function is shown below.

```
function setTime(time_input){
    var hours = time_input.value.split(':').slice(0);
    var minutes = time_input.value.split(':').slice(1);

    // get current simulation time
    var simulation_time = new Date(simulation)
    var sim_hours = simulation_time.getUTCHours();
    var sim_minutes = simulation_time.getMinutes();

    // calculate the time_offset
    time_offset = (parseInt(hours)*3600+parseInt(minutes)*60)-(sim_hours*3600+
        sim_minutes*60)
}
```

The slider at the bottom of the screen works using the same principle. The slider has a build-in function that gives it a value. This value is then used to determine the offset that has to be added to the simulation time. The slider gives the user the ability to finely tune the time that he uses. The slider is set to be adding or removing up to 20 minutes of the simulation time.

The other option to manipulate time that the user has, is the ability to speed up, stop, or reverse the motion. This is the third option in the menu that the user has to its disposal. The user can increase or decrease

the speed factor of the visualization by clicking the minus and plus buttons. Clicking the one of these buttons activates the *timeFactor()*-function. When the minus button is clicked, the variable that is passed into the *timeFactor()*-function, is -1, otherwise the variable is 1. The function determines the factor by cycling through a predefined list of factors. The list is predefined in the *global_variables.js*-script. Depending on the sign of the input variable, the factor is set to be the next or previous item in this list. The index of the list is changed to provide the value for the time acceleration. The factor is then set to the global variable *time_factor*, which can be used in Equation 3.1. In the *render()*-function this factor is used to calculate the time difference between two frames.

Besides the time manipulation, the user has the option to change the reference frame. The user has access to this function in the user option menu. The option is denoted as *Current reference frame:* . After this the current reference frame is shown to the user. The user can click the button that follows right after. This button activates the *referenceFrame()*-function. The function starts by determining what the current reference frame is. The current reference frame is stored in a global variable named $reference\_frame$. The function then continues by flipping the value of the reference frame. The text in the user menu is also updated.

Next the highlighted orbit is removed and the *addSatellite()*-function from the it main.js-script is called. This function removes the satellites and orbits from the scene and recalculates the buffers for each satellite. In the function, the $reference\_frame$ variable will be different than the last time the function was called. This means that the calculation of the buffer and the orbits will be converted to the desired reference frame. Below the *referenceFrame()*-function is shown.

```
1  function referenceFrame(){
2      if (reference_frame == "ECI"){
3          reference_frame = "ECEF";
4          document.getElementById('reference_frame_value').innerHTML = "Current
               reference frame: ECEF";
5          while (display_orbit.children.length > 0) {
6              display_orbit.remove(display_orbit.children[0]);
7          }
8          addSatellites();
9      }else{
10         reference_frame = "ECI";
11         document.getElementById('reference_frame_value').innerHTML = "Current
               reference frame: ECI";
12         while (display_orbit.children.length > 0) {
13             display_orbit.remove(display_orbit.children[0]);
14         }
15         addSatellites();
16     }
17 }
```

The reference frame change, as well as the initial calculations of the orbits take a considerable amount of time. While the calculations are performed the visualization is temporarily paused. This is because the current frame is delayed due to the enormous amount of computations that have to happen. The faster the user computer is, the better the performance is. On my own laptop, the change of reference frame takes around 10 seconds for the most current date. When less objects are in the visualization, the conversion is faster, since less recalculations occur.

The user is also capable of altering the look of the visualization. The user can change the look of the Earth, change the size of the objects, and toggle the information that is displayed. These option are also available in the user menu that is shown in Figure 3.9.

To change the Earth, the user activates the *changeEarth()*-function by pressing the button that is shaped like the Earth. The function removes the Earth from the *scene*. This does not destroy the *earth*-object, but rather removes it from the visualization. The *earth*-object has an attribute that tells what type it is, either it is *earth_simple* or *earth_complex*. The function reads the type and then reassigns the other visualization of the Earth to the *earth*-object. A visual representation of each Earth is shown in Figure 3.12 and Figure 3.13. After the *earth*-object is changed, it is added to the *scene*. By removing the object and adding it, the visualization is updated. Just changing the *earth*-object does not change the representation in the *scene*.

To change the size of the particles, the user can activate the *particleSize()*-function. The user does so by pressing either the minus or plus button that is associated with the particle size. Similarly to the factor buttons, these buttons pass either -1 or 1 to the *particleSize()*-function. This value is used to alter the global

variable *particle_size*. Increasing and decreasing the size of the particles is done by simply adding the value that is passed to the function. Whenever the particle size exceeds 10, the size is set to 10. Similarly when the particle size is smaller than 1 the size is set to 1. The function bounds the variable to be an integer between 1 and 10. This is done to prevent the user from making the particles disappear or making the particles so large that the visualization becomes useless. Then the size of the particles is set to the *particle_size* variable divided by 10.

The last thing the user can change in the visualization is the information shown. The user can toggle the information on or off. This can be useful for someone to take a screen shot or screen capture. The user activates the *toggleInformation()*-function. The function checks whether or not the HTML elements containing the information are shown. This is done by checking the *style.display* of the elements. When the *style.display* is set to *none*, the elements are invisible. When they are set to *block*, the elements are visible. The function changes the *style.display* from one to the other. Furthermore, the function disables the highlighted orbit, to minimize the clutter of the screen.

Group Selection

Another option the user has is to select predetermined groups. These groups are defined as the groups in the satellite catalog. The user can select these in another user menu. This menu is shown in Figure 3.14.

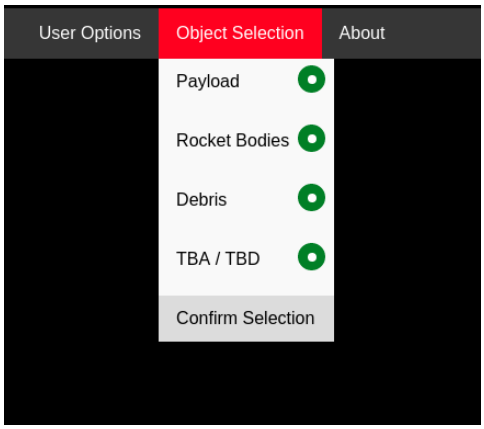


Figure 3.13: The group selection menu

In this menu the user can select which groups he would like to see. When the user toggles one of the groups, it calls a function: the *selectGroups()*-function. Each bullet points sends a variable to the function. The variable is the name of the group that is associated with that bullet point. The *selectGroups()* either removes the groups from the $groups$ list or add the group to the $groups$ list. The $groups$ list is defined in the *global_variables.js*-script and contains the groups of the satellite catalog. For future development, it would be nice to be able change the color of the groups in this menu.

After the user makes the selection final, the user has to confirm the selection by pressing the button on the bottom of the menu. This button triggers the *addSattelite()*-function. Since all satellites are part of one object, calling the *addSatellite()*-function makes sure that the groups that are selected appear in the visualization, whereas the not selected groups will disappear. This function is slow and requires the orbits to be recalculated, even though the buffers and orbits don't change. In the future, each group could potentially be their own object in the visualization. This would decrease the time it takes to change selections significantly. However, it would increase the complexity of the *render*-function.

Search Engine

The last option the user has to interact with the visualization is to search a specific object. The user can type their search term into to search engine, which will prompt the user with the search results. An example of a search result is shown in Figure 3.10. The search input is created in the *user_input.js*-script. When the text in the input bar is changed a function is triggered that tests the time between key strokes. When the time between key strokes is greater than 500 milliseconds, another function is triggered. This function is the *searchSatelliteResults()*-function.

The *searchSatelliteResults()*-function takes the user input after the time between key strokes exceeds 500 milliseconds. The function, located in the *main.js*-script, then starts by deleting the old search results. Once

the results are deleted, the length of the search term is determined. If this term is shorter than three letters, the function directly returns no results, since search terms that are shorter than three letters do not represent a good search.

When the search term is long enough, the actual list of results is build. The search method loops through each item in the satellite catalog and determines if the search term matches either the catalog number, the international designator, or the satellite name. When this is the case, the result has to go through one more check. The check is to see if the result is available to the user. That is, is the result in the TLE data that was send to the front-end? When this check is also passed, the result is added to the result list. This is done for every item in the satellite catalog.

After the list of results is determined, the length of the results is checked. If the length is zero, the function ends. If the function continues, the results are displayed using the format shown in Figure 3.10. Each search result is build up from a string of text and three buttons. The string of text represents the name of the satellite and the number that is used in the catalog. The three buttons are object show, orbit show, and object focus.

The first button is the object show button and looks like a satellite. When the user clicks this button, the *positionShow()*-function is activated. This function checks if the color of the button is green or white. When the button is green, the object is shown, so when the user clicks the button, the object should disappear. If the color of the button is white on the other hand, the object should appear and the button should be green colored. The function does this by adding or removing the object from the $specific\_ids$ list. This list is used in the *render()*-function to show specific objects. By removing or adding the object to the list, the object is shown in or removed from the visualization.

The second button is the orbit show button and looks like an orbit. When the user clicks this button, the *orbitShow()*-function is activated. Similarly to the object show button, the function checks the color of the button. When the color of the button is white, a new orbit object is made by cloning the orbit object that is associated with the shown result. Once the orbit is cloned, the color is adjusted to a color from a preset list. Each time the user types a new search, the next color will be selected. Once the new orbit is created, the orbit is added to the $search\_satellites$ object. This object is shown in the visualization, which means that the new orbit is shown as well. When the button of the green, the orbit is already shown. Clicking the button changes the color of the button to white and removes the orbit from the $search\_satellites$ object.

The last button the user is able to click is the button that is shaped like a crosshair. This button focuses the view temporarily on the object. This does not follow the object, it rather points the camera to where the object was at that specific time. To do this, the button activates the *cameraView()*-function. This function calculates the position of the object and then adjusts the position by multiplying the values of the x, y, and z position by 1.5. Then the camera is set to this new position. Another thing this function does is changing the color of this specific object to an orange-yellow color, to make the object more easily identifiable for the user. If the button is pressed again, the color is set back to the color of the group the item is part of.

Besides the buttons for specific satellites, the user has another button. This button is located next to the search bar and looks like a looking glass. This button allows the user to select all objects and orbits from the search results. To do this, the button activates the *searchSatellite()*-function. This function goes through each element in the search list and activates the first two buttons, the object and orbit show buttons. The function also removes any orbit that is highlighted, since this might be confusing to the user. The *searchSatellite()*-function is shown below.

```
1  function searchSatellite(){
2      for (var search in search_list){
3          search_num = search_list[search]
4          document.getElementById('button_' + search_num[1]).click();
5          document.getElementById('position_' + search_num[1]).click();
6          while (display_orbit.children.length >0) {
7              display_orbit.remove(display_orbit.children[0]);
8          }
9
10     }
11 };
```

4

Finished Product

This chapter takes a closer look into the software that has been outlined in chapter 3. The chapter looks at the software from the users point of view, the limitations of the software, and addresses several use cases for the software.

4.1. Overview of the tool

To start the tool, the user has to access the web page on which the tool is located. The current location of the tool is orbits.tudelft.nl. The code can be found following https://gitlab.com/SvenKardol/Orbit-Visualization-Tool. When the user makes the first call to the page, the user will be greeted by the first load page as described in subsection 3.3.1. The first load page is shown in Figure 4.1.

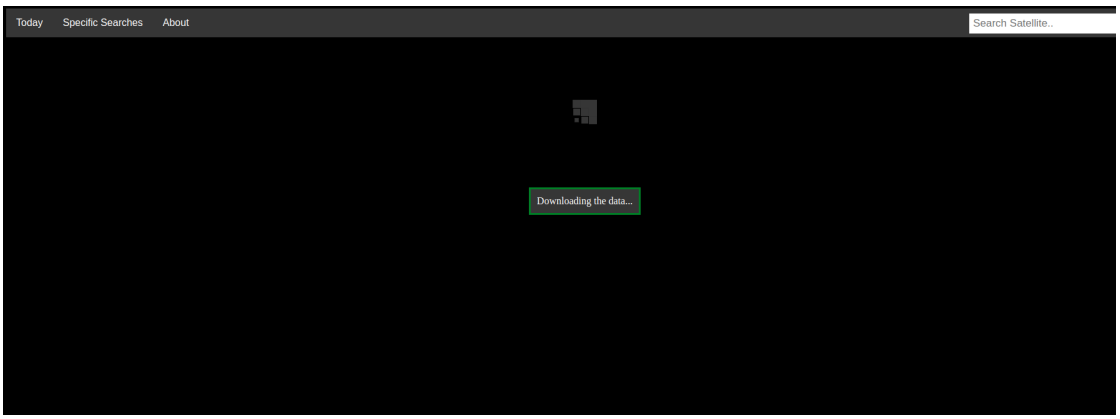


Figure 4.1: The page that greets the user upon first loading the tool.

In this first load page, the user is not able to use the menu or the search bar. Technically neither one is disabled, but when the page is done loading it redirects the user to a new page. This redirection causes the menu to be disabled other than the color change when the item is hovered. The user is redirected to the */home* end point.

The */home* end point contains a live visualization of all objects. Once the new page is done loading, which happens after the data is send to the front-end, the user sees a new message in the green box in the middle. It shows that the calculations for orbits are being performed. Once the calculations are done, the user sees the visualization as depicted in Figure 4.2.

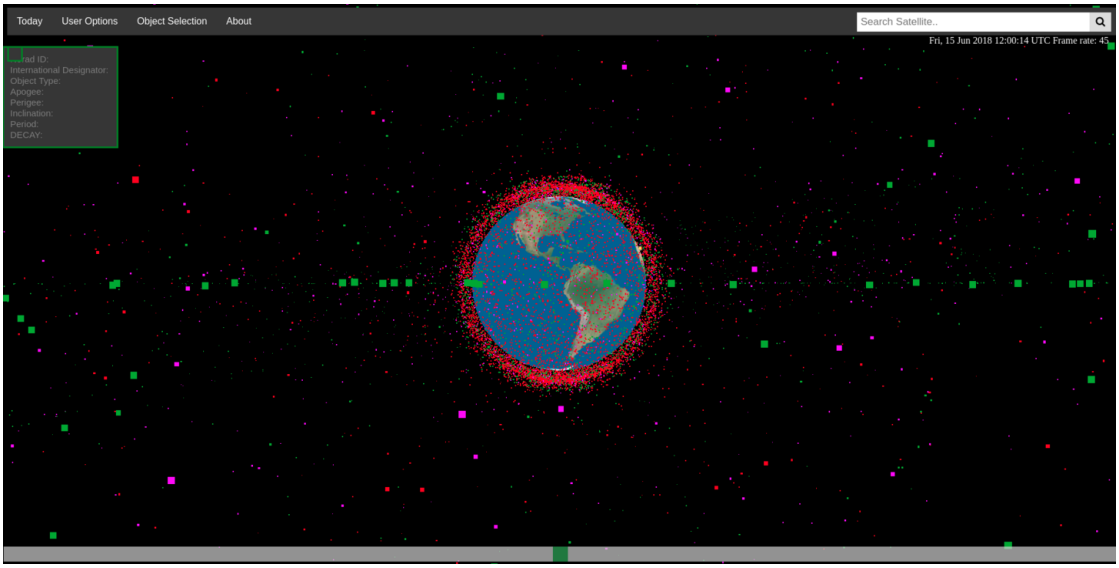


Figure 4.2: The page depicting the live data.

This page should start as a live visualization of the items that are currently in orbit. In the right top corner, the user can see the time of the simulation and a frame rate at which the visualization is operating. In the left top corner, the box that contains information about a hovered satellite is shown. At the start this box is empty, hence no information is shown in Figure 4.2.

The bar on the bottom of the screen is the slider that allows the user to precisely search around the current simulation time. The user is capable of sliding about 20 minutes behind or ahead of the simulation time. When the user likes to jump more time, the user should enter the desired time in the "Jump to time" input under the user options menu.

Another option the user has, is the search for specific date. When the user clicks the "Search Date" option in the user option menu, a message box with a date input is prompted. If the user click the input box, a calendar is shown to guide the user in picking a date. The user can also type the date, if that is more convenient. Figure 4.3 shows how the user sees the search date option to select a specific date.

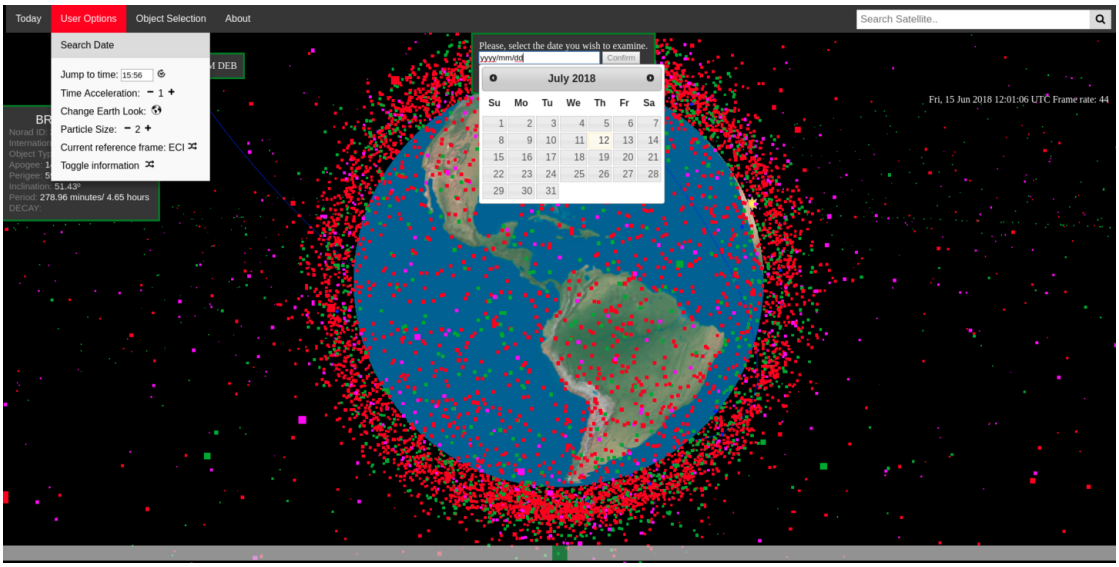


Figure 4.3: The view of the user when they want to search a specific date.

The other menu option, like time acceleration and change of reference frame, are conveniently located under the user option as well. The user can find the selection of certain object types under the "Object Selection" menu item. The user is presented the option to show different groups. These groups are payload, rocket

bodies, debris, and to be announced/determined object. In Figure 4.4, the payload objects are depicted and the menu that shows the options is shown.

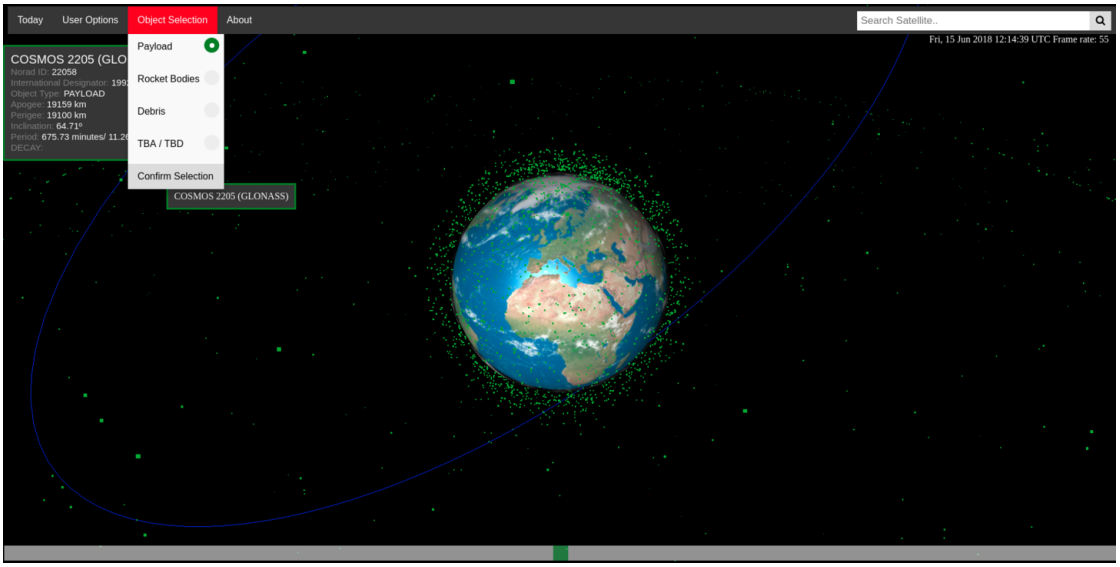


Figure 4.4: The view of the user when the payload objects selected and the menu that shows the options is activated.

The user is also capable of searching for specific satellites. When the user searches for "flock", a list of results is shown. Clicking the looking glass button results in selecting all objects and orbits from the search. This is shown in Figure 4.5.

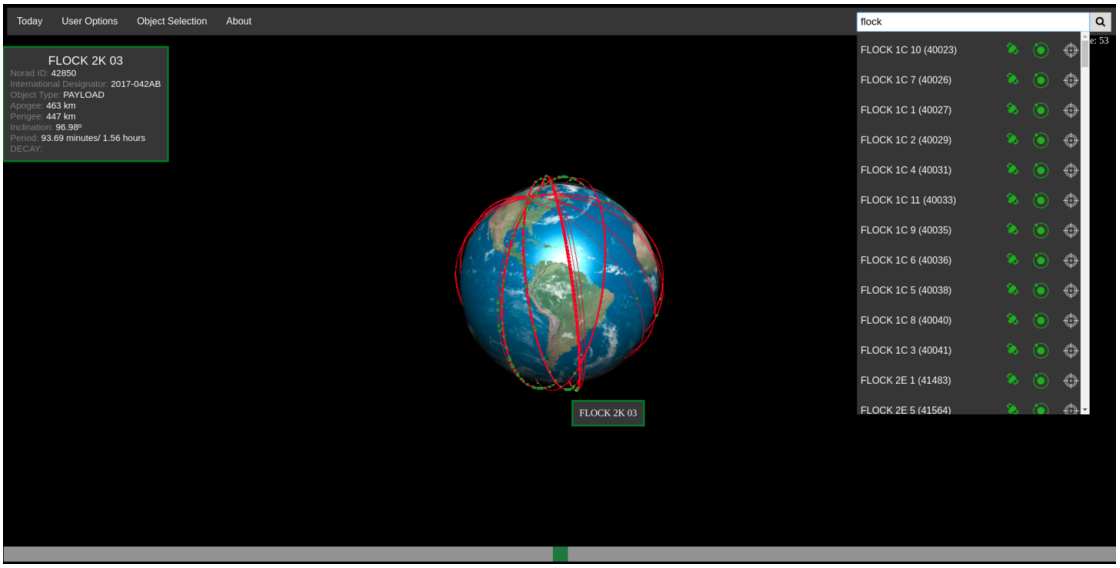


Figure 4.5: The view of the user when they search "flock" and select all objects and orbits.

4.2. Criteria and limitations of the tool

In the introduction, the conclusion of the literature study was mentioned. The conclusion of the literature study dictates the criteria for the tool. The criteria were:

1. The tool should be a web-based application.
2. The user should be able to easily manipulate time.
3. Comparison of multiple satellites and their orbits should be included.

4. The tool has to be fast and responsive.

5. The tool should be easy to use.

The criteria of the tool were used in designing the tool. This should mean that all the criteria are met.

Starting with the first criterium: The tool should be a web-based application. Was this criteria met? In the current version, the tool is set up to be a web-based application. However, the tool has yet to be released on the web for everyone to use. The reason why the tool is not available online yet, is that there was not enough time to figure out where and how to host the application.

The time manipulation criteria is linked to one of the biggest limitations of the developed tool. The user is capable of manipulating time easily, but some options result into the tool struggling to keep the performance up. When the user either jumps time or has a high time acceleration set, the tool requires the recalculation of multiple buffers at the same time. This leads to a drop in frame rate, because the calculations are heavy on the system. This is not a problem when only a few objects are shown, but when using the data of today the visualization struggles to keep up. This is mainly because the visualization has to account for the changing buffer of 18,000 objects very quickly. The more items and the higher the time accelerations, the more often the buffers have to be calculated. These calculations are done one after each other, which means they are not optimized. There are multiple options to optimize this in the future. One of the options is to reduce the number of buffer points, which results in faster calculations, but needs to be calculated more often. Another solution is to perform the calculations using multiple threads. This means that the calculations are done in the background, while the visualization continues. This seems to be the better option, but it requires a better understanding of the algorithms that make this happen.

The time manipulations do not stop at just the time acceleration or time jumping, the slider also manipulates time. The slider is capable of precisely determining the time within a 40 minute time window around the current simulation time. When the slider is used in a calm manner, the visualization can handle the interpolations without a problem. When the user plays with the slider and slides them from left to right multiple times, the interpolation algorithm struggles to keep up. When this happens the visualization can look like the objects are exploding outward. A comparison of regular visualization and exploding visualization is shown in Figure 4.6 and Figure 4.7. Some solution that could be taken into consideration is to put a damping effect on the slider. This would prevent the user to slide the slider faster than that the tool can keep up. Another option is to increase the buffer time, or decrease the maximum time that the slider can access. For example decrease the maximum time for the slider to 10 minutes instead of 20.

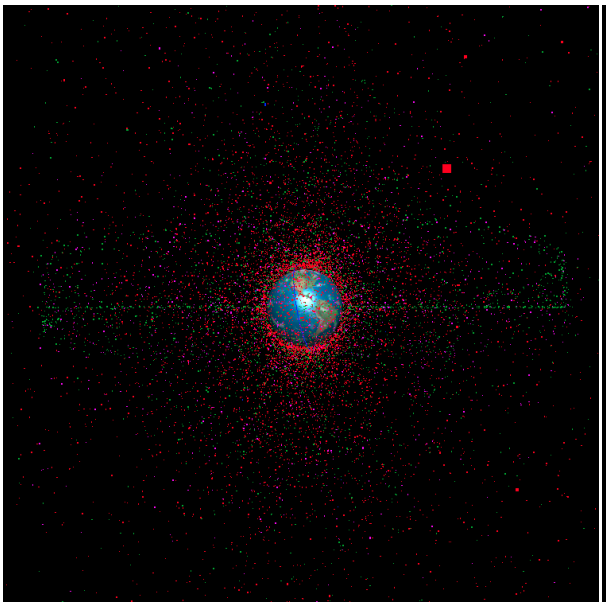


Figure 4.6: Exploded visualization due to time manipulation

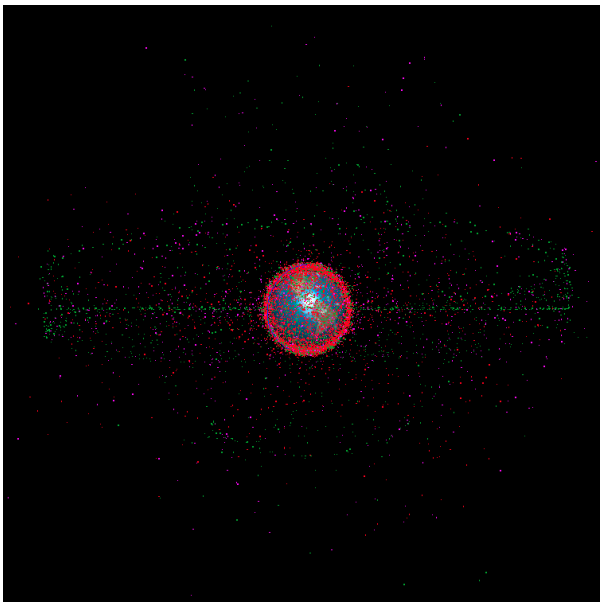


Figure 4.7: Regular visualization

The third criterium is the ability to compare the orbit of multiple satellites at the same time. It is possible to compare the orbits of two or more satellites by searching these satellites and show their orbit using the buttons in the search results. Furthermore, it is possible to select satellites that are part of the same group.

For example, searching for "falcon" objects allows the user to see all objects that contain "falcon" in their name. More about these type of visualizations in section 4.3.

The fourth criterium about how fast and responsive the visualization is go hand-in-hand with the second criteria of time manipulation. Whenever the user is using the tool without pushing the limitations of time manipulation as described earlier, the tool is fast and responsive. When the user does operate the tool at the edges of the limitations, the performance of the tool is linked to the capabilities of the users computer. The more demanding the tasks, the more computing power the user needs to keep the visualization running smoothly.

Besides the time manipulations, the user can access other options during the visualization. Some of these options are computationally heavy, because new buffers have to be calculated. One of these options is the change of reference frame. When changing the reference frame, the buffers have to be recalculated and the positions of all objects have to be redone. By selecting this option, the user will experience a pause of the visualization of around 15 seconds. This is about the same time it takes the visualization to calculate the orbits at the start. Similarly to the change of reference frame, selection objects from the same group takes a long time. This is because the objects are recalculated as well. In the future, the groups should be separated to decrease the waiting time when these groups are selected.

The last criterium is about the ease of use. To conclude whether or not the tool is easy to use, the user should be able to understand how the tool works. The user can easily figure out how to move the camera position, as it is just a click with the mouse button to rotate the visualization. This is nearly native compared to any other application that uses a similar style of visualization. In this regard, the user should experience the visualization easy to use. The other user options are clearly shown in the menu bar and are easily accessible. When the user is unsure what each button does in the option menu, the user can simply try the button and see the effect it has on the visualization.

Besides the design criteria, the tool has other limitations. At the moment the only data that is shown the user is downloaded and stored on the server side of the tool. However, a user might want to add their own data to the visualization. Currently, this is not possible. The user is limited to the information that is available in the TLE's from the *space-track.org*-website. Furthermore, the visualization is bound to the objects around Earth and can not be extended to a visualization that includes the moon, or other planetary bodies. For these visualizations other methods for determining the position of objects needs to be developed.

4.3. Use cases

Now that the tool is operational, we can look into the ways the tool can be used and beneficial. Several use cases were selected to demonstrate the versatility of the tool and the ways in which it is beneficial. The first use case is the visualization of entire constellations in a matter of seconds. This is demonstrated by using Planet's constellation of their dove satellites (FLOCK). Planet is also known as Planet Labs.

4.3.1. Planet Constellation Visualization

> *"Today Planet successfully launched 88 Dove satellites to orbit—the largest satellite constellation ever to reach orbit. This is not just a launch (or a world record, for that matter!); for our team this is a major milestone. With these satellites in orbit, Planet will reach its Mission 1: the ability to image all of Earth's landmass every day."*[10]

This was the news of the company planet after launching their 88 Dove satellite into an orbit on February 14, 2017. The company has the goal to image the entire Earth on a daily basis. These 88 satellites increases the size of their satellite constellation to 149. These satellites take high resolution images of the surface of the Earth mapping most of the surface. Using the developed tool, a user could see the constellation that Planet build over the years. Focusing on the Dove satellites, known as "FLOCK" satellites in the satellite catalog, the tool is able to visualize the constellation in a matter of seconds. This is shown in Figure 4.8.

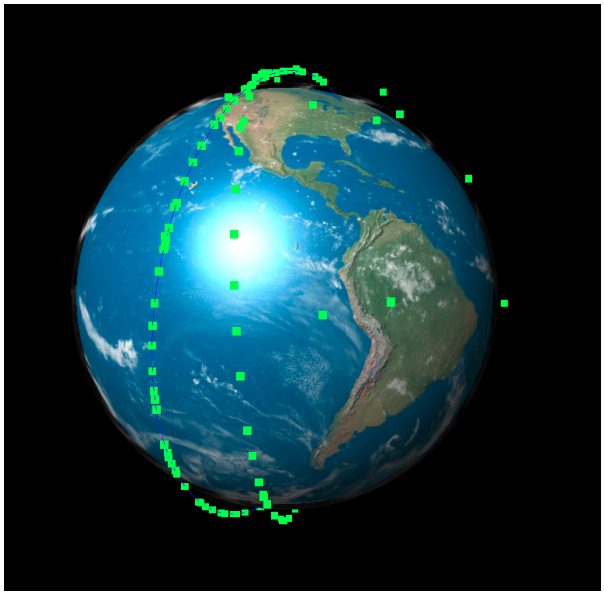


Figure 4.8: FLOCK satellites that were tracked on April 18th of 2018.

This image shows where the satellites were at 21:45 (CET) on April 18th of 2018. It can be seen that most of their satellites are located in one of two orbits. This is due to the fact that the company launches a lot of small satellites using the same rocket. Then letting the satellites deploy at certain times, the satellites spread out over the original orbit of the rocket. In the Earth Centered Inertial reference frame, the orbits of object launched with the same launch vehicle should have very similar paths. This can also be shown with the tool. This is shown in Figure 4.9.

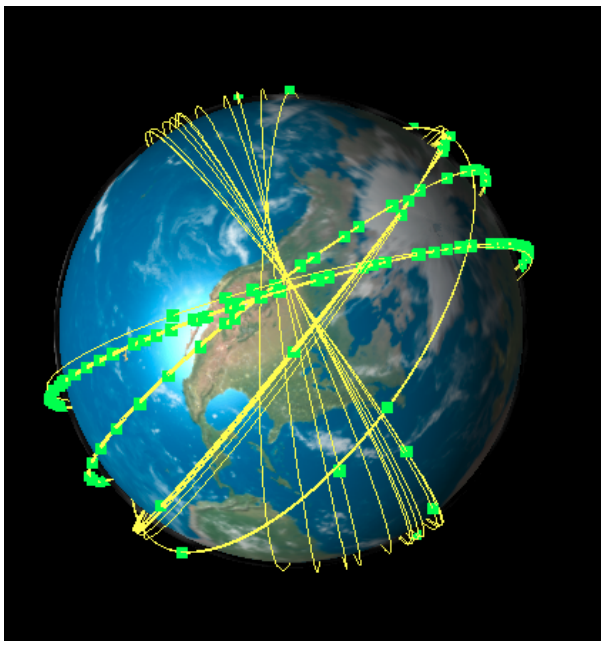


Figure 4.9: FLOCK satellite orbits that were tracked on April 18th of 2018.

From the image it can be concluded that the satellites indeed follow orbits that are very similar. The longer the satellites are in orbit, the more the orbit will deviate from the original launch orbit. This also seen in the image. In the image we see a third cluster of orbits from an earlier launch. These satellites were part of the FLOCK 1 deployment. Over time the orbits drift more apart.

Another way to look at the orbit is in the Earth Centered Earth Fixed coordinate system. When looking at the satellites in this coordinate system, we see their relative track to the surface of the Earth.

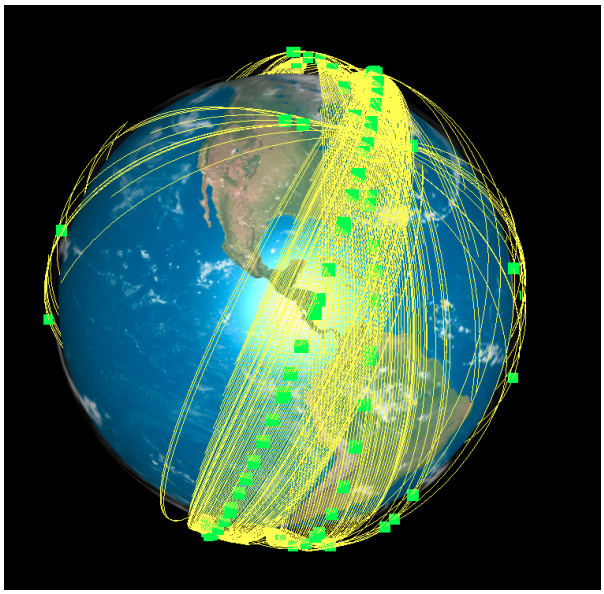


Figure 4.10: FLOCK satellite orbits that were tracked on April 18th of 2018 at 17:30 (GMT).

This reference frame can be used to show coverage of the constellation over one orbit. Using the time acceleration function, it can also show the coverage over time and predict when new data of a certain location could be expected. The orbits shown can be compared with ground tracks when the visualization is on a flat map. These orbits however show it in three dimensions. In Figure 4.10 the ECEF orbits are shown for the FLOCK constellation April 18th of 2018 at 17:30 (GMT). Then using the time acceleration function, we can fast-forward over two hours in advance to 19:45 and see how the orbits have progressed. This is shown in Figure 4.11.

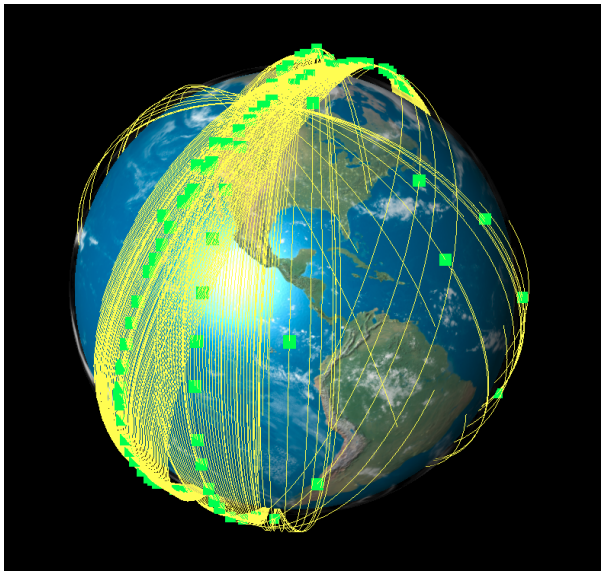


Figure 4.11: FLOCK satellite orbits that were tracked on April 18th of 2018 at 19:45 (GMT).

This will also indicate when a certain area will be in view of the constellation. From these 2 images it can be seen that the constellation moves towards the east in reference to the surface of the

Earth. The tool is capable of quickly showing the progression of the constellation over time.

This use case demonstrates the power of comparing multiple orbits at the same time. Furthermore it can demonstrate that the constellation of Planet is capable of capturing images of the entire Earth surface every 24 hours. This can be achieved by running the visualization at the maximum time acceleration and then determining the area where the constellation has captured images over a 24 hour time period. One of the things the tool is not capable at the moment is to keep the track the satellite has traveled over the time of the simulation. Updating the orbit track regularly results in less memory use of the tool, enhancing its performance and lightening the load on the user's computer.

4.3.2. Iridium 33 and Cosmos 2251 collision

On February 10th 2009, two satellites accidentally collided while in orbit. These satellites, the Iridium 33 and the Cosmos 2251, left a huge debris field in the wake of the collision. One of the first articles that appeared on the subject was "Satellite Collision Leaves Significant Debris Clouds"[5] published in the Orbital Debris Quarterly News vol 13(2) in April 2009. The article mentioned that about a month and a half after the collision 823 large debris objects were tracked and related to the collision, but that there were more to be cataloged. The article goes on to show a prediction of how the debris spreads out over time. This is shown in Figure 4.12. Six months after the collision would be August 10th, 2009. Using the tool, we can compare how this prediction held up with the tracked items at the time. The tool provided the image shown in Figure 4.13.

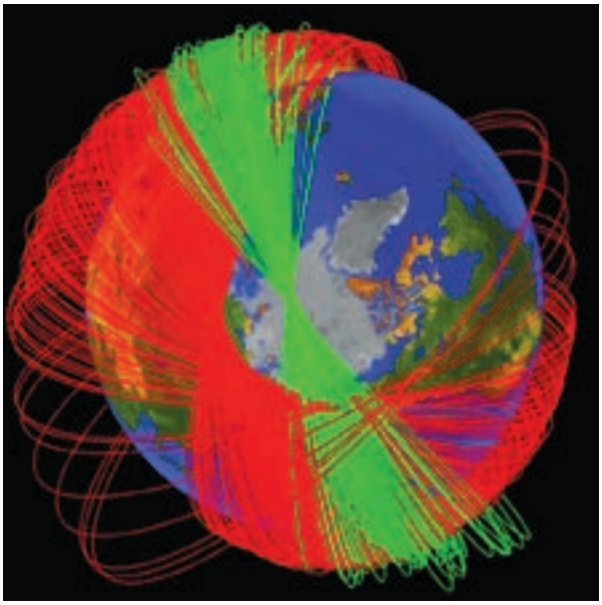


Figure 4.12: Predicted evolution of the Iridium and Cosmos debris planes six months after the collision[5].

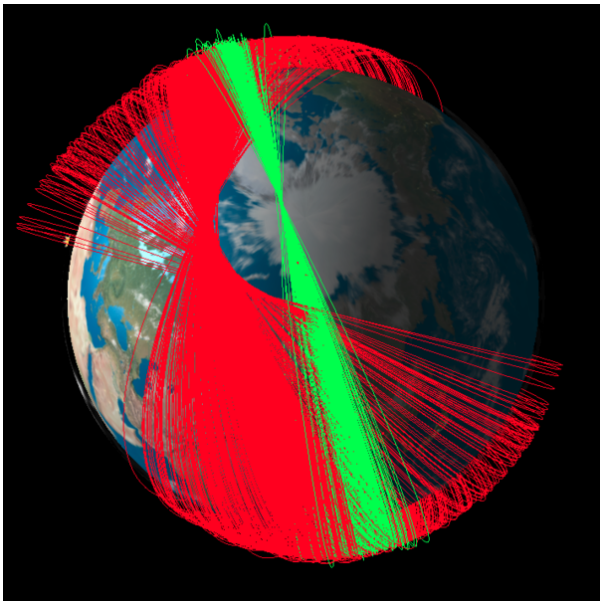


Figure 4.13: The Iridium and Cosmos debris planes six months after the collision according to the tool.

Both Figure 4.12 and Figure 4.13 show the debris from the Iridium 33 and Cosmos 2251 using the same color scheme. In the figures, red depicts the Cosmos 2251 debris objects, whereas green shows the Iridium 33 debris objects. Both figures also show that the debris indeed spread out over the time period of 6 months from the initial orbit of the satellite. However, it can be seen that the prediction is spread out more than the orbits provided from the TLE data. The green orbits show a much wider spread in the prediction model that the tracked objects show. Furthermore, the red orbits show that a few objects were predicted to have a larger divergence from the original orbit.

Another thing the tool is capable of doing, is to show the collision and the objects that came from the collision. This is done by showing the 2 satellites before the collision. Then searching dates that happened after the collision, we can see how fast the new debris objects were tracked over time. Before the collision, the two satellites had orbits that were nearly perpendicular. This is shown in Figure 4.14

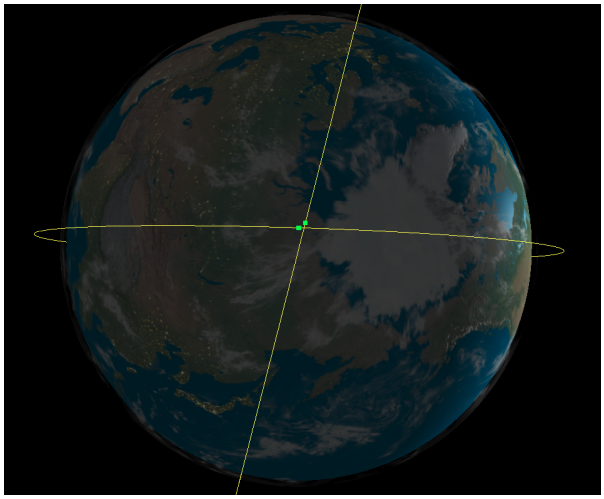


Figure 4.14: The Iridium 33 and Cosmos 2251 orbits about 15 seconds before they collided.

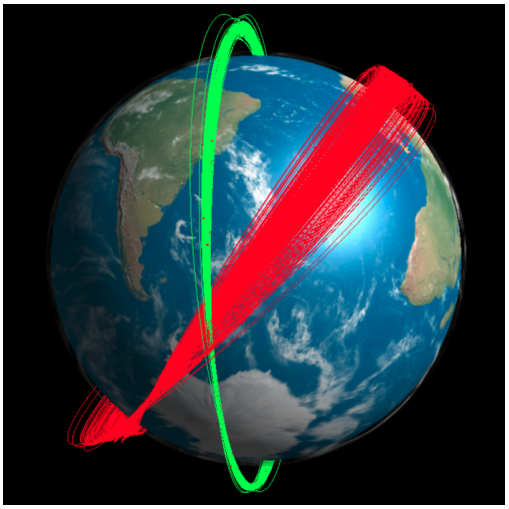


Figure 4.15: Debris one month after the collision (3-10-2009).

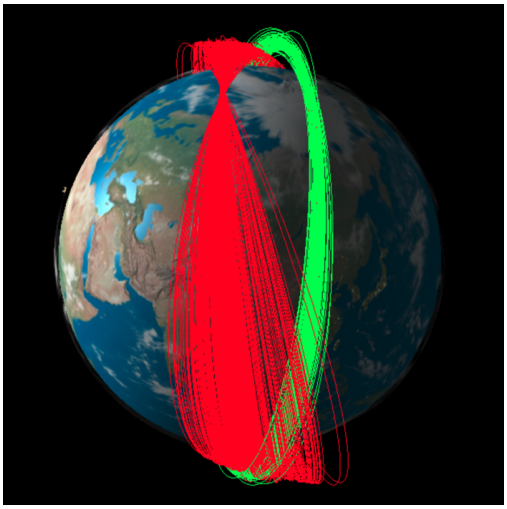


Figure 4.16: Debris two months after the collision (4-10-2009).

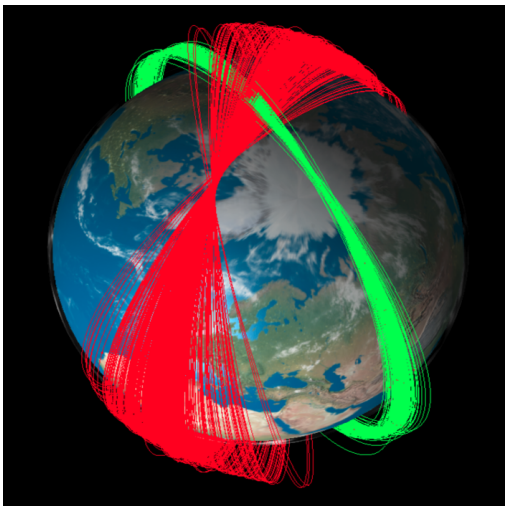


Figure 4.17: Debris three months after the collision (5-10-2009).

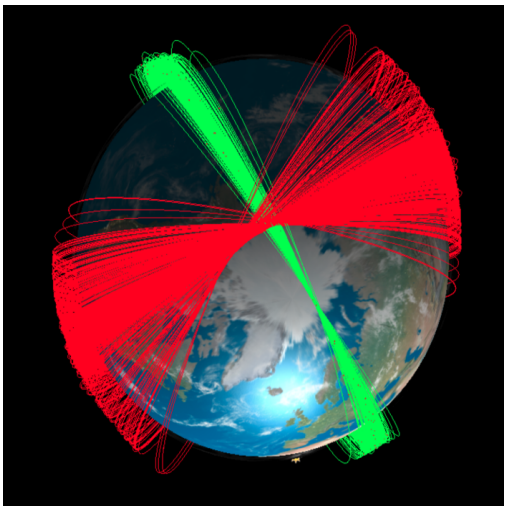


Figure 4.18: Debris four months after the collision (6-10-2009).

Over the course of the next months, the tracking of the newly formed debris started. The debris had to be found, cataloged, and tracked. At the end of March 2009, 823 pieces of debris were tracked and confirmed to be debris from the collision[5]. Furthermore, due to their orbits, the debris can be traced to either one of the original satellites. Due to the collision, debris got flung into different orbits, some objects ended up being higher in altitude, whereas others were lower and even burned up in the atmosphere. Due to the small deviations from the orbit, the orbital period of these pieces of debris are slightly different. This difference in orbital periods of the debris will gradually separate their orbital planes form a shell about the Earth.

This can be demonstrated by comparing the orbits of debris at different points in time. Figures 4.15, 4.16, 4.17, and 4.18 show that the orbits of the debris spread out over time. It can be noted that the debris from the Iridium 33 satellite, depicted as the green orbits, spread out more slowly than the debris from Cosmos 2251 due to their higher inclination, as was predicted in "Satellite Collision Leaves Significant Debris Clouds".

Using the same method, we can look at the debris 5 years after the collision. At this point in time, the debris should have form a shell around the Earth. This is shown in Figure 4.19. The spread of the orbits from the Cosmos 2251 debris is much more uniform than the spread of the Iridium 33 debris. The Iridium 33 debris shows clearly that most of the debris is still clustered around the original orbit of the satellite. Since this debris spread out more slowly, the shell is not uniform yet.

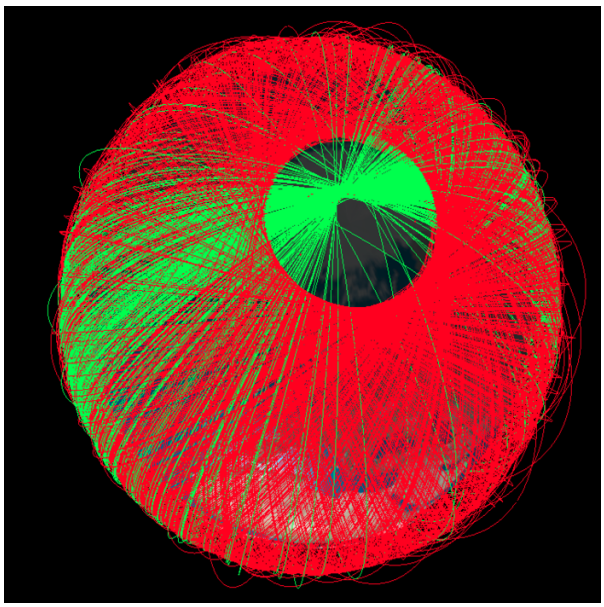


Figure 4.19: The Iridium and Cosmos debris orbits five years after the collision according to the tool.

4.3.3. Educational Purposes

The tool can be used in an educational setting. The tool provides a great way to show students various orbits and reference frames during lectures. And while students are studying the material, they can grab the tool and use the features to visualize the orbits themselves. I have always found that images and visualization help really understand what is going on with these orbits. This tool could therefore be an addition to the book as a study mechanism.

For example, during Astrodynamics in chapter 23.7 from [13], special orbits were discussed and shown. These specialized orbits exploit the effects of specific orbit perturbations. The orbits can be visualized by the tool, if there are or were missions that every flew, that specific orbit. The first specialized orbit that is mentioned in this part of the chapter are sun-synchronous orbits. The sun-synchronous orbit is a high inclination orbit, which precesses through one complete revolution per year. This means that the satellite passes over locations at the same local mean solar time, making the angle of illumination at that point nearly constant as observed from the

satellite. To demonstrate this orbit, a student or teacher could select a mission that flies in a sun-synchronous orbit, like *PROBA 2*. To see whether or not the orbit precesses as intended, a snapshot can be taken from the orbit at the same time of the day for several dates throughout the year. The orbit should be located in the same orientation every single time. For other orbits this is not the case, the orbits precess and will show different rotation of the orbit compared to the Earth. The following four figures show how the orbit of the *PROBA 2* precesses over the year compared to the solar direction.

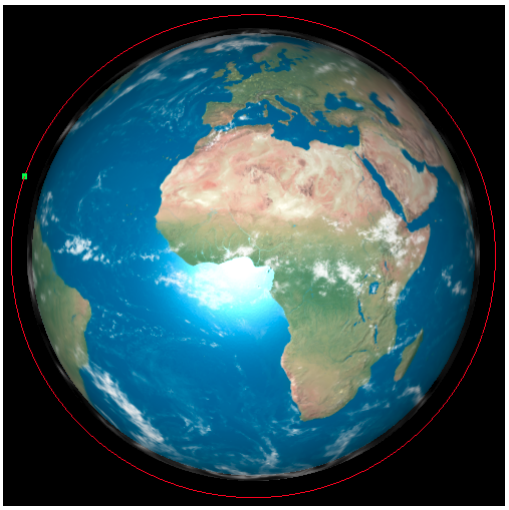


Figure 4.20: PROBA 2 orbit on September 15th 2017 at noon.

Figure 4.21: PROBA 2 orbit on December 15th 2017 at noon.

Figure 4.22: PROBA 2 orbit on March 15th 2018 at noon.

Figure 4.23: PROBA 2 orbit on June 15th 2018 at noon.

As it can be seen from these images, the *PROBA 2* does not change its orbit angle to the sun. This means that the *PROBA 2* satellite is indeed sun-synchronous. For a student it can be very useful to see this and actually experience this using the tool. The concept will be cleared for the student and it might help them with the material from the book.

Another orbit that is mentioned in chapter 23.7 from [13], are the GPS orbits. The book mentioned that originally there were 24 satellites in 6 different planes. Nowadays there are more alternatives to the original *NAVSTAR* GPS satellites. For example, the *GLONASS* and *GALILEO* satellites. When we visualize these two constellations, we can see how the constellations are formed and which orbital planes the satellites operate. The *GALILEO* project is still being completed by the European Union to offer an alternative form of GPS signal. The constellations is not very old yet, which means that the orbital planes of the satellites have not drifted far apart yet.

The *GLONASS* project has been around longer and shows that some orbits are deviating from the original plane. These drifts are caused by inactive satellites within this constellation. This is shown in Figure 4.24.

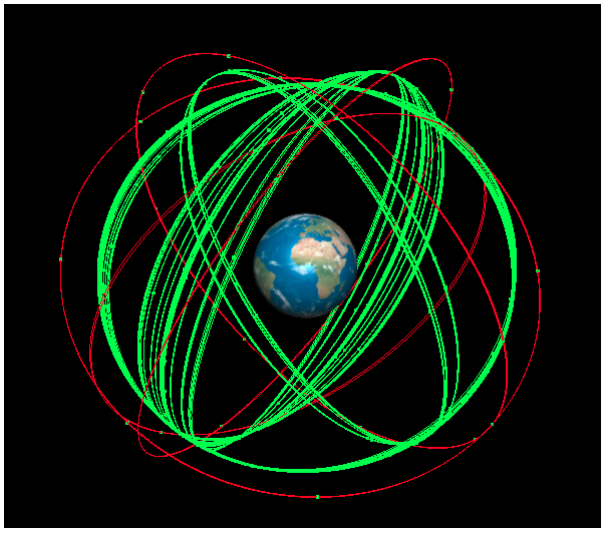


Figure 4.24: The GALILEO satellites (red) and GLONASS satellties (green) on June 15th 2018.

Visualizing the GPS orbits has the benefit that it shows how important the orbital planes are. When the network would not use the orbital planes, the number of satellites that are needed to create the same coverage would increase, increasing the cost of each GPS project. For educational uses, the tool can show how the constellations came to be, using historical data, as well as how the constellations change over time.

A third orbit that is mentioned in the book is the Molniya orbit. This orbit was designed to maximize the time the satellite can look at the Northern pole of the Earth. The orbit is very eccentric, meaning that the perigee and apogee differ a lot. This leads to a difference in time spent pointing at a certain area. The perigee, usually located on the Southern pole, makes the time spent in the vicinity of the Southern pole limited. But the large apogee around the Northern pole, increases the viewing time for this region. The visual aid from the book is the figure shown in Figure 4.25.

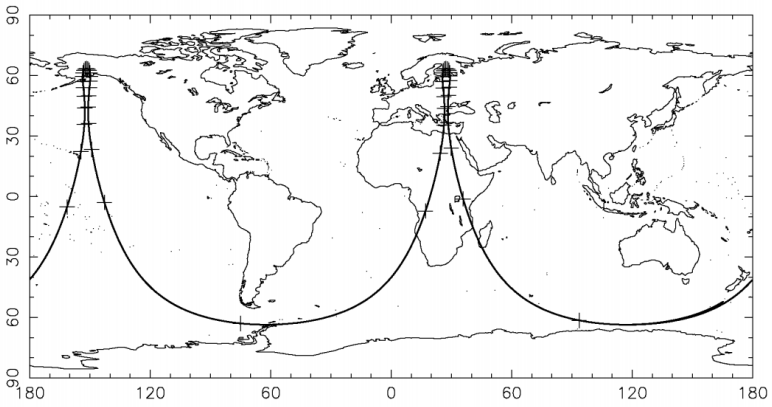


Figure 23.14: Ground track of a Molniya repeat orbit.

Figure 4.25: Figure 23.14 from [13]. The visual aid that is provided in understanding the Molniya orbits

The ground track of the Molniya orbit is shown. In this ground track, the time is indicated by the dashed through the line. Around the Northern pole, the amount of dashed in the ground track

is significantly higher than at the Southern pole. The visualization tool can step in and add to this plot. Instead of having a student see the ground track with these time dashes, the student can open the tool and visualize the tool. Using the time acceleration function, the student can clearly see the difference in time spend in each of the regions. The student can do this for a single satellite, but also for more. This is shown in Figure 4.26.

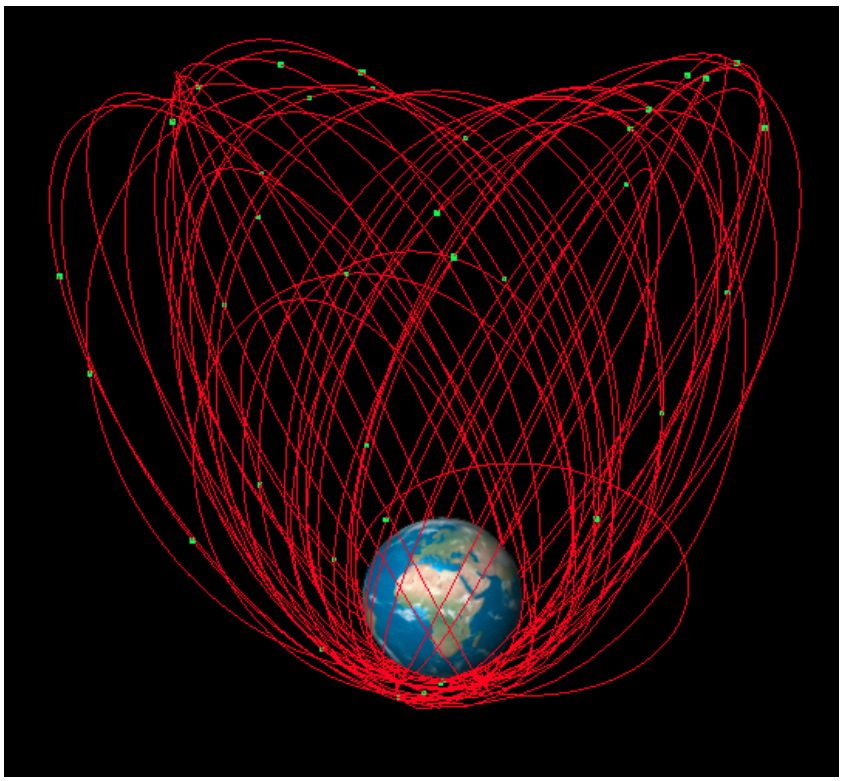


Figure 4.26: Molniya orbits on June 15th 2018.

This one image can already be helpful. It can be seen that the number of satellites that are located near the top of their orbit is much greater than the number of satellites that are in the bottom of their orbit. If a line were to be drawn about halfway, the number of satellites that are above that line is roughly 24, below the line it is only 11 or 12. A visualization with time acceleration enhances the experience even more than this static image.

The last educational tool that can be accessed is the ability to transform the coordinate systems. The tool is capable of transforming between the Earth Centered Earth Fixed coordinate system and the Earth Centered Inertial coordinate system. The first one shows a non-rotating Earth, whereas the second one shows the rotating Earth. Both reference frames have the Earth at the center, meaning that the rotation of Earth around the Sun is not displayed. In chapter 23.7 from [13], one of the orbits that is shown in a graph is a circular geosynchronous orbit with an inclination of around 30^o. In the ECI reference frame, the orbit looks indeed circular, as shown in Figure 4.27.

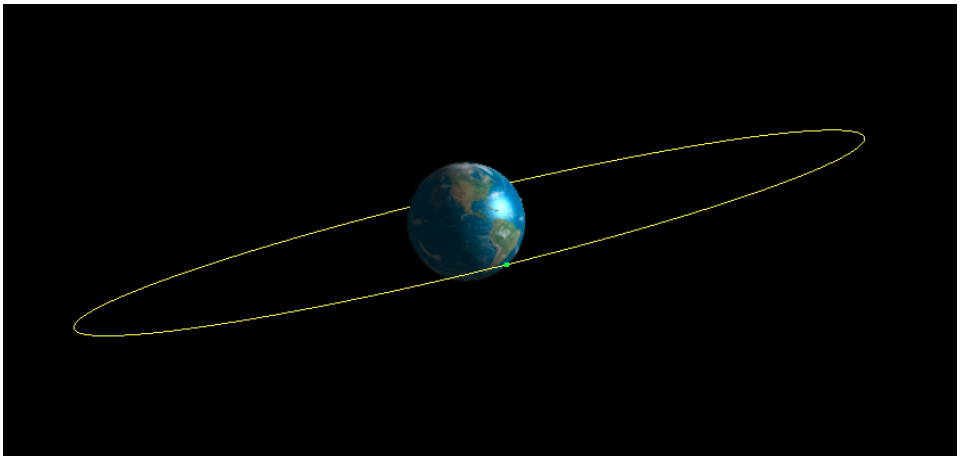


Figure 4.27: A geostationary orbit on June 15th 2018 in the ECI system.

When this orbit is converted to the ECEF reference frame, the orbit turns into something that looks like an 8. This is because at the geosynchronous altitude the period of the orbit allows each satellite to maintain its position over the same region of the world. The inclination creates the shape that is seen, since the satellite moves in latitude during one orbit. This is shown in Figure 4.28.

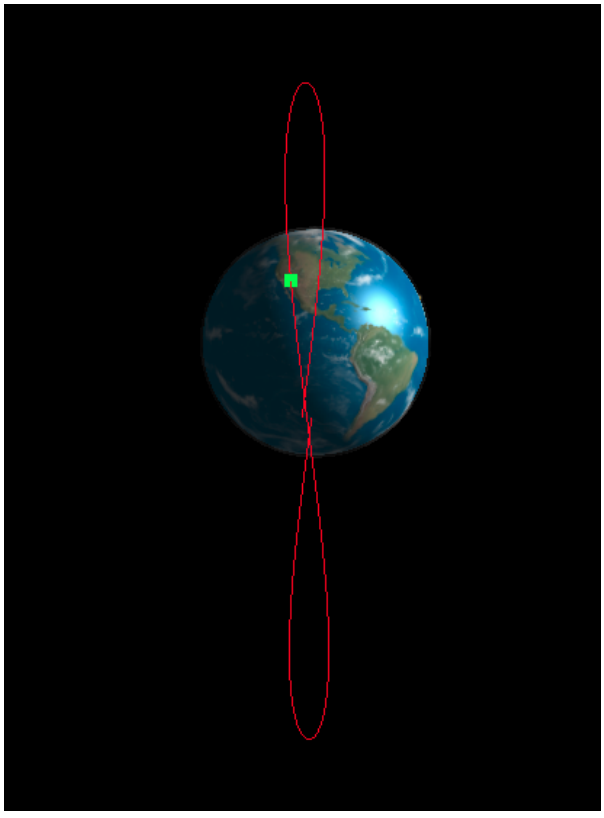


Figure 4.28: A geostationary orbit on June 15th 2018 in the ECEF system.

Similarly to the other educational uses, the tool adds a visual component to the studies and allows the user to understand the material better by providing visual aids. The visualization is easy to use and makes it therefore a great addition to classes, like Astrodynamics.

4.3.4. Space Debris Visualization

Over the years, the amount of space debris has been steadily rising. Using the group menu, the user of the tool can select to visualize just the space debris to show how it increases over time. A visualization that shows how space debris grows over time has been done before, by for example Dr. Stuart Grey, who created an interactive space debris visualisation for the Royal Institution Christmas lectures in 2015 using *space-track.org* data and WebGL[3]. His animation focused on the visualization and does not allow the user to take control over what they see. The video of this visualization can be found on here: https://www.youtube.com/watch?v=wPXCk85wMSQ. The interactive tool is very limited as it only shows the visualization that Dr. Grey created.

The developed tool has very similar features. Not only is it capable of visualizing the satellites in orbit, it also can distinguish between the different types of orbits. For example, instead of showing both payload and debris in a visualization, they can be shown separately. This can be done over time as well, since the tool is capable of obtain historical data. For this use case, we will look into the growth of the Space Debris, jumping a decade at a time, starting on January 1st 1960. The debris that is shown are the objects that are classified as debris, as well as rocket bodies, since these objects are a specific type of debris. The next images show this growth of space debris over time. Space-track.org categorizes payloads that are no longer functioning as payloads, whereas they should be classified as debris. This means that there is actually more debris than shown in the images.

Figure 4.29: Space debris on January 1st 1960.

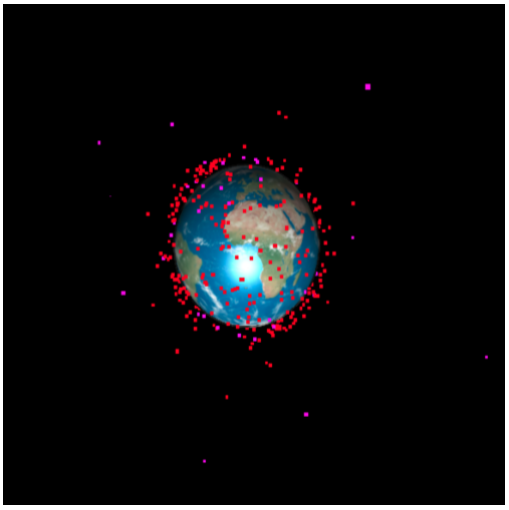


Figure 4.30: Space debris on January 1st 1970.

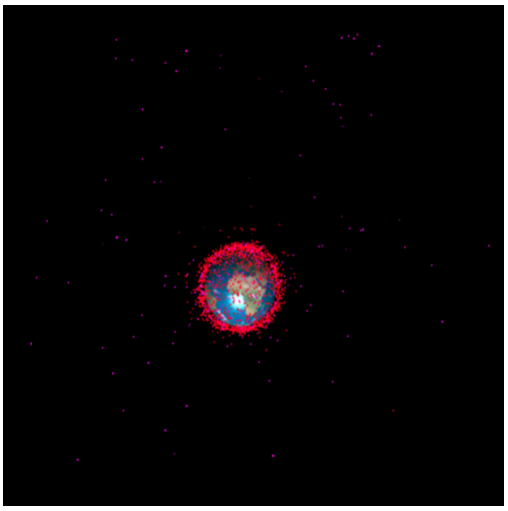


Figure 4.31: PSpace debris on January 1st 1980.

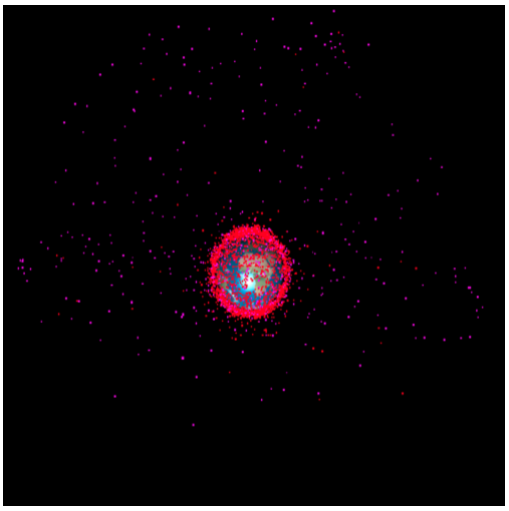


Figure 4.32: Space debris on January 1st 1990.

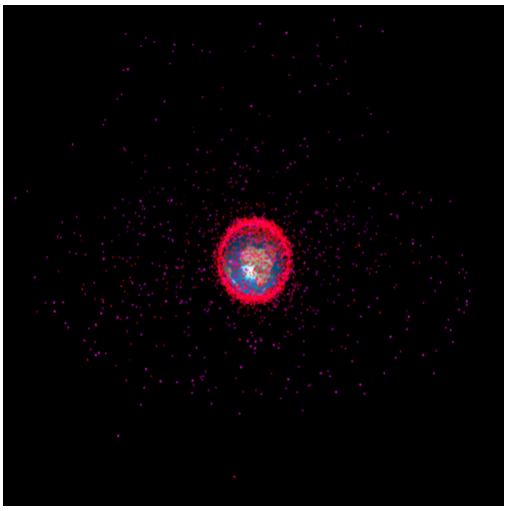


Figure 4.33: Space debris on January 1st 2000.

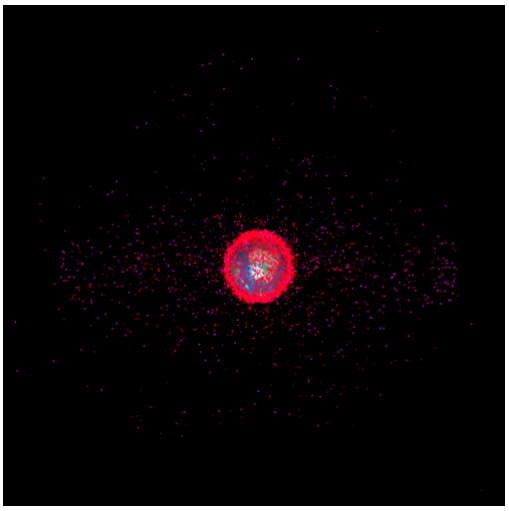


Figure 4.34: Space debris on January 1st 2010.

Compared to the tool that was created by Dr. Grey, this tool allows for more interaction and functionality. It also can show the distinction between the types of objects and contains information about the object itself.

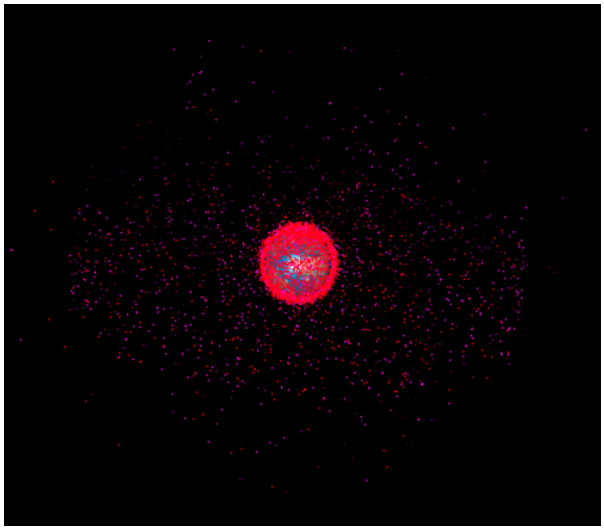


Figure 4.35: Space debris on June 15th 2018.

4.3.5. SWARM and EPOP

Another way to use the tool is to determine the position of satellites at certain points in time with relation to phenomenon on the Earth's surface or atmosphere. This is the case for an event that happened on March 11th, 2016. At 6:47 am (GMT), an aurora was detected above Canada. At the time, the e-POP, Swarm A, and Swarm C satellites were in the neighborhood of the event. Magnetic fields and disturbances in the field can be detected by these three satellites. In a paper that appeared earlier this year, the measurements of the satellites were compared. The paper was "Alfvénic Dynamics and Fine Structuring of Discrete Auroral Arcs: Swarm and e-POP Observations" written by Miles, et al [6]. In their paper, a graph that shows the paths of the satellites as well as the data along the path is shown. The figure is shown in Figure 4.36.

The tool can reproduce this situation. To do this a user simply selects the date, and then jumps to 6:47 am on that day. Then searching for the Swarm A and C and e-POP satellites, the result is as shown in Figure 4.37. The simplified Earth is shown to clearly see the land mass. At 6:47 am northern Canada is dark, making it hard to make out where the location is exactly.

In the article, these were the only satellites that were used. But the tool can be used to determine if there were more satellites in the neighborhood of this aurora. To do this, the user can select the payload objects, eliminating all debris, and go to the time and location of the event. When a user does this, the result is as shown in Figure 4.38. In the figure it can be seen that a fourth satellite is near the aurora. This is cosmos 1733. This satellite, launched in 1986, was a Soviet ELINT (Electronic and Signals Intelligence) satellite launched from the Plesetsk cosmodrome.[1]. Since this is an old intelligence satellite, which is probably long dead in 2016, there is most likely no data recorded by this satellite for this event.

Even though this specific search might not have resulted in another satellite that looks down at event, this way of quickly finding satellites for other events might proof to be fruitful.

[1] https://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1986-018A

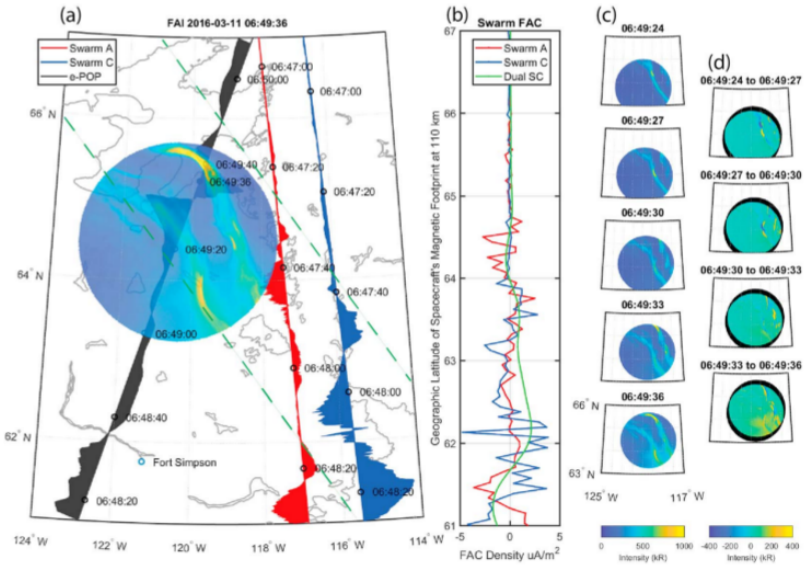


Figure 1. Mapped, near conjugate measurements from e-POP (black) and Swarm A (red) and C (blue) spacecraft. (a) Spacecraft tracks overlaid on a coastline map; the dashed green lines extrapolate the general region of the arc, FAI near-infrared image from 06:49:36 UT, with superposed cross-track magnetic perturbations. (b) Field-aligned current density as estimated by single spacecraft methods for Swarm A (red), Swarm C (blue), and by the Swarm dual spacecraft integral method (green). (c) e-POP FAI images at 3 s intervals showing dynamic aurora. (d) Differenced FAI images showing equatorward and poleward motions of southern and northern arc features, respectively. See text for details.

Figure 4.36: Figure 1 from [6]

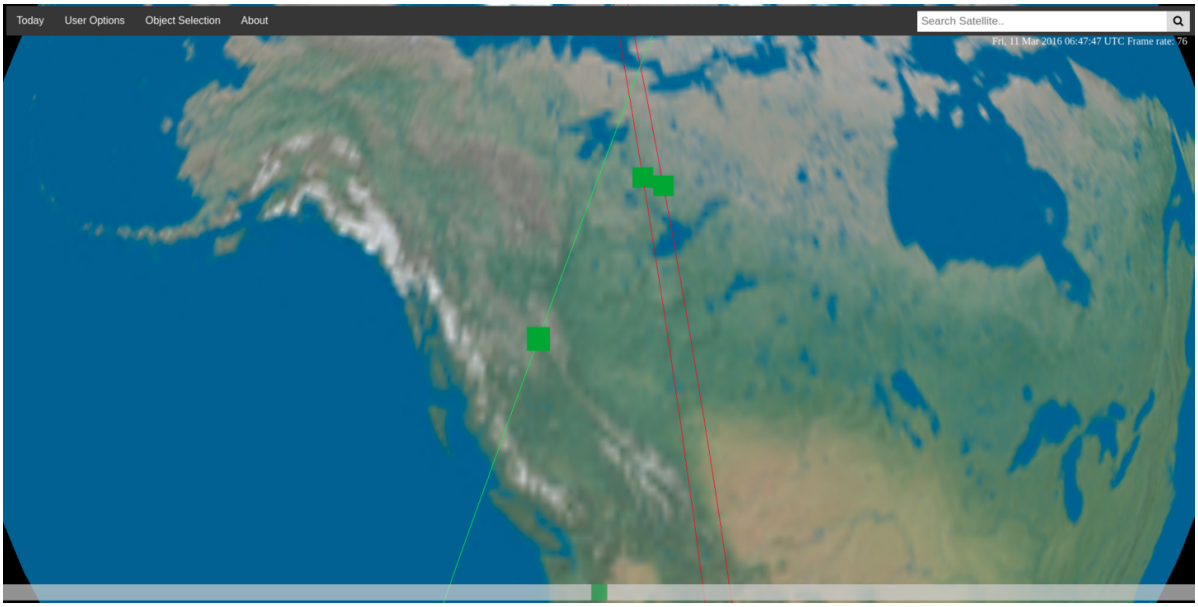


Figure 4.37: Recreation of the Aurora location on March 11th 2016 at 6:47.47 am (GMT).

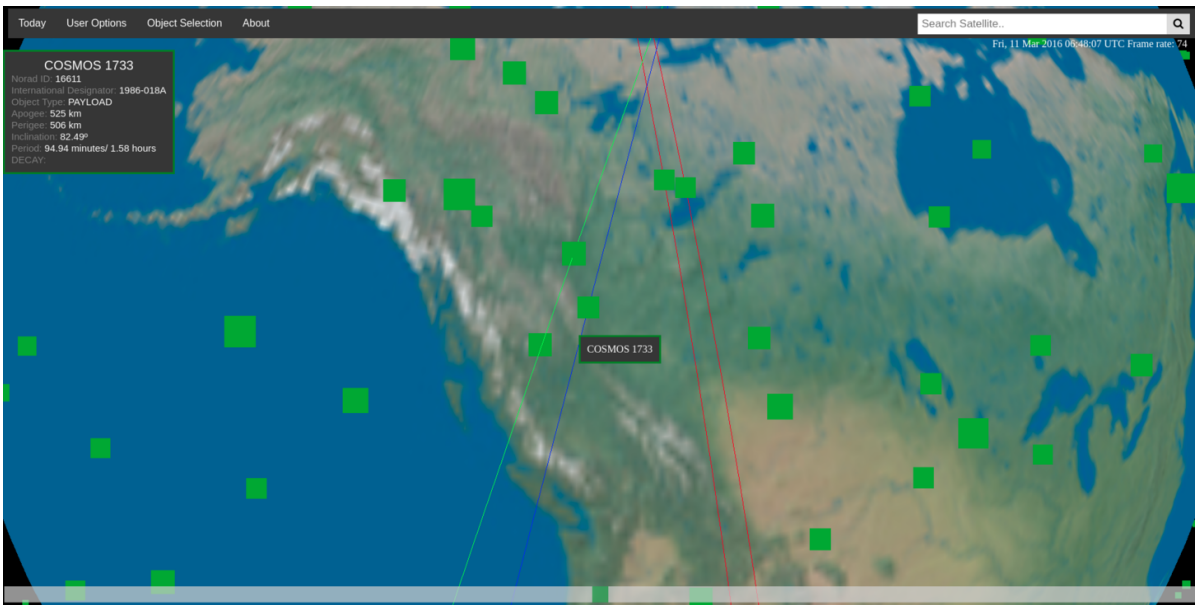


Figure 4.38: Searching for satellites around the aurora event on March 11th 2016 at 6:48.07 am.

5

Conclusions and Recommendations

For this thesis project an interactive, web-based, near-Earth orbit visualization tool was developed. This was done to answer the research question: *What are the uses of another interactive, web-based, near-Earth orbit visualization tool?*

The tool has proven to be useful in several cases. The tool is capable of quickly visualizing orbits, constellations, and debris. This is not only possible for the current situation but also for situations in the past. One of the cases that was looked at was the collision of the Iridium 33 and Cosmos 2251 satellites. Prediction of how the orbits would spread can be compared with how the situation actually turned out to be. It seems like the predictions might have overestimated the rate of spreading. This is one situation where having a tool like this saves researchers a lot of time actually visualizing these orbits. The tool can also be used to help students study certain orbits and trajectories, as well as reference frames. Adding a visualization next to the textbook might help the students understand the material better.

Furthermore, the tool is able to quickly switch between dates, which can be used to take timed snap shots. This can be used to demonstrate the growth of a constellation, or growth of debris.

Lastly, the tool provides an easily accessible platform for everyone interested in space to visualize orbits. These visualization can be used in presentations or as a tool to teach others. The ease of use also won't discourage people from using the tool, unlike some desktop tools that require additional knowledge or installations to work.

In section 4.2, several limitations on the tool were presented. In the future these limitations should be addressed and solved. However, looking beyond the limitations, features can be added features in the future as well.

5.1. Custom TLE or ephemeris files.

One of the limitations that were mentioned in section 4.2 is the ability to add TLE information that is not in the data. A user might want the ability to combine current orbits with historical orbits. This could be the case when the user wants to compare satellites that have flown similar profiles, but at different times. A user could potentially change the original TLE information of two missions so that the timestamps are the same. The orbits for these missions could then be compared in one way or the other. To achieve this feature, the user should be able to add their own adjusted TLE or ephemeris files. This can be done either in a different end point or have it as an option available to the user. There should be an algorithm that checks if the data that is entered, is valid.

5.2. User Profiles

Each user might have their own specific searches that they always go back to. They might be interested in a single satellite or constellation. To avoid having to go through the steps that show these orbits and satellites each time the user opens the tool, user profiles should be created. Each user would have a profile with their most recent searches stored. This feature could potentially be combined with the custom TLE feature. A user would be able to add a TLE and have it ready for use. Once the user is logged in to the profile, the user could combine their last search and add the stored TLE to the search. This would enhance the usability of the tool for users that want specific data from the tool.

5.3. Camera Perspectives

Currently the tool possesses one camera perspective. The camera is always looking at the center of the Earth. The user is capable of rotating the visualization around this point, but is not allowed to change where the camera is pointing. In the future, two camera perspective should be added.

The first perspective that should be added is one where instead of focusing on the center of the Earth, an object is the focus. The user could experience the visualization from the eyes of an object. This could be useful for visualizing the view from the international space station or for missions of two satellites that are in close proximity. In case of the latter, the user could see how the satellites move with respect to each other.

The second perspective that should be added is a ground view. The user should be able to type in coordinates or click on an area on the ground to select their ground point. After the selection, the camera should be pointed from this ground point upward in the sky. This would allow the user to see which satellites are coming over at what time. This could be used to predict GPS coverage in remote areas, or even when the international space station comes over. It could also be used for the last use case. In this case the view could be from the ground to the sky and see the SWARM and EPOP satellites move over.

A related feature would be adding labeled markers on the Earth, for example cities or observatories.

5.4. Ground Tracks

In the visualization the tracks of the satellites are shown, but it is not possible to project these tracks onto the ground. This can be done by taking the ECEF positions. Then convert these positions to unit vectors and multiply by the radius of the Earth. It could even be projected at a certain altitude. This could be used for aurora studies, for example.

Another options is to give the option to show the ground track on a two-dimensional plane. The user could potentially select a number of satellites and then having the option to project it onto a two-dimensional surface.

5.5. Sub-solar Point

The determination of the sub-solar point is done in the back-end. Once the epoch and the coordinates at this epoch are send to the front-end, the sub-solar point is just rotated around the Earth with the rotation speed of the Earth. The algorithm only takes the longitudinal motion into account. The latitudinal motion is not addressed. For short visualizations this is not a problem. However, for long term visualizations the sub-solar point deviated from the true sub-solar point. This should be addressed in the future.

Another option that should be made available to the user is to add the terminator, the line that clearly indicates the division between day and night. Using the complex Earth model, this is currently shown in the visualization, but it is not entirely clear where the line is located exactly. In future versions of the tool, this should be a consideration.

5.6. More ideas

Besides the ideas mentioned above, there are plenty more ideas that could be implemented in future versions of the tool. Some more ideas are:

- Tick marks along the orbit, with labels indicating the UTC time, at 10 minute intervals, for example.
- Allowing the user to set each orbit/object color manually with a color picker widget.
- Customized groups of satellites, for example based on ESA DISCOS data.

Bibliography

[1] Hejduk, M. D. and Ghrist, R. W. Solar Radiation Binning for the Geosynchronous Orbit. AAS 11-58. NASA Technical Reports Server., 2011.

[2] Berrut, J.P., Trefethen, L.N. Barycentric Lagrange Interpolation. *SIAM REVIEW Vol. 46, No. 3, pp. 501–517*, 2004.

[3] Grey, S. Royal Institution Advent Calendar: A History of Space Debris .

[4] Kardol, S. Orbit Visualization: Literature Study, 2017.

[5] Liou, J.C. Satellite Collision Leaves Significant Debris Cloud. *Orbital Debris Quarterly News, Vol. 13(2), pp. 1-2*, 2009.

[6] Miles, D. M., Mann, I. R., Pakhotin, I. P., Burchill, J. K., Howarth, A. D., Knudsen, D. J., Yau, A. W. Alfvénic Dynamics and Fine Structuring of Discrete Auroral Arcs: Swarm and e-POP Observations. *Geophysical Research Letters, 45,*, 2018.

[7] Montenbruck, O., Gill, E. *Satellite Orbits.* Springer, 2001.

[8] Pogurelskiy, O. Lecture 3: Coordinate systems. LinkedIn: SlideShare. Accessed 2018-04-07.

[9] Rhodes, B. Python-sgp4 library. `https://pypi.python.org/pypi/sgp4`, 2018. Accessed 2018-04-08.

[10] Schingler, R. Planet Launches Satellite Constellation To Image The Whole Planet Daily. `https://www.planet.com/pulse/planet-launches-satellite-constellation-to-image-the-whole-planet-daily/`, 2017.

[11] Seidelmann, P.K. *Explanatory Supplement to the Astronomical Almanac.* Mill Valley, CA: University Science Books., 1992.

[12] Vallado D. A., Crawford P., Hujsak R., et al. Revisiting Spacetrack Report 3 - Models for Propagation of NORAD Element Sets. *The AIAA/AAS Astrodynamics Specialist Conference, Keystone*, 2006.

[13] Wakker, K. *Fundamentals of Astrodynamics.* Delft University of Technology, 2015.