POLITECNICO DI TORINO

Corso di Laurea Magistrale in ingegneria Aerospaziale

Tesi di Laurea

Evaluation of lighting condition for SROC docking at Space Rider

Relatore
Prof. Sabrina Corpino

Candidato
Antonio Scalise

Co-Relatore
Ing. Guglielmo Daddi

Index

Abstract

The following thesis aims to evaluate the lighting conditions during the docking phase of SROC to Space Rider, through graphical simulation in Blender environment. Within the same there are: an introductory chapter that describes the work that will be analyzed in the following chapters, emphasizing the key elements of the project; a chapter on the implementation of the Python code necessary to perform the simulation within the said program, containing a detailed analysis of all the steps taken, justifying the choices made, to achieve the objective; a chapter describing the realization of the Earth, complete with atmosphere and clouds, in which we focus more on how to build the shader of these three elements, in order to ensure a good level of photometric accuracy of the images, respecting the physical hypotheses relative to phenomena involved; and, finally, a final chapter where the results obtained are shown and commented on. In addition, in the appendix, the code in full form and some file-type to read to the code to perform the simulation correctly.

1. Introduction

1.1 Mission

Within the following thesis will be analyzed the steps that have been taken to achieve a correct simulator, from the geometric and photorealistic point of view, the process of docking the satellite SROC to the satellite Space Rider. The SROC satellite, developed by the CUBESAT Team of the Politecnico di Torino, aims to closely observe Space Rider (Ref. [1]). The main challenges in the design of the satellite include: navigation, propulsion (since the mission involves proximity operations), the definition of mechanisms to be extracted/retracted and, finally, observation. The following thesis was born because of this last challenge, in fact, graphically simulating the mission, through the software Blender, you can have a clear and immediate idea of what SROC will see during its mission of observation. The collection of images by SROC takes place through a multispectral camera that operates in the visible, near infrared and thermal infrared; the following simulation deals with the images acquired in the frequencies in the visible field of the spectrum of light. The other main element of the mission is Space Rider, visible in Figure 1.1.

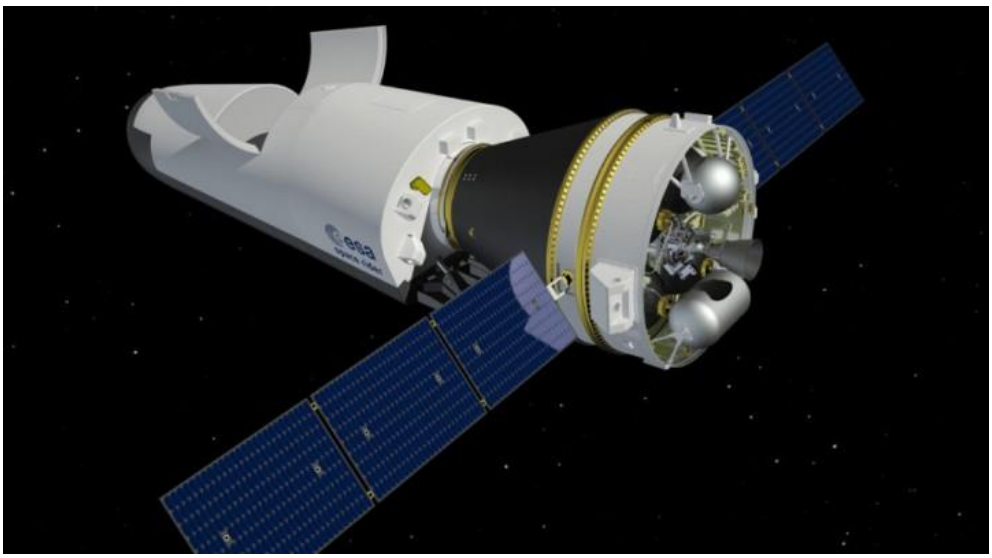

Fig. 1.1: Render of Space Rider, Ref. [2]

This unmanned spacecraft is automated and capable of re-entering Earth so it can be reused. This project belongs to the ESA (European Space Agency) and sees a strong participation of Italy through its space agency (ASI, Italian Space Agency). It is appropriate to mention a project that predates Space Rider, but with the same objective of making an autonomous flight until the return to the atmosphere. This project is the ESA IXV mission, consisting of a space system with no wings (Fig. 1.2) which was launched on 11 February 2015 by the spaceport of Kourou, French Guiana via a Vega launcher (Ref. [3]). The vehicle was experimental in order to test the re-entry capability and proper

functionality of the systems mounted on it to perform this task. The suborbital flight of IXV lasted 102 minutes, in which the vehicle reached the maximum altitude of 412 Km (Ref. [3]), before starting the re-entry into the atmosphere. The orbit has been designed in such a way as to have a re-entry speed of 7.5 Km/s, necessary to create the same conditions that are found in the re-entry of vehicles from the low orbit, in order to test the systems in this condition to assess their compatibility with Space Rider vehicles. The return, coordinated by ALTEC (Advanced Logistic Technology Engineering Center) of Turin, has been completed correctly, ending in the Pacific Ocean, today the vehicle is visible inside the airport of Turin Caselle (Fig. 1.2).

Fig. 1.2: IXV exhibited at Torino Caselle Airport after flight

This mission opened the doors to the world of unmanned vehicles capable of re-entering the atmosphere and, therefore, capable of being reused. The time of Space Rider's stay in space largely exceeds 100 hours, in fact, will be able to stay in space for two months, in which will perform various activities. The 1200 l cargo bay (Ref. [4]), which holds up to 800 Kg (Ref. [2]), can be used as a real space laboratory, within which experiments can be carried out in microgravity, other possible missions that the vehicle can perform are: Observation of satellites and Earth and robotic space exploration, i.e. without the presence of astronauts (Ref. [4]). Space Rider will be launched in 2023 via a VEGA-C launcher (Ref. [4]). The observation activity that SROC performs on Space Rider is very important in order to assess its operation and integrity (crucial task since the vehicle is reusable). The activity carried out by SROC is part of the use of CUBESAT as support to the activities, and therefore to the missions, of larger spacecrafts, such as Space Rider. Other examples of such supporting activities are: the observation of satellites arrived at the end of life to properly predispose the disposal then remove it correctly from the orbit, without leaving space debris; the inspection of objects in deep space which would make it easier to create, for example, a lunar space station. The

computer simulation activity of the mission largely supports the design of this type of missions, for example to assess the correctness or the possible modification of the trajectories and the settings of the observant satellite; or to evaluate, visually, whether the chosen optical sensor is suitable for the purpose; or, as in the case of this work, assess the lighting conditions of the satellite observed to dock. As you will see in Chapter 2, the code has been created in such a way as to be very flexible to changes in trajectories, structures, bodies and sensors, precisely in order to ensure the possibility of use even outside the subject of this work.

1.2 Optical sensor

Space Rider's observation activity takes place thanks to a Hyperscout camera that allows to collect hyperspectral images, i.e. in the fields of the visible and near-infrared in the spectrum of light. Extending the frequencies collected from the camera beyond the visible range allows the sensor mentioned above to be used for applications such as: monitoring of vegetation, volcanic activity, the risk of fire, oil spills, water quality, etc. (Ref. [5]). In the specific case of the SROC mission, this sensor is used to observe satellites, as explained in the previous paragraph. Wanting to define in a more general way the optical sensors used in space, these are part of the payload, along with all the systems necessary to carry out the mission assigned to spacecraft. Specifically, optical sensors are placed between passive sensors, since they collect electromagnetic radiation emitted, or reflected, by the observed body, just like the human eye. To better understand the distinction between active and passive, before proceeding with the operation of optical sensors, it is appropriate to mention an active sensor, the LIDAR. It is placed between the active sensors since it emits a laser beam to perform the measurements, that is, it is itself to produce the electromagnetic radiation necessary for the measurement, unlike the passive ones that collect one produced by other bodies. The passive sensor consists of two main elements: the collector and the detector. The first has the task of collecting electromagnetic radiation through a system of lenses that, depending on their arrangement, is able to eliminate effects such as: chromatic aberration; spherical aberration, incorrect distribution of photons that are not conveyed exactly to the focal point, which causes a distortion of the image in which the photons farthest from the axis are perceived at a distance other than those close to the axis; coma; astigmatism, or the inability to locate a precise point on the detector (you see the halo even if the source is punctual); and finally, the distortion. The element that follows the collector is the detector, this element is the first to be hit by electromagnetic radiation and, when this happens, the elements that compose it vary their physical and chemical properties and, these variations, are used to detect images. The first detectors were made with films that, struck by the light radiation, made the image

through the chemical reaction of the materials that composed it. The technology of the detectors has undergone a considerable step forward with the advent of semiconductor materials, with which it is possible to collect images both in the field of the visible and in the infrared. The realization of the image through semiconductors takes place through the production of an electrical signal of the same when hit by electromagnetic radiation, such signal is related to the radiation that strikes the detector. So it's easy to understand that image information is translated into an electrical signal. Therefore, building a matrix of semiconductor material elements (photosite), you can store the image related to electromagnetic waves that enter the collector. Today's detector technology is divided into two types: CCD and CMOS. The first are a matrix of coupled photosites (Charge-Coupled-Device), so that an element can transfer the charge to another adjacent element. Each photosite acts as explained above, generating an electrical signal through which the matrix of pixels that make up the image is reconstructed. Note that the concept of photosite is related to the detector and is a physical element, while the pixel is a computer concept. The CMOS is a less expensive alternative than the CCD which, however, generates more defined images. The main elements on which the CMOS is based are the inverter and the MOSFET transistor. Obviously, whenever you have an electrical signal produced by a system, to convert it into information (audio, video, etc.) you need some steps: the signal undergoes the sampling process, that is, the value of the signal in a series of points of the wave, after which it is converted from analog to digital, so that it can be stored in a memory or displayed on a monitor. This process takes place in a controlled manner and requires the provision of electrical power, so the optical sensor requires connection to the EPS and satellite control systems in the case of space. As mentioned above, semiconductor sensors are able to operate in the field of infrared and visible, in this regard there are different types of sensors:

- Pancromatic Camera (PAN): operates within a spectrum band related to what you want to detect (e.g. the entire visible band, produces high spatial resolution images and is chosen when this requirement is relevant);
- Multispectral chamber (MSI): operates between 3 and 50 spectrum bands, useful for identifying rocks, trees, water, etc;
- Hyperspectral Chambers (HSI): this is the case of the chamber mounted on SROC, these chambers operate between 30 and 300 bands with a high spectral resolution; they are used to assess which materials is composed of a rock, the type of harvests, etc;
- Ultraspectral chambers (USI): they work between 300 and 3000 bands, typically in infrared, to study the emission spectrum of Earth and atmosphere; they are characterized by very high spatial resolution;

If you list the differences between the different camera types, you should define the parameters that characterize the camera and understand how they affect the captured image. In this respect it is useful to observe Figure 1.3.

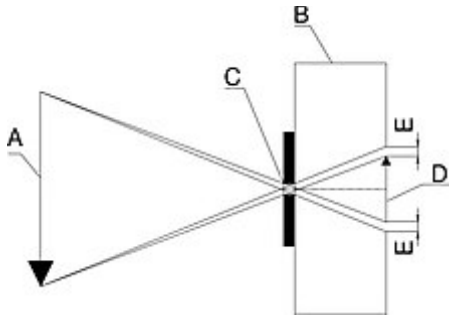

Fig. 1.3: Geometry of camera

- The dimension A represents the diameter of the scene, that is the diameter of the part of subject that the camera frames and that will be, therefore, immortalized by the same;
- The dimension D represents the diameter of the image, that is the size of the image that will be produced by the camera, that can be calculated in pixels (it will be set by the user in the simulation);
- The dimension C represents the diameter of the lens;
- The distance between A and C is the distance between the scene and the camera (in the case of a satellite framing the Earth coincides with the flight altitude);
- The distance between D and C (B) is the focal distance, that is the distance between the lens and the plane on which the image is projected, that is where the detector is located (this will also be set during the simulation);

Another important parameter that defines the camera is the aperture, this is indicated as f followed by a number, that number the smaller the light enters inside the sensor, the larger the hole from which the light enters the sensor is small. This parameter greatly affects the image. In fact, if the opening is large (e.g. f 4) the camera will focus the object close by, blurring the background according to the value (the more marked effect, the smaller the value), thus creating a portrait effect. On the contrary, for opening values like f22, you have a good focus of the entire image. This parameter can also be modified in Blender, in the simulation that will be explained in the next chapter. However, it has not been modified since in Blender the default value (f2.8) coincides with that of both sensors selectable to perform the simulation.

1.3 Software

To carry out the graphical simulation of the mission described in paragraph 1.1, it has been chosen to use the Blender software. This software is a three-dimensional graphics program, whose source code was first released in 2002. Today the software is open source and can be downloaded, free of charge, from the website https://www.blender.org/. The first version that was used, in this work, to design the simulator was the 2.80, then it was moved to 2.83 and, finally, to 2.93 for the presence of additional functions and shortcuts useful to the process of writing and debugging of the code. Within Blender there are several useful functions: such as the ability to create geometries, even complex, in the form of mesh, only by accessing a menu on the home page; the ability to import different CAD file formats within the scene; the possibility to export objects created in the scene in different formats (eg. file .obj); the presence of a graphics engine for gaming and the presence of two graphics engines for rendering. The first graphical engine that manages the renders is Eevee, such engine is very fast but is not able to render all the nodes of the shader (see ch. 3). The second engine, present inside the software, is Cycles, that is of type unbiased, that is photorealistic. The difference between engines like Cycles and the others is that the first ones have as their only objective to realize rendering physically accurate with respect to the path of light in the scene, simulating also the interaction between the various elements of the scene; while biased motors certainly simulate lighting within the scene but with optimized algorithms to perform the calculation in a short time. The need to carry out simulations that maintain a high level of photorealism, in order to make them as close as possible to reality, has led to the need to choose Cycles as the graphics engine for the simulation, despite the considerable increase in render times. Cycles is a path tracing graphic engine, that is, it works in such a way as to integrate all the light that reaches a point of a surface and it evaluates how much light the optical sensor reaches through the application of a BDRF function, this for all the pixels of the image. This type of algorithms allows you to obtain high quality images when applied to a physically based model, they are also able to simulate various effects that with other graphics engines are not displayed (e.g. shadows with Eevee). Other examples of effects that are included in path tracing engines are: motion blur, light propagation in volumes to simulate scattering, etc. In order to reduce image noise you should increase the number of rays that are traced, always taking into account the cost, and therefore the time, computational necessary. Cycles is a graphical engine that uses path tracing with next event extimation, a technique that allows to render more detailed and larger scenes, but not optimal for all types of light effects. In particular, Cycles acts in inverse to what was shown before for path tracing engines, that is, it traces the rays of light from the camera to the scene (looking for surfaces or light sources whose rays enter the camera). The advantage is to work only on the rays that actually enter the camera, the difficulty is to find useful paths of light, that is, that, from a surface or

a light source, reach the camera. As mentioned, the renders produced by this type of engines are affected by noise, the higher the lower the number of rays traced. To better understand the phenomenon it is appropriate to give an example, think of a sphere illuminated by a light source, the source is in a known position and, therefore, does not generate noise, on the contrary the sphere, according to its material, will reflect more or less light. The algorithm calculates pixels per pixel the amount of reflected light (BRDF function) evenly distributing the rays on the sphere. In doing so, however, it is possible to leave out some bright spots and identify another that maybe is not, all this generates noise. In Blender, in addition to increasing render sampling, you can activate the denoising that acts as a filter for the image, through the information collected during the rendering (feature pass), in order to reduce noise. Note that, even in this case, the higher the quality, the higher the calculation times. To conclude the paragraph on software it is appropriate to explain what was the main reason that directed the choice towards the same. Within the software there is a section 'Script', in which it is possible to insert lines of code that carry out the same task as the GUI but in programmed way. Such function is fundamental from the point of view of the realization of the simulation that can be, therefore, flexible regarding any change (type of bodies involved, trajectories, arrangements, etc.), without the user having to do anything other than change the path of the files to read, selecting those of his interest. The ability to program, in Python language, the simulation is a main function to meet one of the most important requirements of the project, ie the flexibility of the simulation with respect to any changes. As a result, the choice fell on Blender, despite this software involves problems, including the management of scenes in which there are very large bodies and very small bodies, as in our case where we have Earth, Sun very large compared to Space Rider and SROC. This problem prevented the realization of a simulation with the ICRF system coincident with that present within the scene of Blender but it was necessary to switch to a configuration in which Space Rider is positioned in the center of the scene, only position where it is rendered by the room. However, despite some difficulties, the software proved to be suitable and full of useful features to achieve the desired ideas with a few simple commands.

1.4 Possible application

Space simulators are a very important tool for the mission, especially when it involves the presence of the crew. Speaking of space simulators in a more general way, in fact, their main purpose is to train the crew, simulating, as accurately as possible, the conditions of the space environment (Ref. [6]). The work of this thesis is placed in the category of space flight simulator while simulating the docking of a spacecraft not equipped with crew. In fact, one of its possible applications, as well as providing

an immediate idea of what SROC will see and a simple tool to choose some design parameters (such as camera orientation), is to be used to make machine learning for neural networks in order to make it totally autonomous docking. The use of machine learning for space applications is not very developed, by Ref. [7] it is clear, in fact, that the current autonomous applications are: collision avoidance, detection of volcanic activity, or even autonomous driving capabilities of Martian exploration rovers. The increased use of machine learning in the mission would reduce its cost, increase its duration and return in terms of scientific data collected. The field in which the simulator of this work for the use of machine learning is placed is the analysis of images, so as to provide information to targeting and navigation tools in order to autonomously complete the docking. In addition to docking applications, this type of simulators, correlated with image analysis algorithms (Ref. [8]) simplify deep space navigation, limited by the constraints of communication with Earth stations, especially the time necessary for the exchange of information. Such systems, therefore, allow to make the system able to navigate to sight, making the operations totally autonomous.

1.5 Previous work

The work from which we started to analyze the steps to be taken in order to get to a simulator that performs the desired functions is the one reported in Ref. [9]. The aim of this work is to develop a simulator capable of generating images of a satellite in space with Blender, and then to extract and analyze the light curves of these images in order to obtain important information about the object observed, with the aim of developing a deep learning classifier of objects in space. The diagram of the simulator is shown in Figure 1.4 below.

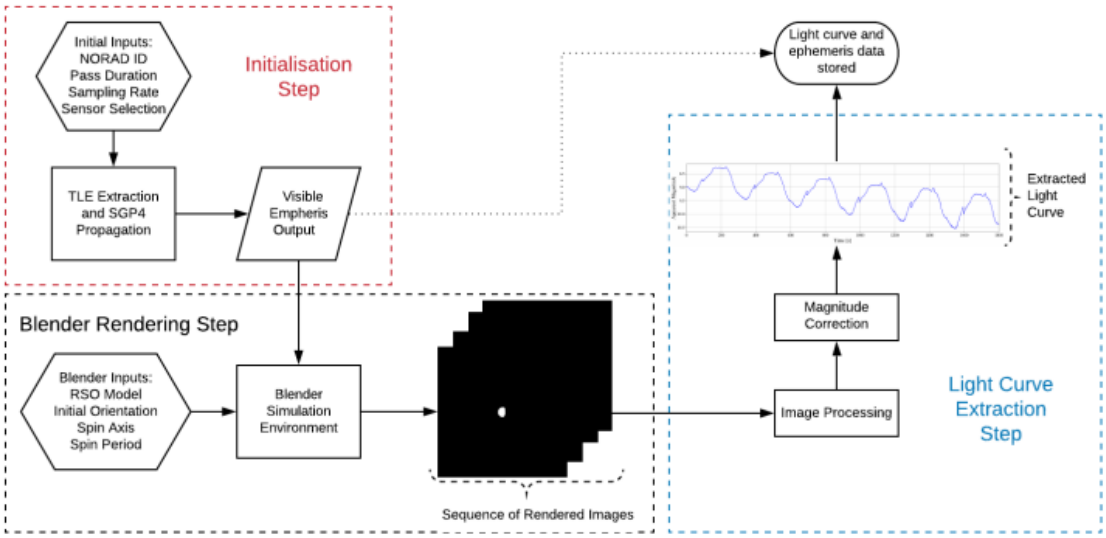

Fig. 1.4: Scheme of simulator in Ref. [9]

The first step of the simulator is to calculate the orbit of the body in question, once you get the ephemeris of the body, you move to the realization of mission geometry in Blender. In this phase, as in this work, models of the bodies present in the scene are inserted (in particular Sun, satellite and camera). Inserted the bodies are assigned the trajectories and the frames and, finally, is made the render of the images that will be, then, processed with the algorithm useful for calculating the curves of brightness of the images. An important information that can be seen from the following paper is the use of motion blur, which allows to simulate the correct visualization of a body, framed in the camera, when this has a spin motion around one of the axes. In fact, in the case of motion of the subject the collected image will be different according to the exposure time, to better understand this effect it is useful to look at Fig. 1.5, reported below.

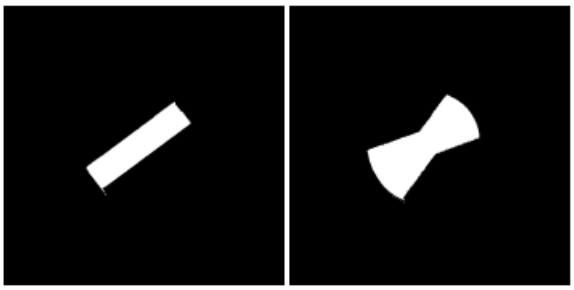

Fig. 1.5: Effect of motion blur in render, Ref. [9]

In Fig. 1.5 you notice exactly the difference between the image with motion blur and that one without, such difference will be reflected, inevitably, on the curves of brightness of the images. Observing the diagram in fig. 1.4 it is noticed that once the realization of the images in Blender is completed it passes to the analysis of these images, going to extract the light curves. One of the major difficulties of this step is to bind the luminous intensity values of the object in Blender to an accurate physical value of apparent magnitude, that is, the apparent luminosity of a body seen from Earth. When making the calibration of an instrument is used to compare the value of the magnitude acquired with that of a known star; while, in the case of a simulation, this can not be done, having no measure from which to start. The stratagem that was found to solve this problem, in the case of the work in ref. [9], is to calculate the fraction of solar intensity reflected by the body, taking into account that the magnitude of the sun is equal to -26.7 (note that the more the object is bright, the lower the magnitude is). Once the extraction of the light curves is completed, taking into account the distance with the reference object for the calculation of the magnitude and the scale adopted to realize the simulation (problem not present in the following job seen that the inserted bodies have the real dimensions), we pass to the analysis of the curves. The following is an example of curves drawn to understand the kind of considerations that can be made.

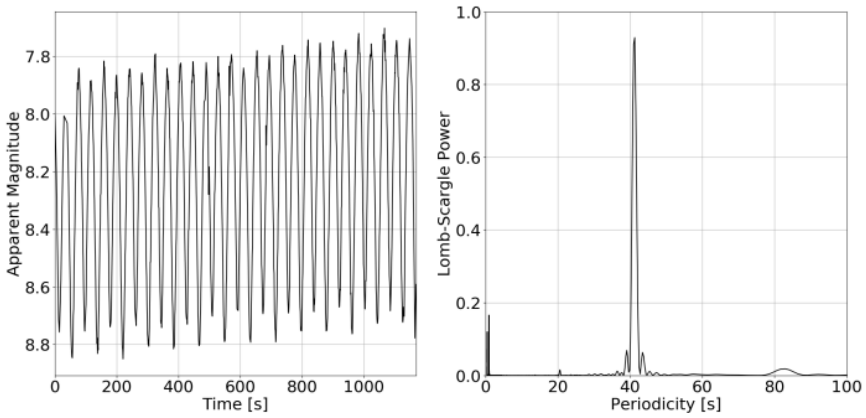

Fig. 1.6: Light curve of Falcon 9, on 7 July 2019, Ref. [9]

From the first curve we see a cyclical trend that testifies to the fact that the object is rotating, this consideration is confirmed by the analysis of Lomb-Scargle that shows a peak at about 40 s, which represents half rotation period (given the symmetry of the curve), which, therefore, globally will be equal to 82.6 s. This and the many other information obtained from the curves are very useful in the process of identification of a body by a neural network. The work of this thesis deals with the second part of the diagram in fig. 1.4, that is that of the realization of a simulation in Blender environment, and differs from the point of view of the inserted elements, since the Earth is inserted with its shader and the sun, both in the form of a lamp as in the work just illustrated, both as a sphere that is modeled in compositing (or after rendering) to achieve the lens flare. With regard to shader, it is worth mentioning another work related to the realization of materials in the simulation (Ref. [10]) in which we talk about how to make a physically accurate material through a BRDF model (bidirectional reflectance distribution function) the characteristics of which define the material. In particular, the realization of this model allows to model the reflected light according to two directions: the first is that of the light that strikes the object and the second that of the observer who looks at the object. In particular, the material is defined by a diffusive quantity and a specular that are set according to the same (for example, a metal with smooth surface will have a diffusive component equal to 0 and, therefore, only a specular component). In addition, in the model there are two other parameters that allow you to control the shape of the lobe (Fig. 1.7), this parameter coincides with the anisotropy parameter of the blender nodes (see ch. 3).

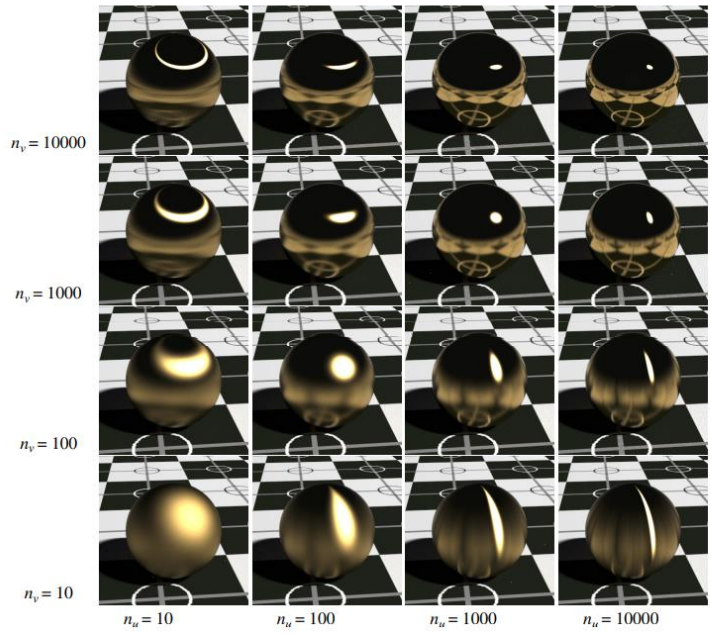

Fig. 1.7: Different metals with BRDF model, Ref. [10]

In the figure it is noted, as explained above, the difference in shape of the luminous lobe to vary n_u and n_v parameters. The metals in the picture are different and see their diffusive component increase to the detriment of the reflective one from top to bottom. If you want to know more about the calculation of the various contributions, see Ref. [10]. No calculation of the BDRF function has been made in the work in question, since the graphics engine in use in Cycles is very accurate from the point of view of calculating the light interaction between objects and how light reaches the observer. However you can find great similarities in setting the Earth shader with this paper, as the diffusion parameters, Reflection and anisotropy are all present in the *Principled BSDF* node of Blender under the names of *Subsurface* and *Subsurface Color* for diffusion, *Metallic* and *Specular* for reflexion and *Anisotropy* to regulate the shape of the lobe. To witness of the analogy they are succeeded to construct in Blender the spheres shown in fig. 1.7 simply changing the parameters of the node aforementioned. It is concluded, therefore, that Cycles independently performs calculations of this type without the need to implement an algorithm to calculate the BRDF function. The design of the materials in Blender definitely becomes more complicated when it comes to atmosphere. The majority of the works examined, in fact, generates a LUT (Look Up Table) that accurately calculates the effects of the scattering of the atmosphere point by point, however this has not been applied to such work since it makes greatly increase the memory (already high) to be used to simulate the mission. For completeness, two of the works examined concerning the realization of an accurate atmosphere are cited: Ref. [11], [12]. The effects that both works are concerned with developing are mainly related to the scattering of light through the atmosphere (see par. 3.3), the first, as said creates a LUT, the

second calculates the depth values in 3 parts: in front, behind and inside the atmosphere, values that are used, then, inside a code to make the pixel shader of the same. Even this solution is, however, heavy for calculation, since the Earth moves in the simulation and, therefore, these values should be calculated point by point. The realization of an accurate atmospheric shader was made, once again, using the Cycles graphics engine and the possibilities that it guarantees, as the node *Volume Scatter* that correctly simulates the phenomenon of scattering of light within the atmosphere, in addition to being adapted to have a fall in density with the height (to deepen the phenomena involved and the way they were modeled see par. 3.3).

2. Simulation

The software chosen to run the simulation is Blender. This open source software is used for 3D modeling and, among the various functions, allows you to realize an entire simulation, or in general a scene, by writing a command file in Python language (API). In the specific case of this work has been chosen to use this functionality in order to make the simulation adaptable to various trajectories and settings, so as to be able to verify, in various concepts of operations, the lighting conditions for docking. The software, however, has limitations. In fact, in the first phase of the work it was chosen to put the Earth in the origin of the axes of the space created by Blender, so as to use directly the coordinates of the various bodies in the ICRF system. Approach proved, then, unsuitable, since the high difference in scales, which will be discussed later, between the Earth and spacecraft (SROC and Space Rider) created difficulties for the graphics engine (Cycles) to render Space Rider. The problem was solved by inserting the center of the space of Blender Space Rider and, around, the bodies (Earth and SROC), thus creating a simulator that works in coordinates related to the Space Rider reference system. The following will analyze the various steps and the code created to realize the simulation. Note that the full code is given in the Appendix.

2.1 Libraries and useful functions

The first library that is imported is *bpy*, such library allows the use of Python API in Blender so as to access the various menus and objects created in the software. The second *os* is required to write a code that can always access the documents needed for the simulation, whatever the device from which it is executed (see the code below). The *datetime* library is used in calculating the simulation duration, from the files. csv of the coordinates, so as to determine the frames in which the scene changes, to have exact correspondence with the aforementioned files, and then with the mission. The other libraries are necessary for mathematical calculations to determine latitude and longitude and rotation matrices, useful for defining the relative motion between bodies. As for the functions defined: the first is used to determine the length of the files read, in order to assign a priori a length to the lists in which the quantities read will be stored; the second serves to switch from quaternions to Euler angles, with radiant outputs (roll, pitch and yaw). Note that the input of the *euler_from_quaternion* function requires first the vector part of the quaternion (q_v) and then the scalar part (q_0). The formulae used to define the latter function are given below.

$$t_0 = 2 * (q_0 + q_1 + q_2 + q_3)$$

$$t_1 = 1 - 2 * (q_1^2 + q_2^2)$$

$$\varphi = atan2(t_0, t_1) \quad (1)$$

$$t_2 = 2 * (q_0 q_2 - q_3 q_1), -1 < t_2 < 1$$

$$\vartheta = \text{asin}\,(t_2) \quad (2)$$

$$t_3 = 2 * (q_0 q_3 - q_1 q_2)$$

$$t_4 = 1 - 2 * (q_2^2 + q_3^2)$$

$$\psi = atan2(t_3, t_4) \quad (3)$$

Equation (1) is used to calculate the Roll angle from the quaternions, (2) for pitch angle and (3) for yaw angle.

2.2 Definition of file paths

```
current_path=os.getcwd()
my_path=current_path + '\\Resources2'
SROC_file_path=my_path+'\\Blender'+'\\SROC.obj'
SROC_in_SR_file_path=my_path+'\\Trajectories'+'\\SROCCenter_inSRbodyaxis.csv'
SROC_attitude_in_SR_file_path=my_path+'\\Trajectories'+'\\SROC_D31Bodyaxis_inSRbody
axis.csv'
SpaceRider_file_path=my_path+'\\Blender'+'\\Space_Rider.obj'
SpaceRider_attitude_in_ICRF_file_path=my_path+'\\Trajectories'+'\\SR_quaternions_wr
t_ICRF.csv'
SpaceRider_LIGHT_in_SR_file_path=my_path+'\\Trajectories'+'\\SRsunVector_inSRbodyax
is.csv'
SpaceRider_in_ICRF_file_path=my_path+'\\Trajectories'+'\\SR_coordinate_ICRF.csv'
Earth_in_SR_file_path=my_path+'\\Trajectories'+'\\Earth_position_inbodySR.csv'
Camera_data_input=my_path+'\\Camera parameters'+'\\Camera1.txt'
Name_file_output='Render Relative'
Earth_path=my_path+'\\Blender'+'\\Earth'+'\\Earth2.blend'
Sun_path=my_path+'\\Blender'+'\\Sun.blend'
```

In this part of code are defined, as explained by the title of the paragraph, the paths of the useful files within the simulation. These files are: the . obj files of Space Rider and SROC to import; the files .csv of trajectories, trim, and orientation of sunlight; and finally the file contains the parameters of the chamber mounted on SROC and Earth that was modeled within another file (.blend) to realize only once the shader of the same and of the atmosphere (Cap. 3) and to import the all inside of the simulation. Note that using the *os* library allows you to locate the workbook and move within it, defining the paths. This operation, as explained in the previous paragraph, allows the software to locate the files, whatever the device on which it is launched.

2.3 Setting simulation

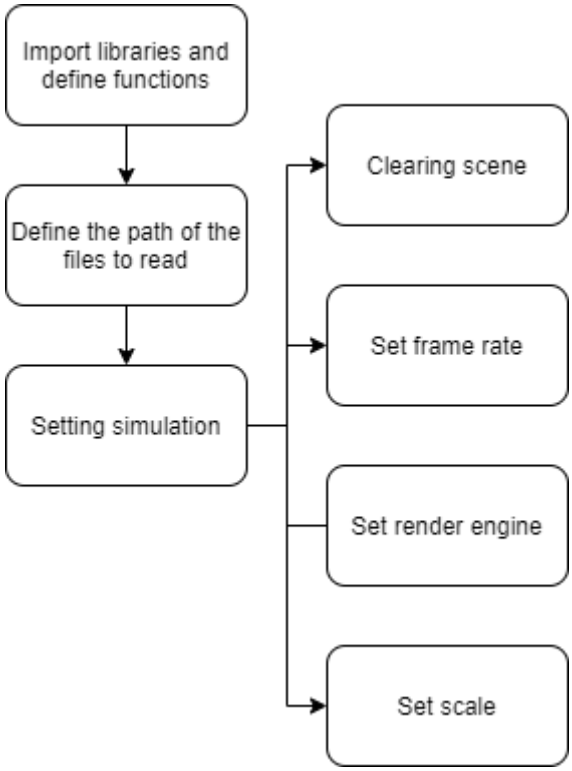

Fig. 2.1: Code scheme (setting simulation)

Before starting to import the various bodies within the simulation it is necessary to set some parameters of the same. In the lines of code that follow the definition of the paths of the files it is part erasing all the eventual elements contained within the scene, this in order to be able to relaunch the simulation without carrying behind residue of previous simulations. The next step is to set the frame rate, a useful quantity in defining the various frames of the simulation (in case the output is video). It ends by defining the render engine, which in this case is Cycles, and finally the scale parameter that, so defined, allows the user to modify it autonomously and once for the whole code.

2.4 Adding Earth

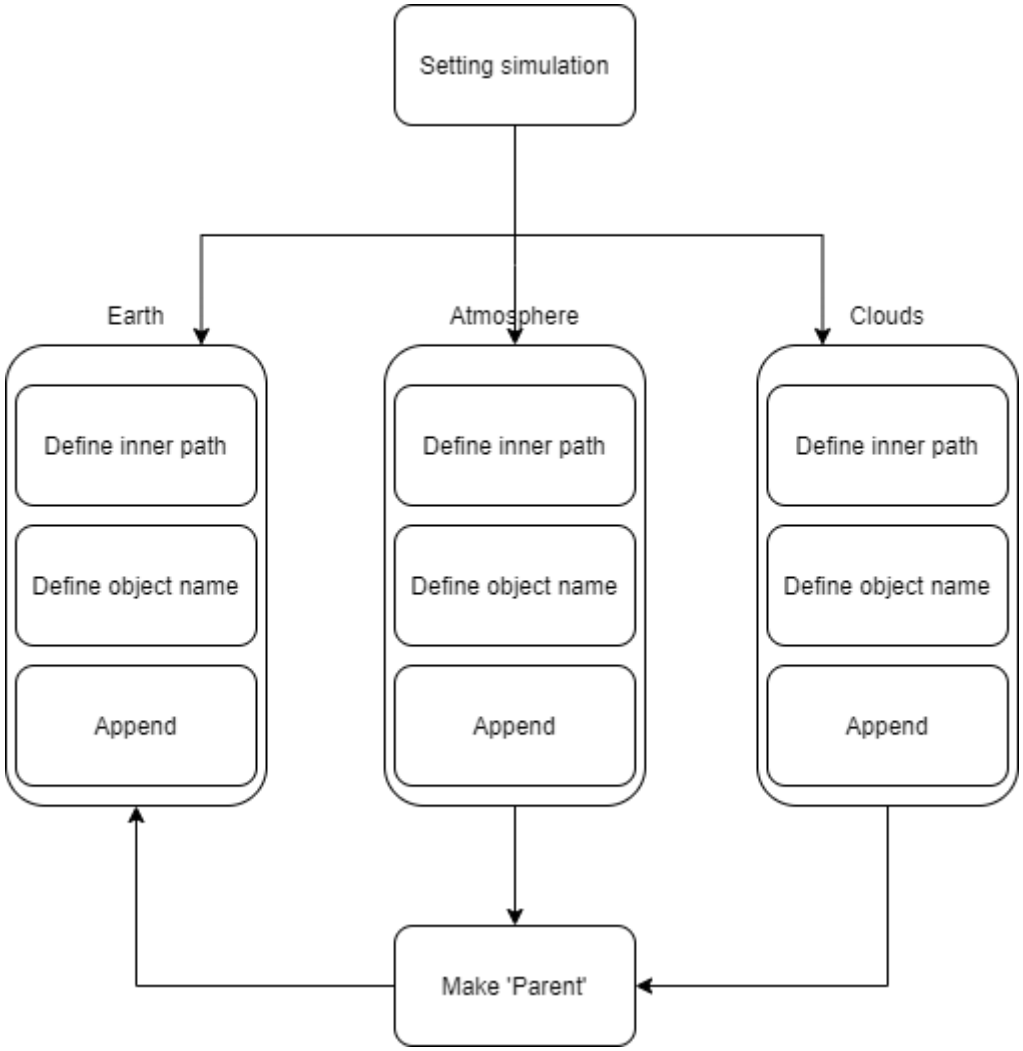

Fig. 2.2: Code scheme (import Earth)

The function, in the software, which allows you to transport elements (objects, textures, etc.) from a file. blend to another is the function ‘append' (in the code in appendix you can see how you use that function by programming it in Python). Since the Earth.blend file consists of 3 objects, that is three spheres representing the Earth, the atmosphere and the clouds, it is necessary to apply this function 3 times to the above file. To ensure the same motion to the three elements during the simulation it is necessary to establish a relationship between them. In particular, the relationship used is ‘parent’ which carries out, in fact, the task of moving an object with respect to a relative reference system (of the other object with which it is related). So, in our specific case, atmosphere and clouds will move relative to the Earth and, therefore, if the Earth moves they will move with it[1], generating the desired effect. Note that, by applying this relationship, it is necessary to make sure that the elements are

[1] The position (0.0.0) of atmosphere and clouds, once the ‘parent’ relationship is applied, is referred to the Earth, so if they remain in that position they will be fixed to the Earth

related to the Earth (that is, that the Earth remains at a higher level), since trajectories and rotations will be applied to the Earth.

2.5 Import Space Rider

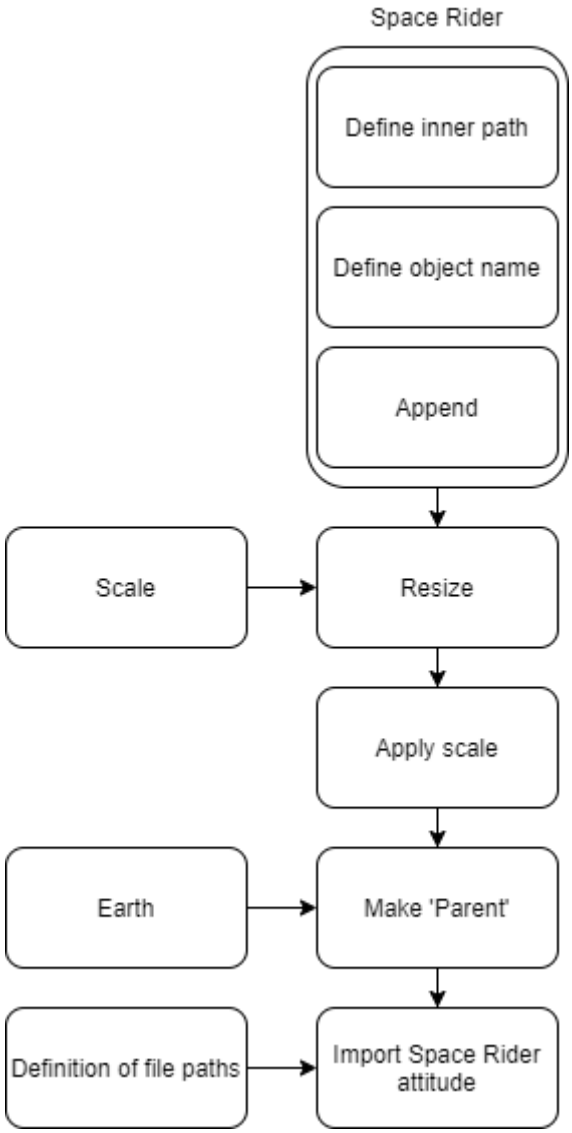

Fig. 2.3: Code scheme (import Space Rider)

In this section is reported the code scheme to import Space Rider. The first step, known the path of the file, defined in par. 2.2, import the. blend file into the Blender scene. Once this step is completed, it is necessary to resize the spacecraft with the aim of having it have realistic dimensions. In fact, as seen in tab. 2.2, once imported, Space Rider, does not have the correct size. Below you will find the actual dimensions of Space Rider, and the proportions necessary to obtain the scale parameters present in the code.

Real dimensions of Space Rider	
X-axis	4.5 m
Y-axis	1 m
Z-axis	1 m

Tab 2.1: Real Space Rider dimensions

In Table 1 and 2 the X axis is the one along the spacecraft fuselage, Y and Z are transverse to the spacecraft.

Imported dimensions of Space Rider	
X-axis	0.074915 m
Y-axis	0.024558 m
Z-axis	0.024558 m

Tab 2.2: Imported Space Rider dimensions

Before showing the proportions carried out to derive the values of scale it should be noted that the unit of measurement chosen within the software is in meters. This choice, however, is totally irrelevant from the point of view of the software, since it reasons in units, the possibility of modifying it is to facilitate the user in writing the position data, size, etc. In our case is kept the default unit (meters) but we think in Km, then 1 m, or a unit, corresponds to 1 Km. Made this premise we can go on to list the proportions:

$$X)\ 0.074915\ m{:}\ 1 = 0.0045\ m{:}\ x \quad \rightarrow x = 0.06006807715 \tag{4}$$

$$Y)\ 0.024558\ m{:}\ 1 = 0.001\ m{:}\ x \quad \rightarrow x = 0.04071992833 \tag{5}$$

$$Z)\ 0.024558\ m{:}\ 1 = 0.001\ m{:}\ x \quad \rightarrow x = 0.04071992833 \tag{6}$$

Note that all parameters of scale, in the code, are multiplied by 'scale', so that, as explained in par. 2.3, the user can modify this parameter only once for all elements of the simulation. Going back to the code, the next step is to apply the scale. This operation ensures that during the simulation the dimensions remain correct, and is preceded and followed by two lines of code that deselect all elements, to avoid applying changes to other components not involved. The next step is to make the Earth move relative to Space Rider, as the simulator reads relative coordinates. Once again, the 'parent' relationship is used. The last part of the code is used to import the Space Rider attitude in the various moments of time, which will be translated into frames in next par., contained in the file .csv. Note that the evolution of the attitude over time is in the form of quaternions.

2.6 Import SROC

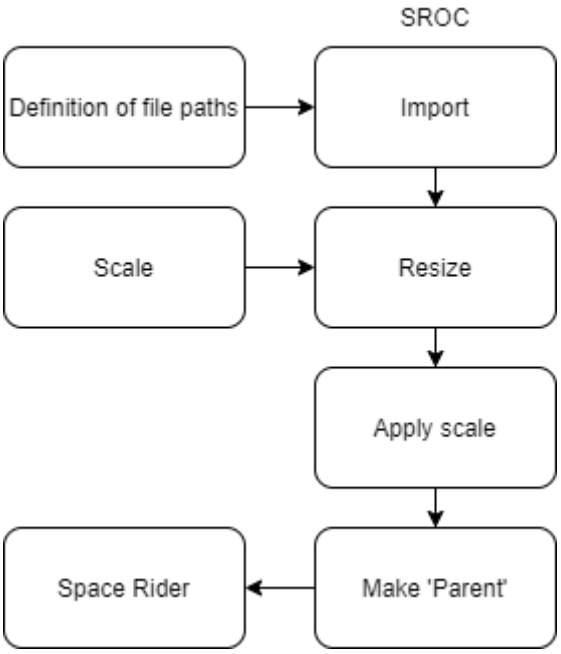

Fig. 2.4: Code scheme (import SROC)

This section contains most of the calculations needed for the simulation. In the first part the SROC model is imported inside the simulation, the steps are exactly the same as those made to import Space Rider, however, in this case, the function that have been used is import and not append (since the SROC shader does not matter since it is never framed). In this case the transformations to be applied to the object are two: the scale and a rotation. As far as the scale is concerned, the following tables show the actual and imported dimensions of SROC and, with the appropriate proportions, will show how the scale parameters present in the code have been derived.

Real SROC dimensions	
X-axis	0.1 m
Y-axis	0.1 m
Z-axis	0.2 m

Tab 2.3: Real SROC dimensions

Note that the dimensions of SROC are those defined by the CUBESAT standard for a two-unit satellite.

Imported SROC dimensions	
X-axis	2 m
Y-axis	2.5 m
Z-axis	4 m

Tab 2.4: Imported SROC dimensions

$$X)\ 2\ m{:}\ 1 = 0.0001\ m{:}\ x \quad \rightarrow x = 0.00005 \tag{7}$$

$$Y)\ 2.5\ m{:}\ 1 = 0.0001\ m{:}\ x \quad \rightarrow x = 0.00004 \tag{8}$$

$$Z)\ 4\ m{:}\ 1 = 0.0002\ m{:}\ x \quad \rightarrow x = 0.00005 \tag{9}$$

As for the reason for which the real quantities divided by 1000 appear, see par. 2.5. The second transformation to be applied to SROC is a rotation. In fact, as we see, in Figure 2.1 the axes are oriented so that Z is along the most extended satellite direction and X and Y belong to the square section.

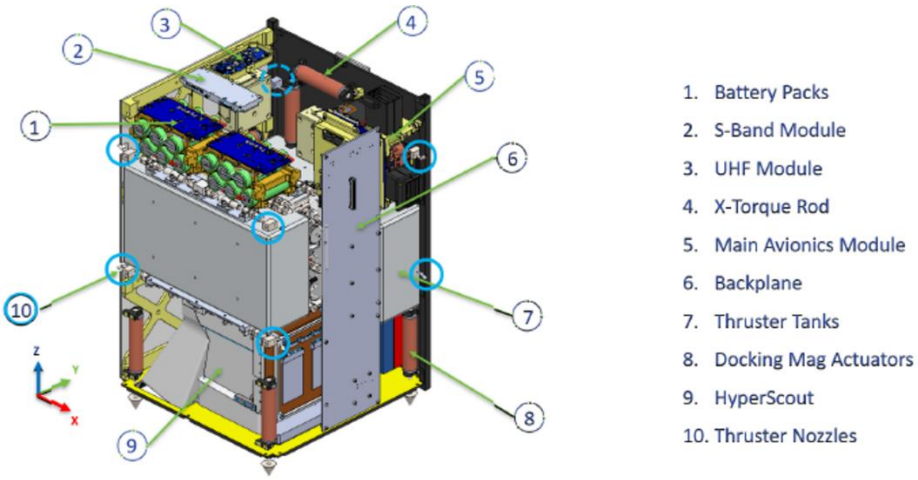

Fig. 2.5 = SROC body axis orientation

It is verified that, by importing the S/C in Blender, this one is oriented with the X_{body} and Y_{body} axes rotated of -90 , turns out, therefore, necessary, a rotation on Z (coinciding with Zbody) of 90°. This rotation is then applied to the object, so that, by rotating the same, following the data of the files .csv, maintain correct orientation during the simulation. Having already established that the simulator works in relative coordinates, it is necessary to relate 'parent' SROC to Space Rider, using the same lines of code seen in par. previous, in particular, in the last line, the wording *keep tranform = false* avoid deformations of objects caused by the introduction of this relationship.

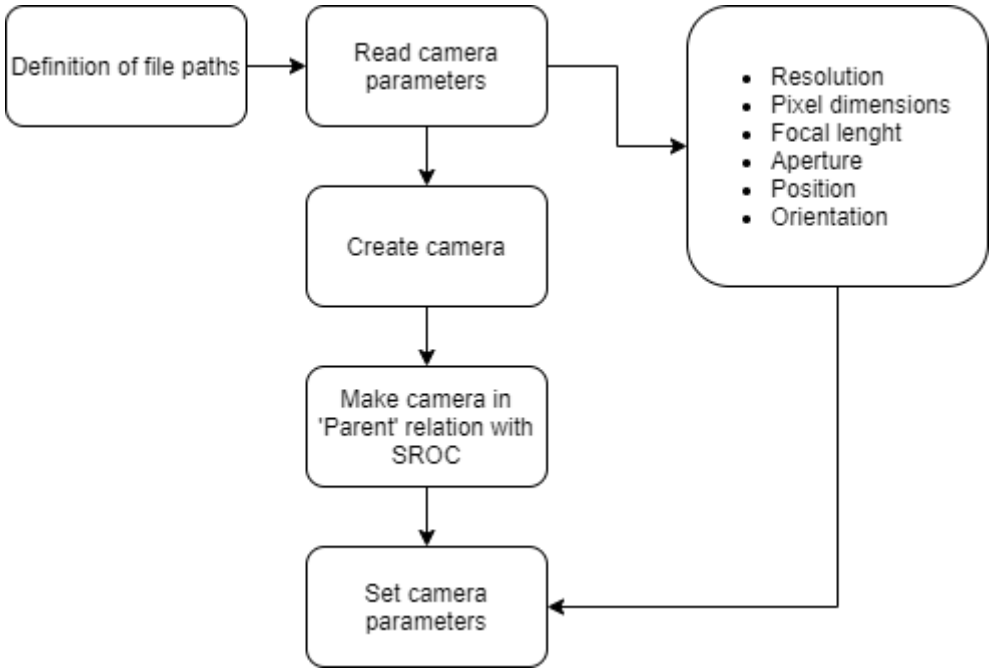

Fig. 2.6: Code scheme (create camera)

The following lines of code are used to read the parameters of the camera. This code function allows the user to independently choose which sensor to use during the simulation. Specifically, the parameters of the camera are: the resolution in x and y, the pixel size, the focal length, the opening, the position and the orientation, some of these will be used in the definition of the camera (next step), others in the definition of render parameters. An example of a file containing the optical sensor parameters is given in the appendix. We continue, in the analysis of the algorithm with the creation of the camera. The first phase is, in fact, to insert the camera (‘Camera SROC') inside the scene created in Blender, then the parameters are applied to it, in particular: the focal length, the position and the orientation. It is important to note that you do not have reliable data on the location of the camera in SROC, however, to create a meaningful simulation, it was chosen to place it on the lower face that, as you will see later, looks towards Space Rider, this is supported by the position of the chamber in Fig. 2.1. The last trick to define the camera is to set the distance of start and end clipping, this parameter does not find physical correspondence in the sensor, but it serves to indicate to the software from which and up to which distance sees the camera, has been chosen, then, to set them to 0.004 (or 0.004 mm) and 10^9 for start and end respectively. The chamber is mounted on SROC, so it is appropriate to make sure that it remains fixed to the same, using once again, for this purpose, a 'parent' type relationship.

Fig. 2.7: Code scheme (read SROC coordinates)

The next section, that one regarding the import of SROC and the relative files of trajectory and order contains, as it can be seen from the quantities initialized before the reading of the files, between the other things, the calculation of the frames and the duration of mission. You may notice, in the definition of the amount *n_points*, the use of the *file_len* function that, as explained in par. 2.1, serves to calculate the length of the file read. Later, it is noted that it is used in initializing the *frames* list, where *n_points* is reduced by one unit to eliminate the count of the first line that contains no useful information. Once the useful variables are initialized, read the SROC coordinate file. The first loop that appears in the code is a *while* loop, that loop is used to read the file line by line until the last one. In order to avoid reading the first line that, as mentioned before, does not contain information, an *if*

has been inserted in which you enter only when the quantity *i*, that is the index of the lines, is greater than 0. Entered in this *if* cycle there is the reading of the coordinates that happens in three phases: the first one previews the substitution of the characters ':' and ',' with a space, such passage concurs to approach the second phase, where the split string command is used to split the string whenever a space appears in the string. Finally, the third step transforms the values in position 6, 7, 8 into float-type numbers, multiplying them by the scale parameter. We notice the presence of a sign '-'. to the values of position in Y_{body} and Z_{body}, this because of the fact that importing Space Rider in Blender is assigned a body referral system with the Z axis facing upwards while, as shown in figure 2.8 (extracted from the video produced with the software STK), the tern of axes, with which the files were produced . csv, presents the Z axis downwards.

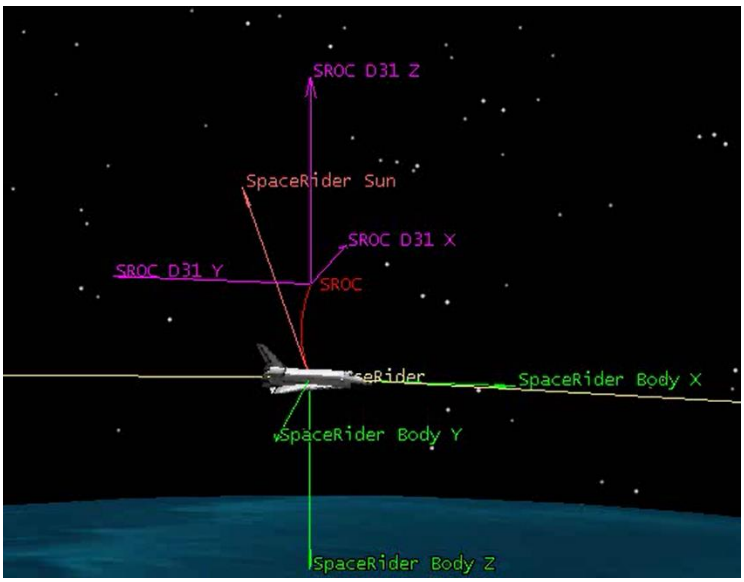

Fig. 2.8: SROC docking at Space Rider with body axis

In addition to reading the coordinates, it is necessary to locate the frames corresponding to the time moments indicated in the coordinate file. In this regard, the code is divided into three more *if*: the first to read the first useful line (*i*=1), the second to read the lines between the first and the last and the third to read the last (*i*=*n_points*). This subdivision was made to keep track of the first time value and the last in order to use the *datetime* library to calculate the duration of the mission. Inside of the first *if* there are 12 *if* in series that serve to convert the month in number, so like demanded from the library *datetime*, moreover, only in this case, it has another line of code that has the scope to count the days passed until the month contained in the first row, for the calculation of the angle α_{g_0}, that is the longitude at which the meridian of Greenwich is located, with respect to the direction pointing towards the constellation of Aries, at the initial instant of the coordinate file. This quantity is calculated using the formula:

$$\alpha_{g_0} = \alpha_{g_0}{}^{*} + 360.98564736692 \left[\frac{^\circ}{s}\right] * \Delta t \tag{10}$$

Where the quantity $\alpha_{g_0}{}^{*}$ is equal to 280.46061837°, ie the angle α_{g_0} at the1° January 2000, while the quantity Δt represents the days elapsed since that date.

$$\Delta t = (date1[2] - 2000) * 365 + delta_{mese} + date1[0] + 5 \tag{11}$$

The quantities appearing in equation (11) are:

- *date1[2]*[2]: the year read in the first line of the .csv file contain the SROC coordinates;
- *delta_mese*: the days elapsed in the year up to the month read in the first row (calculated in the month-to-number conversion *if* cycles);
- *date1[0]*: the day read in the first line of the file;
- *5*: leap days elapsed until the date of mission (2023);

The time unit of measure of the saved quantities for the simulation is the second, in this regard it should be emphasized that the *datetime* library does not allow to insert between its inputs seconds in float format. For this reason, in the code we see that all the values, before being passed to the function of the aforementioned library, are transformed into integers and, the tenths of a second, saved in an additional variable and, as we will see later, added at the end. The definition of quantities with the suffix '*_old*' is necessary for the calculation of frames, that is to set a calculation cycle keeping track of the previous quantity to subtract from the current one. The second *if* cycle, as mentioned above, reads the temporal information in the middle rows. After converting the month into a number, we move on to the date calculation (*date*), which is subtracted from the previous date (*date_old*) to calculate the time elapsed between the next two times in seconds, then, the calculated amount that will be used for the calculation of frames, which happens immediately after, and the angle α_g, is added to the *delta_i* list. The frames are calculated by adding to the previous value (*frames_old*, initialized before the 0) the new *delta_t* multiplied by the *frame_rate*, defined in par. 2.3. When the new frame is calculated, replace the '*old*' variables with the new ones to proceed to the next step. Once the file has been read, the useful quantities remain to be calculated, in particular it starts by changing with *datetime* the last date and, through the same library, it calculates the duration of the mission in seconds (*Simulation_time*). Known this data is possible to calculate the number of total frames of the simulation that must be indicated to the software. The calculation takes place by multiplying the *frame_rate* by the *Simulation_time*. Note that the elements of the *frames* vector do

[2] Note that the index 2 is due to the fact that the lists in Python count the elements starting from 0, it follows that the third element, that is the year as you see from the file in appendix, has the index 2.

not represent all the frames of the simulation, but only those corresponding to the time moments read in the file, that is those to which will be set the changes of position and attitude, as you will see in the following paragraphs.

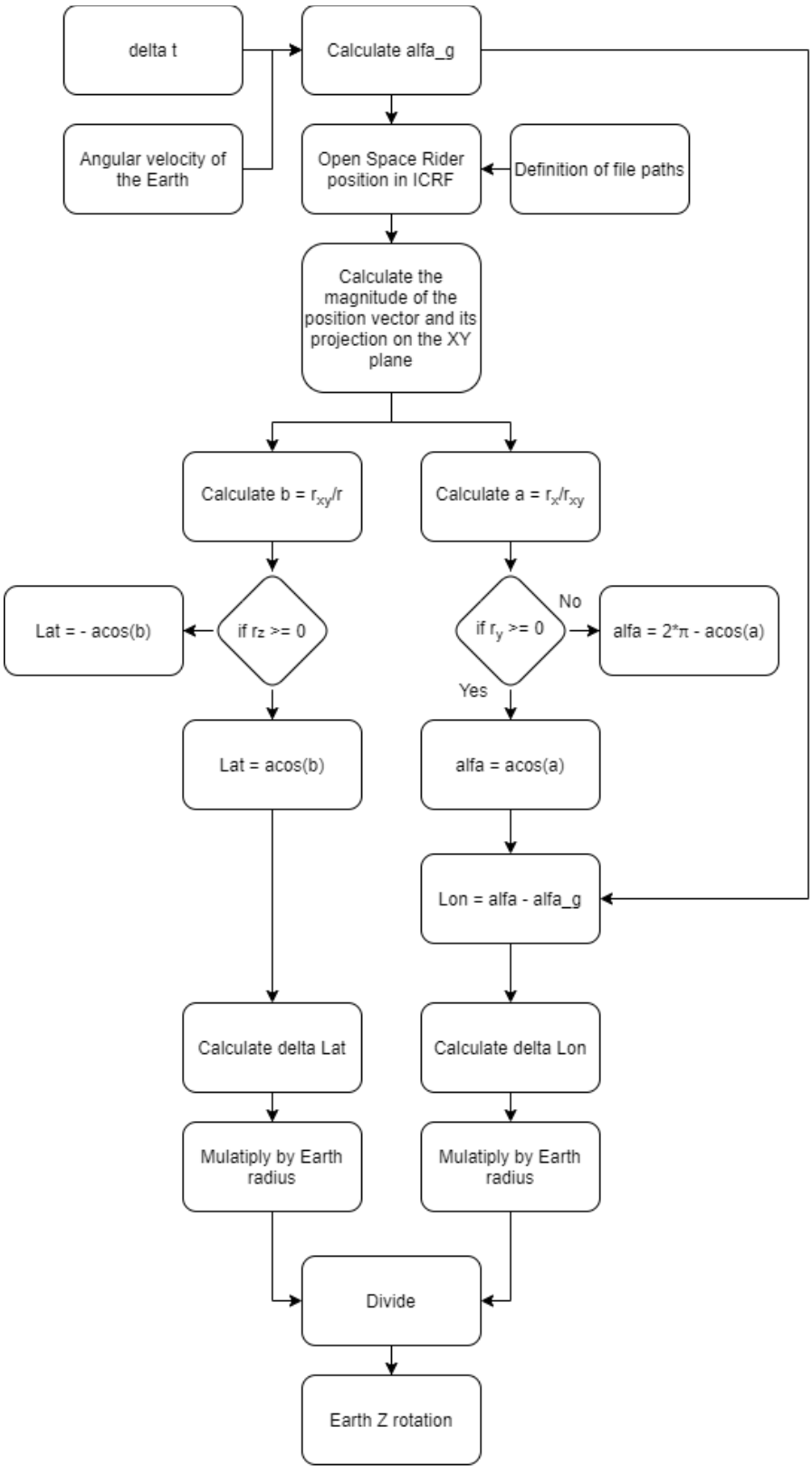

Fig. 2.9: Code scheme (calculate Lat and Lon)

Continuing to analyze the code we have the calculation of α_g which is determined by adding to the same amount, calculated at the initial instant (α_{g_0}), the Earth's angular velocity multiplied by the time elapsed in seconds, contained in the *delta_t* list.

$$\alpha_g = \alpha_{g_0} + \omega_t * \Delta t \tag{12}$$

Where ω_t is equal to 7.29*10^{-5} rad/s.

The motion of the Earth, however, is not the only one to consider to define the rotation of the same in the reference system of Space Rider, in particular you have to define the longitude and latitude of Space Rider during the simulation. To do this you should read the Space Rider coordinate file in the ICRF reference system and use the formulas below

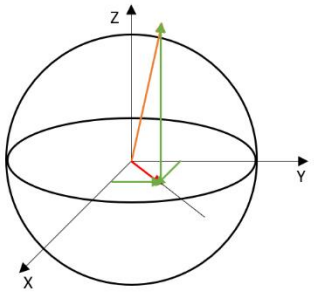

Fig. 2.10: Vectors to calculate Lat and Lon

In Figure 2.3 the orange vector represents the position of Space Rider at the i-th instant. This vector breaks down into 3 directions: X, Y, Z, represented by the green vectors. The red vector is the projection of the position on the XY plane. Having the coordinates on the three axes it is necessary, to calculate the latitude and longitude of Space Rider, to calculate the vector position and its projection on the XY plane:

$$\|\bar{r}\| = \sqrt{r_x{}^2 + r_y{}^2 + r_z{}^2} \tag{13}$$

$$r_{xy} = \sqrt{r_x{}^2 + r_y{}^2} \tag{14}$$

Known these quantities by determining the angle between the X-direction and the projection of the position on the XY plane, calculating the ratio between the position in X and the projection to XY (thus '*a*' in the code) if the ratio is positive or negative, within these cycles two further *if* cycles are distinguished: if the y-coordinate is positive or negative. In the first case, to obtain the angle, just apply the acos function to the ratio, while in the case where the y-coordinate is negative the value of the acos(a) function is subtracted to 2π. With regard to the calculation of latitude, however, it is

necessary to relate the value of the projected position on the XY plane to the module of the position (i.e. '*b*' in the code). Also in this case it is necessary to divide the calculation into two *if* cycles, distinguishing the case in which the coordinate z is positive and that in which it is negative. In both cases the calculation is done using the acos function, applied to the above mentioned ratio, so that when the z coordinate is positive, the latitude is the result of the application of the above mentioned function, while in the case where the coordinate z is negative the latitude is the result of the same function, but negative. For the calculation of longitude it is necessary, now, to subtract the rotation of the Earth α_g from the *alpha* values previously calculated, to have the angular position on the XY plane of Space Rider, with respect to the meridian of Greenwich. After this step you have two lines of print of longitude and latitude. The next step is to rotate the earth around the Z_{body} axis of Space rider so that it will be aligned to the trajectory, we know that the X axis remains tangent to the trajectory during the mission. To do this, by calculating the swept angles in latitude and longitude, we calculate the arcs of equator and meridian that the S/C traces on the Earth's surface, known such arcs it is possible to calculate how much the Earth must rotate on the Z_{body} axis through the application of the atan function to the ratio between the side on the XZ plane (*lato_Lat*) and that on the equatorial plane (*lato_Lon*).

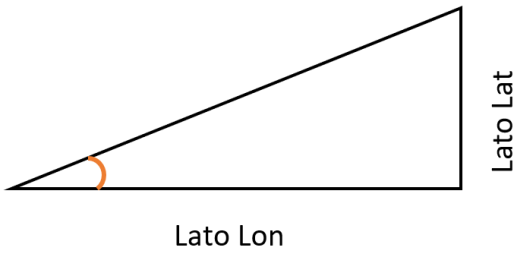

Fig. 2.11: Calculate Earth rotation angle on Z_{body} to align Space Rider to trajectory

The following section concludes with the application of the newly calculated rotation to the Earth.

2.7 Creation of light and Earth path import

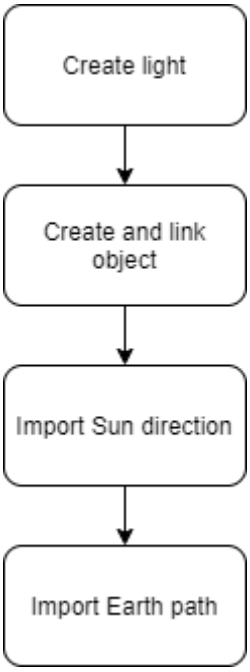

Fig. 2.12: Code scheme (create light)

The first 4 sections of this part of the code concern the creation of a Sun-type light source in the scene under construction in Blender. This type of light source does not require placement in the scene, but only the direction in which the sun's rays point, that direction will be imported into the next section of the code. The first section is used to set the data of the light to be created, the second to create an object type light with the data set previously; third and fourth sections, instead, serve, respectively, to connect the object to the scene and activate it. When you import the coordinates of the solar ray vectors you can see the presence of the sign '-' to the y and z coordinates, the reason is the same as the one explained above, that is the fact that the Space Rider body Y and Z axes, in Blender, are opposite to those used by the STK software to calculate these coordinates. The same consideration applies to the last section, which concerns the import of Earth coordinates, as to all related coordinate files that are imported. Finally, the Sun is created, dividing the elements into collections and view layers to ensure that the Sun does not illuminate the scene (function assigned to the lamp).

2.8 Create the motion of the elements

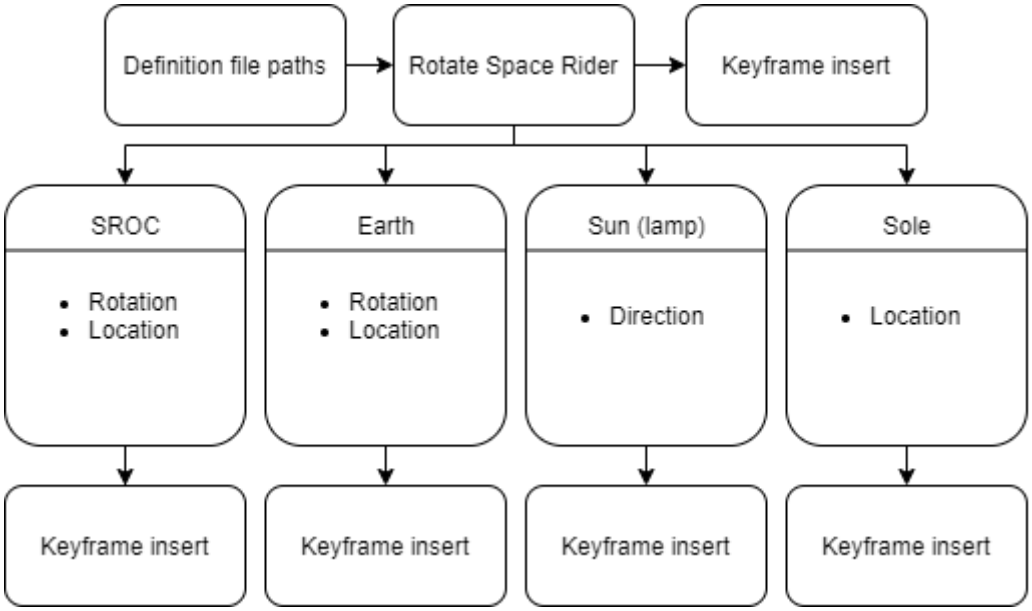

Fig. 2.13: Code scheme (create motion)

In this paragraph is explained the part of code that deals with moving the elements created and imported within the simulation. The first step is to set the rotation of Space Rider by following the previously imported setup file. Having put all the elements in relation to Space Rider allows, now, to make them move relatively to the same, then it becomes essential, first, rotate the above S/C. Assigned the 4 elements of the quaternion to Space Rider you calculate, through the *mathutils.Quaternion* function, the rotation matrix of the system, which will be useful later in defining the motion of the other elements. Once you set that specific rotation value, read from the files, it is assigned to its corresponding frame (scanned by the counter '*j*'). We understand, now, the importance of inserting a series of lines of code that calculate exactly the frame corresponding to the date in the file .csv. The first element after Space rider to be set in motion is SROC, by means of a command, the position of the cubesat is set, while a further measure of rotation is necessary. Up to now the problem of the inversion of the axes Y and Z of Space Rider has been evidenced during the import of the files of coordinates, however it is evident that also the rotations are affected, since they are also in relative coordinates. To overcome this problem it is difficult to work directly on the quaternions, while a much simpler solution is to convert the trim values into angles of Euler and insert a sign '-' on the pitch angles (Y) and yaw (Z). This solution requires the use of the *euler_from_quaternion* function, defined in par. 2.1. This solution is then applied to the attitude values imported for SROC and the *rotation_euler* command to rotate SROC is applied. The definition of the SROC motion ends by matching this rotation, and also the position, to its frame. The next step is the motion of the Earth. In this case the position has already been imported (par. 2.7) and the rotation must simulate the variation

of longitude and latitude of Space Rider around the Earth. To simulate the variation of longitude you rotate the Earth around Y_{body} of Space Rider, note that this axis is already contrary to what it should be and then the sign '-' that should have applied does not need to be inserted because of this inversion. Conversely, as far as latitude is concerned, it is necessary to rotate the Earth on X_{body}, but in this case the axis is not inverted with respect to the calculations, it follows the need to add a sign '-' in front of the latitude value. Here, too, the rotation and the position coincide with the corresponding frame. The next step is to move the Sun, in this case it is necessary to insert a command that is able to rotate the Sun in order to follow the direction saved in the *coords_Sun* list. The rotation that the Sun undergoes is calculated in quaternions and the command to be given is *direction.to_track_quat('Z','Y')*, a command that calculates the quaternion to be assigned to the body, in this case the Sun, to remain aligned to a certain indicated direction. Note that the *direction* quantity has been calculated previously by multiplying the Space Rider rotation matrix by the relative orientation vector of the Sun. This procedure is an alternative to introducing a 'parent' type raltionship between objects, since the product of rotation matrix by relative vector provides the same vector in the absolute reference system. At the end of the paragraph, once again, the calculated rotation of the Sun (lamp) and the sphere 'Sole' position are assigned to the corresponding frame.

2.9 Definition of render parameters

```
####### Render parameters #######

bpy.context.scene.render.resolution_x = resolution_x
bpy.context.scene.render.resolution_y = resolution_y

Output_path=os.getcwd()[:-10] + 'Results\\'
Output_path=Output_path + Name_file_output
bpy.context.scene.render.filepath = Output_path #Path of output file
bpy.context.scene.render.image_settings.file_format = 'AVI_JPEG'
```

The render parameters are defined in this paragraph. Note that the first set parameters (resolution in x and y) were imported from the optical sensor file, as was already anticipated in the relevant paragraph. The next three lines have the task to create and set the path to save the output file, note that the string '*name_file_output*' has been defined in par. 2.2. Finally, in the last row, select the output file format, in this case 'AVI jpeg'.

There are two versions of this part of code, one for a video output and another that allows you to render in the form of a file . png frame by frame, with the possibility of having a code that, unlike the one illustrated, generates a frame for each line of the file .csv.

2.10 Camera constraint

In the latter section of the code a constraint is imposed on the camera. The need for this constraint arises from the fact that there is no exact information on the position and orientation of the camera mounted on the SROC. The following constraint, of type ‘Track-To’, then, it takes care to keep the camera pointed towards Space Rider during the entire duration of the simulation, in order to have images that always point to the target vehicle of the docking.

3 Earth shading and Sun compositing

This chapter describes the process of defining the Earth.blendfile. This file contains the Earth, complete with atmosphere and clouds. It was chosen to create the Earth in a separate file given the complexity of that activity when one has to take into account the physical hypotheses of the phenomena involved. Also, using this approach lightens the computational load that you have every time you run the simulation. In fact, in the event that the Earth is created with a script, with OSL (Open Shading Language), supported by Cycles, for shading, the computational load to generate each simulation would be much higher. The choice to realize the planet only once and to copy it in every created simulation, therefore, brings with itself many advantages. Inside the scene in Blender there are 5 components: three spheres , a light type Sun and a camera to test the quality and photorealism of the images produced (fig. 3.1, 3.2, 3.3). The three spheres make up the Earth, the clouds and the atmosphere respectively. As for the Earth, this has been realized, like the whole file, of the real dimensions: r = 6371 Km (also in this case without changing the unit of measure of the software in order to avoid the problems evidenced in the previous chapter). Within the Earth's shader, three textures were used, one for the Earth's crust, one for the sea and one for the topography to realize the extruded effect of the mountains, and in general of the reliefs, on the Earth's surface. As for the layer of clouds, generated with the second sphere, this is positioned at a height of 2 km from the earth's crust, according to the Ref. [13], or the sphere has a radius of 6373 Km. Inside the shader of the clouds you have two equal textures, but with different names to overcome an inconvenience that you have in Blender, which will be discussed later. These textures are a topography of the clouds, one serves to give color to the clouds, the other to realize the volumetric effect, in the same way as it is realized for the Earth's surface. Finally, as for the atmosphere, in this case the shader is volumetric, designed to reduce the density from the Earth's surface up to the line of Kármán (100 Km from the Earth's surface), conventional limit of end of atmosphere and start from outer space (Ref. [14]). The volume is a 'scatter' to simulate the effect of Rayleigh's scattering. As for color, the calculation leading to the definition of RGB channels for the atmosphere will be explained later in the relevant paragraph. The other two elements, light and camera, as mentioned, are useful to produce only renders of the Earth and thus verify the photometric accuracy of the images. Below, before going into the details of the various spheres and how they were built, we report Earth renders of the Earth file Earth.blend and real images of the Earth seen from space (fig. 3.4) in order to make a comparison with those produced.

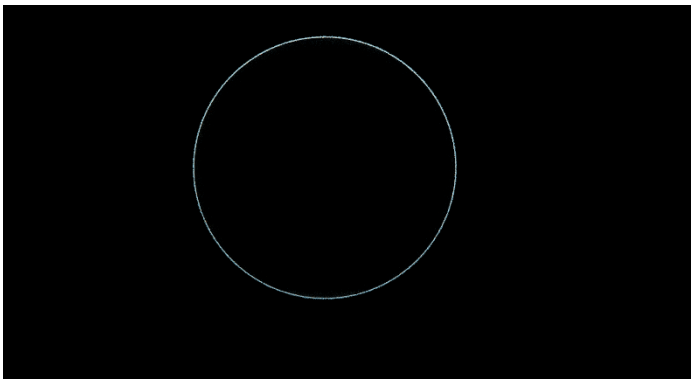

Fig. 3.1: Earth and atmosphere seen from behind

Fig. 3.2: Earth and atmosphere with evidence of red sunset highlights

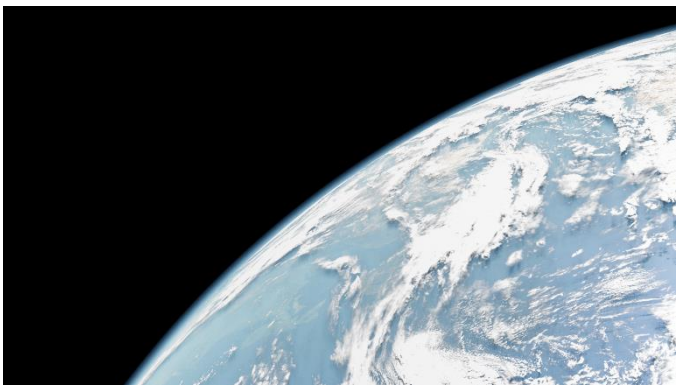

Fig. 3.3: Earth and atmosphere seen up close

Fig. 3.4: Real Earth and atmosphere image, Ref. [15]

From the renders of the Earth.blend file, compared with a real image of the atmosphere it's possible to see how the effect of atmospheric fall off, caused by the reduction of density with altitude, is well represented in the model created. Even the color of the atmosphere is very similar and it's also also simulated the 'corona' effect when you look at the Earth placed exactly between the observer and the Sun. An ulterior verification of the photometric accuracy of the model is had looking the fig. 3.2, from such image, is, in fact, possible to detect the red color of the atmosphere, and therefore of the sky seen from the Earth, to the border with the zone in shadow (night), which simulates the red color of the sky at surise and sunset, a phenomenon due to the Rayleigh Scattering which will be discussed in the paragraph related to the atmosphere. Before proceeding with the detailed explanation of how the shaders of the elements have been designed it is appropriate to understand what is the function that they perform. The shader defines how light interacts with the surface or how it propagates within the volume, in case of *Surface* or *Volume* output, as well as defining the color of the surface. There are different types of shaders in the software: BSDF, which deals with modeling reflection, refraction and light absorption when it meets a surface; Volume, which simulate the passage of light through a volume (used for the atmosphere); Emission, whose purpose is to describe the emission of light of a surface or a volume; Background, which deal with the emission of light from the environment. Such shaders can be combined through specific nodes (which will be seen in the following par. above), so that the result of this combination can be used by the graphic engine to calculate the interaction of light with surfaces and volumes.

3.1 Earth

In this section we will analyze the steps taken to achieve the Earth as a surface of the Earth, that is without clouds and atmosphere, which will be treated separately in the following paragraphs. Referring to the introduction of this chapter, this paragraph deals with the realization of the first sphere of the Earth.blend file. To start you need to create a UV sphere and apply a scale of 6371 in all three directions, so as to make the real size from the beginning. The next step consists in the subdivision of this sphere into several elements through the surface modifier '*Subdivision Surface*' , in which two parameters are set: *Viewport* and *Quality* respectively equal to 6 and 5. To conclude the realization of the geometry the last step is to set the visualization of the sphere as Shade smooth, so as to make the visualized surface smooth. It should be emphasized that, among the hypotheses made to create the Earth, there is that of perfect sphericity of the same. Once the geometry has been realized you move on to make the shader of the Earth. To create the shader of an object means to create the material of the same one, in such sense the software Blender previews already of default some effects,

as an example glossy, glass, transparent, diffused, etc. In the specific case of the Earth is brought back, in the Fig. 3.5, the realized shader, that will be analyzed below.

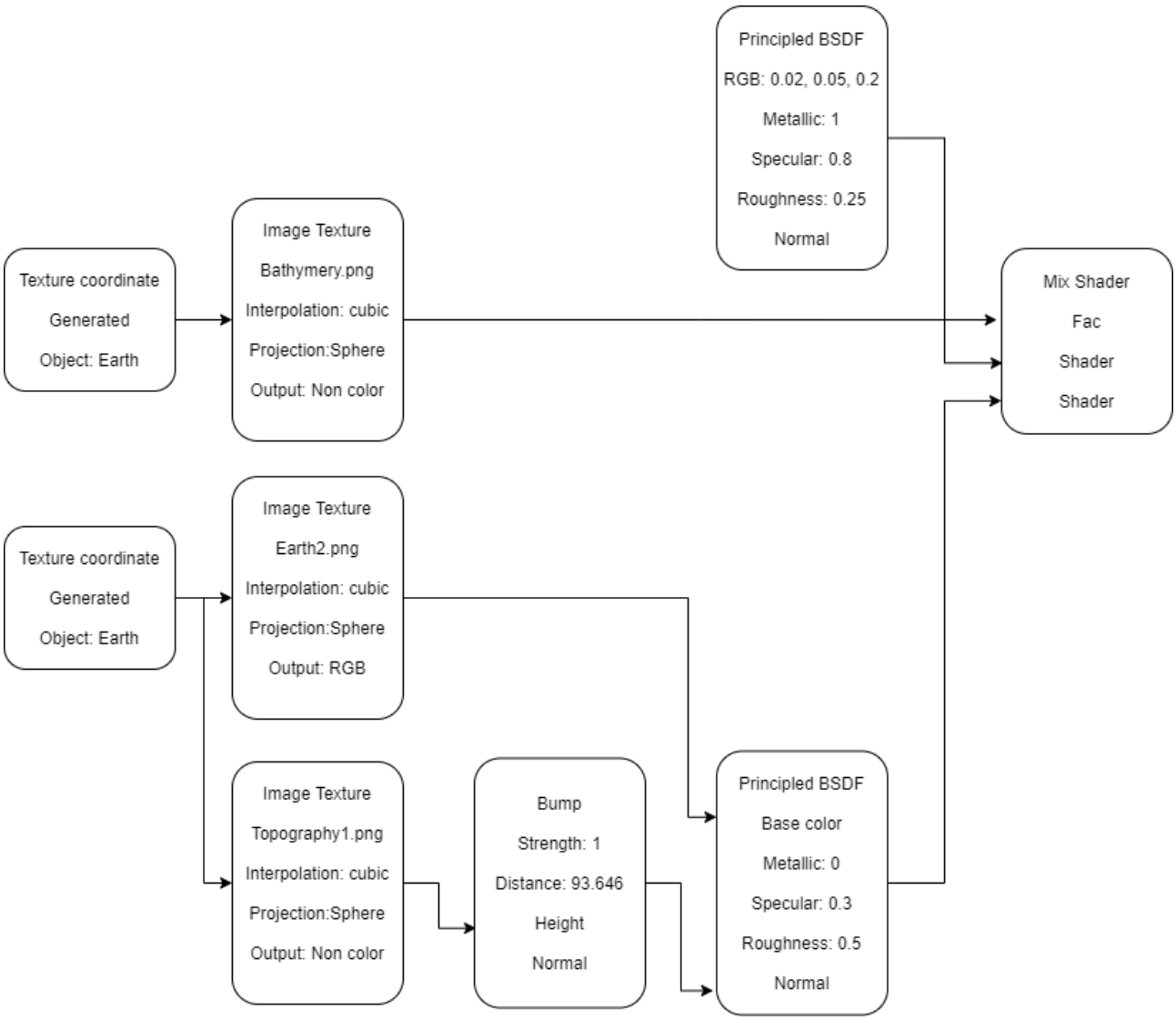

Fig. 3.5: Earth shader

To understand the logic, functions and relationships between the various blocks of the shader it is appropriate to analyze it from left to right and divide the top from the bottom. Starting from the lower part of fig. 3.5, the first block that appears is the *Texture Coordinate*, this block serves to correctly map the body on which the texture will be applied, in this case the Earth (*Earth*), in order to apply the same in the correct way. This step, in fact, could be avoided by applying the texture as *flat* on the sphere, but the result would be unsatisfactory especially in the areas of the poles, where the image 'wrapped on the surface' in this case, should be adapted to a very narrow space relative to the equator, which is why the result obtained in those areas would not be accurate at all. Once inserted the *Texture*

Coordinate block, we proceed with the insertion of a new *Image Texture* block, whose function is, in fact, to generate a texture from an imported image. Observing the fig. 3.5 we realize that such block presents of the parameters to set, in particular, the first one, regards the type of interpolation, in our specific case it has been chosen to change the type of interpolation of default, that is that linear one, in cubic, for greater precision in texture application. The second parameter concerns the type of projection of the texture on the body, this parameter has been set to *Sphere*, a choice necessary having inserted as an input vector in the block the mapping of the sphere through the *Texture Coordinate* block. To return to the previous example, in which it was assumed to apply the texture as *flat*, this would be the parameter to set, without the need to map the sphere, confirming the low accuracy of the results. Another parameter that has been modified to create the shader is the *Color Space*, this parameter indicates the type of output generated by the block, in particular it can be set to '*RGB*' to have an output in RGB channels, typically used when the output has to be connected to an input of type '*Color*'; or it can be set as '*Non Color*' in topography block, in the case where the texture is a topography or a map, this choice is typically made when the output of the block goes in *Bump* block or in an input of type '*Fac*' of a *Mixer Shader*. Remaining on the bottom of the shader you notice a further subdivision, in particular this subdivision is highlighted through the presence of two blocks *Image Texture*, the upper one containing the texture of the Earth (downloaded from the NASA site, Ref. [16], Fig. 3.6) and the lower one containing the topography of the Earth (Ref. [16], Fig. 3.7). It is noted, as previously mentioned, that the *Color Space* parameter is set to '*RGB*' in the Earth's texture block and '*Non color'* in the topography block, being, in fact, this, a map. It should be noted at this point that in the folder containing the Earth.blend file there are two subfolders containing textures, one with lower resolution 21600 x 10800 pixels, and one with higher resolution containing the texture of the Earth at 86400 x 43200 pixels and the other (topography, bathymetry and clouds) at 43200 x 21600 pixels. The higher resolution textures were created by combining the various pieces of textures, downloaded from the site in Ref. [16], with the Photoshop software. It should be noted that the Fig. 3.1, 3.2, 3.3, 3.6, 3.7 and all the figures containing renders of the Earth have been realized with the textures at lower resolution because the times to render, already remarkable, are acceptable regarding the use of the textures at higher resolution, which is why the lower resolution textures are also maintained in the file folder.

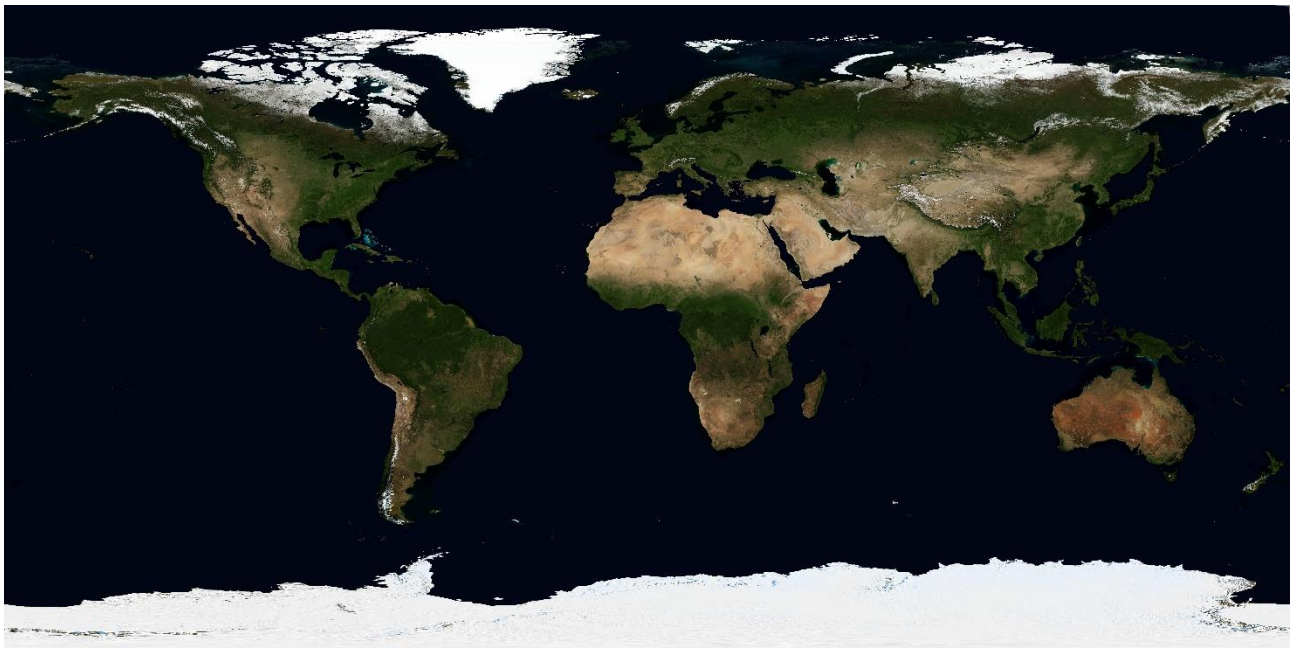

Fig. 3.6: Earth texture

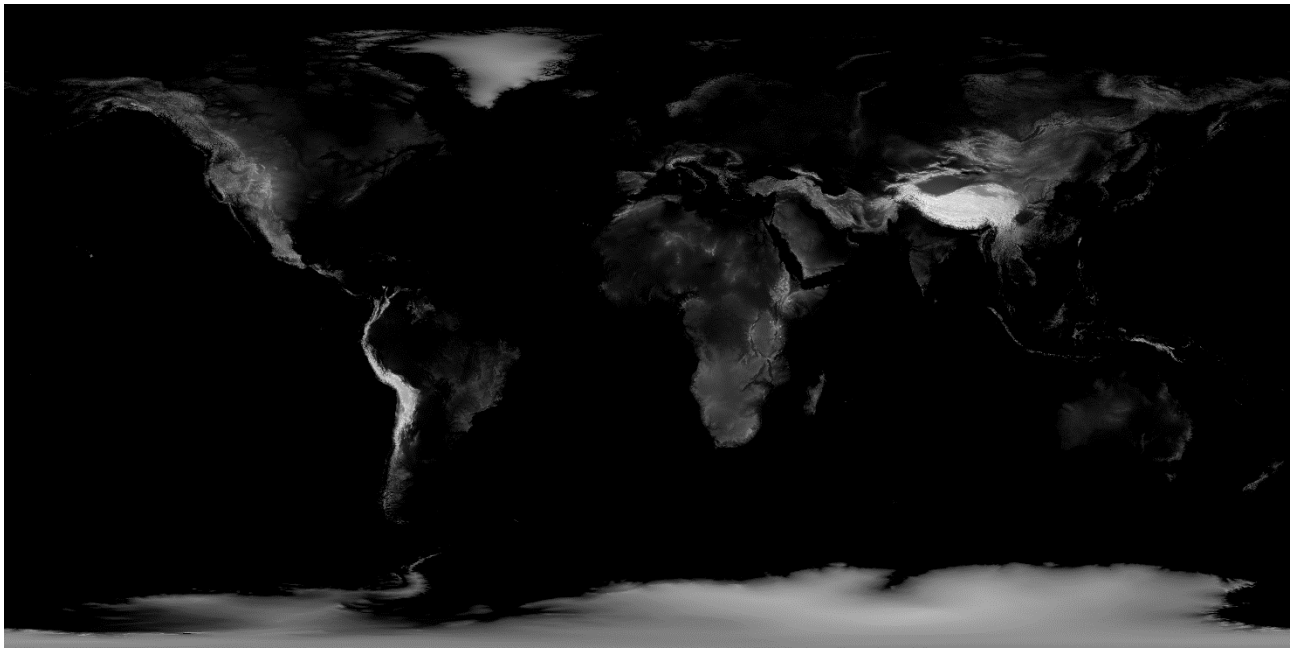

Fig. 3.7: Earth topography texture

Continue the analysis of the shader looking at the lower part, in this case, moving to the left, you notice the presence of a block type *Bump*. This block is inserted, in fact, downstream of the texture with the topography of the Earth. Its task, in fact, is to change the surface of the Earth, shaping its mountain ranges and, more generally, the reliefs. From that we understand the need to introduce a topography texture that does not have an RGB output, which enters within the *Bump* block's *Height* input slot. Before describing how to manage the output of this block it is advisable to describe the

parameters of this block. The *Strenght* parameter is used to set the intensity with which the block is modeling the surface, while the main parameter, for the physical accuracy of the model, is the *Distance* parameter. This parameter makes it possible to adjust the height to which the measurements are 'extruded'. In particular, in our case, a verification has been made in the area of the Calabria region, Italy, for the ease of acquisition of images in the correct position (or cut to measure the height of the reliefs), the figures describing this verification and the proportion carried out to calculate the value of the parameter are given below. Before illustrating, in Fig. 3.8 and 3.9, the height verification, the actual data used for this verification should be reported. In the aforementioned region the altitude varies from 0 Km to 2267 Km, in the highest point, the choice, to set the parameter *Distance* was to use an average value between the two, or 1100 Km (in the simulation 1.1 m for the reasons already explained in the previous chapter).

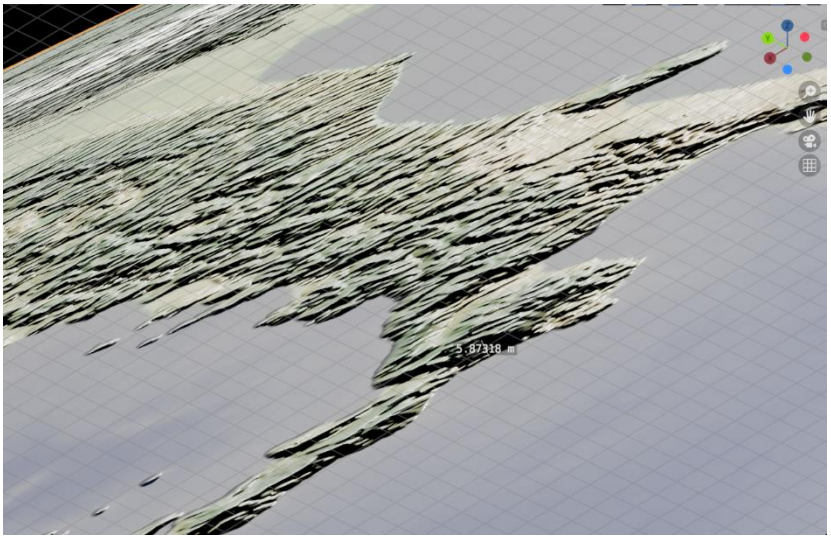

Fig. 3.8: Altitude of mountains with 'Distance' parameter = 500

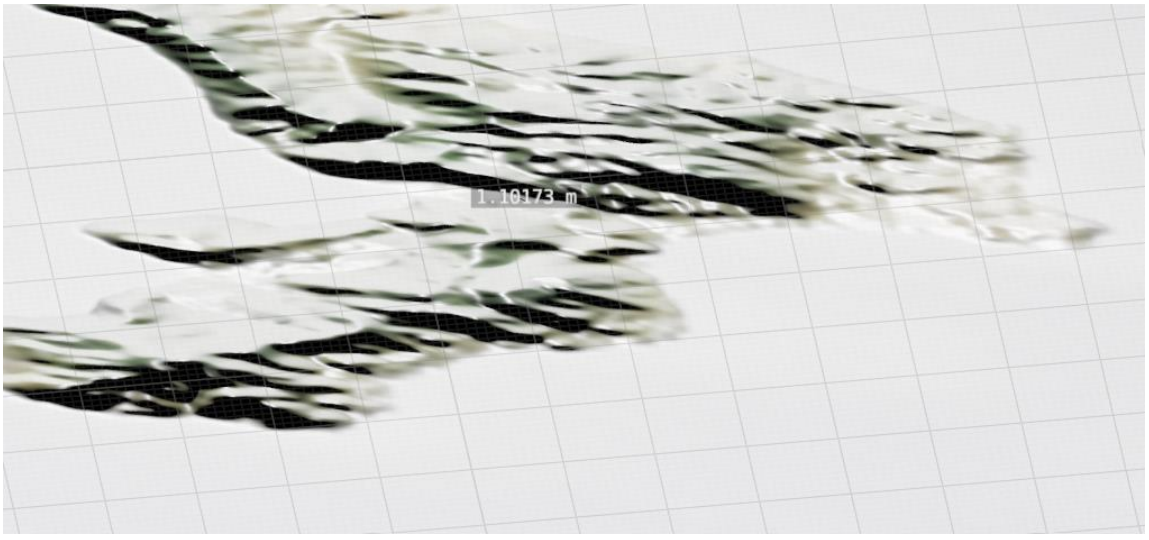

Fig. 3.9: Altitude of mountains with 'Distance' parameter = 93.64603

Remaining in the lower part of the diagram and moving to the right you can notice a block of type *Principled BSDF*, this block encloses all the various blocks type glossy, glass, diffuse, etc. In particular this block allows to realize all the desired effects. To understand its use within the model of the Earth you start by illustrating the two inputs that come from the external blocks: the first, or *Color*, is connected to the block that deals with importing the texture of the Earth, logical process since the intended effect is that such an image is projected onto the Earth; the second, at the bottom, is the *Normal* vector which, in fact, allows the application of the modification of the *Bump* block, In fact the latter maps the various heights in the points without changing anything, insert the output of this block in *Principled BSDF* allows the application of these changes. Before proceeding with the description of the parameters contained in the block it is appropriate to point out, as will be explained when talking about the *Mixer Shader* node, that the choices are referred only to the emerged part of the Earth, for seas and oceans, in fact, you choose a different path. The first parameter that appears within the block is the value of the *Subsurface*, in this case, place zero. This parameter is set to values other than zero when the surface emits with colors different from that of the *Color* parameter, set previously. The second parameter is *Metallic*, which, as the name suggests, defines a metal surface, in this case, placed equal to 0. The third parameter modified to model the Earth is *Specular*, that is, how much the Earth, understood as surface emerged, reflects the light. The setting of this parameter, as well as the next one, *Roughness*, were done with tests until an accurate result was achieved, having regard to the absence of material that allows to relate the parameter contained in the *Principled BSDF* of Blender with physical parameters, which, however, is taken into account in the realization of the atmosphere. Continuing to move to the right in Fig. 3.5 you find the *Mixer Shader* block, which, however, is connected to the blocks located at the top of the diagram. So, to fully understand the functionality of this block, you should first analyze the top of the diagram. Starting from the left you find the same blocks as before, then: the *Texture Coordinate* block to map the surface, the *Image Texture* block to import a texture (Fig. 3.9), which in this case is a Bathymetry. The need to introduce this type of texture comes from the fact that, following the Ref. [17], the water-covered part of the earth's surface has three effects on light: reflection, scattering for photons crossing the surface layers of water and then succeeding and attenuation due to the fact that some photons reach deep layers of water without being re-emerged. This consideration implies the need to divide the shading of the emerged part from that covered by water. In this sense the presence of a Bathymetry type texture is fundamental since it allows, once inserted as an input in the Shader, to divide the two parts, it is reported in Figure 3.10 and 3.11 the application of the texture only on the emerged part, to verify the correct functioning of the shader. In order to generate this verification image a .blend file is used made with a sphere of radius 100 Km, to accelerate the render times, without affecting the

applicability of the considerations, since the blocks work in the same way whatever the size of the sphere.

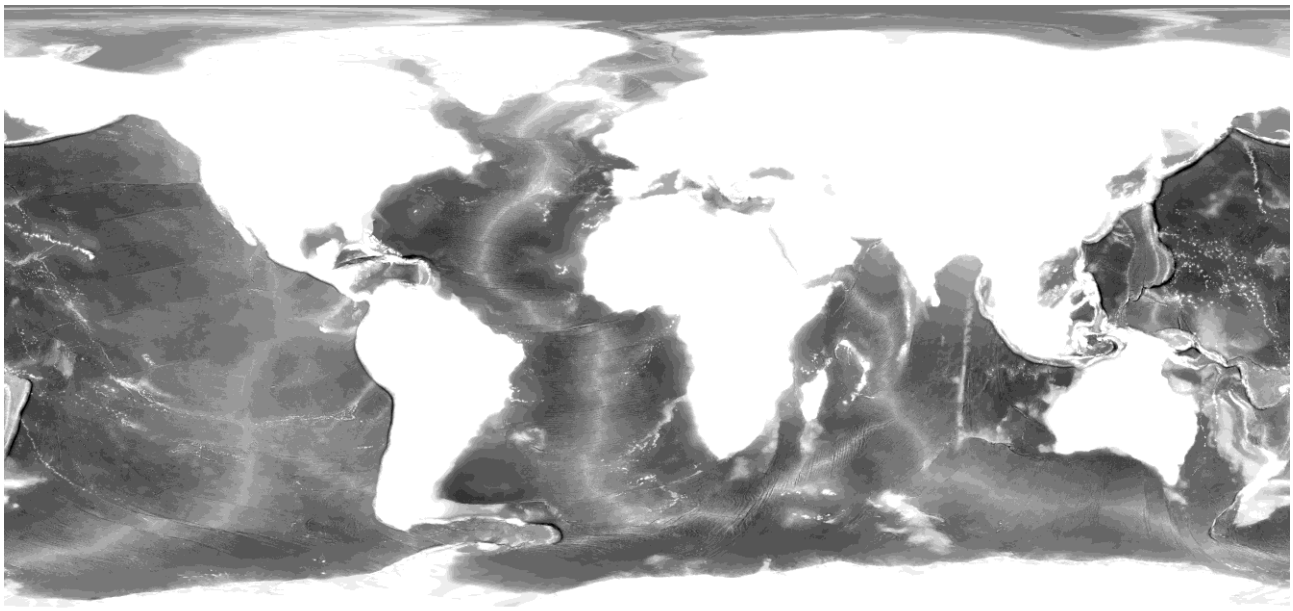

Fig. 3.10: Bathymetry texture

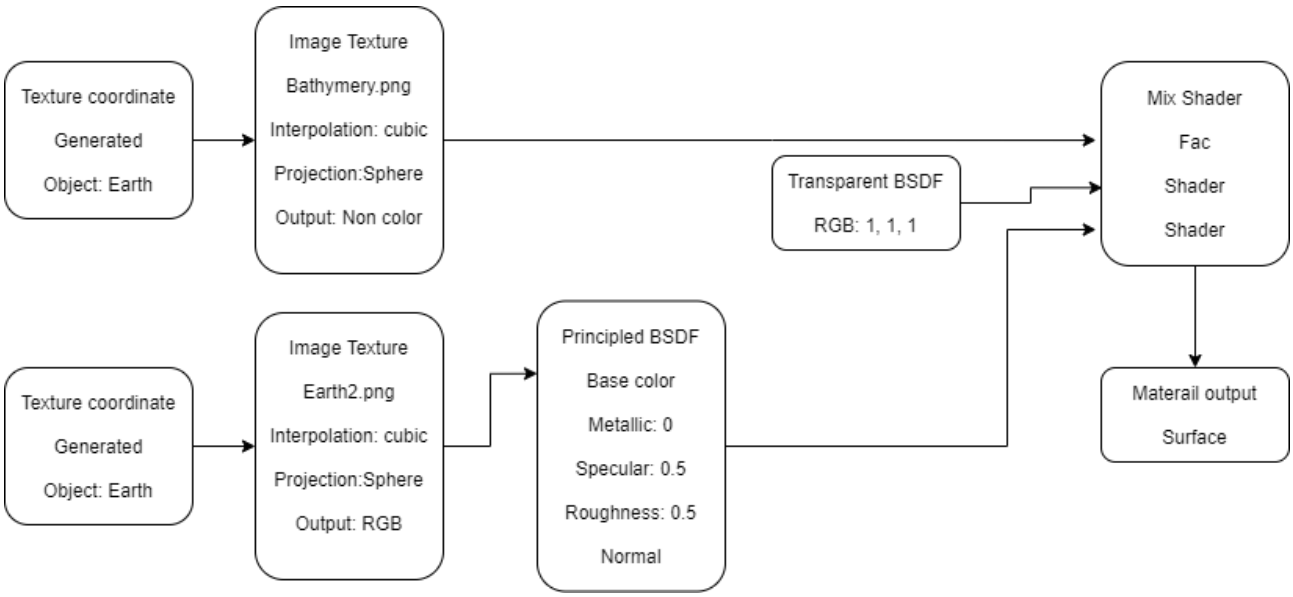

Fig. 3.11: Shader for verification model

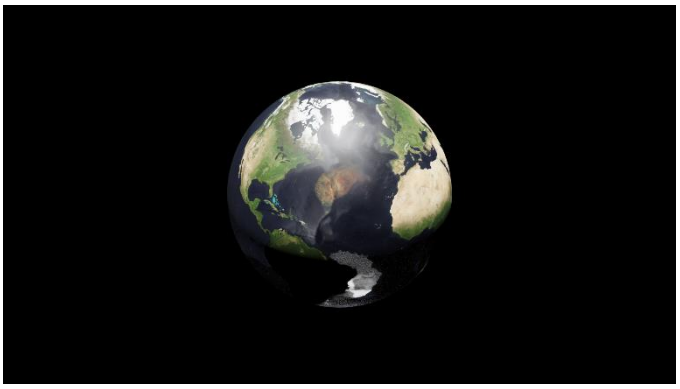

Fig. 3.12: Render of verification model

In fig. 3.11 it is noticed that the emerged part (*Principled BSDF*) is well defined and not transparent, to the contrary of the waters on the Earth that are transparent (in fact in the image it is visible Australia), as it is expected in the definition of the shader in this way. This verification confirms that the *Mixer Shader* block works as expected. Before moving on to the next step it is appropriate to make a further observation: one might wonder why the oceanic and marine part of the earth has a slight blue color despite its block is *Transparent BSDF* type. This is due to the fact that the bathymetry (Fig. 3.9) is not completely black in the zones covered by water, but presents a scale of greys that represent zones more or less deep. The lighter the zone, the more the output of the *Image Texture* block is different from 0, and therefore, each time the value is not zero, the texture of the Earth is applied more or less intensely. All this does not affect the final result, on the contrary it improves it, in fact, applying a block different from the *Transparent BSDF*, as in our case (Fig. 3.5), we have a cover of this blue layer little intense, maintaining, however, the scale of colors that define more or less deep areas. Including the operation of the *Mixer Shader* block, only the choices made in the *Principled BSDF* that makes the material of the ground water remain to be justified. Starting from the color, this was extracted from the texture of the earth via the site in Ref. [18], in particular the value of the RGB channels of the sea are: R=0.02, G=0.05; B=0.2. The second parameter set to realize the material is the *Subsurface*, this parameter simulates the scattering of light when it crosses the surface layers of water. As mentioned above there is no relationship between the parameters inside the block and the physical ones, it is therefore necessary to set these parameters with a series of tests until a photorealistic effect is reached. In this case it has been chosen to assign, to the above parameter, the value 0.2, noting that the color of the light that undergoes scattering is the same as that of the sea previously set. The parameters *Metallic* and *Specular* have been set in the same way to obtain the effect of reflection of light on the surface of the sea (Ref. [17]), in particular the first is equal to 1 and the second to 0.8. Finally, the last parameter that has been modified, compared to the default values,

is the *Roughness*, whose effect is to modify, enlarging or reducing, the surface affected by greater brightness, that is, the one around the area radiated perpendicularly by the Sun. To increase the roughness means to reduce the luminosity in that around but to widen the zone interested from the phenomenon, while, reducing it, the opposite effect is obtained. The definition of the Earth's shader ends with the *Material Output* block, which, in fact, applies the shader to the sphere. It is important to note that, in this case, all the effects created are surface, in fact the output of the *Mixer Shader* enters the *Surface* input of the *Mixer Shader*, you will see, in par. 3.3, the use of volume effects.

3.2 Clouds

The second sphere that has been inserted into the model of the Earth is the one that serves to represent the clouds. In this case the number of steps to be taken to achieve the desired result is smaller, but it is appropriate to list them from the beginning. The first step is, as in the previous case, to create a sphere of UV type that is immediately scaled in order to have a radius of 6373 Km, that is to start the clouds at 2 Km from the Earth's surface, according to the Ref. [13]. The surface modifier Subdivision Surface is applied to this sphere in order to divide the mesh into smaller elements for greater accuracy in the render results. The values assigned to set this modifier are the same as in the previous case, namely Viewport=6 and Quality=5. Completed this step you can switch to the shading of that sphere. Below, in Figures 3.13 and 3.14, the shader and texture used to create the cloud layer are shown.

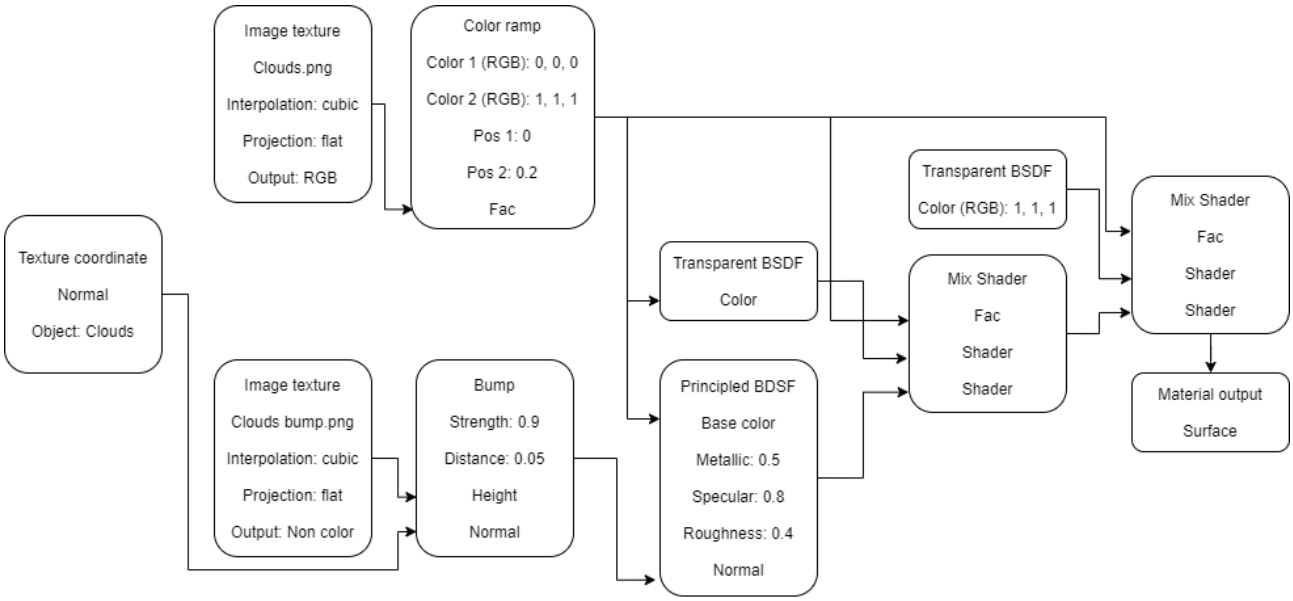

Fig. 3.13: Clouds shader

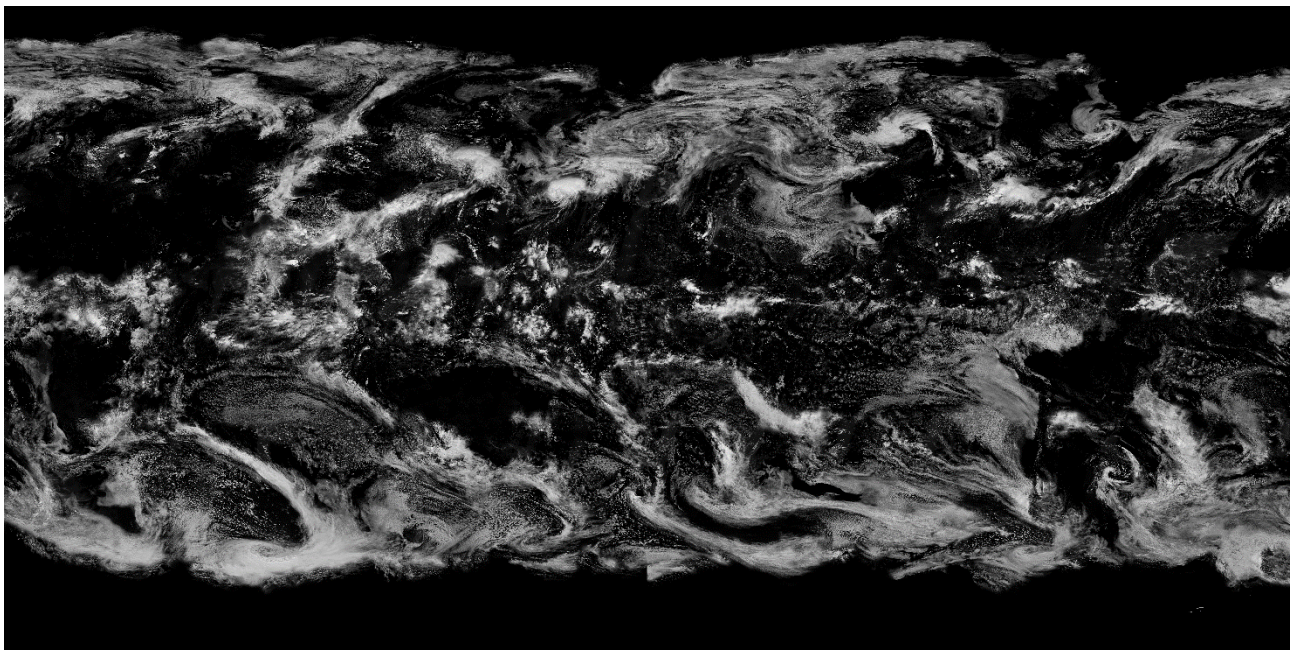

Fig. 3.14: Cloud texture

The following texture, in figure 3.14, has been realized joining the two pieces of texture, downloaded from the Ref. [16], through the Photoshop software, reaching a resolution of 21600 x 10800 pixels. Note that, there is also, within the folder containing the Earth, the texture with higher resolution 43200 x 21600 pixels. We proceed, now, with the analysis of the shader made to simulate the layer of clouds above the Earth's surface. Looking at Fig. 3.13 you begin to analyze the shader from the top. In this case it is possible to identify similarities with the previous case, in fact, there are both a *Texture Coordinates* block to map the sphere, created previously, on which to apply the texture in Fig. 3.13,

is an *Image Texture* block to import that texture into the scene. In this case we set the above block in order to realize a cubic interpolation and ensure a flat projection, in order to obtain an accurate application of the texture, avoiding visible errors and strongly impacting on the photorealism of the simulation. In the upper part of the diagram the output of the block *Mixer Shader* is of type RGB, in fact, this block will be, successively, connected to the two blocks *BSDF* that define the material and, therefore, the color of the clouds. Before proceeding with the right part of the diagram the underside is illustrated. In this case the same texture shown in Fig. 3.13 is used, but it is changed in name ('clouds 21600 x 10800 bmp'), this because Blender does not allow to apply, to the same texture, two Image texture blocks with different output. In particular what happens inside the software is that, if you find in the same shader two *Image Texture* blocks that import the same image, going to change a parameter, it changes, automatically, also the same parameter but in the other block. In our case this problem is strongly impacting, since the same image must be used once as a texture with *RGB* output and once as a topography to make the clouds volumetric, then with *Non Color* output. The solution to this problem was found, in fact, by renaming the same file and copying it in the folder containing the various textures, so as to have the same identical image, but with different name to circumvent the problem. Once understood the motivation for which are inserted two blocks *Image Texture* in this shader you can continue with the analysis of the remaining blocks that are within the shader. Proceeding from the bottom and, moving to the right, you notice the presence of a *Bump* block[3]. Its function is to map the height of the various points of the mesh according to the indications of the texture (used as topography), however, a detailed explanation of the operation and setting of this block is made in par. previous. Before proceeding, however, with the analysis of the shader it is appropriate to point out a difference in setting from the previous case. In this case, in fact, the *Distance* parameter of the Bump block is chosen equal to 0.05. Completed this step, to understand the operation of the shader you need to consider, first the blocks *Principled BSDF* and *Transparent BSDF* (bottom) and then talk about the function of the other block *Transparent BSDF* that combines the effect of the two blocks above. The *Principled BDSF* block is used to define the material of the clouds, in particular the color in input comes from the texture after the passage from the *Color Ramp*, while the *specularity* has been chosen equal to 0.8 and the *roughness* equal to 0.4. This block, however, alone, does not guarantee photometric accuracy of the images, in fact, the effect obtained using this as a single block, are clouds with volumetric extension opaque. It's immediate, therefore, to understand that to simulate the transparency of clouds (more or less marked according to density) it's necessary to introduce a *Transparent BSDF* block. This block, like the previous one, will have in input the

[3] Note that for the proper functioning of the block you need to map the normals via the *Texture Coordinate* node

texture color (with *RGB* output). To make accurate, and therefore physically correct, the variation of transparency of clouds according to density is used, once again, the only exact input that you have, namely the texture. In particular, the need is to find a way to dose, point by point, how transparent the layer of clouds is. For this purpose, it should be remembered that the texture of the clouds had been imported into a block with *RGB* output, and it is precisely this block that is necessary to achieve the mapping of more or less dense areas (after *Color ramp*). In fact, this output will be smaller (tending to 1) in areas tending to white and smaller (tending to 0) in areas tending to black. Everything is also reflected in the physics of the phenomenon, in fact, where the clouds are less dense their color in the texture will be more tending to black and, on the contrary, where they are denser will have a color more tending to white. To apply this density dosage it is necessary to assign to the input voice of the Mixer Shader block, '*Fac*', the *Color* output of the *Image Texture* block. The parameter *Fac*, in fact, is a parameter between 0 and 1 and, therefore, assigning the input described above, you get exactly the desired effect. The two blocks (*Pricipled BSDF* and *Transparent BSDF*) constitute, substantially, the central body of the cloud shader, are these, in fact, to define the material of the same. Once the definition of these blocks has been completed, then the description of the function that the other *Transparent BSDF* block performs. This block has the purpose of making the Clouds sphere transparent in the points where the texture is darker, that is in the points where there are no clouds. It should be noted the difference between the *Mixer Shader* block before the *Material Output* and the one illustrated above, the latter, in fact, deals with modulating the transparency of the clouds where they are present, the first, instead, applies the texture only in the points where clouds are present, ensuring complete transparency in the other points. Figure 3.15 is very explanatory in this respect.

Fig. 3.15: Clouds sphere ahead violet plane

To create the figure 3.14, a violet plane has been inserted inside the scene, to highlight the transparency of the sphere of clouds in the black points of the texture. The fact that you see the violet plane inside the sphere, where there are no clouds, certifies the proper functioning of the shader. The *Mixer Shader* block in analysis then takes in input 3 quantities: the output of the previous *Mixer Shader* block; the output of the *Transparent BSDF* block (top) and, finally, the output of the Image Texture block that imports the texture with *Non Color* output. The logic behind the choice of the *Fac* input of the block in analysis is the same as that described above, that is to have the clouds where the output tends to 1 and have transparency where it tends to 0, consistent with what is present in the texture. The shader ends, as in the previous case, with the *Material Output* block, noting that, also in this case, the effects applied to the sphere are all surface.

3.3 Atmosphere

This section describes the process that was followed to create the third, and last, sphere contained within the Earth.blend file. Surely the atmosphere is the most complex sphere from the physical point of view, within it take place, in fact, several phenomena, including the Rayleigh Scattering and Mie Scattering, beyond the reduction of density with the altitude that is cause of the phenomenon of the fall off atmospheric, that is, from the visual point of view, the variation of color in the atmosphere that gradually tends to disappear (See fig. 3.4). The first effect, the Rayleigh Scattering, is responsible for the blue color of the sky. This phenomenon is due, in fact, to the scattering of photons in the atmosphere and is related to the wavelength of light, in particular the luminous intensity that reaches the observer is inversely proportional to the fourth power of the wavelength.

$$I \propto \frac{1}{\lambda^4} \tag{15}$$

Looking at the formula it is easy to see that lower wavelengths reach the observer more easily, so, for this reason, the sky is blue in the eyes of an observer on Earth. These considerations are valid when the angle between the vector joining the observed point and the observer and the vector joining the observed point and the light source, the Sun, is quite large. In fact, in the hours of light, the sky turns blue in all conditions except sunrise and sunset. This consideration implies that the light intensity that the observer achieves also depends on other parameters. The following is the complete formula of Rayleigh Scattering (Ref. [19]).

$$I = I_0 \frac{8\pi^4 N\alpha^2}{\lambda^4 R^2}(1 + \cos\vartheta^2) \quad (16)$$

Where α is polarizability, N the scatter sumer, R the distance and ϑ the angle between the vectors mentioned above. The reason why the sky at sunrise and sunset turns red is that light, in these two particular conditions, must pass through a larger portion of the atmosphere and, as blue light tends to suffer more the phenomenon of scattering, the light reaching the observer is red/orange. It's necessary, to ensure a high accuracy of the images, that this phenomenon is simulated, the way you do it will be illustrated in the following, when it will be explained step by step the shader made for the atmosphere. A further consideration on Rayleigh Scattering concerns the color of the clouds, in this case the phenomenon is independent from the wavelength, or at least not from those in the field of the visible, because of the size of the water particles contained in the clouds, which is much bigger than air. For this reason the clouds appear white and not blue like the sky. Note, finally, that Rayleigh's Scattering considers the constant energy in the process of photon dispersion in the atmosphere.

As for the Mie Scattering, this results to be little dependent on wavelengths and is caused by particles larger than wavelengths in the light spectrum. This effect is, therefore, relevant when particulate matter is present in the air, or in fog conditions, in fact, it is precisely because of this phenomenon that fog gives a blurred perception of the environment. In the case of Rayleigh Scattering light is dispersed more or less evenly in all directions, on the contrary, in Mie Scattering, a much more pronounced lobe is formed in the forward direction, taken as a reference the vector that from the light source reaches the point observed. The extension of the lobe in forward direction compared to the backward direction is all the greater the larger the particles, it is for this reason that the more the air is charged with large particles (particulates or fog) the more accentuated the effect of blurring. The effect of Mie's scattering is viewable by looking at the Sun, because the lobe is formed in the forward direction that, in the case of looking at the Sun, coincides with the vector that connects the observer and the observed point. For the same reason it occurs that, on clear days, the scattering that prevails is that of Rayleigh, in fact, looking at a point far from the light source (that is, at an angle ϑ greater than 0 in equation 16), Rayleigh's scattering scatters light almost evenly in all directions, while Mie's scattering mostly in the forward direction, which, in this case, is not in the direction of the observed point. To fully understand this concept it is useful to observe Fig. 3.16, below.

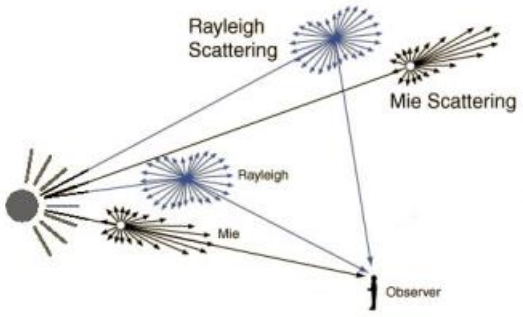

Fig. 3.16: Sun light Rayleigh and Mie Scattering, Ref. [20]

In the figure we see, in fact, the two cases, the one at the bottom where the observer looks at the Sun and, at the top, where the point observed has a direction so that the forward lobe of Mie's Scattering is not intercepted by the observation vector (i.e., the vector connecting the observer to the observed point). Finally, the last phenomenon to describe, before justifying the choices that have been made to realize the shader, is that of the reduction of the density of the atmosphere with the altitude. Within the atmosphere you can distinguish several layers within which thermodynamic quantities have a certain behavior, modeled with the standard atmosphere. In the case of density it occurs that , in all layers, it decays with exponential trend, following the law:

$$\rho = e^{-\frac{h}{H_0}} \tag{17}$$

- The following equation has been derived from Ref. [17], describing the quantities that appear within it we find:
- ρ: that represents the *Densisty ratio* and not density, in fact at altitude of 0 m is equal to 1 and not to 1.225 Kg/m^3;
- h: that is the height at which the *Density ratio* is calculated, in m;
- H_0: Equal to 7994 m (Ref. [17]), value corresponding to the height extension of the atmosphere if density was uniform;

Having a mathematical law that regulates the trend of density in the atmosphere is very useful for the realization of a shader physically accurate, in the following, in fact, you will see its implementation within the same. The drop in the density of the atmosphere generates, as previously mentioned, the phenomenon of the atmospheric fall-off. This phenomenon can be visualized by looking at the Earth from space (then to be realized in the simulation). In particular, what is displayed is a variation of color from the lower layers that appear tending to white to the higher layers that, after becoming blue due to Rayleigh's scattering, tend to black, which testifies to the reduction of molecules in those areas, until disappearing. In the following simulation it was chosen to follow the Kármán line convention,

however there is no exact limit beyond which there are no particles, hence the need to use a conventional limit. Defined in detail the phenomena related to the interaction of light with the atmosphere, we pass, now, the description of the shader created to realize the atmosphere and, therefore, simulate such phenomena. Then, in Figure 3.17, the shader of the atmosphere is reported.

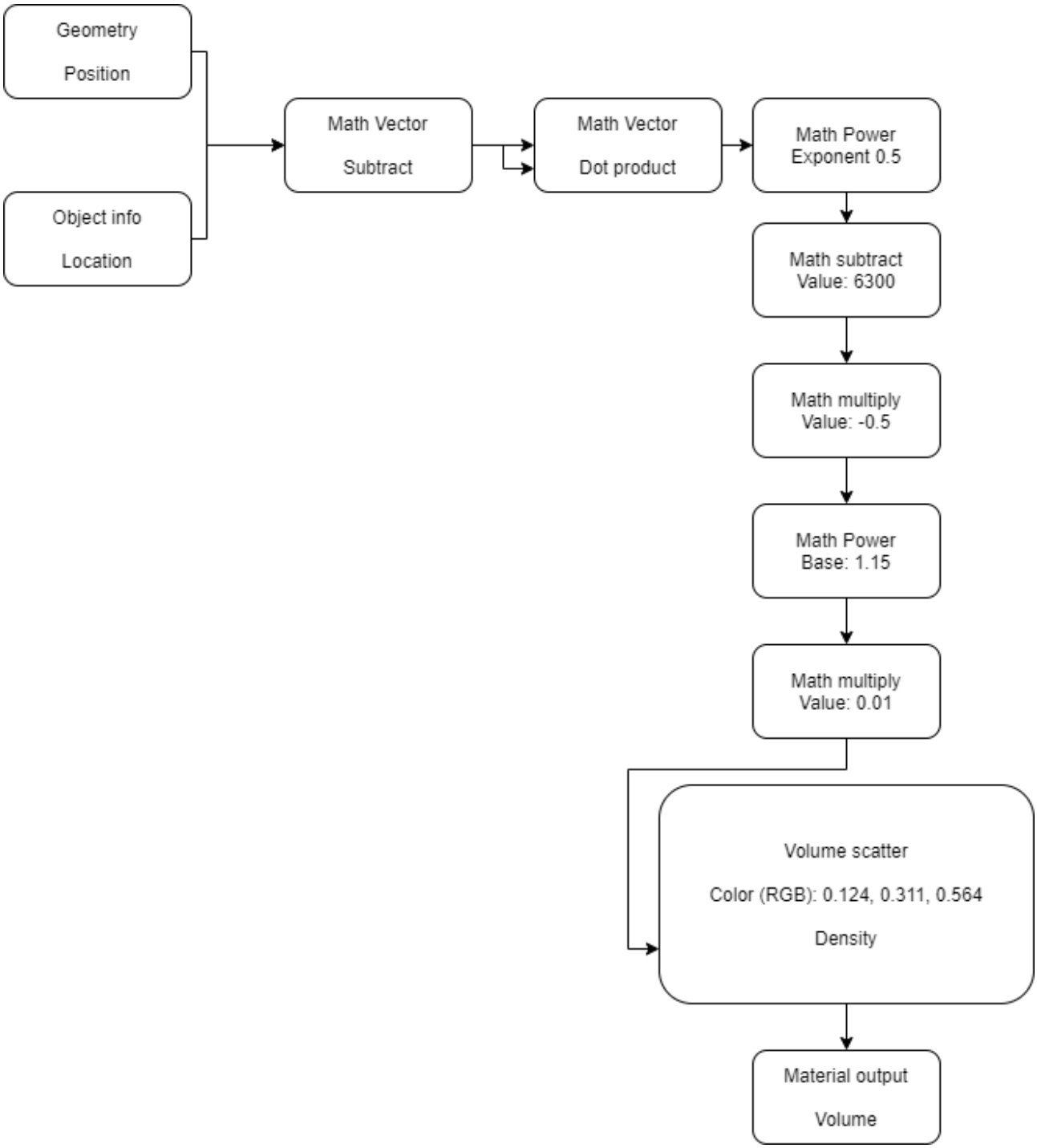

Fig. 3.17: Atmosphere shader

Before creating the atmosphere shader, as for the other two spheres, it is necessary to create a UV sphere, which is subdivided with the modifier *Subdivision Surface*, using the same parameters as par. previous: *Viewport*=6 and *Quality*=5. Once this is done and, set the display of the sphere on *Shade smooth*, we proceed to the creation of the shader. To describe the shader, according to what has been done so far, we start from the left of Fig. 3.17 and proceed to the right. The first two blocks that have been inserted are: the *Geometry* block and the *Object Info* block. The first block, whose output is *Position*, is necessary to create vectors that, from the origin of the space created by Blender identify all the points of the surface. The second block, on the other hand, aims to identify the position of the sphere, that is, its origin. The presence of these two blocks allows, in the next block (*Math, Subtract*) to calculate the vectors that from the origin of the sphere identify the points of the same. By subtracting, in fact, from the position of the points in the absolute reference system, the position of the sphere, we obtain the positions of the points of the surface in the relative reference system (centered in the sphere).

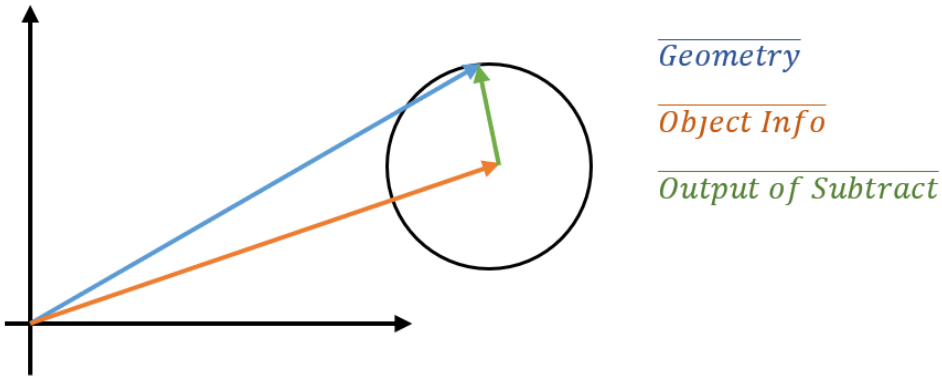

Fig. 3.18: Explanation of Subtract block output

The vectors in Fig. 3.18 are explained in the legend, the reference system in black is the space created by Blender. Once the relative position of the points of the sphere is obtained, it is necessary to find the vectors that identify these points from the Earth's surface. The need to find these vectors is due to the fact that the atmospheric density varies precisely in this layer. In fact, looking at the formula (17), more than the vector, what is of interest for the calculation is the value of the height of the various points. It is important, at this point, to emphasize that all the blocks that follow, except the last two, serve to realize the equation (17), the effect of the scatter, in fact, is enclosed in a single block, which will be discussed later. Continuing with the implementation of equation (17), we insert another *Vector Math* block, this time to realize the scalar product between the previously obtained vectors and themselves. This operation is part of the calculation of the module of these vectors, to

complete this calculation is inserted inside the shader a block *Math* (scalar in this case), which has the task of performing the square root of the result of the previous block. At the end of this operation you have the value of the height of the various points from the center of the sphere. The value *h* that appears in (17), however, is not calculated from that point, but from the Earth's surface. It is necessary, at this point, to insert another block, always of *Math* type, to subtract from the obtained value the value of the radius of the Earth. Observing the shader in Fig. 3.17, however, we note that the value is not equal to 6371 but is equal to 6300, the motivation behind this choice is found, once again, in the non-correspondence in Blender of the parameters of the blocks with the physical parameters, this consideration also applies to other changes that are present in the following blocks. Having now the parameter *h*, it must be multiplied by $-1/H_0$. Also in this case, as mentioned before, the value inside the *Math, Multiply* block differs from the one provided by equation (17). In fact, this value should be -1/7.994 (~ -0.125), while it is known to be equal to -0.5. At the end of this block you have, therefore, the exponent of the formula that you are importing. The task of the next block, *Math, Power* is to raise this exponent to power, observing the figure you notice, once again, a deviation from the formula, necessary to make the atmosphere accurate, the value chosen for this block is 1,150. Although the formula has been, at this point, completely converted into blocks of the shader, we notice the presence of an additional block *Math, Multiply*, which scales the value of the *Density ratio* of a parameter 0.01 to ensure the correctness of the images generated. After the part of shader that deals with simulating the fall off atmospheric you pass to the block that defines the material of the atmosphere. In the introductory part of this paragraph we talked about the scattering of light in the atmosphere, to simulate all this the software provides a block that has just this function (*Volume Scatter*). This block has 3 inputs: the color, density and parameter of anisotropy that remained at the default value. Inside the input *Density* the output of the series of blocks described above is inserted. As for the calculation of the RGB channel values of the color of the atmosphere, the Rayleigh equation was used. From (15) it is clear that the light intensity that reaches the observer is inversely proportional to the fourth power of the wavelength, therefore, for the calculation of the three channels it is necessary to use the wavelengths of red, of green and blue, which are shown in the table below.

Color	Wavelenght
Red	625-740 nm
Green	520-565 nm
Blu	435-500 nm

Tab 3.1: RGB wavelenght

From these wavelengths can be obtained the percentage of intensity I0 that reaches the observer, for each of the 3 colors.

- Red: $I \propto 4.609*10^{24}\ I_0$;
- Green: $I \propto 1.154*10^{25}\ I_0$;
- Blu: $I \propto 2.093*10^{25}\ I_0$;

Note that the mean wavelength between the extremes given in Table 3.1 has been used for the calculation. From these values, to obtain the calculation of RGB channels you calculate the percentage of each of the total, this percentage will be equal to the corresponding channel:

- Red: $4.609*10^{24}/(3.708*10^{25}) = 0.124$;
- Green: $1.154*10^{25}/(3.708*10^{25}) = 0.311$;
- Blu: $2.093*10^{25}/(3.708*10^{25}) = 0.564$;

This calculation is based on the fact that, of all the light reaching the observer, obtained by adding the various I0 by their proportionality factor, one part will be red, one green and one blue, these parts are calculated in proportion to the total, or by carrying out a percentage calculation. Set these values you insert a *Mixer Shader* to improve image quality and , as anticipated in par. 3.1, in this case the input of the *Material Output* block is that of type Volume, being that the atmosphere introduces effects of volume. The following two figures are reported: the first (3.19) is a render of the atmosphere without the Earth and clouds, useful to demonstrate the correct operation of the series of blocks that simulate the fall of atmospheric density; the second (3.20) is a render, complete with Earth atmosphere and clouds, in case the parameters of (17) are inserted within the atmosphere shader.

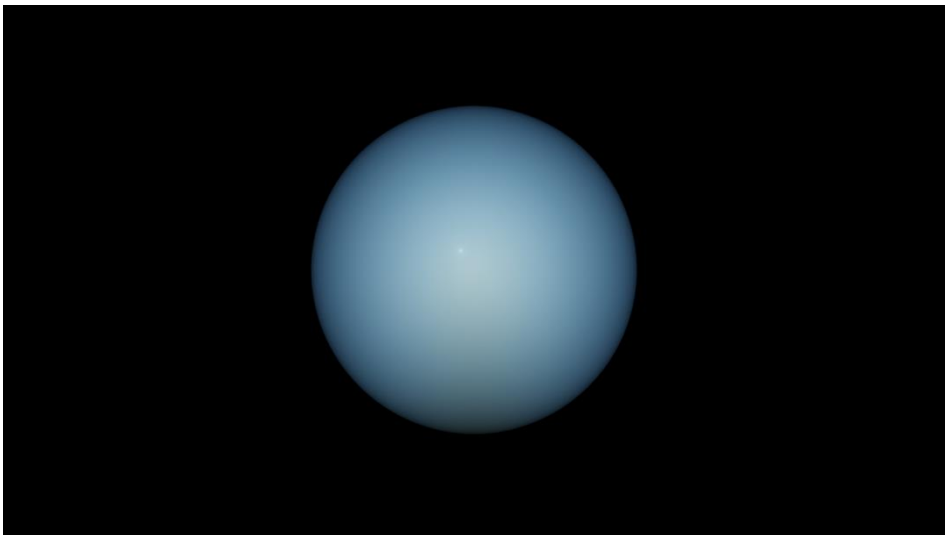

Fig. 3.19: Render of atmosphere with fall off and scatter effect

Fig. 3.20: Render of atmosphere shader with parameters of equation (17)

Looking at Figure 3.19 you can appreciate the proper functioning of the atmosphere shader, note that this render was made with a sphere of smaller size (r = 50 m). Figure 3.20, on the other hand, justifies the fact that the parameters of equation (17) are not included in the shader. However, even in the absence of photorealism in this image you can appreciate the formation of the red color at the edge of the shadow zone, witness to the fact that the *Volume Shader* block correctly simulates the phenomenon of scattering.

3.4 Sun compositing

Insert the Sun inside the simulator is necessary to have the lens flare that allows a possible image analysis algorithm to identify whether the Sun is present or not within the same. The phenomenon of lens flare, according to the Ref. [21], is due to the passage of light through an optical sensor (a camera for example), in particular to the reflection between optical elements (which generates ghosting). Note, in fact, that the node of compositing in Blender that is used to realize the lens flare is just the *Glare-Ghost*. As said it is the phenomenon of reflection that causes this visual effect and, from the law of Fresnel, it is clear that it is influenced by the polarization of the material. From the law cited above, in fact, they derive different coefficients of reflection and transmission for different states of polarization of the material. In this regard, two considerations can be made: for an accurate model from the point of view of the lens flare it would be appropriate to keep track of the beam-surface interaction from the point of view of polarization, however, in the Ref. [21], it is explained that the accuracy advantage is minimal compared to the computational cost involved. In fact, in the above-mentioned work it is assumed that light is p-polarized and s-polarized. Another consideration that comes out of the influence of lens flare from polarization is that to achieve anti-reflective lenses it is necessary to act on the polarization of the material. Within the following work you do not get to this

level of accuracy of the lens flare. The reason is that what is interesting is that the lens flare is simulated, to recognize the presence or not of the Sun in the frame, without the need to opt for complex models. In our model, what we want to achieve, is a Sun that emits light and generates lens flare but that does not modify the lighting inside the scene, since there is already a sun-type lamp that deals with the lighting of Earth and Space Rider. For the same reason, the Sun lamp should not illuminate the Sun, to avoid interfering with the lens flare generated. For this purpose it is necessary to divide the scene into two view layers: the first contains Earth, Space Rider and SROC (grouped in the appropriate collection), the second contains the Sun. It is important to leave the camera outside the collections in order to make it present in both view layers, in order to render both the collection of Space Rider and Earth, both that of the Sun. The realization of the sun begins with the creation of a mesh of radius 696340 m (corresponding to the Km, see par. previous), then, also in this case, of real size. Before realizing the compositing, the shading of the Sun is done, inserting an *Emission* node with an emission value of 20. Then it is realized the compositing of the entire scene to obtain the desired effects, of continuation they bring back the nodes used.

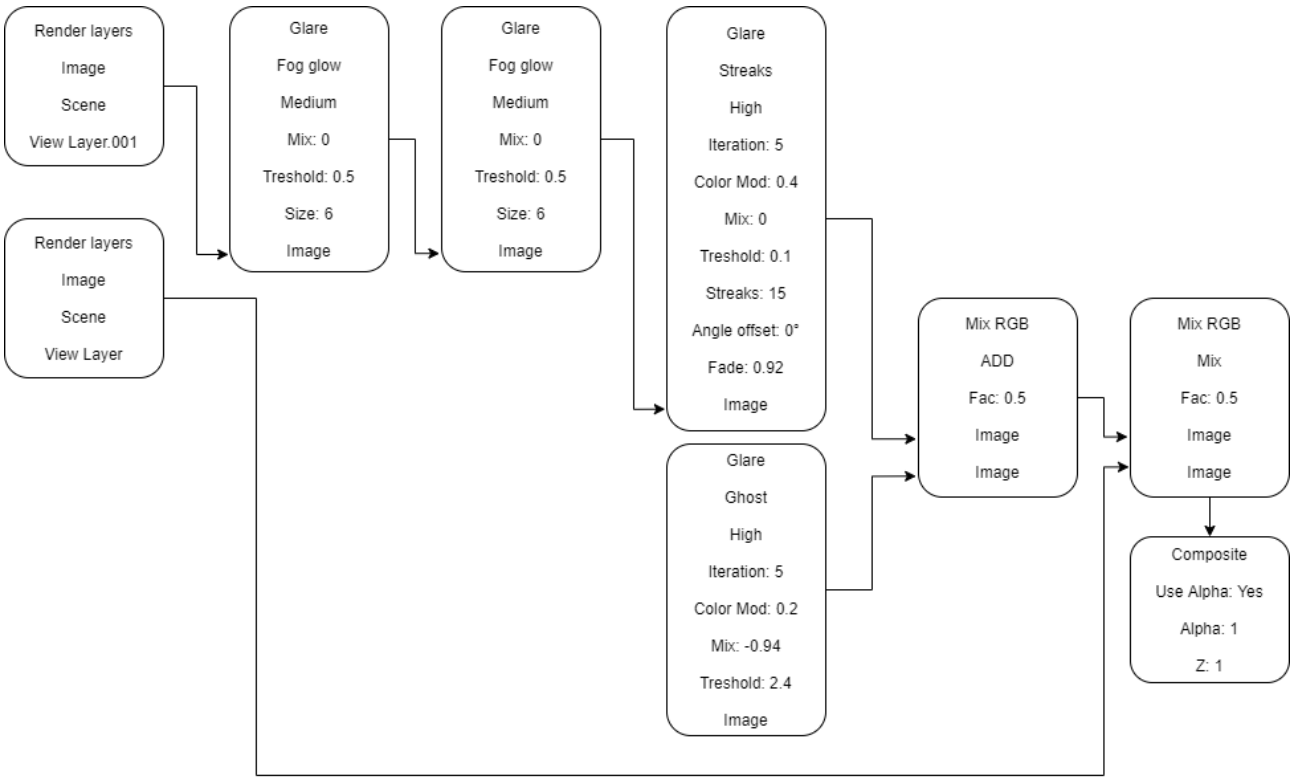

Fig. 3.21: Compositing of scene

In the following diagram two levels can be distinguished: an upper and a lower one. The upper one concerns the view layer containing the Sun, the lower one the second view layer in which the elements of the scene described above appear. Starting from the upper level you notice the presence of two

nodes of type *Glare-Fog Glow* that are necessary to realize the halo around the Sun, what you want to achieve is a effect type *bloom*, however, using Cycles, this can not be activated with a command, But that's how it's done. The third node*: Glare-Streaks* serves to realize the rays, in the definitive version of the file has been eliminated because of the low quality of the result obtained, that is of the rays too thick and, therefore, little realistic. The most important node of the diagram is the one that follows, that is the *Glare-Ghosts*, that allows the realization of the lens flare, main reason for which the Sun has been inserted. The effect of this node is added to others through a *MixRGB-Add* node, unlike the others that are in series, this to make sure that the lens flare is only due to the sun that emits light and not to the halo and other effects of compositing, basically they go to separate the effects. The lower level, as you can see, is not modified with compositing effects. Note that compositing is applied to render when completed.

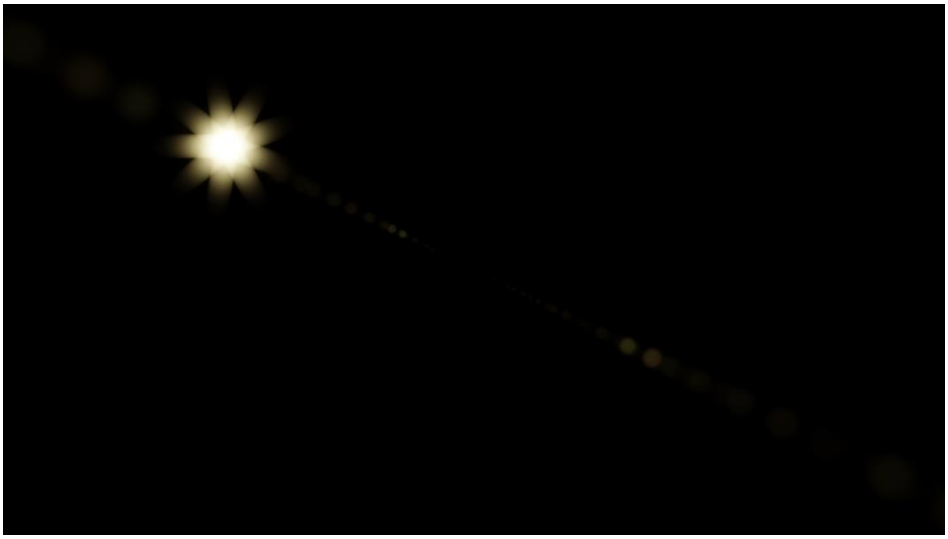

Fig. 3.22: Sun Render with streaks, form LEO orbit

Figure 3.22 shows, as already mentioned, the unsatisfactory accuracy of the sun's rays (to see the final Sun render, therefore without rays, see ch. 4). However, from this image, you can appreciate the correctness of the lens flare. It is appropriate to make a further remark about Blender's compositing, which is to be strongly influenced by the distance from which you look at the phenomenon. Such consideration implies that, once designed the compositing, it works well only for that determined distance. Note, however, that, regarding this observation, the Sun can be considered equidistant from all objects orbiting around the Earth, this problem, therefore, does not concern this simulator

4 Conclusions

In the following chapter, which concludes the thesis, the results and possible applications of this code will be explained. Before proceeding with the results it is appropriate to illustrate a summary diagram of how the code operates (fig. 4.1).

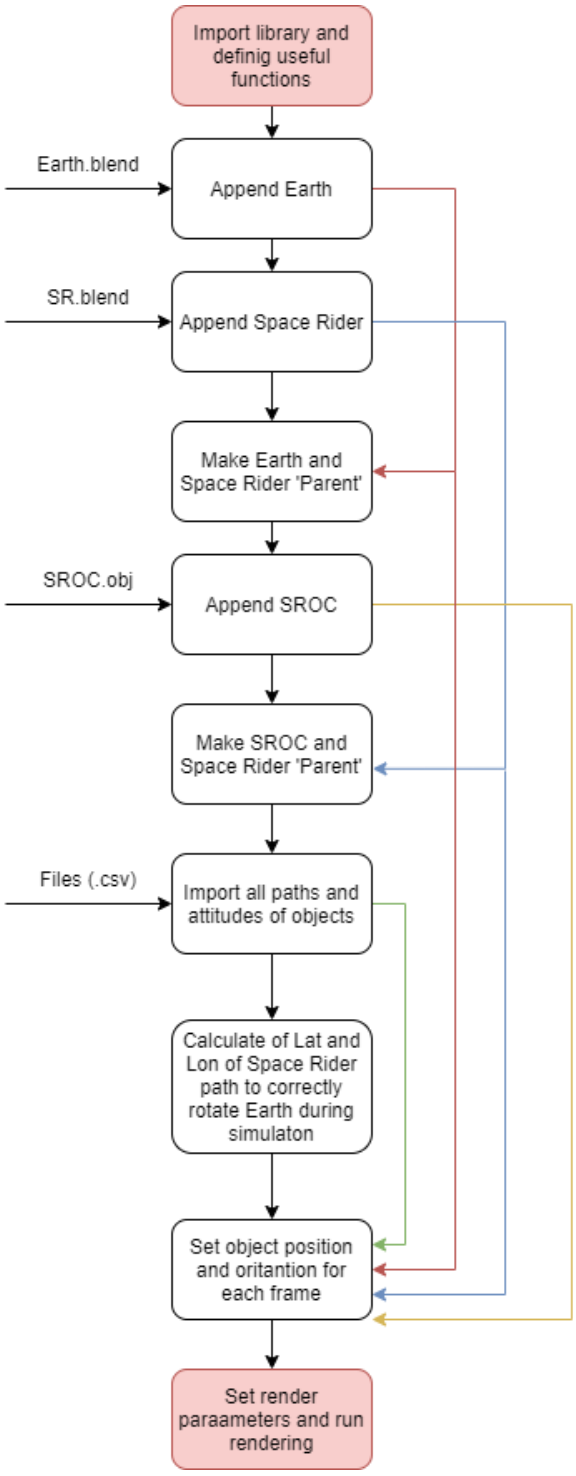

Fig. 4.1: Code simplified scheme

To summarize in a nutshell, the code always follows the same logic for all three created bodies, that is, it imports them from another file and relates them to Space Rider in order to assign relative coordinates to objects. Once the objects and coordinates are imported from the files (.csv), the position and orientation of each object is assigned for each frame. The main requirements of the simulator are three: the flexibility with respect to changes in bodies and trajectory, the correctness of the simulation of mission geometry and, finally, a high level of photorealism of images. The first requirement is highlighted both in the code diagram (fig. 4.1) and in chapter 2 where it is explained in depth. It is noted, in fact, that it is possible to use different spacecrafts instead of Space Rider and SROC, different trajectories and arrangements, but also bodies other than Earth if you have the relevant file (.blend) to import. To highlight the second requirement, that is the correctness of the mission geometry, two images are shown below: one representing the real mission geometry (fig. 4.2) and the other representing the simulated mission geometry in Blender (fig. 4.3).

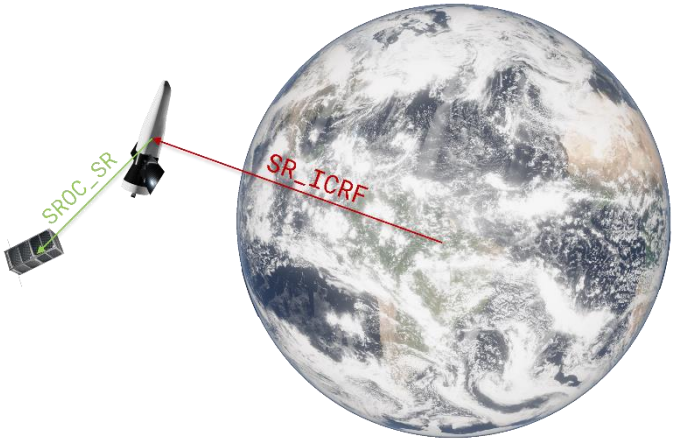

Fig. 4.2: Representation of mission real geometry

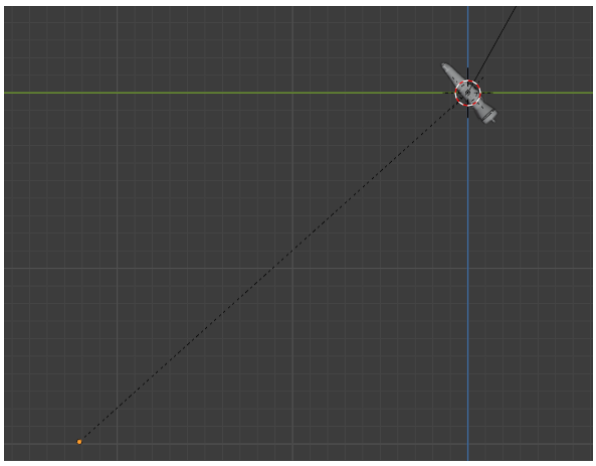

Fig. 4.3: Representation of Blender simulation geometry

Observing the fig. 4.3, that is that relative to the simulation, it is seen that SROC (dot orange) is positioned correctly regarding Space Rider. Choosing to make the bodies with their real size prevents capturing a snapshot that simultaneously shows the correct positioning of Earth, Space Rider and SROC within the simulation. To verify everything, however, just run the simulator, immediately noticing the correct geometry of the simulation. With regard to the third requirement, namely the creation of a model with a high level of photorealism, the verification is more complex. What is done, to validate the model from this point of view, is to compare the images produced with the software and real images of the Earth, in order to determine whether the model is accurate or not. Beyond this comparison, it is appropriate, for the same purpose, to verify if the model is able to capture effects that occur due to the physical phenomena involved in the propagation of light between the Earth and the atmosphere (see ch. 3). Below are some renders of the Earth and real images of the same view from space, to make the comparison described above.

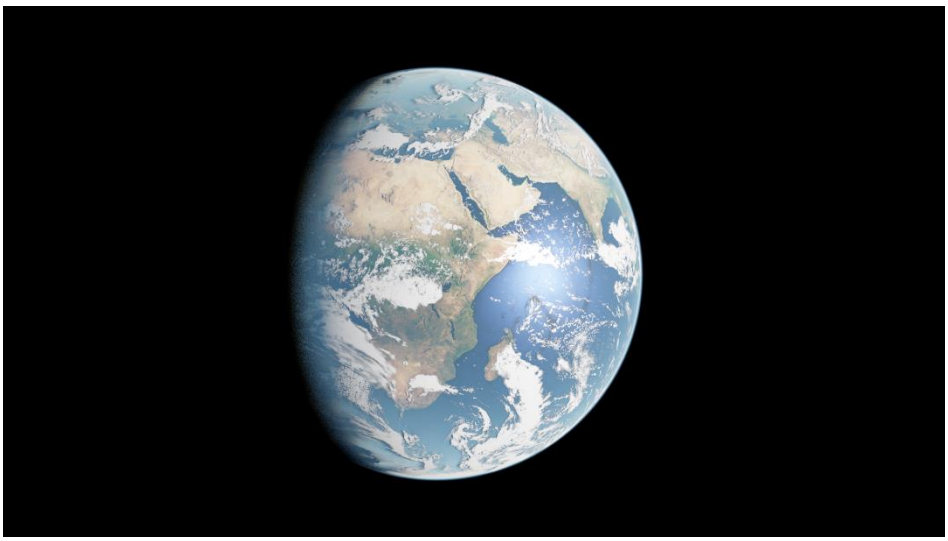

Fig. 4.4: Earth render 1

Fig. 4.5: Earth render 2

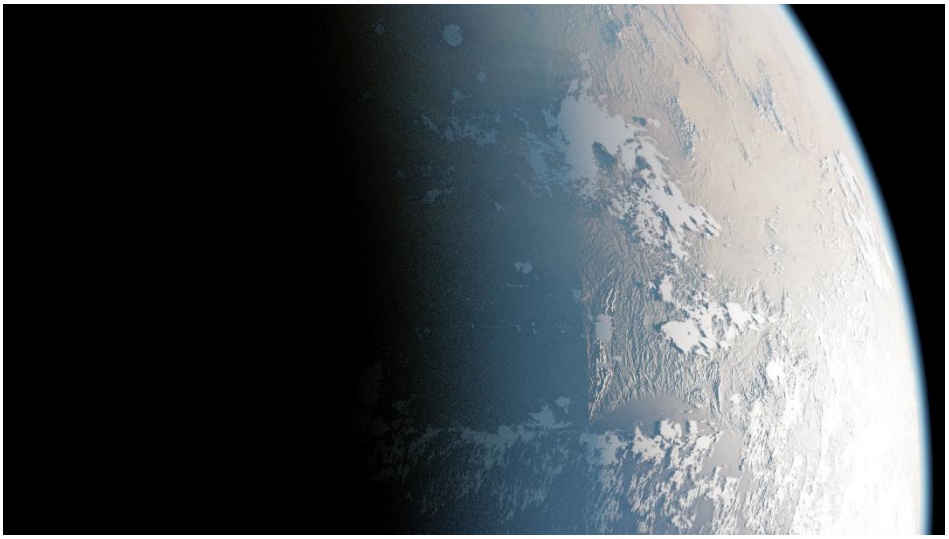

Fig. 4.6: Earth render 3

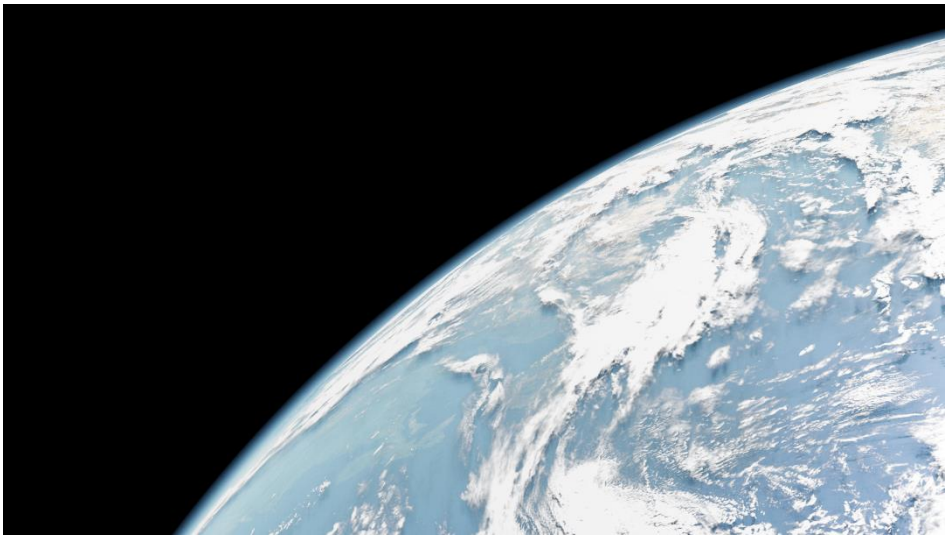

Fig. 4.7: Earth render 4

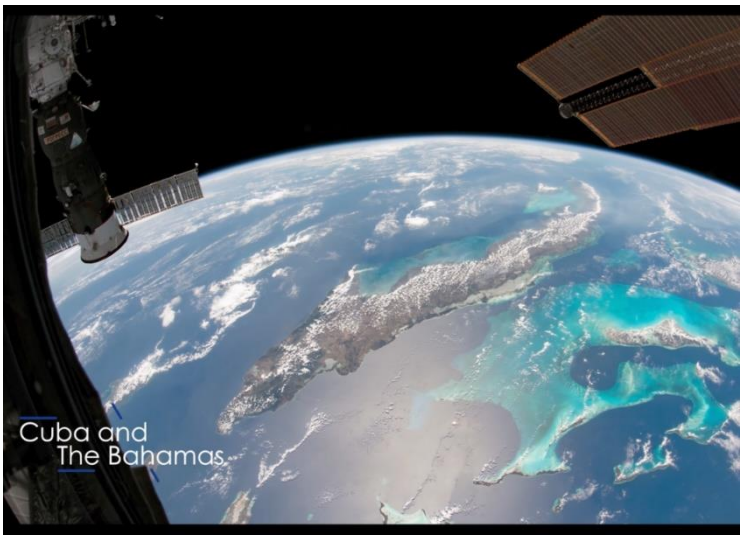

Fig. 4.8: Earth real image 1

Fig. 4.9: Earth real image 2

From the comparison of these last four images we can see that the model well simulates the drop in density of the atmosphere as the altitude increases, that is the phenomenon of the atmospheric fall-off. In addition, there is also a high accuracy in the realization of clouds, a fundamental element in order to have a realistic model. Another consideration regarding the accuracy of clouds can be made by observing that clouds produce shadows on Earth, as happens in reality. Observing, then, the figures 4.5 and 4.6 you can see that the model well simulates the colors of the atmosphere (and therefore of the sky) at dawn and sunset, with a small red stripe close to the area of the earth in shadow (night). This is due to the scattering of light in the atmosphere, a theme extensively explored in cap. 3. Finally, the other two characteristics of the model that contribute to making it realistic are: the three-dimensional extrusion of mountain ranges and land reliefs (through a topography); and the reflection of the much more intense light on the seas and in general on the waters than on the land emerged.

The other element, inserted in the simulator, is the Sun, below we report renders of the same to show how the phenomenon of lens flare has been modeled. Before proceeding with the illustration of the renders it should be noted that in the realization of the Sun was not taken into account the physics of the phenomena, for two reasons: the first is that the compositing of Blender is extremely limited in the possibilities (even just making convincing rays creates problems), the second is that the only reason why the Sun is inserted in the simulator is to have, in fact, the lens flare, in order to identify, through an algorithm of image analysis, whether or not the sun is framed by the camera.

Fig. 4.10: Sun render

In the image you can see the absence of sun rays for the reasons explained above, while you can see how the phenomenon of lens flare is correctly simulated. Below is a further rendering of the Sun that verifies the fact that the presence of the same does not illuminate the other elements of the scene, this is necessary because the solar lighting is simulated through the appropriate lamp in Blender.

Fig. 4.11: Sun and Earth render

In the image the Earth is located to the left of the sun and, as you can see, it is not illuminated by the same. This behavior is achieved by inserting a second view layer in the simulation, within which there is only the Sun.

The last render that is reported is relative to the first frame of the simulation

Fig. 4.12: 1st frame simulation render

Also in the following render you can see a high accuracy of the elements involved. The final output of this work is these renders. The natural development of this work is the creation of an image analysis algorithm that, taken in input these renders, can recognize the elements framed (eg Sun, Space Rider, etc.) or, in a more advanced version, also to derive the kinematic characteristics of the bodies (Ref. [9], analysis of the period of spin of the spacecraft framed).

4.1 Limits and future works

Between the limits of the simulator illustrated in this thesis there is, sure, the elevated rendering time, it is noticed, in fact, that for every frame such temporal interval is equal, in average, to 25 minutes. Another limitation in the simulator is the need to put Space Rider in the center, to ensure its appearance in render. Such problem comes from the difficulty of Blender to manage objects of too different dimensions inside the same scene and, the eventual presence of two small bodies to frame, could send in crisis the simulation, although the applied solution seems to function properly and responds robustly to the trajectory changes used as a test for the simulator. Another problem lies in the rotation of the camera when it is applied the constraint 'Track-To' (necessary to maintain the pointing), problem solved by manually setting an angle of view to frame the satellite (Space Rider)

for the duration of the simulation. In this regard it is noted that the simulator is useful in the design phase of the mission in another area, namely the choice of positioning, and orientation, of the camera in order to keep the target framed for the duration of the mission. Finally, the last limit of the simulator is the low physical accuracy of the lens flare. It has already been explained above that this problem is not impacting on the purposes of the simulator since the only necessary feature is the lens flare, which is generated. As for the future works, one of the possible fields of development of the simulator is the expansion of the 'catalogue' of bodies from which the simulator draws to import them into the scene. Think for example of other planets such as Mars, Saturn or satellites such as the Moon (which would make the lighting conditions in LEO orbit even more precise, at the price, however, of a higher computational cost needed for the simulation). Another possible field of use of the simulator is in the design of operations to remove disposed of satellites. Certainly a visual simulator of the activities to be carried out would bring to light, in a clear way, any problems of design, helping to find the right solution. As explained above, another possible application of the simulator is to develop an image analysis algorithm in order to create a neural network that is able to derive from the images information necessary for the identification and characterization (position velocity, etc.) of the target. All this aims to make the neural network, and therefore the satellite, able to autonomously complete the docking or a possible operation to remove space debris (connecting to the previous possible field of development of the simulator).

Appendix

A.1 Code

```
#Blender version: 2.8

import bpy
import os
import datetime as dt
import math
import mathutils
from mathutils import Vector, Matrix
from math import acos, atan

############ FUNZIONI UTILI ###############
def file_len(fname):
    with open(fname) as f:
        for i, l in enumerate(f):
            pass
    return i + 1

def euler_from_quaternion(x, y, z, w):
        """
        Convert a quaternion into euler angles (roll, pitch, yaw)
        roll is rotation around x in radians (counterclockwise)
        pitch is rotation around y in radians (counterclockwise)
        yaw is rotation around z in radians (counterclockwise)
        """
        t0 = +2.0 * (w * x + y * z)
        t1 = +1.0 - 2.0 * (x * x + y * y)
        roll_x = math.atan2(t0, t1)

        t2 = +2.0 * (w * y - z * x)
        t2 = +1.0 if t2 > +1.0 else t2
        t2 = -1.0 if t2 < -1.0 else t2
        pitch_y = math.asin(t2)

        t3 = +2.0 * (w * z + x * y)
        t4 = +1.0 - 2.0 * (y * y + z * z)
        yaw_z = math.atan2(t3, t4)

        return roll_x, pitch_y, yaw_z # in radians



current_path=os.getcwd()
my_path=current_path + '\\Resources2'
SROC_file_path=my_path+'\\Blender'+'\\SROC.obj'
SROC_in_SR_file_path=my_path+'\\Trajectories'+'\\SROCCenter_inSRbodyaxis.csv'
```

```
SROC_attitude_in_SR_file_path=my_path+'\\Trajectories'+'\\SROC_D31Bodyaxis_inSRbody
axis.csv'
SpaceRider_file_path=my_path+'\\Blender'+'\\Space_Rider.obj'
SpaceRider_attitude_in_ICRF_file_path=my_path+'\\Trajectories'+'\\SR_quaternions_wr
t_ICRF.csv'
SpaceRider_LIGHT_in_SR_file_path=my_path+'\\Trajectories'+'\\SRsunVector_inSRbodyax
is.csv'
SpaceRider_in_ICRF_file_path=my_path+'\\Trajectories'+'\\SR_coordinate_ICRF.csv'
Earth_in_SR_file_path=my_path+'\\Trajectories'+'\\Earth_position_inbodySR.csv'
Camera_data_input=my_path+'\\Camera parameters'+'\\Camera1.txt'
Name_file_output='Render Relative'
Earth_path=my_path+'\\Blender'+'\\Earth'+'\\Earth2.blend'



# Clearing scene
for item in bpy.data.objects:
    bpy.data.objects.remove(item)

# Set frame rate (fps)
frame_rate=0.1
bpy.context.scene.render.fps = frame_rate

# Set render engine
bpy.context.scene.render.engine = 'CYCLES'

# Set scale
scale=10**0

#Append Earth
#Earth
inner_path = 'Object'
object_name = 'Earth'

bpy.ops.wm.append(
    filepath=os.path.join(Earth_path, inner_path, object_name),
    directory=os.path.join(Earth_path, inner_path),
    filename=object_name
    )

#Atmosphere
inner_path = 'Object'
object_name = 'Atmosphere'

bpy.ops.wm.append(
    filepath=os.path.join(Earth_path, inner_path, object_name),
    directory=os.path.join(Earth_path, inner_path),
    filename=object_name
    )
```

```
#Clouds
inner_path = 'Object'
object_name = 'Clouds'

bpy.ops.wm.append(
    filepath=os.path.join(Earth_path, inner_path, object_name),
    directory=os.path.join(Earth_path, inner_path),
    filename=object_name
    )

#Make Atmosphere and Earth parent
bpy.data.objects['Atmosphere'].select_set(True)
bpy.data.objects['Earth'].select_set(True)
bpy.context.view_layer.objects.active = bpy.data.objects['Earth']
bpy.ops.object.parent_set(type='OBJECT', keep_transform=False)

#Make Clouds and earth parent
bpy.data.objects['Clouds'].select_set(True)
bpy.data.objects['Earth'].select_set(True)
bpy.context.view_layer.objects.active = bpy.data.objects['Earth']
bpy.ops.object.parent_set(type='OBJECT', keep_transform=False)

#Import SpaceRider model
file_loc = SpaceRider_file_path
imported_object = bpy.ops.import_scene.obj(filepath=file_loc)
obj_object = bpy.context.selected_objects[0]
obj_object.name="SpaceRider"
print('Imported name: ', obj_object.name)
bpy.ops.transform.resize(value=(0.06006807715*scale,           0.04071992833*scale,
0.04071992833*scale), orient_type='GLOBAL', orient_matrix=((1, 0, 0), (0, 1, 0), (0,
0,  1)),  orient_matrix_type='GLOBAL',  mirror=True,  use_proportional_edit=False,
proportional_edit_falloff='SMOOTH',                          proportional_size=1,
use_proportional_connected=False,               use_proportional_projected=False,
release_confirm=True)
for i in bpy.data.objects:
    i.select_set(False)
bpy.data.objects['SpaceRider'].select_set(True)
bpy.ops.object.transform_apply(location=False, rotation=True, scale=False)

for i in bpy.data.objects:
    i.select_set(False)

# Make Earth and SpaceRider parent (to have relative motion)
bpy.data.objects['Earth'].select_set(True)
bpy.data.objects['SpaceRider'].select_set(True)
bpy.context.view_layer.objects.active = bpy.data.objects['SpaceRider']
bpy.ops.object.parent_set(type='OBJECT', keep_transform=False)
```

```
#Import SpaceRider attitude
rot_1=[""]
rot_SR=[]
with open(SpaceRider_attitude_in_ICRF_file_path) as file:
    for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
            rot_1=[float(d) for d in line[6:10]]
            rot_SR=rot_SR+[rot_1]

#Import SROC model
file_loc = SROC_file_path
imported_object = bpy.ops.import_scene.obj(filepath=file_loc)
obj_object = bpy.context.selected_objects[0]
obj_object.name="SROC"
print('Imported name: ', obj_object.name)
bpy.ops.transform.resize(value=(0.00005*scale,    0.00004*scale,    0.00005*scale),
orient_type='GLOBAL',  orient_matrix=((1,  0,  0),  (0,  1,  0),  (0,  0,  1)),
orient_matrix_type='GLOBAL',       mirror=True,       use_proportional_edit=False,
proportional_edit_falloff='SMOOTH',                          proportional_size=1,
use_proportional_connected=False,                use_proportional_projected=False,
release_confirm=True) #scalare dimensioni di 100
bpy.data.objects['SROC'].rotation_mode="XYZ"
bpy.data.objects['SROC'].rotation_euler=0, 0, math.pi
for i in bpy.data.objects:
    i.select_set(False)
bpy.data.objects['SROC'].select_set(True)
bpy.ops.object.transform_apply(location=False, rotation=True, scale=False)

# Make SROC and SpaceRider parent (to have relative motion)
bpy.data.objects['SROC'].select_set(True)
bpy.data.objects['SpaceRider'].select_set(True)
bpy.context.view_layer.objects.active = bpy.data.objects['SpaceRider']
bpy.ops.object.parent_set(type='OBJECT', keep_transform=False)


#Read file with camera parameters
with open(Camera_data_input) as file:
    file.readline()
    resolution=file.readline().replace(':', '').replace('x', ' ')
    resolution=resolution.split()
    resolution_x=float(resolution[1])
    resolution_y=float(resolution[2][:-1])
    pixel_dimensions=file.readline().replace(':', '')
    pixel_dimensions=pixel_dimensions.split()
    pixel_dimensions=float(pixel_dimensions[2])
```

```
    focal_length=file.readline().replace(':', '')
    focal_length=focal_length.split()
    focal_length=float(focal_length[2])
    aperture=file.readline()
    position=file.readline().replace(':', '')
    position=position.split()
    position=[float(d) for d in position[1:4]]
    orientation=file.readline().replace(':', '')
    orientation=orientation.split()
    orientation=[float(k) for k in orientation[1:4]]

######## Camera attached to SROC ##########
camera_data = bpy.data.cameras.new(name='Camera SROC')
camera_object = bpy.data.objects.new('Camera SROC', camera_data)
bpy.context.scene.collection.objects.link(camera_object)
camera = bpy.context.scene.objects['Camera SROC']
camera.data.lens=focal_length
camera.parent = bpy.context.scene.objects["SROC"] # Setting SROC as parent tells it
to look at its local reference frame
camera.location[0] = position[0] # Modifying location in S/C local reference frame
camera.location[1] = position[1]
camera.location[2] = position[2]
bpy.data.objects['Camera SROC'].rotation_mode='XYZ'
bpy.data.objects['Camera SROC'].rotation_euler=orientation
bpy.data.objects['Camera SROC'].data.clip_start = 0.004
bpy.data.objects['Camera SROC'].data.clip_end = 1e+09

#Import SROC path
coord=[""]
coords_SROC=[]
delta_t=[0] # for Earth rotation
n_points=file_len(SROC_in_SR_file_path)

frames_old=0
frames=[0]*(n_points-1)
with open(SROC_in_SR_file_path) as file:
    for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
                        coord=[float(line[6])*scale,   -float(line[7])*scale,   -
float(line[8])*scale]
            coords_SROC=coords_SROC+[coord]
            if i == 1:
                if line[1]=='Jan':
                    line[1]=1
                    delta_mese=0   # Per il calcolo di alfa_g_0
                if line[1]=='Feb':
                    line[1]=2
```

```
                    delta_mese=31
                if line[1]=='Mar':
                    line[1]=3
                    delta_mese=31+28
                if line[1]=='Apr':
                    line[1]=4
                    delta_mese=31+28+31
                if line[1]=='May':
                    line[1]=5
                    delta_mese=31+28+31+30
                if line[1]=='Jun':
                    line[1]=6
                    delta_mese=31+28+31+30+31
                if line[1]=='Jul':
                    line[1]=7
                    delta_mese=31+28+31+30+31+30
                if line[1]=='Aug':
                    line[1]=8
                    delta_mese=31+28+31+30+31+30+31
                if line[1]=='Sep':
                    line[1]=9
                    delta_mese=31+28+31+30+31+30+31+31
                if line[1]=='Oct':
                    line[1]=10
                    delta_mese=31+28+31+30+31+30+31+31+30
                if line[1]=='Nov':
                    line[1]=11
                    delta_mese=31+28+31+30+31+30+31+31+30+31
                if line[1]=='Dec':
                    line[1]=12
                    delta_mese=31+28+31+30+31+30+31+31+30+31+30
                date1=[int(float(k)) for k in line[0:6]]
                 date_old=dt.datetime(date1[2],date1[1],date1[0],date1[3],date1[4],d
ate1[5])
                sec1=float(line[5])-date1[5]
                sec_old=sec1
               delta_data_alfa_g_0=(date1[2]-2000)*365+delta_mese+date1[0]+5 #Ultimo
termine per includere i giorni bisestili fino all'anno 2023 (anno di lancio)
                date1=date_old #per poter fare la differenza dopo
                alfa_g_0_star=280.46061837 #al 1° Gennaio 2000
                alfa_g_0=alfa_g_0_star+360.98564736692*delta_data_alfa_g_0
                while alfa_g_0>360:
                    alfa_g_0=alfa_g_0-360 # riporto in angoli compresi tra 0 e 360
            if i > 1:
                if line[1]=='Jan':
                    line[1]=1
                if line[1]=='Feb':
                    line[1]=2
                if line[1]=='Mar':
```

```
            line[1]=3
        if line[1]=='Apr':
            line[1]=4
        if line[1]=='May':
            line[1]=5
        if line[1]=='Jun':
            line[1]=6
        if line[1]=='Jul':
            line[1]=7
        if line[1]=='Aug':
            line[1]=8
        if line[1]=='Sep':
            line[1]=9
        if line[1]=='Oct':
            line[1]=10
        if line[1]=='Nov':
            line[1]=11
        if line[1]=='Dec':
            line[1]=12
        date=[int(float(k)) for k in line[0:6]]
        sec=float(line[5])-date[5]
        date = dt.datetime(date[2],date[1],date[0],date[3],date[4],date[5])
        delta = (date-date_old).total_seconds()
        delta_t_alfa_g_0=(date-date1).total_seconds()
        delta_t=delta_t+[delta_t_alfa_g_0]
        frames[i-1]=frames_old+delta*frame_rate+frame_rate*(sec-sec_old)
        frames_old=frames[i-1]
        date_old=date
        sec_old=sec
        if i==n_points-1:
            if line[1]=='Jan':
                line[1]=1
            if line[1]=='Feb':
                line[1]=2
            if line[1]=='Mar':
                line[1]=3
            if line[1]=='Apr':
                line[1]=4
            if line[1]=='May':
                line[1]=5
            if line[1]=='Jun':
                line[1]=6
            if line[1]=='Jul':
                line[1]=7
            if line[1]=='Aug':
                line[1]=8
            if line[1]=='Sep':
                line[1]=9
            if line[1]=='Oct':
```

```
                        line[1]=10
                    if line[1]=='Nov':
                        line[1]=11
                    if line[1]=='Dec':
                        line[1]=12
                    date2=[int(float(k)) for k in line[0:6]]
                    sec2=float(line[5])-date2[5]

date2 = dt.datetime(date2[2],date2[1],date2[0],date2[3],date2[4],date2[5])
Simulation_time=(date2-date1).total_seconds()+(sec2-sec1)
Frames_number=frame_rate*Simulation_time
bpy.context.scene.frame_end = Frames_number
print(Frames_number)
print(frames)
omega_Earth=float(7.29*10**-5)
alfa_g=[]
for i in delta_t:
    alfa_g=alfa_g+[float(alfa_g_0*math.pi/180+omega_Earth*i)]


#Earth orientation from Space Rider coordinates in ICRF
alfa=[]
Lat=[]

with open(SpaceRider_in_ICRF_file_path) as file:
    for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
            coord=[float(d)*10 for d in line[6:9]]
            rx=coord[0]
            ry=coord[1]
            rz=coord[2]
            r_xy=math.sqrt(rx**2+ry**2)
            r=math.sqrt(rx**2+ry**2+rz**2)
            a=rx/r_xy
            if a>=0:
                if ry>=0:
                    alfa1=acos(a)
                if ry<0:
                    alfa1=2*math.pi-alfa1
            if a<0:
                if ry>=0:
                    alfa1=acos(a)
                if ry<0:
                    alfa1=2*math.pi-alfa1
            alfa=alfa+[alfa1]

            b=r_xy/r
```

```
            if rz >=0:
                Lat1=acos(b)
            if rz<0:
                Lat1=-acos(b)
            Lat=Lat+[Lat1]

Lon=[float(alfa[i])-float(alfa_g[i]) for i in range(len(alfa))]
print(Lon)
print(Lat)

# Calculate Earth rotation along Space Rider Z-axis to align Space Rider to path
delta_Lon=Lon[len(Lon)-1]-Lon[0]
delta_Lat=Lat[len(Lat)-1]-Lat[0]
lato_Lon=6371*abs(delta_Lon)
lato_Lat=6371*abs(delta_Lat)
Earth_Z_rotation=atan(lato_Lat/lato_Lon)
print(Earth_Z_rotation)

# Rotate Earth to simulate orbit inclination
bpy.data.objects['Earth'].rotation_euler[2]=Earth_Z_rotation

#Import SROC attitude
rot=[""]
rot_SROC=[]
    #Attitude of body in ICRF
with open(SROC_attitude_in_SR_file_path) as file:
       for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
            rot=[float(d) for d in line[6:10]]
            rot_SROC=rot_SROC+[rot_1]

#SpaceRider SUN LIGHT

# create light datablock, set attributes
light_data = bpy.data.lights.new(name="SpaceRider LIGHT", type='SUN')
light_data.energy = 30

# create new object with our light datablock
light_object = bpy.data.objects.new(name="SpaceRider LIGHT", object_data=light_data)

# link light object
bpy.context.collection.objects.link(light_object)

# make it active
bpy.context.view_layer.objects.active = light_object

# Import SUN LIGHT direction
```

```
coord=[""]
coords_Sun=[]

with open(SpaceRider_LIGHT_in_SR_file_path) as file:
    for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
            coord=[float(line[6]), -float(line[7]), -float(line[8])]
            coords_Sun=coords_Sun+[coord]

# Import Earth path
coord=[""]
coords_Earth=[]

with open(Earth_in_SR_file_path) as file:
    for i, line in enumerate(file):
        if i>0:
            line=line.replace(':', ' '). replace(',', ' ')
            line=line.split()
                        coord=[float(line[6])*scale,    -float(line[7])*scale,    -
float(line[8])*scale]
            coords_Earth=coords_Earth+[coord]

# Append Sun
inner_path = 'Object'
object_name = 'Sole'

bpy.ops.wm.append(
    filepath=os.path.join(Sun_path, inner_path, object_name),
    directory=os.path.join(Sun_path, inner_path),
    filename=object_name
    )


# Key Framing attitude
bpy.data.objects['SpaceRider'].rotation_mode = "QUATERNION"
j = 0
for i in rot_SR:
    bpy.data.objects['SpaceRider'].rotation_quaternion[0] = i[3]
    bpy.data.objects['SpaceRider'].rotation_quaternion[1] = i[0]
    bpy.data.objects['SpaceRider'].rotation_quaternion[2] = i[1]
    bpy.data.objects['SpaceRider'].rotation_quaternion[3] = i[2]
    quat=mathutils.Quaternion((i[3], i[0], i[1], i[2]))
    SR_RF=quat.to_matrix()
     bpy.data.objects['SpaceRider'].keyframe_insert('rotation_quaternion',  frame  =
frames[j])
    # SROC
```

```
            rot_SROC_eul=euler_from_quaternion(rot_SROC[j][0],     rot_SROC[j][1],
rot_SROC[j][2], rot_SROC[j][3])
    bpy.data.objects['SROC'].location=coords_SROC[j]
    if j==len(rot_SR)-1:
        print(SR_RF @ Vector(coords_SROC[j]))
    bpy.data.objects['SROC'].keyframe_insert('location', frame = frames[j])
      bpy.data.objects['SROC'].rotation_euler=rot_SROC_eul[0],  -rot_SROC_eul[1],  -
rot_SROC_eul[2]
    bpy.data.objects['SROC'].keyframe_insert('rotation_euler', frame = frames[j])
    # Earth
    bpy.data.objects['Earth'].location=coords_Earth[j]
    bpy.data.objects['Earth'].keyframe_insert('location', frame = frames[j])
    bpy.data.objects['Earth'].rotation_mode='XYZ'
    bpy.data.objects['Earth'].rotation_euler[1]=Lon[j]
    bpy.data.objects['Earth'].rotation_euler[0]=-Lat[j]
    bpy.data.objects['Earth'].keyframe_insert('rotation_euler', frame = frames[j])
    # Sun
    direction = SR_RF @ Vector(coords_Sun[j])
    bpy.data.objects['SpaceRider LIGHT'].rotation_mode = 'QUATERNION'
              bpy.data.objects['SpaceRider      LIGHT'].rotation_quaternion      =
direction.to_track_quat('Z','Y')
       bpy.data.objects['SpaceRider   LIGHT'].keyframe_insert('rotation_quaternion',
frame = frames[j])
    # Sole
    bpy.data.objects['Sole'].location=direction
    bpy.data.objects['Sole'].keyframe_insert('location', frame = j)

    j += 1

# Divide objects in collections
bpy.data.collections.new('SpaceRider, Earth, Sun')
bpy.context.collection.children.link(bpy.data.collections['SpaceRider, Earth,
Sun'])
bpy.data.collections.new('Sole')
bpy.context.collection.children.link(bpy.data.collections['Sole'])
bpy.data.collections['SpaceRider, Earth,
Sun'].objects.link(bpy.data.objects['SpaceRider'])
bpy.data.collections['SpaceRider, Earth,
Sun'].objects.link(bpy.data.objects['Earth'])
bpy.data.collections['SpaceRider, Earth,
Sun'].objects.link(bpy.data.objects['Atmosphere'])
bpy.data.collections['SpaceRider, Earth,
Sun'].objects.link(bpy.data.objects['Clouds'])
bpy.context.scene.collection.objects.unlink(bpy.data.objects['SpaceRider'])
bpy.context.scene.collection.objects.unlink(bpy.data.objects['Earth'])
bpy.context.scene.collection.objects.unlink(bpy.data.objects['Atmosphere'])
bpy.context.scene.collection.objects.unlink(bpy.data.objects['Clouds'])
bpy.data.collections['Sole'].objects.link(bpy.data.objects['Sole'])
bpy.context.scene.collection.objects.unlink(bpy.data.objects['Sole'])
```

```
####### Render parameters #######

bpy.context.scene.render.resolution_x = resolution_x
bpy.context.scene.render.resolution_y = resolution_y

Output_path=os.getcwd()[:-10] + 'Results\\'
Output_path=Output_path + Name_file_output
bpy.context.scene.render.filepath = Output_path #Path of output file
bpy.context.scene.render.image_settings.file_format = 'AVI_JPEG'

# Maintain SROC camera pointed on Spacerider

for i in bpy.data.objects:
    i.select_set(False)
camera=bpy.data.objects['Camera SROC']
pointing=camera.constraints.new(type = 'TRACK_TO')
pointing.target=bpy.data.objects['SpaceRider']
pointing.track_axis='TRACK_NEGATIVE_Z'
pointing.up_axis='UP_X'
```

A.2 Example of coordinate file (.csv)

Time (UTCG),"x (km)","y (km)","z (km)","Velocity x (km/sec)","Velocity y (km/sec)","Velocity z (km/sec)"						

A.3 Example of camera parameters file

Resolution: [pixels]	
Pixel dimensions:	
Focal length: [mm]	
Aperture:	
Position: (x, y, z)	
Orientation: (x, y, z)	

References

[1]:https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/New_Cubesat_Mission_for_Inspection_Developed

[2]: https://www.asi.it/trasporto-spaziale/space-rider/

[3]: https://www.esa.int/Enabling_Support/Operations/IXV_mission_timeline

[4]: https://www.esa.int/Enabling_Support/Space_Transportation/Space_Rider

[5]: https://www.cosine.nl/cases/hyperscout-2/

[6]: Introduction to Space Simulators, Gengis Kanhg Toledo;

https://www.researchgate.net/publication/331288019_Introduction_to_Space_Simulators

[7]: Machine learning in space: extending our reach, Amy McGovern, Kiri L. Wagstaff; https://link.springer.com/content/pdf/10.1007/s10994-011-5249-4.pdf

[8]: Image Processing Algorithms For Deep-Space Autonomous Optical Navigation, Shuang Li, Ruikun Lu, Liu Zhang, Yuming Peng; https://www.cambridge.org/core/journals/journal-of-navigation/article/image-processing-algorithms-for-deepspace-autonomous-optical-navigation/9998B9ECC6C60D3B3CF38C30988EE1D1

[9]: Development of a High Fidelity Simulator for Generalised Photometric Based Space Object Classification using Machine Learning, James Allworth, Lloyd Windrim , Jeffrey Wardman, Daniel Kucharski , James Bennett , Mitch Bryson

[10]: An Anisotropic Phong BRDF Model, Michael Ashikhmin, Peter Shirley

[11]: A Scalable and Production Ready Sky and Atmosphere Rendering Technique, Sébastien Hillaire

[12]: Real Time Atmosphere Rendering for the Space Simulators, Radovan Josth

[13]: https://spaceplace.nasa.gov/review/posters/GOES_clouds/goes-r-cloudsat-front-8x11.pdf

[14]: https://en.wikipedia.org/wiki/K%C3%A1rm%C3%A1n_line

[15]: https://www.grc.nasa.gov/www/k-12/airplane/atmosphere.html

[16]: https://visibleearth.nasa.gov/collection/1484/blue-marble

[17]: Display of The Earth Taking into Account Atmpospheric Scattering, Nishita, Sirai, Tadamura, Nakamae; http://nishitalab.org/user/nis/cdrom/sig93_nis.pdf

[18]: https://www.site24x7.com/it/tools/selettore-codice-colore.html

[19]: http://hyperphysics.phy-astr.gsu.edu/hbase/atmos/blusky.html

[20]: Modelling of Daylight for Computer Graphics, 2006, Thomas Kment, Michael Rauter, Georg Zotti;
https://www.researchgate.net/publication/228608165_Modelling_of_Daylight_for_Computer_Graphics

[21]: Physically-Based Real-Time Lens Flare Rendering, Matthias Hullin, Elmar Eisemann, Hans-Peter Seidel, Sungkil Lee

List of figure