OPTASAT: An Open-Source, Flexible Software Framework for Small Satellite Operations

by

Thomas Joseph Murphy III

SM, MIT, 2021
SB, MIT, 2019

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN SPACE SYSTEMS ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

Authored by: Thomas Joseph Murphy III
Department of Aeronautics and Astronautics
September 20, 2024

Certified by: Kerri Cahoy
Professor of Aeronautics and Astronautics, Thesis Supervisor

Accepted by: Jonathan P. How
R. C. Maclaurin Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

OPTASAT: An Open-Source, Flexible Software Framework for Small Satellite Operations

by

Thomas Joseph Murphy III

Submitted to the Department of Aeronautics and Astronautics
on September 20, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY IN SPACE SYSTEMS ENGINEERING

ABSTRACT

The unprecedented growth in access to space has created a corresponding growth in the number of spacecraft and the number of people operating spacecraft. This has meant that many of these operators are operating spacecraft for the first time. Gone are the days when the only operators of spacecraft were national governments, militaries, and massive corporations. The operators of small spacecraft today include many early-career individuals who need the tools to enable them to make strong decisions in the behavior of their spacecraft.

The tools for operating spacecraft are often overlooked by teams focusing on the spacecraft themselves, but these operating tools are critical for mission success. Spacecraft operations tools have not developed in a similarly low-cost, widespread fashion as the spacecraft have. The best tools for modeling and understanding the situation of a satellite in space remain locked behind high barriers to entry including high cost, long training, and complex interfaces. In the same way that satellites have gone from the size of automobiles to the size of toasters, the software for operating them needs to go from expensive, complicated, high-performing suites to simple, flexible, approachable options that are accessible to the democratized space operators. New spacecraft operations staff need straightforward, direct interfaces which give them the knowledge of where their spacecraft is, where it will be, and what it will be able to do, and they need to know when all the options at their disposal are viable. Operators also need to be given the capability to adjust their software in whatever ways are necessary to tailor it to the particular parameters of their missions, to reflect the incredible variety of spacecraft and missions that exist today.

A gap exists in spaceflight software. Users need software that can perform their mission planning tasks in the short term and to inform them of the upcoming parameters of their spacecraft which concern them, whether this is the spacecraft's location, solar illumination, orientation, or any other property which is relevant to their particular mission. This software must also allow the users to be aware of the expected output of their sensors, especially imaging sensors, such that they may have an understanding of what they are imaging and what it ought to look like. Finally, this software must be open-source, enabling the user to audit the software and make changes to the software to customize it to their preferences, which may differ from anything the original software developer could have imagined. Such spaceflight software does not yet exist.

This dissertation develops and presents OPTASAT, the Open-source Python Tool for Awareness of Spacecraft and Analysis of Telemetry, which provides an extensible, modular

interface for incorporation of multiple tools which contextualize spacecraft data in a manner which maximizes usefulness for the operators. A priority is visualization of data to facilitate rapid understanding and distillation of the complexity of a spaceflight operation. This software has been released as a fully-featured, open-source software toolkit which performs the mission analysis components deemed most crucial to those who stand to benefit from it.

This software is intended to fulfill the needs of small spacecraft missions. Several particular application cases are studied, including that of an Earth Sensing mission, and Astronomy mission, and modeling communications for a real laser crosslink mission. These case studies are evaluated for their ability to present the relevant information to the operator. For Earth Sensing, this involves displaying information regarding the spacecraft's location with respect to the Earth, and enabling the selection of ground targets for imaging. For astronomy, the relevant information concerns the stars visible in the sky, and the spacecraft's relationship to sources of interference like the Sun and Moon. For the laser crosslink example, we study the operator's understanding of the spacecraft as they pass over a ground station and determine the operational configurations available for this communication.

OPTASAT fills gaps in the field. OPTASAT presents users with a tool which is flexible and intuitive to use for understanding data from spacecraft in a way that is not currently available in the offerings on the market. Additionally, it takes functionality that is currently available in proprietary paid software and makes it available for free, in an open source offering that is accessible to everyone. OPTASAT will allow spacecraft operators (especially those operating spacecraft for the first time) to confidently know the state of their spacecraft, enabling them to make the best decisions for their satellites. This will reduce barriers to entry and smooth the learning curve, reducing the amount of overhead to new spacecraft operators. OPTASAT will be yet another step in the ongoing process of making space more accessible to a larger pool of users.

Thesis supervisor: Kerri Cahoy
Title: Professor of Aeronautics and Astronautics

Acknowledgments

A complete acknowledgements section featuring all the individuals deserving thanks for this thesis would be too long for anyone to read; I will attempt to present an abridged version here.

My first thanks go to my advisor, Kerri Cahoy, who has supported my work for over 7 years now, and who I look forward to continuing to have as a source of expertise and assurance. Her thoughts and feedback have been incredibly valuable through this process and will continue to benefit me for my entire career. My time in her lab has set me up for everything coming next and will be a solid foundation for everything else I do from here.

Next, I acknowledge the contributions of my thesis committee, whose inputs shaped OPTASAT into what it came to be. The inputs they provided were critical to take my initial Python code and ultimately create a tool that can actually be useful to someone. Their ideas through this process are hugely appreciated. Naming them explicitly, thank you to Dr. Kerri Cahoy, Dr. Mary Knapp, and Dr. Richard Linares.

This work would not have happened without the experiences I was exposed to through the different projects I have worked on throughout my time as a graduate student. Chronologically, thank you to the MiRaTA team and MIT Lincoln Lab for trusting me with their ground station. Thank you to the DeMi project (especially Paula, Christian, Rachel, Kerri, Greg, Yinzi, Jenny, and John) for my chance to build and operate a satellite and dive straight into real spacecraft development. Thank you to BeaverCube (highlighting Paula, Kerri, Adam, Alex, and Alex) for the lessons in resilience through difficult times and the push to keep making things work. Most recently, thank you to CLICK for the lessons in large, long-term projects with many stakeholders.

That covers the primary acknowledgements for contributions directly toward the development of this thesis. Next, I acknowledge the support of the people who made it possible for me to do this.

First, I acknowledge my family for their confidence in me. As I reach my tenth year at MIT, I am glad to always have them behind me as I push toward whatever is next.

Next, thank you to Paula do Vale Pereira for her wisdom and kindness through everything I have struggled with in grad school. Your guidance has made me feel like success is possible, and I appreciate every bit of advice and insight. I am very glad that, through my time in this lab, I've created a friendship for life.

Next, for a bit of levity, I thank Joe Barnard, Xyla Foxlin, and James Dingley for welcoming me to provide a small amount of help on their various projects. They have been a breath of fresh air.

I thank the MIT Radio Society for being my main source of extracurricular social engage-

ments and for creating my basis for understanding of radios, which ultimately kickstarted my research career. Individually, thank you to Daniel, Mitch, Mary, Christian, Greg, and Jordyn for being the core members of the club when I initially joined it.

This brings me to my final acknowledgement - thank you Jordyn for your patience, kindness, reassurance, and accommodations through this whole process. You're my favorite person from the radio club and I hope to maintain this friendship for a long time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

There has been a large growth of new space systems being deployed by non-traditional entities. Where previously spacecraft were chiefly operated by major sovereign states or large companies (such as satellite television operators), it is now possible to deploy systems into orbit for comparably low cost [1]. This has led to a massive increase of groups, especially educational institutions, operating spacecraft with little to no prior experience [2]. These groups are entering new operating environments and handling the realities of spaceflight without the decades of flight heritage and mentorship available within organizations like NASA. They also lack the software tools for operating their missions, leading to most teams creating their own tools.

When operating a spacecraft, there is a large variety of telemetry that groups must handle. It can be difficult to manage all of the information the first time someone operates a spacecraft [3]. A simple example of this is the task of interpreting thermal data to determine whether the spacecraft is properly managing its temperatures. Most importantly, this data must be understood in the context of the solar environment - if a spacecraft is 60 degrees Celsius while in eclipse, this is much more concerning than if it were the same temperature in full sunlight. Groups may be handling data represented in forms they have never seen before, such as spacecraft attitude represented using quaternions. Being able to visualize the attitude represented by a quaternion may be helpful to understand the physical orientation represented by that quaternion.

Another common activity in spacecraft operation is identification and resolution of anomalies [4]. When a spacecraft misbehaves, it is essential to understand the circumstances the system is in in order to identify potential causes. If a flight computer has a single-event upset, and mission operators can quickly be aware that the spacecraft has passed through the South Atlantic Anomaly [5], this helps the fault resolution process by giving a reasonable, relatively mundane explanation for the upset, rather than generating worries about more major failures in the flight computer. Other operating parameters such as solar beta angles are essential for evaluating power and thermal data on a spacecraft [6]. By giving mission operators direct information about the operating state of the spacecraft in a way that makes insights clearer, instead of just providing a datastream, we can enable missions

to make decisions more accurately and more confidently.

Space missions exist in a territory where there are several common categories of vehicles, environments, and goals, but also a large degree of uniqueness from one mission to another. Many spacecraft, with the exception of constellations, are one-of-a-kind, and may be flying instruments of completely new types. It would be difficult for a single suite of software to fully account for every mission's needs. At the same time, however, there are enough similarities between missions that some types of tools are universally, or at least broadly, useful. As an example mission, we can look at NOAA's (National Oceanic and Atmospheric Administration) JPSS (Joint Polar Satellite System) satellites [7]. Their mission involves tracking weather systems on Earth. Like many other Earth-imaging spacecraft, their geographic location will always be a critical piece of information. We can consider geographic awareness to be broadly useful for many types of spacecraft doing any type of remote sensing. This means it would benefit many spacecraft operators to provide software that identifies the locations of their satellites. But there are other, more specialized items of interest that will be more bespoke mission-to-mission. Again looking at the JPSS satellites, their VIIRS (Visible Infrared Imaging Radiometer Suite) instrument goes through a monthly calibration which involves imaging the moon [8]. Moon imaging from low Earth orbit is not a common behavior among satellites, and therefore building in features to give awareness of the state of the moon (position, phase, whether it is behind the Earth from the satellite's perspective, etc.) would ultimately go unused by many mission operators. It is therefore important that software which is intended to be helpful to a wide range of missions is flexible and modular. It must include utilities which are common, so that different users are not re-writing the same functionality and duplicating work, and it must also be constructed in such a way that the unique needs of each mission can be addressed by the people responsible for that mission.

When developing software, particularly software that is meant to be configurable by the end user (as opposed to software that is developed, distributed, and meant to be used as a completed tool), there are generally two schools of thought in how to allow the user to interact with the software's capabilities [9]. One way is through the use of Application Programming Interfaces (APIs). An API provides means for a user (assuming they have the programming skills) to implement their own programs which will "talk to" the given program, and exchange data back and forth such that extra functionality can be added. For example, some flight simulators offer APIs where a user's own programs can ask for the aircraft's position, speed, throttle levels, etc., and then the user's program can request the flight simulator to apply control actions to the simulated aircraft [10]. The flight simulator can then have new functionality added. A user could code their own auto-pilot, even for a game which was originally developed to be only flown by hand. However, APIs have their limitations - namely, that the user can only develop functionality which interacts with the target program in ways that the target program's developers envisioned. If the flight simulator's API does not provide a function for changing the gravity in the simulation, then the user will not be able to change it with their external program. A user who wants to pretend to fly a plane on Mars would be unable to do this, if the API does not allow changing the gravity (and the density of the air, and any other parameter of interest). The other way to allow users to tailor software to their needs is with open-sourcing. Open-source software is software where the source code is provided to users, and they are able to see how the software operates "under the hood" [11]. Most open-source software is distributed with licenses which

allow the end user to modify the code to their own liking. Again, in our flight simulator example, if the flight simulator were open-source, a user would be able to find the constants in the code which define gravity, and simply change the numerical value. In order to allow users to modify code for the vast variety of space missions, it is essential that a tool which aims to serve all missions, and allow users to adjust things to their needs, be open-source.

1.2 Background

Of course, given the rapid growth in the space sector, many tools already exist which operators rely on for their missions. However, these tools all have limits which restrict the degree to which they can serve missions which aim to use them. For example, few of them are open-source, restricting the user's ability to extend their functionality. Others may serve highly specialized roles in operating a mission, but require the assistance of other tools to fully suit the needs of the mission. Additionally, many of the most capable tools come at high price points. At the most extreme, some pieces of mission operations and planning software can approach the budgets of entire small space missions. This becomes prohibitive for small teams operating a spacecraft for the first time.

Broadly speaking, we can divide mission planning into three scopes, categorized by the time range of a given plan. There is the large-scale planning that occurs when establishing a mission, selecting instruments, determining an appropriate orbit, and other activities needed to create the groundwork for a spacecraft. Then, there is smaller-scale planning, where mission operators determine in the short term what goals exist - an astronomy mission might choose a list of stars of interest that they would like to target in the near future. And finally, there is the immediate planning, where spacecraft operators select individual commands to send to the spacecraft over their communication link.

Of course, this last example of command software is the most directly attached to the physical mission, as it involves the low-level interface by which the human operators interact with their systems. The importance of sending commands means that robust software exists for this role. The existing options for this software will be discussed in further depth in the Literature Review (particularly section 2.2, but will be briefly explained here. One example is COSMOS, an open-source framework created by Ball Aerospace. It allows users to predefine commands, and sequences of commands, to be sent to a spacecraft. It also processes incoming telemetry packets and interprets the values encoded within, based on configuration files.

On the other end, in the large-scale planning space, tools exist as well, particular examples being STK (Systems Tool Kit, made by Ansys) and GMAT (General Mission Analysis Tool, made by NASA). STK is ubiquitous in the spaceflight industry and is used for many types of mission planning tasks, especially simulating orbital mechanics. These tools are extremely useful for planning a mission's orbital trajectory and any maneuvers involved in traveling anywhere in the solar system.

The issue comes in the space between these scopes of large scale and immediate planning. Mission-scale needs are well served by STK and GMAT, and second-by-second needs are served by COSMOS, but day-to-day planning is underserved. There is a notable gap in the space of tools that are helpful for short-term planning and contextualization of spacecraft

data to enable operators to make the best decisions in the scenarios they are confronted with. Operators must rely on either raw interpretation of numbers, or creating unique software from square one to plan the behavior of their missions. We have filled this void by creating an open-source software toolkit for operations teams to use on the day-to-day level.

1.3 Author's Experience

The initial inspiration for of OPTASAT comes from the DeMi mission [12]. DeMi (Deformable Mirror Demonstration Mission) was a 6U CubeSat which aimed to demonstrate a new type of space telescope, using a MEMS (Micro Electromechanical Systems) deformable mirror (which has heritage among ground-based telescope adaptive optics systems) to correct for wavefront errors and enable sharper imaging than is possible with traditional space telescopes. DeMi was the first demonstration of MEMS adaptive optics technology in space, and serves as a starting point for future large telescopes which may use this type of component.

During operations for DeMi, the team found that there were a large number of day-to-day tasks which required specialized software which was not available. For example, the team needed to not only know when the spacecraft would be passing over our ground station, but also which astronomy targets to image, the direction of passes across the sky, when it was in eclipse or in sunlight, and more. While proprietary software exists on the market to serve some of these purposes, nothing could address these needs in exactly the way the team wanted. Thus, individual Python scripts were created to do these jobs. One set of scripts generated overpasses which could be copied into the Google Sheets document that was used for scheduling. Another set of Python scripts would give a view of stars in the sky for selection of targets. Still another would generate spacecraft attitude quaternions to point to a given star. Others would tell us when the satellite was in eclipse, or generate a summary of the parameters of a given overpass for populating a pass log document. This wide array of scripts served the team's needs, and more scripts were created whenever new needs arose. Team members eventually became familiar with which version of many scripts to run at a given time, but this process was unwieldy, and presented a large barrier to entry for any new satellite operators who needed to learn what each script was, when to use it, and what kinds of input it needed.

After the end of the DeMi mission, when there was no longer a time crunch to get minimum viable tools as soon as possible, development began on OPTASAT, to replace all the scripts that the team needed, and ultimately to serve future teams, who will hopefully never need to create another ad-hoc script to serve the same tasks DeMi needed.

1.4 Primary Contributions

There are three primary contributions which will be explicitly enumerated. These will be referred to later as well, by these same numerical designators.

Contribution 1 is the development of an open-source Earth-orbiting satellite operation planning tool, which will have a strong emphasis on visualization of data and modularity. The goal is to maximize the ability of end-users to derive the largest degree of useful understanding

distilled from the raw data available to them. Making things as visual as possible, and as adjustable as possible, is a top priority.

Contribution 2 is the close integration of existing open-source utilities into this software, and building upon those utilities to achieve a cohesive understanding of the mission state. These include open-source tools for orbit dynamics, a star catalog, Geographic Information Systems (GIS) for Earth targeting, and keep-out restrictions for instruments.

Contribution 3 consists of functionality enabling users to export data from the software into a text file, which can then be used to drive any external analyses or systems. Most importantly, this data exporting utility is intended to allow data generated in the software to be used with real spacecraft systems. This way, exported data can be converted into commands for a spacecraft and can close the loop between a spacecraft's state, the actions taken as a result of that state, and the final state reached because of the action.

The sum of these three contributions will result in a fully-featured toolkit which will be ready for spacecraft operators to use in operations of their satellites. The tool will be sufficiently featured to have some common tools ready for use, and will also have the full capability for users to adjust in ways that the original software developer is unlikely to have imagined.

1.5 Thesis Organization

This chapter described the context behind the creation of OPTASAT. The next chapter consists of a literature review, analyzing the types of software which perform purposes adjacent to OPTASAT itself, illustrating the gap which OPTASAT seeks to fill. Then, in Chapter 3, we will discuss the approach behind creating OPTASAT and some of the motivating case studies, followed by Chapter 4 sharing a discussion of the software that came before it, and then Chapter 5 discussing the details of implementation of the software, followed by Chapter 6 discussing some of the implemented case studies of possible applications for OPTASAT. We will then finish with Chapter 7, sharing some ideas for future work which would serve to improve OPTASAT further.

Chapter 2

Literature Review

This literature review will be approached from three main angles, representing three of the primary capabilities our tool aims to deliver. These will demonstrate the existing landscape of available tools, and will make it clear that, while some tools represent some capabilities of the final software we envision creating, none capture the full scope of what is intended, and therefore a gap exists, which will be filled.

2.1 Sensor Data Simulation

One important capability of the software is the ability to simulate the output of a spacecraft sensor, so that operators can have an established awareness of what to expect the spacecraft to return, and thus have an understanding of whether the data they receive matches what would be predicted. Particularly, we are interested in creating simulations of imaging sensors, such that mission operators can choose when and where to collect data. If the spacecraft is commanded to take a photo of the ground, and the command is activated when the spacecraft is over featureless ocean, operators will get an image that may be less useful than one containing land features of interest. Therefore, it is useful for them to have a generated version of the image that may be expected from the spacecraft.

For this work, our interests are limited to passive optical imagers, with the potential to include multispectral imagers, infrared, and ultraviolet imaging. Other types of imaging such as hyperspectral sensing, radar, and lidar are considered out of scope for the present time and may be considered as part of future work.

DIRSIG

Few options were found in the literature pertaining to simulation of space sensor output. A key example of the technology that does exist is that of DIRSIG [13] (Digital Imaging and Remote Sensing Image Generation). DIRSIG is a tool created by Rochester Institute of Technology's Digital Image and Remote Sensing Laboratory. The main goal of DIRSIG is to simulate a variety of sensors being considered for a system in order to facilitate trade studies. DIRSIG performs complex physical calculations of the behavior of the light illuminating a scene (as in Figure 2.1), including ray-tracing. DIRSIG aims to give mission engineers a

strong sense of their system's performance prior to deployment. While a copy of DIRSIG was not obtained for the sake of this review, DIRSIG does not appear to offer integration with mission operations. It would not, for example, be able to predict a spacecraft passing over a given target and thereby generate an image of that target. It is also not clear if it is capable of simulating an arbitrary spacecraft attitude and specific sensor field of view geometries.

Figure 2.1: An example of a rendering of a scene (in this case, the Port of Tacoma) as created by DIRSIG's simulation [14]

Rendered.ai

While DIRSIG is intended to simulate a sensor's view of the world, other tools exist which create views of computer-generated scenes. These include Rendered.ai (a name shared by the tool, its company, and its website) [15]. Rendered.ai is intended for the sake of generating datasets, especially for training machine learning models. While the Earth has a finite set of structures in one arrangement, Rendered.ai can generate more structures, patterns of structures, and views of structures (for example, the view in Figure 2.2) in order to train models to increase their ability to classify and detect objects. Rendered.ai is offered in the form of a "Platform as a Service" (PaaS), and runs on Amazon Web Services. It is not advertised for use in operations of real space platforms.

Figure 2.2: A Rendered.ai simulated image of a ground-based scene. They also have offerings for Earth Observation image generation. [15]

MATLAB Simscape

MATLAB is software that needs no introduction, especially in the aerospace field. Its heritage is extensive, and its add-on packages (especially Simulink) make it appropriate for many system modeling needs. Simscape [16] is a MATLAB-based software utility for modeling of physical systems using physical models of individual components which are then able to interact with each other. Each component is defined with its dependent equations (for example, a motor can have a speed and torque which are dependent on the voltage, current, and mechanical load applied to it). By combining these components (like shown in Figure 2.3), Simscape can simulate the behavior of the system as a whole. While Simscape is not directly designed for space sensing applications, it may be useful for modeling some of the sensor subsystems present on a spacecraft and how they would operate in space.

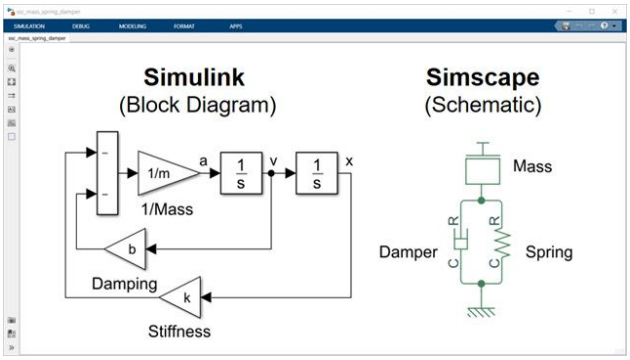

Figure 2.3: An example of Simscape juxtaposed with a traditional Simulink model of the same system, illustrating the components and their interactions. [16]

2.2 Mission-planning and operations

Spacecraft mission planning and operations has a broad scope and involves many factors that operators need to consider. Tools are available which serve to help with some portions of these operations. These tools can be categorized in several ways. One way is using the chronological phases - from mission architecture, to component design, integration, operations, and end-of-mission closeouts. Another division is by topic; for example, one category could be orbit simulation, another thermal modeling, and a third communications availability periods. Whatever way these portions of operations are divided, specific software exists to serve mission needs. Here we give an overview of some of the major offerings.

STK

One of the primary tools that dominates the space operations market is STK, now Systems Tool Kit [17]. STK is a commercial product which is distributed by Ansys. STK is the industry standard in spacecraft mission planning. It has high-quality simulation of orbits, propulsive maneuvers, Earth targeting, sensors, and many of the other tools a spacecraft operator needs. Some of these features are highlighted in Figure 2.4. While STK is highly

capable, the complexity that comes with it can be challenging for new operators. The company frequently offers dedicated trainings which are intended to give users the capability to operate the software, indicating that there is enough complexity that an average user may struggle to begin to use it. Additionally, it comes at a cost. A fully-featured professional STK installation with expansions is a non-trivial cost, though the exact pricing is not advertised publicly. The value of the tool comes from the years of equivalent engineering time that is required to build such a detailed and validated software tool. The value of expert software support is additionally high. STK is a fantastic tool with great capabilities, but the details of its implementation mean that it is generally suited to larger organizations, including governments, which are operating high-cost missions where STK's value comes from its time and cost effectiveness.

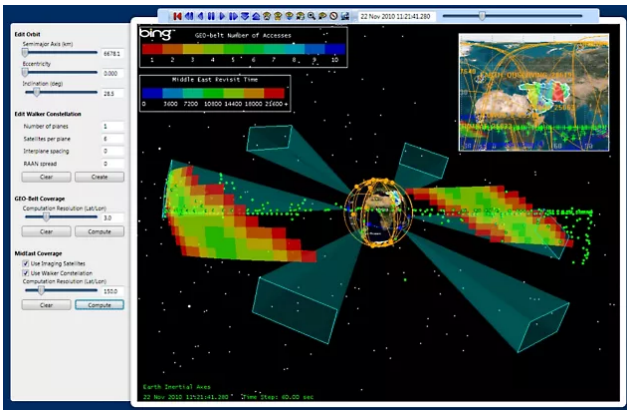

Figure 2.4: A screenshot of an STK scenario displaying an Earth imaging simulation of multiple spacecraft attempting to image a target. [17]

GMAT

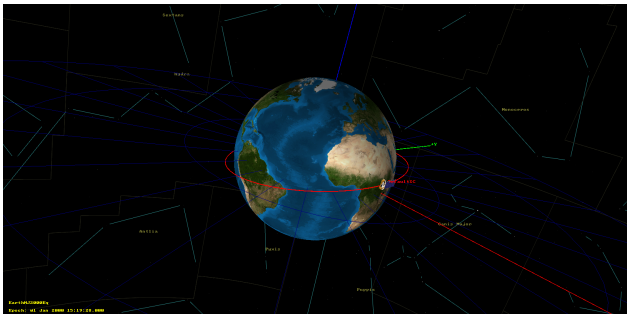

Figure 2.5: The default GMAT scenario, showing a spacecraft orbiting the Earth, along with axes, the Equatorial Plane, and the constellations.

Another mission simulation tool called GMAT exists [18], but GMAT differs from STK in a few notable ways. GMAT is an open-source spacecraft simulation tool developed by NASA Goddard. It's written in C++ and distributed through SourceForge. Its most recent release, R2022a, was made public in late January 2023, indicating active development.

GMAT uses a graphical interface to create visual renderings of spacecraft scenarios, such as the scenario in Figure 2.5. It can do some of the basic planning STK can (for example, simulation of orbits propagated over time with perturbations, and simulation of a spacecraft's access opportunities to a ground station), though does not fully replicate all the features. A full comparison of STK and GMAT is out of scope for this work. STK and GMAT both exist as large, mature tools that were developed by corporations and government agencies respectively, and are designed for use by groups of a similar caliber. Both STK and GMAT provide support for MATLAB integration through APIs, allowing users to write their own code to control the simulations.

Orekit

STK and GMAT both provide a strong user interface to operate spacecraft. While these are fully-featured software packages, there are also software libraries available on the market which provide similar types of orbital determination and propagation capabilities. One such example is Orekit (ORbits Extrapolation KIT) [19]. Orekit is an open-source Java library for processing spacecraft orbits. It can perform orbit determination from a set of observed points, propagate an orbit forward and backward in time (this capability is shown in Figure 2.6), predict spacecraft collisions, and manage spacecraft attitude. Orekit is available under an Apache 2.0 license, meaning that other individuals are allowed to use it and build other software around it without payment, as long as credit is given that Orekit was used in developing their software. Orekit, being a software library and not user-facing software, consists of a set of functions and routines which may be called by other software, including the software that would perform user interfacing.

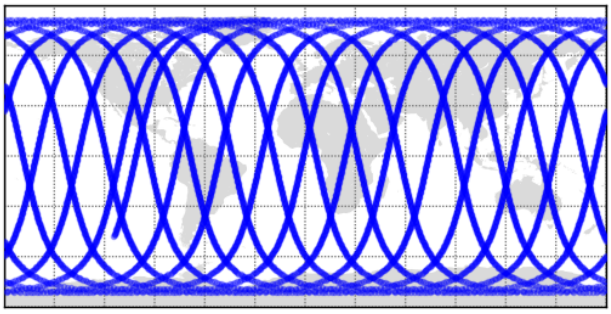

Figure 2.6: An Orekit simulation of the track of a spacecraft over various parts of Earth. [20]

Gpredict

Rounding out the selected examples for orbit prediction software is Gpredict [21]. Gpredict is a satellite tracking and orbit prediction tool which is intended for use by individuals seeking to keep track of a satellite's live location. Gpredict is oriented around TLE (Two Line Element Set) representations of satellites, and will show a user the satellite's location above the Earth at any given time. An example of the main screen of Gpredict is shown in Figure

2.7. Additionally, by providing Gpredict with the user's coordinates, Gpredict can identify when the satellite will be in the user's sky, and give a live readout of its position in the sky. This is especially useful for applications like controlling rotators to point communication antennas at a given satellite to perform mission data uplink and downlink. Gpredict is very well-suited to mission operations in real-time. It is a well-made tool for its purpose, but has a fairly specific purpose. For example, it cannot simulate when satellites have a line of sight to each other, and it is not suited for determining long-term evolution of an orbit, or simulating orbits that involve any type of maneuvers. It simply propagates a TLE and shows where that TLE predicts the spacecraft to be at a given time.

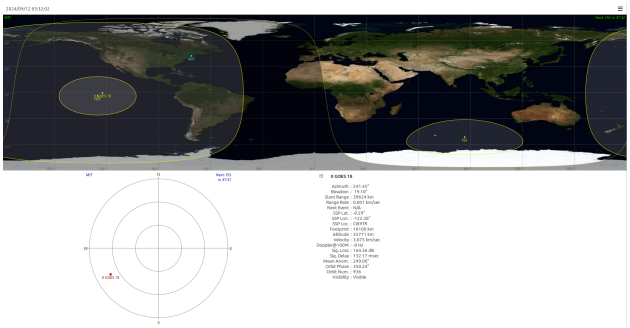

Figure 2.7: A Gpredict simulation of several spacecraft in orbit, with a map shown on the top and data below.

AMMOS

Moving on from the orbit simulation tools, there are many more tools available which center around mission operations and planning. The first tool of interest is a NASA tool called Advanced Multi-Mission Operations System (AMMOS) [23]. AMMOS is designed to be a flexible tool for use on many of NASA's missions, especially those exploring the solar system. It presents spacecraft data through a graphical interface shown in Figure 2.8. AMMOS uses a model where the tool includes Common Mission System Elements, those components which are broadly useful across NASA missions, to avoid duplication of effort, along with Customized Mission System Elements to fine-tune the capabilities for each mission's needs.

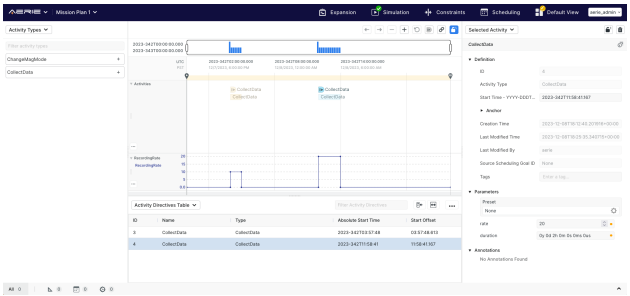

Figure 2.8: The output of a simple AMMOS scenario, displaying simulated data in a control panel interface. [22]

AMMOS is intended to be used through a full mission cycle, from the conceptual phase up through development, operations, and decommissioning. AMMOS is used heavily within NASA, but is not available to the general public without obtaining a special license.

Open MCT

While AMMOS is restricted to NASA's own missions, another NASA tool, Open MCT (Open-Source Mission Control Tool) [24] is not only available to the general public, it is also open-source. Open MCT runs on a web server, and is accessed through a browser. It is a visualization toolkit for data analysis, planning, and operation of space systems. An example of the type of visualization possible in Open MCT is shown in Figure 2.9. Several NASA missions, especially those based at the Jet Propulsion Laboratory like ASTERIA, JASON-3, and Mars 2020, use Open MCT. Because Open MCT is a browser-based tool, it is written almost exclusively in Javascript. Open MCT is actively developed on Github, and had its version 4.0.0 release on September 10, 2024. Notably, Open MCT is primarily concerned with the display of telemetry from a spacecraft, and not any type of simulation or analysis of orbits or other situational parameters.

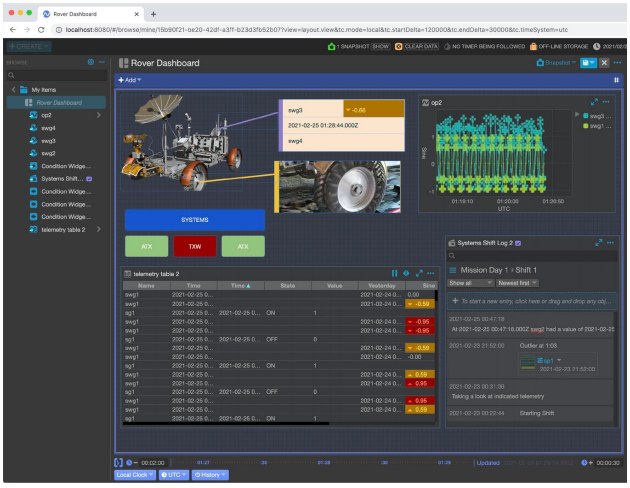

Figure 2.9: An Open MCT dashboard showing data from an imagined rover mission, displaying raw numerical data values as well as some representations of that data in visual interfaces. [25]

COSMOS

Several other mission analysis tools exist outside of NASA. One is COSMOS [26]. COSMOS was originally developed by Ball Aerospace, but has since been spun out into a new company, OpenC3. COSMOS is software which is meant to interface directly with mission communication systems. It takes downlinked packets directly from a ground station and parses their bytes into the data fields defined in the packet structure, before displaying these fields to the user (as shown in Figure 2.10). The user can then send commands which will similarly be built into uplink packets and sent to the ground station. COSMOS does not

perform processing or visualization of these values beyond parsing them from packets. It has a robust telemetry logging system allowing for data from previous communication sessions to be replayed. The original COSMOS software was completely free; modern COSMOS, under OpenC3, continues to offer the free software, which maintains updates and keeps the features of the previous versions, as well as an Enterprise version with extra features available for an annual fee.

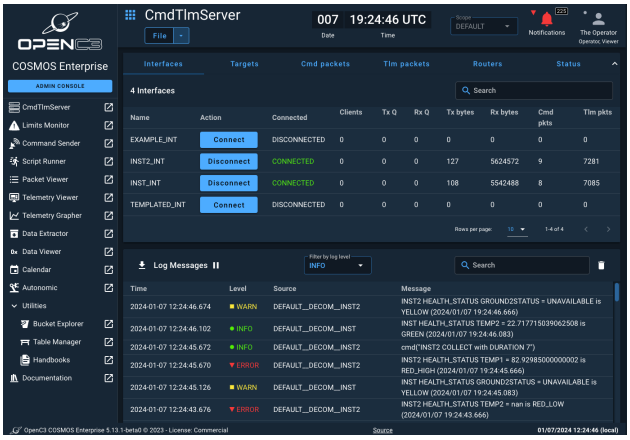

Figure 2.10: A COSMOS command and control interface displaying spacecraft telemetry, along with a window running a script to send commands to the spacecraft. [27]

SpaceTower

Commercial entities have also begun developing software for mission management. Exotrail has developed SpaceTower [28] which is offered in the Software as a Service model. SpaceTower is intended as an automated system which can process some of the needs of a mission

Figure 2.11: An interface window for SpaceTower, showing a constellation of satellites and their locations on Earth, as well as some orbital maneuvering information. [28]

control team, including contact planning, flight dynamics, and spacecraft monitoring. Their products are especially oriented toward constellation management. SpaceTower is advertised to be particularly well-suited for orbit handling, including trajectory and maneuver predictions. An example of the type of data presented by SpaceTower is shown in Figure 2.11. SpaceTower additionally provides an API and a Python wrapper, enabling it to interact with custom software written by the user.

Orbit Logic

Another company, Orbit Logic [29], offers similar services to SpaceTower. They offer a wide range of software meant to aid space situational awareness and mission planning. They have a heavy focus on Earth imaging. Their services range from image planning to networking and sensor tasking. They also have major integration with the aforementioned STK, and have mobile apps which give at-a-glance awareness of the space situation. Their flagship product, the Collection Planning and Analysis Workstation (depicted in Figure 2.12), is distributed as native software that runs on the user's computer, in contrast to SpaceTower's cloud-based approach.

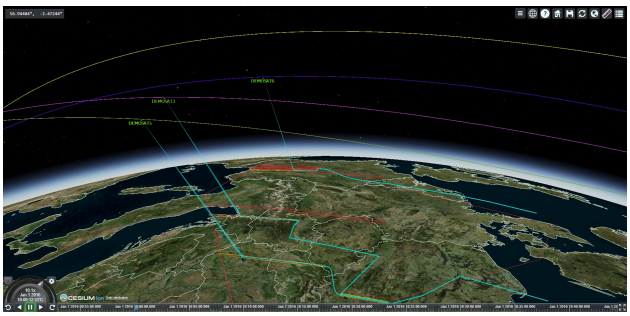

Figure 2.12: A demonstration of a scenario of Orbit Logic's Collection Planning and Analysis Workstation, showing a constellation of imaging satellties planning a multi-satellite imaging session of a region of interest. [30]

2.3 Open-Source

One of the key contributions of this work is that it will be open-source. This is a crucial trait because it ensures that any person in the future who wishes to expand on this work can do so directly by modifying the code to change it in any way they please. Some of the tools mentioned in the previous sections were open-source, namely GPredict, OpenMCT, COSMOS, GMAT, and Orekit, demonstrating a reasonably strong history of open-sourcing the tools created in this domain. In addition to the flexibility benefits granted by making the software open-source, there are many other advantages. One is that the software can never "die", in the sense that, if it is abandoned by its creator, anyone in the future can continue development on their own and potentially re-release the software themselves, rather than the software falling into obsolescence with nobody able to update it. Another is that users of the software can be assured that the software will never shut down. If a company is

being paid a monthly fee, that company could close its doors at any time, at which point the software is not available to run. When the software is open-source and runs on the user's own computer, the limitations on continuing to run it are purely technical (for example, computer crashes) and not legal. Finally, open-source software can greatly benefit from a community being established around it. Many open-source projects not only supply their code freely to the public, they also welcome the public to suggest edits to the code, at which point these modifications may be incorporated into the primary software and be redistributed to all the other users. This is especially important for OPTASAT, because users are expected to implement their own tools suiting their missions. Some of these tools will be useful to other missions, in which case the fact that a tool has already been made will allow new teams to benefit from the work done by those who came before them.

The open-source tools that have been mentioned in Section 2.2 may be, by their nature, used as ready-made code which can assist with the development of OPTASAT. For example, Orekit may become useful as a set of ready-made orbit propagation utilities which could then be used in this tool rather than re-writing everything manually. We will continue by mentioning other tools that are worth mentioning which may be beneficial as a basis for some of the capabilities envisioned in the finished toolkit.

Geant4

A useful open-source tool for simulation of high-energy particles and their interactions with matter is Geant4 [32]. Geant4 is a tool developed by CERN, the European Organization for Nuclear Research. While Geant4 is not designed for space operations, there are instruments which are well-suited for modeling in Geant4, especially instruments dealing with sensing of cosmic rays and other radiation sources. Geant4 is flexible enough to model many types of high-energy particle interactions, including the generation of secondary particles, as shown in the scenario in Figure 2.13, displaying a simulation of space radiation interacting with the International Space Station. Geant4 is also used as a basis for other open-source tools focused on radiation modeling, including MULASSIS [33], developed by the European Space

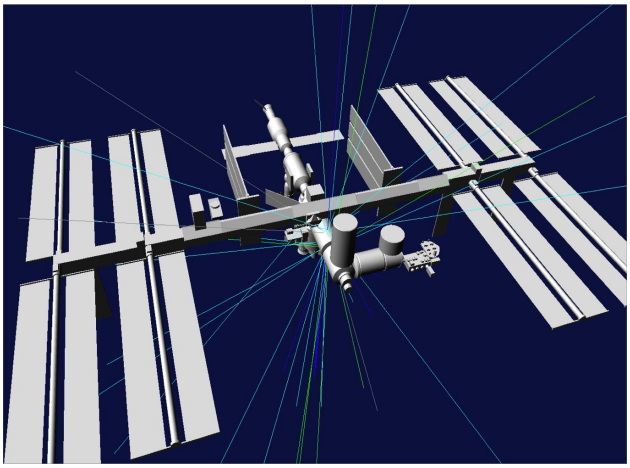

Figure 2.13: A Geant4-based simulation of a high-energy particle interacting with the International Space Station. Tracks of particles are observed in many directions. [31]

Agency (ESA). MULASSIS models the radiation received by a spacecraft, with a focus on predicting the performance of radiation shields. MULASSIS is also available through the ESA SPENVIS (Space Environment Information System) [34], a web-based collection of space environment modeling software. SPENVIS provides access to many different tools (including MULASSIS) for modeling of space sensors.

POPPY

Another useful simulation tool is POPPY (Physical Optics Propagation in PYthon) [35], which is meant to simulate optical systems. This includes the physics behind the refraction, reflection, and diffraction of light as it interacts with the optical elements included in a model. Ultimately, POPPY can simulate the image produced at the sensor of an optical system, aiding in the design and study of different types of optics. It was developed by the Space Telescope Science Institute for use in modeling for the James Webb Space Telescope (an example of its output for this purpose is shown in Figure 2.14), and like the other tools here, is open-source. It could be useful to integrate POPPY for the sake of generating predicted images for space telescope missions that may use OPTASAT.

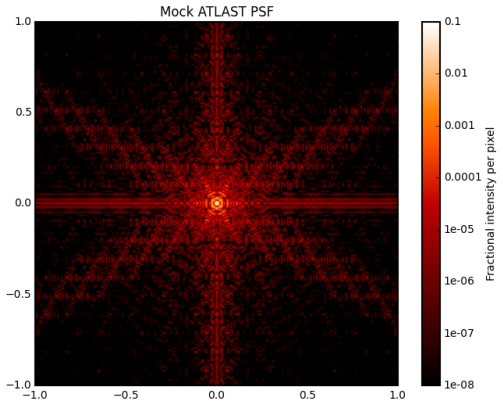

Figure 2.14: A POPPY model of the point-spread function of the James Webb Space Telescope [36]

CesiumJS

Finally, there are several open-source packages and services which may be beneficial to future development of OPTASAT, despite not being directly involved with sensor data or mission planning. The first of these is CesiumJS [37], a Javascript library dedicated to rendering high-quality 3D globes, such as the one in Figure 2.15. This would likely be extremely useful for generating renderings of a satellite or arrangement of satellites and contextualizing their positions over Earth. It provides a ready-made interface for high quality imagery, and leveraging the work already done by the CesiumJS developers could be a benefit to making OPTASAT more user-friendly.

Figure 2.15: A CesiumJS rendering of the Earth, the Sun, and some stars.

OpenStreetMap

Further resources useful in displaying Earth imagery include OpenStreetMap [38], a large project which solicits volunteers to edit a map of the Earth, showing the particular layout of every street, town, river, and other landforms. OpenStreetMap is operated by the non-profit OpenStreetMap Foundation and provides its data openly. While the most basic use of OpenStreetMap is to simply view a map of the land and roads (like in Figure 2.16), the data can also be parsed in different ways to highlight features of interest. Using OpenStreetMap data could be useful for simulating imagery captured by a spacecraft during an overpass, such that operators could ascertain whether a given imaging session should expect to photograph a particular town or body of water.

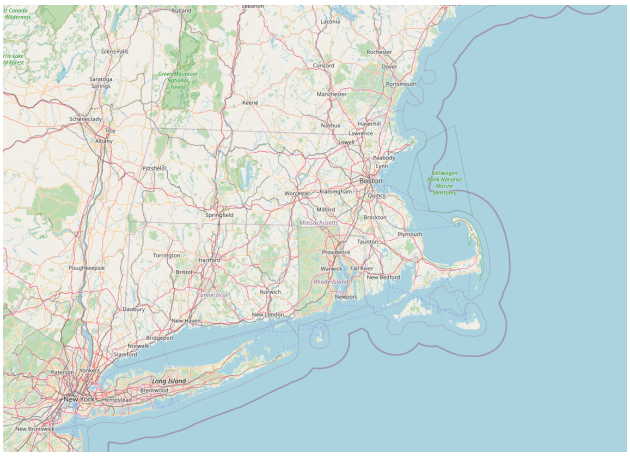

Figure 2.16: An OpenStreetMap image of Massachusetts and the surrounding area. Oceans, lakes, and forests, as well as artificial constructions like roads and cities, are visible.

Skyfield

For modeling of the locations of objects in space, a strong example of open-source software is the Python library Skyfield [39]. Skyfield is open-source and is primarily concerned with astronomy. It simulates the positions of stars, planets, and observers relative to each other

and performs all the relevant coordinate transformations to determine their views of each other in various systems. Skyfield performs simulation of the speed-of-light time delay for observation, and the refraction of the Earth's atmosphere. Skyfield also has built-in utilities for satellite simulation, meaning that it can take in a TLE, determine where a satellite is, and perform simulations of a satellite's ability to image stars and planets. This makes it extremely useful for space telescope missions. Because Skyfield can also model locations on the Earth, it can do all of the basic orbit propagation that OPTASAT needs, at least for version 1.0. For more advanced tasks like orbit determination from onboard GPS data, different software would be needed, but this will be left as a matter for future work.

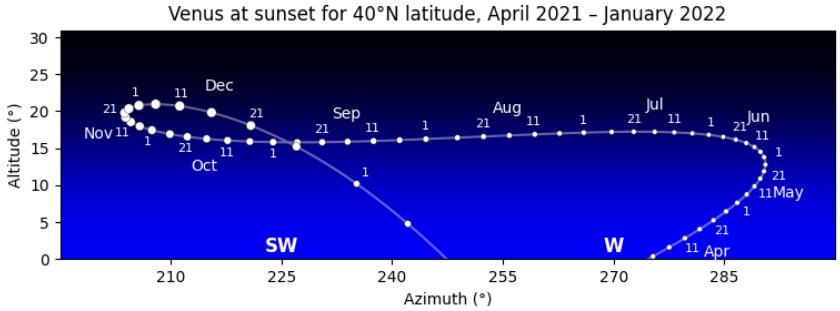

Figure 2.17: A Skyfield simulation of the location of Venus in the sky over time, from the point of view of an observer at 40 degrees latitude. [40]

PyCubed

PyCubed [42] is a project which presents a full open-source CubeSat bus system. Both the hardware and software designs are freely available on the web. PyCubed serves as a strong

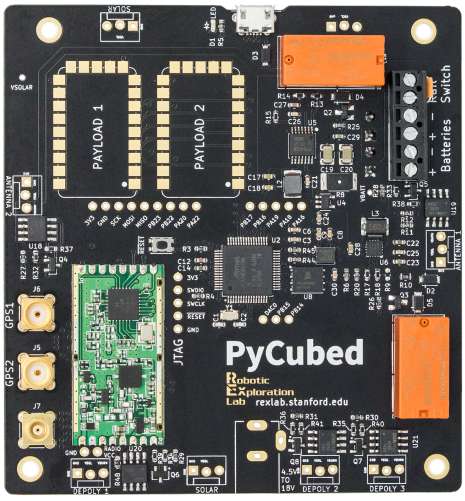

Figure 2.18: The main circuit board for the PyCubed project. This board hosts the primary spacecraft microcontroller, and interfacing hardware to connect to the rest of the systems of a spacecraft. [41]

example of open-source projects being successful in the field of small satellites. The main flight computer circuit board for PyCubed is shown in Figure 2.18. Multiple spacecraft have been flown using the PyCubed framework. Further development is ongoing, including hardware revisions to address issues in prior versions of the design. PyCubed represents open-source systems being successful on the spacecraft side, and complements the way that OPTASAT serves mission operations tools needed on the ground side.

2.4 Collected Literature Review

Figure 2.19 illustrates, at-a-glance, the known works which replicate some of the capabilities we seek to implement in this tool. It shows the overlaps between the three key traits (sensor data simulation, mission planning and operations, and open-source development style), and identifies that there is, at present, no offering on the marketplace for software which possesses all of these traits. This makes the technological gap clear, and motivates the development of the OPTASAT tool to close that gap.

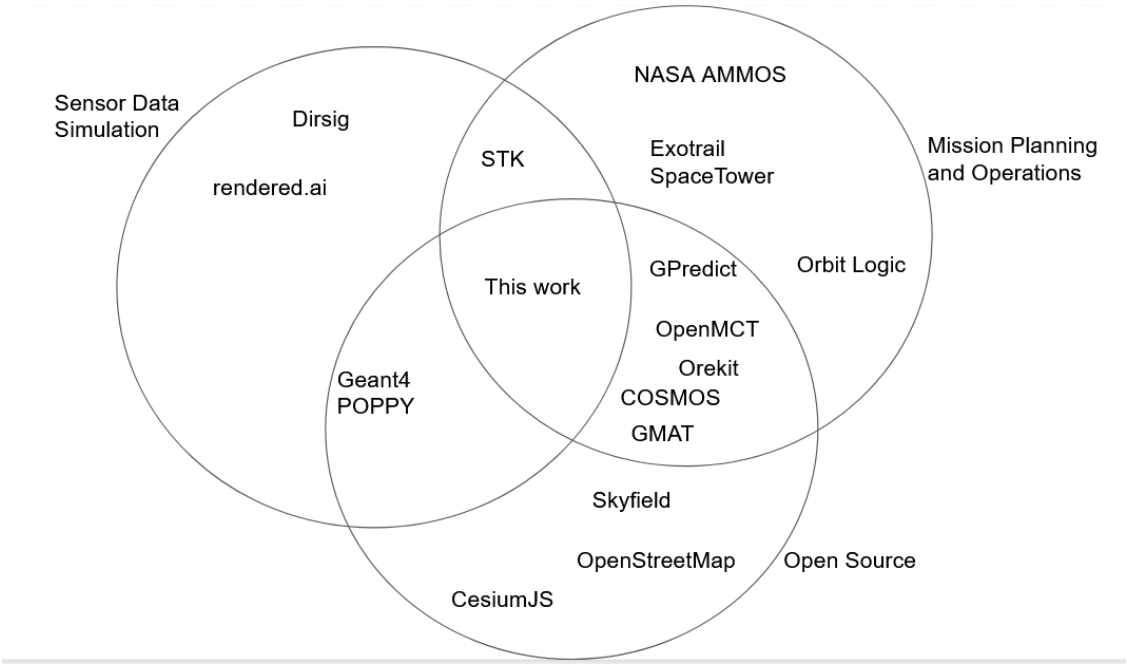

Figure 2.19: Illustrating the current state of the field and the gap which this tool aims to fill

2.5 Overall Capability Comparison

Previous sections have discussed particular purposes for software, and shared several examples of existing software that has capabilities adjacent to OPTASAT. In this section, we will present a few more software offerings which in some ways overlap with the capabilities of OPTASAT. We will then compare these offerings, along with a few that were previously mentioned, to OPTASAT, culminating in a single table which summarizes the capabilities of this cross-section of the software available on the market for spacecraft operations.

Basilisk

Basilisk [43] is a software tool developed by a team from the University of Colorado, Boulder. Basilisk is focused on simulation of astrodynamics scenarios. Basilisk provides modules describing parts of a spacecraft and the actions a spacecraft may take. The modules can then be combined in order to assemble a scenario, such as the Jupiter arrival scenario in Figure 2.20. The modules are combined in a flow-chart manner, similar to tools like LabView, where each module takes in data, processes it, and outputs new data to the next modules in the scenario. Basilisk provides extensive documentation to assist the user in using the software. Basilisk is open-source and allows the creation of custom modules.

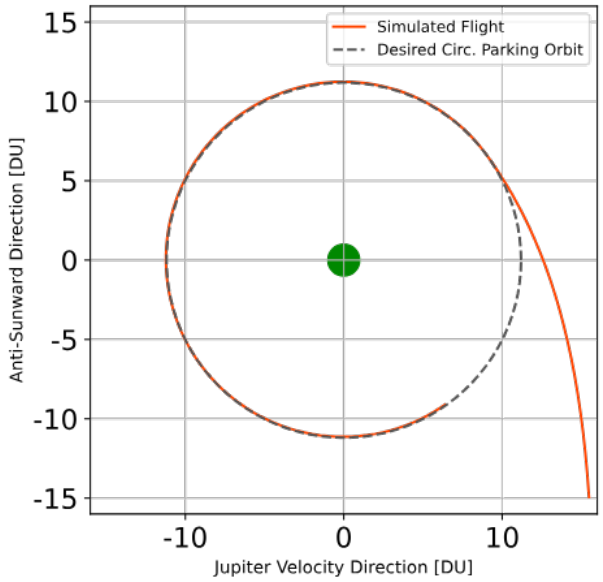

Figure 2.20: A Basilisk astrodynamics simulation of a spacecraft arriving at Jupiter before circularizing into a stable orbit around the planet. [44]

Basilisk modules can be written in C, C++, or Python. Once the individual modules are created (Basilisk comes with built-in modules, and allows the user to make their own modules to supplement the existing ones), Python is then used to describe the relationships between the modules and create a "Task", which describes a scenario in which the simulated

spacecraft will perform some operation. This means that the logic in the simulation is written separately from the logic of the individual modules.

FreeFlyer

FreeFlyer is a modern software suite for mission management, offered by a.i. solutions [45]. FreeFlyer, shown in Figure 2.21, presents a flexible astrodynamics simulation and flight dynamics system for various phases of flight, including design, analysis, and operations of a mission. It is designed to integrate into other aspects of an operations toolchain, including connecting with the mission's ground communication systems. FreeFlyer is commercial software, although a free demo is available. FreeFlyer presents a runtime API which allows integration with multiple programming languages, including C, Java, MATLAB, and Python. It supports Windows and Linux operating systems.

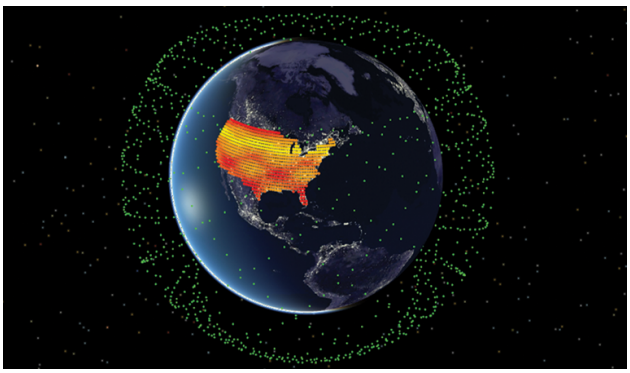

Figure 2.21: A FreeFlyer simulation of a cloud of satellites around Earth, with the United States highlighted as an area of interest for servicing or imaging. [45]

Table of Capabilities

Table 2.1 shows seven of the tools which closely overlap with OPTASAT, and shares each one's capabilities, as well as a few associated metrics such as cost. While this is not an exhaustive list of the capabilities of the tools, it clearly lays out that there is a niche that OPTASAT fits into which is not served by any other tools on the market. We also acknowledge that these tools have some capabilities that OPTASAT lacks, including maneuver simulation. Importantly, the extensibility of OPTASAT means that any capability it lacks could be worked on in the future, but this table represents the features which are operational in OPTASAT version 1.0. For the sake of readability, this table is duplicated in Appendix B, rotated 90 degrees to improve its use of the page.

Ultimately, while many types of software exist to serve some portions of the use cases we are interested in, there remains a gap. Most importantly, OPTASAT represents a mission simulation and planning tool which is available as open-source software at no cost. No other tool incorporates the physical state of a real spacecraft with orbital propagation and astronomy capabilities, and no other tool is written in pure Python to be highly approachable to the users. Many of OPTASAT's individual capabilities are represented in other software

Table 2.1: Comparison of OPTASAT to other space mission operations tools currently available on the market.

Property	Basilisk	COSMOS	FreeFlyer	GMAT	GPredict	JPL AMMOS	STK	OPTASAT
Simulate simple orbital trajectories around Earth	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Simulate complex trajectories to other bodies	Yes	No	Yes	Yes	No	Yes	Yes	No
Simulate propulsive maneuvers	Yes	No	Yes	Yes	No	Yes	Yes	No
Astronomy Target Simulation	No	No	No	Yes	No	Yes	Yes	Yes
Support for custom code interfacing M = MATLAB, P = Python, R = Ruby	P	P, R	M, P	M, P	No	No	M, P	P (Native)
Cost	$0	$0	$$	$0	$0	$0 (requires license)	$$$$	$0
Open-Source	C/C++	Ruby	No	C++	C	Some parts	No	Python
Live Spacecraft Data Integration	No	Yes	No	No	No	Yes	No	Yes
Export Controlled	No	No	Yes	No	No	Unknown	Yes (Some versions)	No

in one form or another (for example, Basilisk does trajectory simulation, but does not pull in the data from a real spacecraft; COSMOS shows the live data from a spacecraft but does not do planning, modeling, or simulation; STK shows the real spacecraft state and allows planning but is not open-source), but none of these tools can do all of what OPTASAT does. Thus, the gap is identified.

This gap motivates the development of the contributions. Chapter 3 discusses the contributions, and how they are fulfilled.

Chapter 3

Approach

3.1 Contribution 1 - open-source operations tool

To develop the core software functionality, Python was used. Python is a well-established language that has been in use and supported since 1991 [46], and has a strong history of being approachable, especially for beginners. This will make modifying OPTASAT familiar for the majority of users, such that they should have a straightforward experience creating and modifying the software components to serve their needs. Additionally, as an interpreted language, Python is highly cross-platform. Python code does not need to be recompiled to run on each system, meaning the source code can be easily distributed and used by anyone who is able to install Python on their system. Python also has strong package management, such that libraries and dependencies can be installed quickly and easily, meaning deployment of the software simple and straightforward, without any compiling or building process.

One of the requirements in making a tool that will be broadly useful to the community is whether it has strong built-in capabilities. While end users can develop the particular capabilities they need, an attractive software offering needs to be one which will handle the basics for the user, and only leave them to make the things that are truly particular to their situation. Therefore, it is important to ascertain these most important capabilities. These capabilities will be found through a two-phase approach. The first phase is identifying standard operations workflows. For example, for the DeMi mission, which had the goal of astronomy imaging, operations involved identifying when the spacecraft would fly over the ground station, identifying whether the spacecraft was in an operational state, selecting an astronomical target, commanding the spacecraft to a particular orientation, taking a photograph of the selected target, and downlinking the data collected. After common workflows are understood, phase two consists of analysis to determine how operators could have their understanding of the situation enhanced. Operators should have straightforward tools to inform them of when they will have communications opportunities, to perform telemetry checks to indicate whether the spacecraft is ready to begin imaging (as opposed to being in any type of safe mode), identify candidate targets for imaging, calculate required orientations, and determine whether an image is successfully received. The more streamlining that can be built into these workflows, the more likely it is that the software will be adopted by users.

3.2 Contribution 2 - Integration of open-source tools for mission status awareness

While OPTASAT is intended to reduce duplicated work for its future users, it is also highly beneficial to reduce duplicated work taken in development of the tool itself. To this end, it is highly useful to leverage the existing marketplace of open-source tools which can perform the needs this tool has. A simple example of a need is that of orbit propagation. Nearly every space mission has a strong interest in knowing their position in space. In the short term, this is of high importance for knowing the spacecraft's position relative to communications stations, ground imaging targets, or the celestial sphere. In the longer term, missions (particularly in Low Earth Orbit) require an awareness of when they might expect to experience de-orbit and the end of the mission. While a custom orbit propagator could be written for these purposes, propagation of orbits is, broadly speaking, an understood problem, which has been solved by many other pieces of software in the past. Therefore, it is preferred to use an established library. For this contribution, the available packages that serve the required purposes (including orbit propagation, and any of the capabilities determined to be essential during the pursuit of Contribution 1) shall be compared against each other, and have one selected to serve these purposes in the finished tool. In addition to the orbit propagation example, other necessary inclusions will be that of a star catalog, Geographic Information Systems, instrument simulation, and management of keep-out restrictions.

Along with the incorporation of open-source tools into this software, this software should be compared against the existing tools available to operators, including STK and GMAT. While these tools are extremely useful and powerful, the goal would be to identify which of their capabilities is most useful in the context of mission awareness for operators, and therefore which capabilities are most useful to seek to implement. Ultimately, if a user can use OPTASAT to understand their mission without needing the background to understand and familiarize themselves with STK or GMAT, this contribution is achieved.

3.3 Contribution 3 - Data exporting utility

Once mission operators are familiar with the state of their spacecraft, they then act upon this state to result in a new state which achieves some desired objective. During initial commissioning, the spacecraft may boot into a default safe mode where science instruments are not operational. Mission operators may then run a status test on these instruments to ensure their health after deployment into space, followed by finally moving into an operational mode. During operations, they may need to command the spacecraft to orient itself to align instruments with a desired target, followed by activating the instrument. Generally, these tasks involve multiple steps, despite having a simple description. For example, the act of "Take ten pictures of the Nile Delta the next time you're flying over it", while being described in a single English sentence, involves many complex steps. The spacecraft needs to know the location of the Nile Delta, its own location, what it means to be "flying over", determine the angular offsets between the spacecraft and the target, engage its attitude control system, command that system to orient to the target, identify the amount of time involved

in the flyover, and send commands to the imaging instruments on a specified cadence. This complexity means users spend a large amount of time translating their desires from their own conception of the situation to the format that the satellite can process.

Many satellites are unique engineering systems, and each have their own required processes in order to result in a specified action. They may use different coordinate systems when defining their attitude control frames, or may have different prerequisites before a particular action can be undertaken. This means that the same broad desire of a spacecraft behavior may be implemented differently on different spacecraft, in ways that end users may prefer to have abstracted away. OPTASAT can take the specifics of a spacecraft's state, and distill it into the relevant parameters for ongoing operations. OPTASAT can then generate output files containing the relevant information for commanding, which can then be passed to other software for further analysis, or for converting into commands for the spacecraft.

The initial intent for Contribution 3 was to add command generation to OPTASAT, but upon review of the existing software for generating spacecraft commands, it was deemed that this is not an area that needs further innovation, and additionally, generating of commands is not directly in line with the other goals of OPTASAT. OPTASAT is focused on taking in information more than sending out commands. The data export utility was developed instead, such that existing command generation software (for example COSMOS) could take OPTASAT's output and make commands, and allow for other uses of OPTASAT's outputs for other purposes.

3.4 Contributions Evaluation

In order to demonstrate OPTASAT's effectiveness at making these contributions, several case studies have been developed to illustrate using the software to serve the needs of some potential future missions. First, we have developed subsystem-oriented case studies, where we will look at using OPTASAT to enhance mission awareness for three selected subsystems: Power, Communications, and Attitude Determination and Control Systems. Next, we examine OPTASAT's use for three particular prototype mission concepts: Earth Sensing, Astronomy, and Multi-Spacecraft Coordination. By developing OPTASAT's capabilities to suit these case studies, OPTASAT's development has remained focused on usefulness to real missions, and its capabilities have been developed to target its suitability to support the operators of these missions. Because these case studies cover a wide variety of missions and subsystems, OPTASAT's flexibility is demonstrated, and thus we can infer that OPTASAT can accommodate many other application cases beyond the ones explicitly demonstrated.

3.5 Contributions Conclusion

We have described the approach for achieving the goals outlined in the contributions. In the future, it is hoped that OPTASAT may be used on an operational spaceflight mission, truly proving its usefulness to real missions. OPTASAT has already been used in the planning stages of a mission; this use will be elaborated more in Chapter 6. We will next discuss the process of developing OPTASAT and the internal implementation details of its construction.

Chapter 4

Initial Work

Prior to the development of OPTASAT, the initial solution to the problems of mission status awareness were solved with individual, bespoke scripts. This chapter will describe those scripts, followed by the presentation of OPTASAT, highlighting its benefits in serving as cohesive, integrated software.

The primary lab working on CubeSats at MIT is the STAR (Space Telecommunications, Astronomy, and Radiation) Laboratory, led by Professor Kerri Cahoy, along with Dr. Paul Serra, Dr. Afreen Siddiqi, Dr. Ajay Gill, and Dr. Danielle Coogan. OPTASAT was developed in association with the STAR Lab. The earliest spacecraft developed in the lab were microwave radiometer sounding instruments intended for gathering new types of weather data. These included the MicroMAS-1 and MicroMAS-2 spacecraft [47], as well as MiRaTA [48]. Later, the lab launched DeMi, the Deformable Mirror Demonstration Mission, which operated adaptive optics in space for nearly two years [12]. More recently, BeaverCube [49] and CLICK-A [50] were unrelated missions developed in parallel, which shared a rocket launch. BeaverCube was an imaging spacecraft intended for sampling ocean temperatures through an infrared camera in space, with a secondary mission of demonstrating operation of a miniaturized electrospray ionic propulsion module. CLICK-A was a risk-reduction mission for the upcoming CLICK-B/CLICK-C mission; CLICK-A performed a one-way optical downlink to a ground station while CLICK-B/CLICK-C will demonstrate two-way optical crosslink between a pair of satellites, using a novel miniaturized laser communication terminal. This author joined the lab during the final assembly phases of DeMi, and therefore had the experience of operating DeMi in orbit. This operations experience motivated the development of software tools, culminating in OPTASAT.

4.1 DeMi Operations Tools

The DeMi mission was launched in February of 2020 and operated until March of 2022. During that time, the MIT team performed science operations and communications with the spacecraft until its deorbit. Because most of the team members had never operated a spacecraft before, there were mission parameters that were relatively unknown to the team and they did not have established workflows. Additionally, funds were not available for expensive software packages. Many individual Python scripts were developed to serve the

team's needs. This was the original inspiration for OPTASAT. Many of DeMi's needs were not unique to DeMi, and therefore it is likely that other teams have similarly developed custom software to serve these needs. Because this represents unnedessary duplicated work, the goal of OPTASAT is to reduce the work needed for future academic research spacecraft operations teams who are operating a spacecraft, such that they will not need to make as many of their own tools as the DeMi team did. We aim to create a single, cohesive tool, in contrast to DeMi's tools which consisted of individual scripts to serve dedicated purposes. It was found that this large set of scripts resulted in a struggle among the team members to remember the purpose of each script, or to remember that some of the scripts existed at all. The goal now is that combining these individual capabilities into a single toolkit will allow mission operators to be more familiar with the tools at their disposal.

Next, we will discuss two simple examples of the types of tools that were developed for DeMi, along with their strengths and weaknesses.

4.1.1 Overpasses

The first script of interest developed for DeMi was intended to predict the times that the spacecraft would be flying over the ground station (located on MIT's campus) and would therefore be available for contact. There are many tools which are capable of this, but there was a desire to create a tool that would prioritize data digestibility and use visualization to allow for a deeper understanding of the mission schedule. This goal persists through the development of the final OPTASAT toolkit. The output of DeMi's version of the overpass finding script is shown in Figure 4.1.

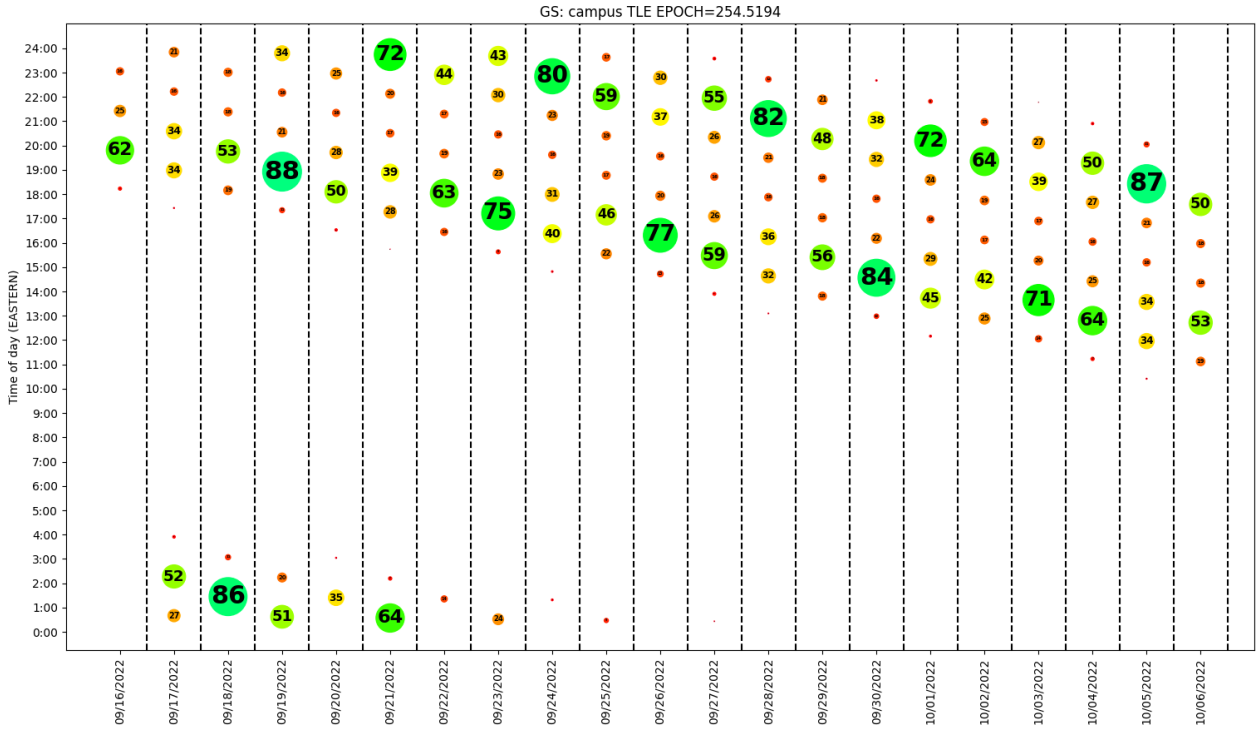

Figure 4.1: Pass prediction script developed for DeMi.

In this data visualization, each day is displayed as a column progressing from 12:01 AM at the bottom, noon in the center, and 11:59 PM at the top. Therefore, time progresses in a zig-zag, starting at the lower left, moving upward, then when it reaches the top, moving to the bottom of the next column to the right. Every circle within a column is a spacecraft overpass where DeMi will be flying over the MIT campus.

When the spacecraft flies over, it starts at the horizon, rises to a particular maximum elevation angle above the horizon, and then goes back down. This maximum angle is the most important driving parameter in judging the quality of a pass. A pass which rises above the horizon, reaches 10 degrees, and goes back down under the horizon is essentially useless for the MIT ground station's communication needs, while a pass at 90 degrees is flying directly overhead and presents a prime communication opportunity. Elevation angle is encoded into the data visualization. The size of each circle, the color of each circle, and the number on each circle all correspond to the elevation angle, with a high-elevation pass having a large, bright green circle and a low-elevation pass having a small red dot. This visualization was highly useful in operations, because it makes several trends immediately obvious. First, each day has approximately six passes. Second, each day has two main opportunities for passes, one at the beginning of the stretch of six passes and one at the end. Sometimes in these opportunities we will have two middle-grade passes (see the two 34's at the top of the second column), and other times there will be one stronger pass. Third, we see that the passes gradually move earlier in the day as the month progresses. Overall, this pass schedule visualization was helpful for the DeMi team's planning, and a variation of this tool is included in the OPTASAT software toolkit.

4.1.2 Astronomical Target Planning

As DeMi was an astronomy mission, selection of astronomical targets was an essential everyday activity. For this reason, scripts were developed that would identify which stars were good candidates for imaging. This needed to take into account many factors. The mission wanted to examine particularly bright stars, because of its relatively low light sensitivity. DeMi also needed to do its observing only when shaded by the Earth, to reduce the amount of stray light entering the telescope from the Sun. It was also desirable to avoid imaging stars that were too close to the Moon, especially when the Moon's phase was particularly full. Any star being imaged also needed to be in a portion of the sky not blocked by the Earth. These factors meant that the task of selecting a star was relatively complex.

Initially, a script was developed that started from a star catalog and combined these factors with weighting functions, such that a score could be established for each star, and a small list of stars with the highest scores would be produced. However, this proved to be difficult to manage, because finding accurate weights was not straightforward, and it was difficult for the human users to validate the script's output and judge whether the chosen star really was the best for the goals they had in mind. Ultimately, it was determined that a more visual representation of the data was needed. For this reason, a software utility which came to be known as the Star Map was developed. An example of this utility's output is shown in Figure 4.2.

In this view, the full celestial sphere, in Right Ascension and Declination coordinates, is shown as a plane in cartesian coordinates, meaning that the celestial poles are stretched to

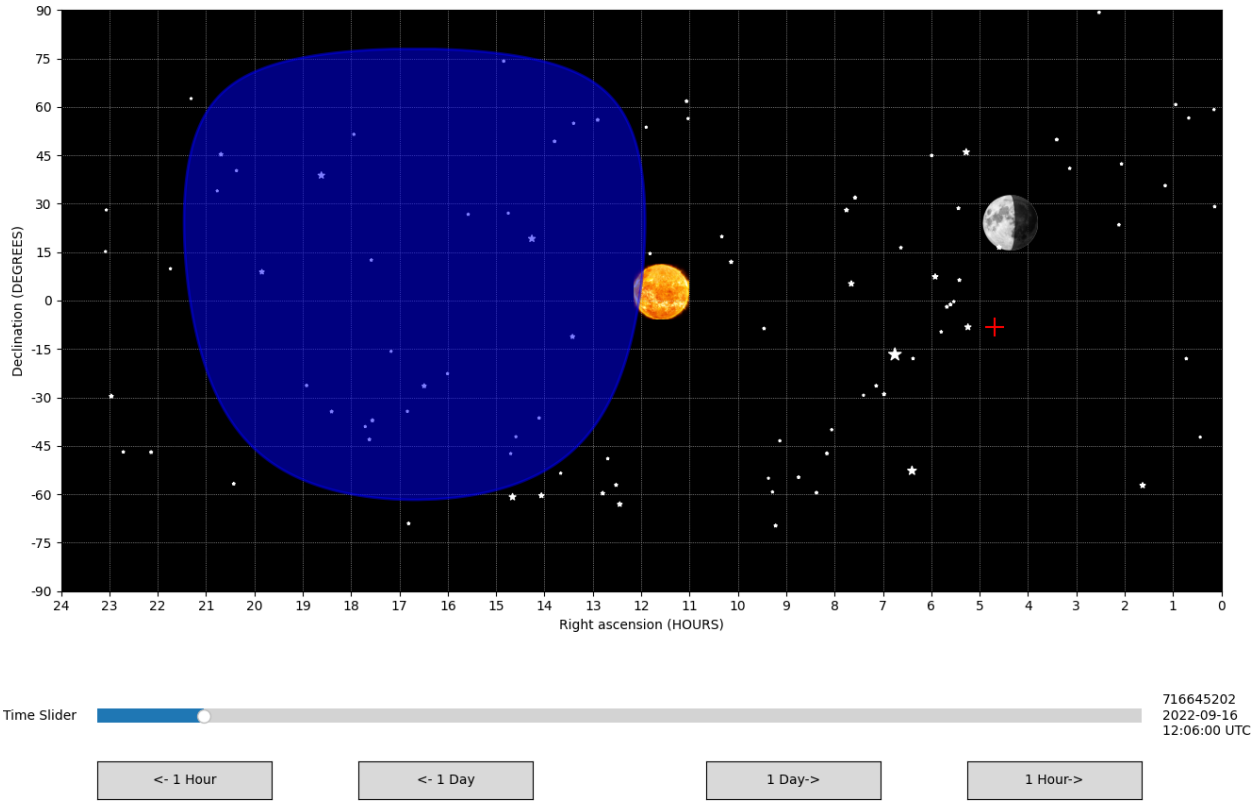

Figure 4.2: The Star Map created for DeMi

cover the entire top and bottom edges. All stars above a chosen magnitude threshold are displayed, and the stars are scaled according to their magnitude. Some familiar constellations and asterisms are evident, such as Orion near coordinates (6,0) and the Big Dipper around (12,60). Note that the coordinates on the X-axis are reversed in order to reflect the fact that Right Ascension runs positively to the left from a viewer's perspective. The Sun and Moon are displayed in their accurate locations at the chosen time, as well as an accurate moon phase being rendered on the displayed Moon. The time of evaluation is shown in the lower right, alongside a slider which the user can use to adjust the simulation's time, to see how the situation evolves with the passage of time. This way, if a particular star is to be imaged, the user can see when the next good opportunity to image that star will come.

The location of the spacecraft in the sky (or, more accurately, the position in the sky, from the spacecraft's perspective, pointing directly away from the Earth) is represented with a red +, which in this view is at coordinates (4,-10). Throughout the course of one orbit, this cross will move across the full width of the plot. The large blue shape on the left half of the display represents the Earth. Any star in this region is a star which is occluded by Earth, and therefore is not an option for imaging. The shape is irregular due to the distortions in the coordinate system mapping a sphere to a rectangle. By using the Star Map, the user can evaluate all of their concerns in target selection themselves. They can choose a star, and adjust time such that the spacecraft is in a good place to image that star. The Star Map also will display the name of any star that the user places their mouse cursor over. This became highly useful for selecting DeMi's targets, and would likely be useful for many other

types of astronomy missions, since many of DeMi's concerns will be common to others. For this reason the Star Map is a key component of the completed OPTASAT software.

4.2 DeMi Tool Limitations

While DeMi's operations tools (including the pass prediction script, and the Star Map script) served the mission's needs, the team quickly identified their limitations. The tools were not intuitive to use, and required long explanations to understand. Users had to clone a GitHub repository containing over twenty different Python scripts, find the one that would do the task they were looking for, configure local libraries to work, and attempt to get their results. The workflow required a large amount of memorization. Further, the scripts were not easy to understand. Because the tools were intended to serve an operational spacecraft mission, the priorities in development leaned more toward speed of development and less toward overall quality of the software. Code that was hard to read and maintain, but operational as soon as possible, was preferred over code that would work well in the long term. No purpose-built graphics libraries were used. These scripts were purely developed in Matplotlib, an extremely popular Python library for plotting graphs of data. Matplotlib is not intended for generating complex user interfaces, so it was being stretched to its limits in order to accommodate the needs of the mission. The Starmap script is over 300 lines of code with little commenting. Changing parameters is performed by directly modifying the Python code in place (rather than calling the script with arguments).

With DeMi, every custom software tool the mission needed was developed independently, meaning each was its own Python script. It became clear that it would be much more approachable if the tools were consolidated into one software toolkit, where the individual tools could work together, and where more tools could be integrated into the toolkit without needing to re-develop all the internal logic from scratch each time. This motivated the development of OPTASAT, which takes DeMi's individual scripts and transforms them into a cohesive single graphical interface for management of a wide variety of spacecraft tasks. The Starmap and Pass Finder utilities are each present in OPTASAT, in enhanced forms. For example, the previous Pass Finder was not interactive - passes were plotted into a static graph display. The full extent of its response to the user was the ability to zoom into the plot. OPTASAT's version introduces interactive mouse-responsive functionality which is described in detail in Section 5.4.4. The Starmap module is similarly upgraded compared to DeMi's, adding in the option to specify sensor keep-out zones as well as to search for stars of interest.

In Chapter 5, we go into detail on how OPTASAT is implemented and the internal structure of the software. We then briefly discuss each of the individual tools included in OPTASAT.

Chapter 5

Implementation

This chapter will describe some of the driving choices in the details of how to implement the software, followed by a description of the overall software structure, and, finally, a rundown of each of the individual software components that have been created.

5.1 Driving factors

Here we will discuss the important considerations that factor into our decisions about how to build the software. Most importantly, these include determining the scope of problems the software aims to solve and the programming language that is used. These two factors lead directly into the overall structure that is used to develop OPTASAT.

5.1.1 Scope

In determining the details of how to implement the software, a key factor is that of scope. It is not practical to develop a single piece of software to serve all the mission needs of every spacecraft, from small Earth-observing cubesats to the James Webb Space Telescope, the Voyager probes, or Mars rovers. It is essential to carefully define what scope this software aims to cover.

One of the most important scope-limiting traits we have is that of budgets. A mission with a billion-dollar budget is likely able to obtain many people with deep expertise in mission operations, along with the most expensive, highly-featured software on the market. Such high budget missions will not have a need for accessible, free, open-source software. We can then conclude that their needs are not as relevant for our decision-making because they are not the target audience.

Instead, we will tailor the software to the relevant missions. Low-budget missions usually will involve small spacecraft, and they will rarely go beyond low-Earth orbit. This allows us to make some simplifying assumptions. One is that the operating environment is heavily limited. We can assume the spacecraft is operating in an Earth-centered reference frame. Of course, this assumption, like all assumptions, is not universally applicable. Any assumption we attach to the nature of missions will be violated by some mission, but if the assumption is widely applicable, the benefits of the majority are deemed sufficient even if a minority are left

underserved. The hope is that the missions needing software which is slightly out-of-scope will be able to extend the software to serve those needs.

We will establish a small set of "reference missions", conceptual mission purposes which we will aim to serve. Broadly speaking, the majority of small satellite missions will fall into a few categories, and by serving these categories, missions can rely on the built-in features and extend upon them for whatever the particular mission's needs are. These reference missions include Earth-observation missions, astronomy missions, and multi-spacecraft missions. We will create enough material to act as a baseline for the universal tools needed for these satellite missions.

5.1.2 Programming language tradeoffs

In any programming project, the choice of what language (or languages) to use is a key one with far-ranging implications on the path of the project. In this case, the chosen language is Python, and this subsection will discuss the reasons for that choice.

Python is one of the most popular programming languages today, according to IEEE Spectrum [51]. Their 2024 popularity ranking places Python as the most popular language, at a normalized popularity of 1, followed by Java at 0.4855, JavaScript at 0.4451, and C++ at 0.3749. With Python being over twice as popular as the next runner-up, it becomes a clear choice as a strong option to maximize expected user familiarity. Further, Python has strong classroom use (for example, MIT's aerospace undergraduate programming requirement is "Introduction to Computer Science Programming in Python" [52]), meaning that many new spacecraft operators who are current or recent students can be expected to have experience with Python. Given that the software is intended to be modified by the users, choosing a language where the users are comfortable will maximize their ability to tailor the software to suit their needs.

The choice to use Python comes with tradeoffs. The most relevant tradeoff when using Python is its performance. As an interpreted programming language, each statement in a Python program is evaluated by the interpreter at runtime and then executed. In contrast, compiled languages like C, C++, or Rust are converted into machine code and run as a native executable on the target platform. Directly operating in the instruction set of the CPU means they can achieve much faster performance than Python. However, these languages are more cumbersome to use. Users need a stronger understanding of the internals of the computer, such as working directly with memory and the use of specific data types for different sizes of integers. In Python, memory is managed by the language, and all integers are collected under the umbrella "int" type, which dynamically changes size as needed. Overall, this means that a Python program is more approachable to write, but will be slower to run. With modern computer hardware continuing to grow in performance, we can support running Python programs which may consume more resources. We conclude that the performance tradeoffs are worth the benefits to the end user in having approachable, maintainable code.

The PyQT5 graphics library is used to create OPTASAT's graphics. This library handles creating a window and placing individual elements, known as widgets, into that window. Widgets exist for clickable buttons, sliders, geometric shapes, and more. Using PyQT5 allows OPTASAT to present a cohesive graphical interface that integrates with the user's system.

5.2 Software Structure

Here we will discuss the overall structuring of the software and how it is implemented.

5.2.1 Modular structuring

The interface of the software consists of a single consolidated graphical user interface. In this interface, individual units, known as modules, exist to provide each functionality that a given mission needs. These modules are each defined by a single Python file, which allows individual modules to be moved in and out of the interface. This also makes it easy for a user to modify any given module, since they are all self-contained in individual files. All modules are optional, and can be removed from the interface if a module is not useful for a given mission (for example, Earth-observing missions will have more Earth-focused modules, while astronomy missions would want to have modules for viewing astronomical targets). Any module can be inserted an arbitrary number of times, which is useful if, for example, the user wants to view the space situation from the perspective of multiple different satellites, or using multiple different ground stations. Each satellite or ground station can have another instance of the same module, and the individual modules will each operate independently. By allowing a single module to be used multiple times (once for each spacecraft), OPTASAT has a strong level of flexibility to accommodate for satellite constellations. Modules can be duplicated as many times as needed to account for each of the satellites being operated by a particular operator.

While modularization is a key feature of OPTASAT, this does not mean that the modules are completely isolated from each other. OPTASAT presents an internal data structure called "cross_module_vars", which is accessible to all modules. Any module may read or write to this Python Dictionary. Ultimately, this acts as a conduit for the different modules to pass data from one to another. The most important example of a cross-module variable is the internal time of the simulation. Most of the modules consist of displaying some state of the spacecraft, at a specified time. In order to keep all the modules synchronized, the time of the simulation is treated as a variable which all time-dependent modules can access. This also means that the simulation time can be controlled from any single module, and all other modules will react to it.

5.2.2 Visual design

OPTASAT's visual layout is highly configurable, but has some overall themes that apply no matter what layout is used. An example of a setup of OPTASAT is shown in Figure 5.1.

In this view, we can see eight modules represented. Each of the individual modules will be described in detail later; for now they are presented only for the purpose of filling the space and presenting a tangible example.

On the left side, we have three graphs which are plotting telemetry from each of the spacecraft's Control Moment Gyroscopes. These are three copies of the same module, running the same code, but configured with different parameters. Sharing the code between these modules allows for rapid adjustments of the software without needing to copy changes across different modules.

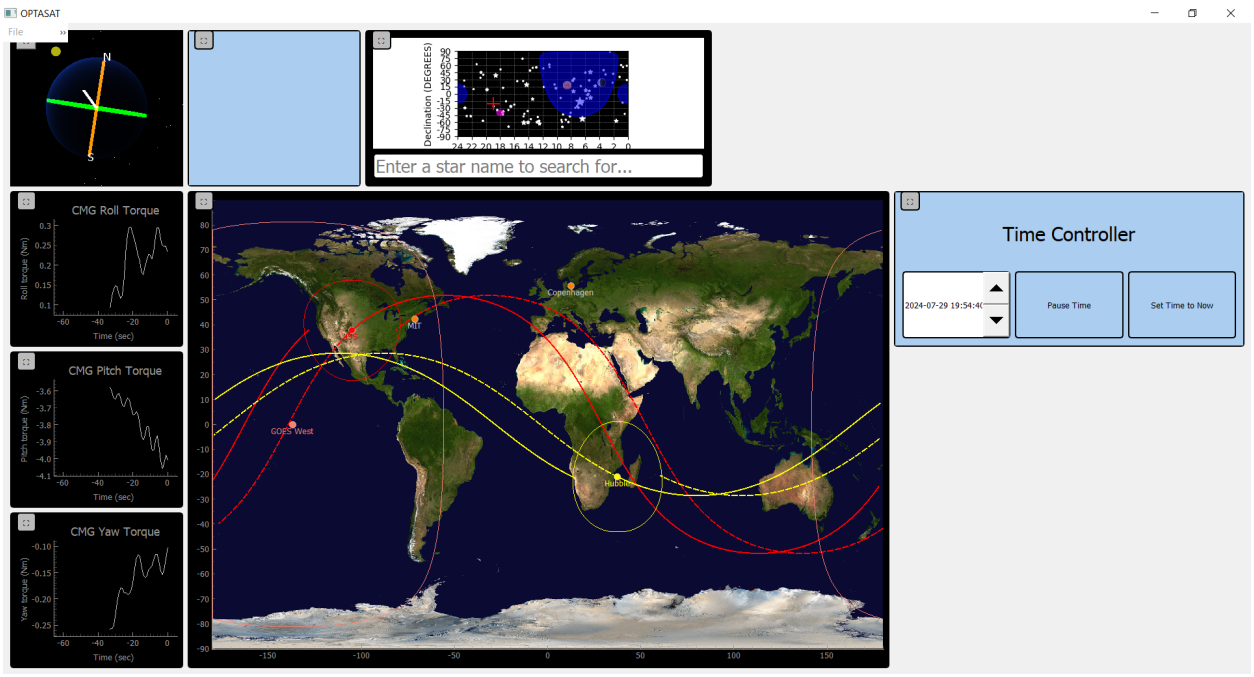

Figure 5.1: An example of a layout of OPTASAT running for a notional low-Earth orbiting mission.

Above these plots, we see the Beta Angle Viewer module. Next, to the right, of that, we have an empty blue rectangle. This exists in order to demonstrate a minimal example of a module which does nothing. It is a blank canvas. Creating a new custom module is easy by copying this module as a starting point, and then adding whatever elements are needed for the custom behavior. Because this module initializes all the code needed to place graphical elements into the view, starting from here allows users to quickly get on their feet with creating a new module.

Next, in the top center, we have the Star Map module, and on the far right, the Time Controller module. Finally, enlarged in the center, the Map Dot module is shown. The Map Dot module was not enlarged in the center when the software was launched with this configuration. Initially, the Map Dot was in the upper right, above the Time Controller. Each module has a grey icon in the upper left, which acts as an "enlarge" button. When any module's enlargement button is pressed, that module will leave its standard docked position, and will fill the large central space. This allows the user to interact with any module of particular importance, and to move between modules without forcing modules to significantly compete for space. The central space starts empty, and any module can be expanded into it. If there is already a module expanded, and a new module is expanded (which would happen if we clicked the enlargement button on any other module shown in the figure), then the existing expanded module (in this case the Map Dot module) will return to its home position where it started, making room for the newly selected module to fill the center.

Modules are positioned according to the built-in PyQT5 QGridLayout system [53]. This system creates a virtual grid on the screen, and any module may be positioned within the

cells of this grid by specifying the cells it should occupy. Modules may take up a single cell, or any rectangular combination of cells. The cells are equally spread across the screen, and the right number of cells on each axis will be allocated in order to accommodate the specified modules. In the example layout we've been examining, the grid has four rows, which are easy to view in the leftmost column, since each module takes up a 1x1 cell space on the grid. On the far right, we can see the Time Controller, which is configured to start in row 1, column 5 (with row zero being the top row and column zero being the leftmost column), with a width of 2 and height of 1. The amount of space reserved for the central module is configurable. The central geometry is set in this case to start at row 1, column 1, with a row span (height) of 3 and column span (width) of 4. The four columns of width can be imagined by looking at the top row and seeing that above the map, we have a 1-wide module, a 2-wide module, and a 1-wide empty space. Overall, this layout has 4 rows and 7 columns.

The modules in the layout are each configured according to the user's preferences. This configuration includes all aspects of the OPTASAT layout, including the location of each module in the layout, as well as the parameters controlling the behavior of each individual module. The details of this configuration process will be presented in Section 5.2.3.

5.2.3 JSON Configuration

All of the options for configuration of the software are performed through the use of JSON text files. JSON stands for "JavaScript Object Notation", but has evolved beyond its original use with JavaScript programs, and is now ubiquitous in many types of software. JSON is also a published standard of the ISO [54]. A JSON file is a simple text file which consists of key-value pairs defining any number of parameters, which can then be parsed by a program to pull in values dynamically.

For an example of the configuration flexibility, we will look at an example of a single module and how it is configured. For the sake of this discussion, we will use the Mapdot module, which will be discussed in further detail in Section 5.4.1. This module is shown in Figure 5.2. Note that this image has been specially generated for this dissertation; specifically, the lines and text have been thickened in order to increase their visibility on a page, compared with their visibility on a computer monitor, as they are intended to be used.

In this map view, there are several parameters of interest which we may want to alter. First is the choice of which satellites to study. In this case, we have chosen the International Space Station, the Hubble Space Telescope, and GOES West. Another parameter is that of ground stations. We have inserted a ground station representing MIT's location in this view. Finally, the colors of each of the satellites can be controlled.

All of these parameters are set through the use of a JSON file. The JSON file is presented verbatim in Figure 5.3. By selecting a single JSON file, the entire OPTASAT environment can be initialized. This means that the graphical layout is preserved and repeatable, and the software can be launched with any configuration at each runtime. If the user has two different sets of satellites of interest, they can keep two configuration files, and switch between them at will. Additionally, users can exchange these configuration files, enabling them to standardize their views, and to adapt each other's views as needed. It is far easier to create a configuration file by editing an existing one than to create one from scratch.

In the configuration file presented here, we start by identifying the ground stations, in

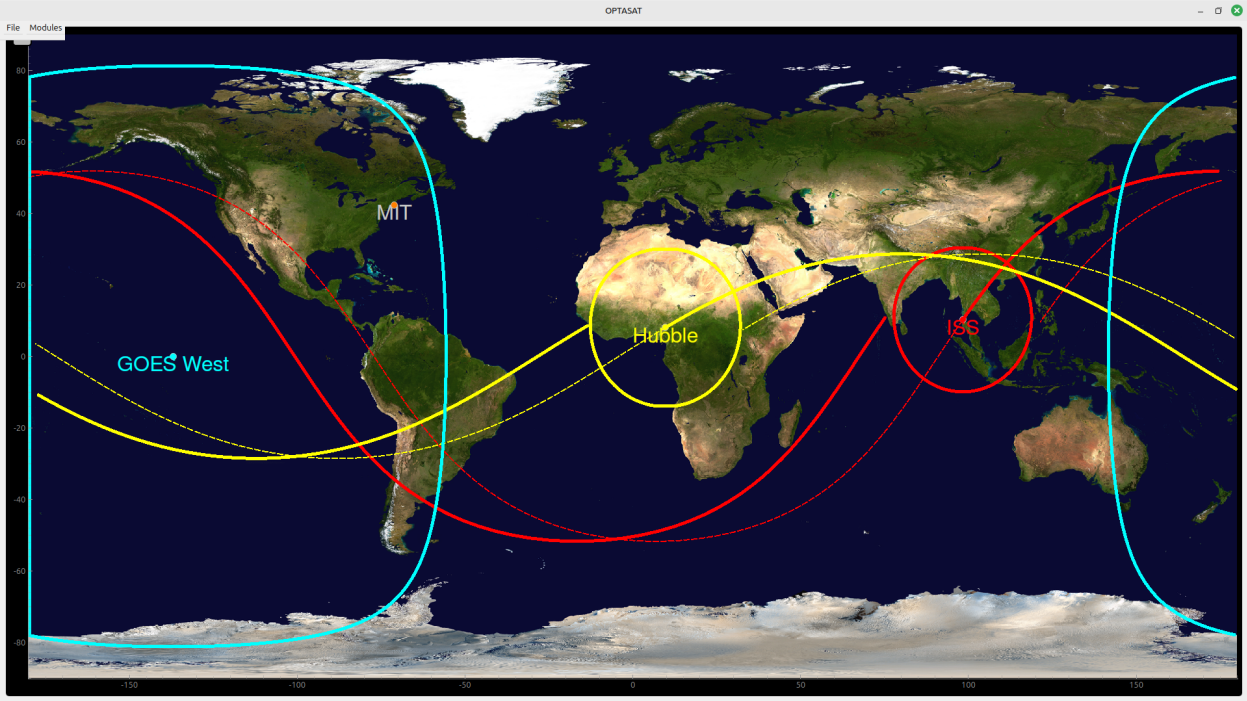

Figure 5.2: An example of the Map View module, showing the International Space Station, the Hubble Space Telescope, and GOES-West. All of the parameters of this map are controlled by the JSON file in Figure 5.3

the "Groundstations" field. This holds a list of dictionaries, where each dictionary describes a ground station. In this case, we have only the single ground station representing MIT. Each ground station has a name and a geographical location represented by a latitude and longitude. These ground stations will each be plotted on the map, and can also be used for modeling of spacecraft passes. In OPTASAT 1.0, ground stations only have these three parameters - they are simply a named point on the Earth's surface. This means a ground station may consist of an antenna for the satellite to communicate with, a landform we may want to take a photo of, or any other location which is fixed to the surface of the Earth.

The next parameters we see in the JSON file are describing the spacecraft ID numbers which will be used in the simulation. All spacecraft which will be used by any modules in the configuration must be defined here. When OPTASAT is initially opened, it will use a built-in process to fetch the Two-Line Element set (TLE) [55] from Celestrak. Once each of the modules is running, they may access these fetched TLE files, but the Spacecraft_IDs field ensures that the TLE will be available when any module requests it.

Next, the JSON defines the central geometry. This describes the position and size of the space which is kept available for whichever module the user expands into the central area of the interface. In this case, since we are only viewing the map alone, we set this to be at cell (0,0) and to have a width and height of 1. There is only one cell in our grid layout, and the single module fills it. In this case, since we are not moving modules in and out of the central area, the expansion feature is not being used.

```
{
    "Groundstations": [
        {
            "Name": "MIT",
            "Lat": 42.36,
            "Lon": -71.09
        }
    ],
    "Spacecraft_IDS": [
        25544,
        20580,
        51850
    ],
    "central_geometry": [0,0,1,1],
    "modules": [
        {
        "source_file":"mapdot.mapdot",
        "initparams":{
            "name":"Mapview",
            "grid_x": 0,
            "grid_y": 0,
            "grid_w": 1,
            "grid_h": 1,
            "self_update_ms":500,
            "SATS":[{
                    "Name":"ISS",
                    "ID":25544,
                    "Color":"red",
                    "FOV":0
                    },
                    {
                    "Name": "Hubble",
                    "ID":20580,
                    "Color":"yellow",
                    "FOV":0
                    },
                    {
                    "Name": "GOES West",
                    "ID":51850,
                    "Color":"cyan",
                    "FOV": 0
                    }
                    ]
            }
        }
    ]
}
```

Figure 5.3: The JSON configuration for the OPTASAT module shown in Figure 5.2.

JSON Module Configuration

All the preceding parameters describe the overall configuration of the window's layout as a whole. The next parameter, "modules", is a list of each of the modules in the interface, and the parameters which will be passed to those modules.

In this case, for simplicity, we are only using one module. The single module has several parameters. The first parameter describes what file contains the module being invoked. All the modules are stored in the main OPTASAT directory, within the modules directory. Within there, simple modules are placed as individual Python scripts, while more complex modules can have their own directories. In this case, the "mapdot.mapdot" parameter indicates that we are looking at mapdot.py, within the directory of mapdot. The period in "mapdot.mapdot" acts identically to the slash which usually is used to separate directories from their contents. Thus, the first "mapdot" indicates the directory, and the second indicates the name of the script. The main OPTASAT script will parse this JSON file, and when it sees this source file, will import that file using a dynamic Python "import" statement, followed by invoking the constructor of the class within the module.

All the remaining parameters in the MapDot module are passed to the Python constructor of the module. Some of these parameters are more universal, while others are particular to the module at hand. Specifically, all modules are given a name. This name is meant to represent the instance of the module. When a module is only used once in a layout, the name is often simply the name of the module itself. When multiple modules of the same type are used, it becomes useful to have a unique name for each. Looking back at the setup in Figure 5.1, we can see that we have three telemetry viewing modules; in their configuration file, they have unique names (CMG Roll Torque, CMG Pitch Torque, and CMG Yaw Torque) describing the data they are plotting, even though they are all generic telemetry modules. Next, the JSON contains the X, Y, width, and height values for the position of the module within the grid of modules. We then have a parameter called self_update_ms, which controls how often the module will update itself to keep the positions of the satellites moving. Finally, we see the set of satellites chosen for plotting.

To see more JSON configuration examples, and to compare the files to each other and see what options exist, please reference Appendix A.

This section has described the overall structure of how JSON files are used to control OPTASAT's behavior at runtime through the use of several examples. The next section will describe another way we use JSON: exporting data from OPTASAT for use in other software.

5.3 Data Export Utility

OPTASAT is a self-contained software suite which can process many aspects of a spacecraft's state. However, OPTASAT can not do everything. Teams may find that their mission needs data to be available outside of OPTASAT. This could include many different applications. One example would be if teams need to process data in an external piece of specialized software (especially if the software is proprietary and can not have its functonality implemented in OPTASAT). Another possibility is that there may be some types of data processing that

require high-performance computing, beyond what OPTASAT (which is written in Python) can deliver. Finally, groups may want to generate command byte sequences to uplink to their spacecraft, which would not currently fit into OPTASAT, as OPTASAT version 1.0 does not have commanding capability. Teams may want to use OPTASAT's data to generate those commands which can then go to the spacecraft.

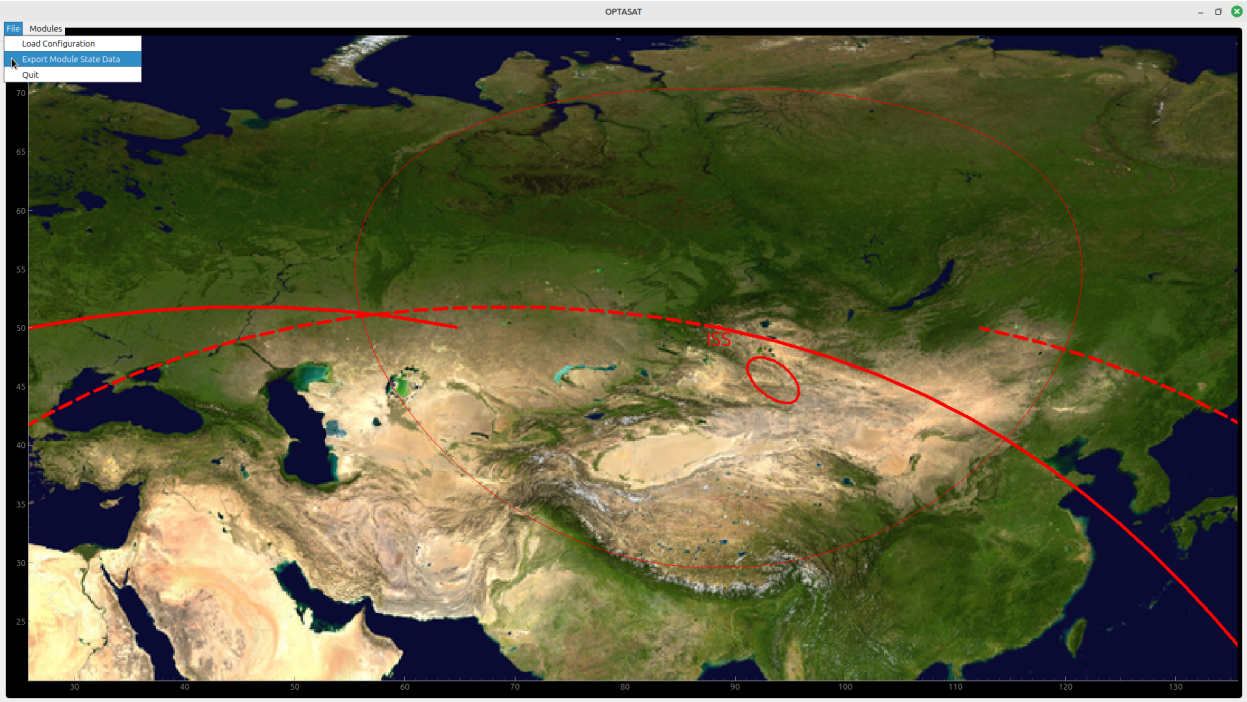

Figure 5.4: A screenshot of OPTASAT with the menu bar open, about to invoke the Data Export Utility. Note that the ISS has a sensor configured which is currently observing an area to the southeast of its position.

OPTASAT's primary form of delivering data to the user is through the visual interface, but it also includes the Data Export Utility. The Data Export Utility produces a "dump" of all the data reported by all the modules in OPTASAT. The utility is invoked through the menu bar at the top of the screen. When the option is selected, the user is prompted to give a filename to save to, at which point every module in the OPTASAT configuration receives a function call to its "export" function. Each module implements "export" in a different way, but the unifying theme is that they all will gather the relevant variables related to their state at the time of exporting, and will write them to a Python dictionary. The Data Export Utility will then gather all the dictionaries for the modules (as well as data relevant to OPTASAT as a whole, such as the internal time), and will export them to a JSON file. This means OPTASAT uses JSON files for two different purposes: declaring the configurations, and representing data exported.

After the data has been exported, other software can import the JSON file (as JSON is a standard format) to then ingest OPTASAT's data and do any further analysis on it. Figure 5.5 is the JSON output from the data that was exported from Figure 5.4.

This data export shows the location the ISS was at, at the time of exporting, and the

```
{
    "Cross module vars": {
        "TLES": {
            "25544": [
                "1 25544U 98067A   24250.60080433  .00073683  00000+0  13481-2
                     0  9992",
                "2 25544  51.6378 272.0371 0007776 321.4625 189.0019
                    15.48652149471177"
            ]
        },
        "globaltime": "2024-09-06 20:24:32.843794"
    },
    "Mapview": {
        "sats": [
            {
                "Name": "ISS",
                "Lat": 50.005234328202356,
                "Lon": 88.49369845310382,
                "Off-nadir mag": 50,
                "Off-nadir dir": 140
            }
        ]
    }
}
```


Figure 5.5: The data dumped from Figure 5.4.

direction the spacecraft sensor was facing (50 degrees off-nadir, at a 140-degree compass direction). This is all of the information needed to represent the area of land which the user chose to target the view sensor toward. We also save the most updated current TLE for the ISS, so that in the future if we wish to refer back to what we exported, we know the exact parameters that fed into it, along with the time. In addition to its usefulness outside of OPTASAT, the Data Export Utility also serves as a method to create a permanent log of the state of OPTASAT, such that the user could re-create a particular scenario in the future, if needed.

5.4 Modules

This section will describe each of the modules which are included with OPTASAT version 1.0, including their parameters, their functionality, and their applications. More modules will likely be added to OPTASAT in the future, but these ones are functional at the time of writing.

5.4.1 Mapdot

The Mapdot module, as was described in Section 5.2.3, displays the positions of satellites on a map of the world. Here we will go into depth on how it works and its full capabilities.

The Mapdot module was heavily inspired by GPredict [21]. It inherits many design choices from GPredict. The core features of the Mapdot are to show the position of an arbitrary number of satellites at a given time, as well as to show the locations of any ground stations of interest. By default, each ground station is rendered as a simple static dot with the name of the ground station, and each satellite is rendered with three elements. The first element is a single dot at the location of the satellite. This was the initial functionality of the Mapdot module, and is the reason it retains this name today. The second element is a drawing of the horizon around the satellite, indicating all locations on Earth which are visible to the satellite (and conversely, all locations where a ground observer would be able to see the satellite). The third element is a pair of lines which indicate the motion of the satellite, propagated one orbit forward and backward. Figure 5.6 shows a basic setup of the Mapdot showing a single satellite and ground station.

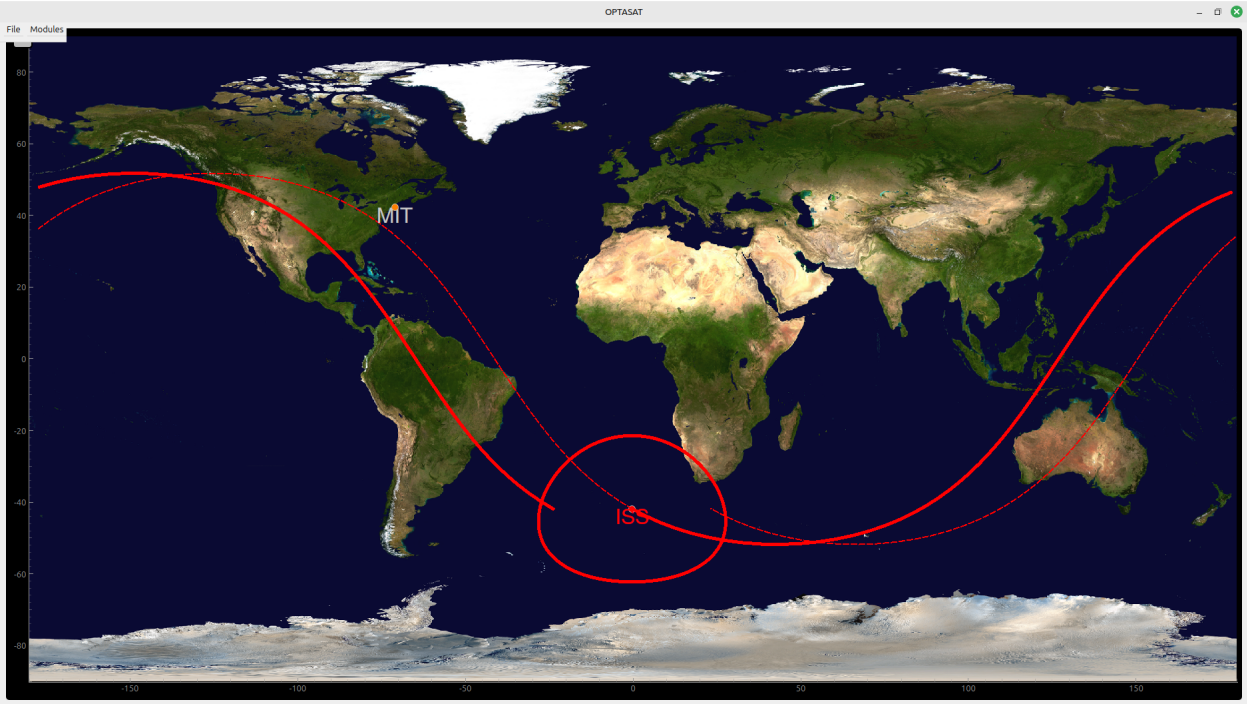

Figure 5.6: A basic Mapdot configuration showing a single ground station (MIT) and a single satellite (ISS), with the satellite's location, footprint, and propagated orbits

The view of the map itself is presented in a relatively simple manner. We use the plate carrée projection, which means that the latitude and longitude of points on the Earth are directly mapped to X and Y coordinates on the map. This projection makes some compromises on visual accuracy, but comes with the large benefit that plotting objects on the map is very straightforward, with no coordinate transformations needed to convert between the latitude and longitude of a plotted point and the screen coordinates of that point. The module is implemented as a graph, using the PyQTGraph Python library. Implementing the map as a graph means that it supports zooming and panning around the world, making it easy to focus in on a satellite or ground feature of interest.

The map is simply a PNG file which is applied as a background on this graph. OPTASAT comes with a map sourced from NASA [56], but users are also able to change it to use any other map they may prefer. This can be done by opening OPTASAT's modules directory, then opening the "mapdot" directory, and replacing the "background_map.png" file. OPTASAT will load in whatever PNG is provided with that file name. They should be careful to ensure that their map image is presented under the same plate carrée projection used for plotting, or the plotted points will not correspond to the locations shown in the map image.

The Mapdot module is useful for one additional purpose: Ground target modeling. Observe in Figure 5.7 that the ISS is passing over the eastern coast of Brazil. We can imagine that an instrument on the ISS may have interest in imaging the Amazon River Delta, which is visible as a triangular tan area roughly to the west of the current ISS position.

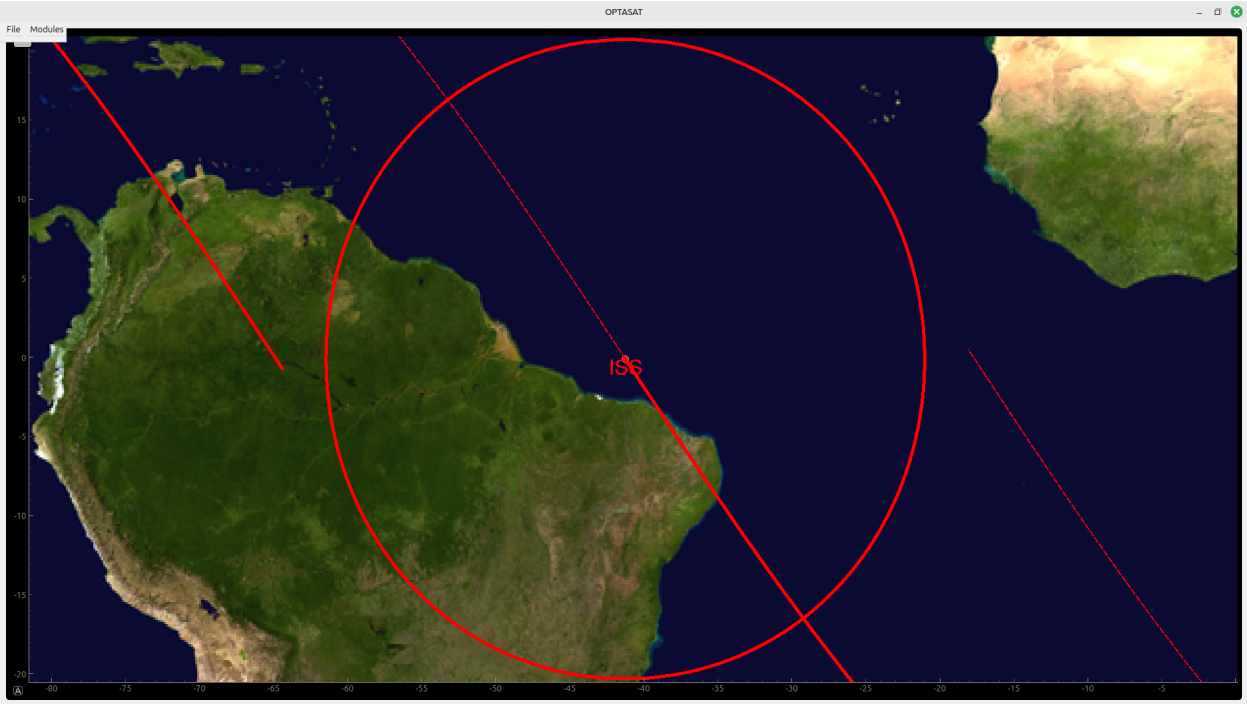

Figure 5.7: A view of the ISS passing over Brazil

The Mapdot module supports modeling of the field of view of a sensor directed off-nadir to image a target of interest. For the sake of our ISS imaging example, let's assume an astronaut wants to take a handheld photo of the river delta. An example of a photo like this

is shown in Figure 5.8. According to the image metadata, it was taken with a focal length of 24 millimeters, using a Nikon D5 camera. According to Nikon [57], the camera's sensor is 35.9 mm x 23.9 mm. We can then use the smaller of these, at 23.9 mm, and use the field of view equation:

$$\text{FOV} = 2 \cdot \arctan\left(\frac{\text{Sensor Size}}{2 \cdot \text{Focal Length}}\right)$$

to find that the FOV is approximately 53 degrees (or 26.5 degrees as a half-angle FOV). This will be used to drive our simulation of the imaging.

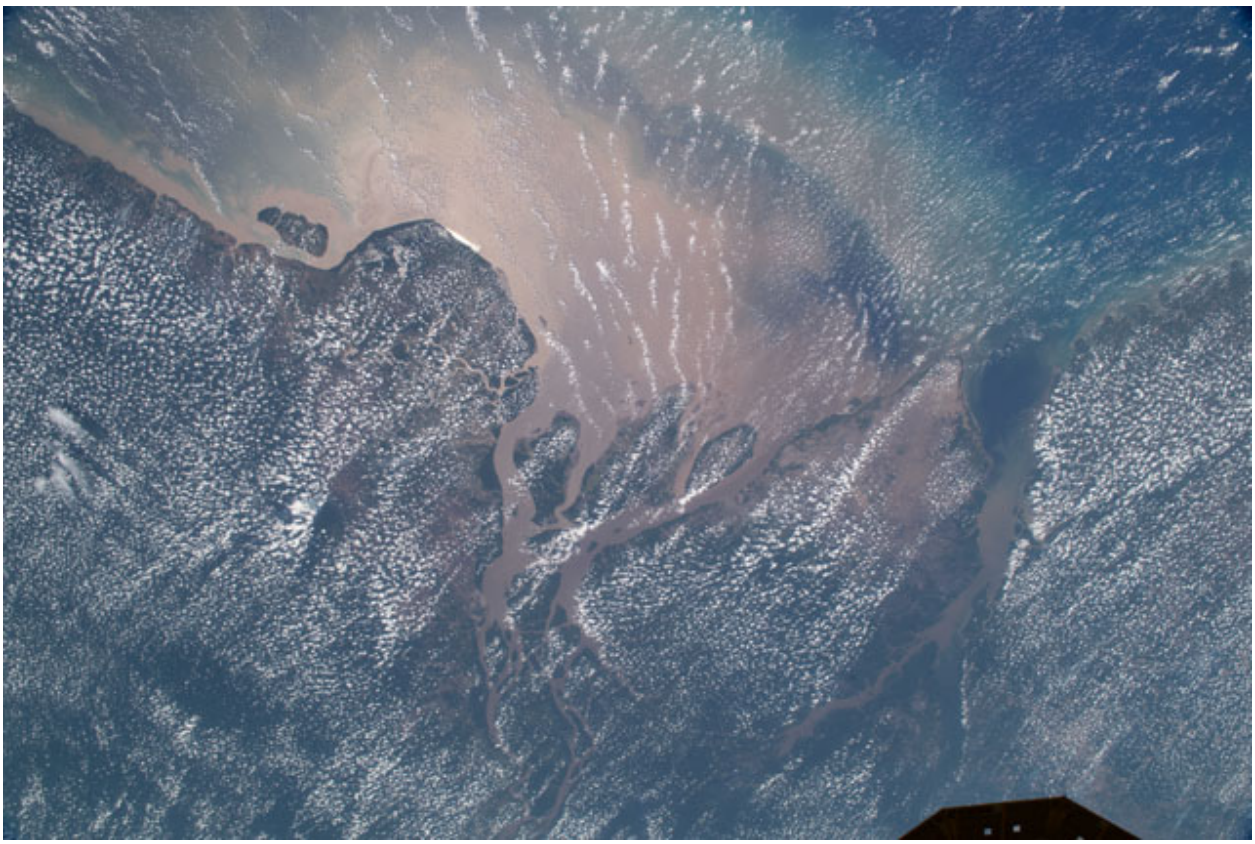

Figure 5.8: An astronaut's photo of the Amazon River Delta. [58]

Now, we can use OPTASAT to model the imaging of that location, using a 53-degree FOV sensor. In the photo (Figure 5.8), the ISS was flying directly over the delta, but in our case we are not so lucky. We would like to determine the required imaging angle in order to capture an image of the delta from our current location. The module's image planning utilities will serve this purpose. In the menu bar at the top of the screen, we can open the boresight control window. This allows us to specify the sensor's field of view, as well as how far, and in what direction, it is tilted off-nadir.

In Figure 5.9, we have used the boresight control window to aim our camera. This interface enables a closed-loop process, where the human operator can adjust the values of angles used to aim the camera and directly see the results of doing so. We see that in this case, we can get the delta centered in our imaging area if we tilt 42 degrees off-nadir, at a

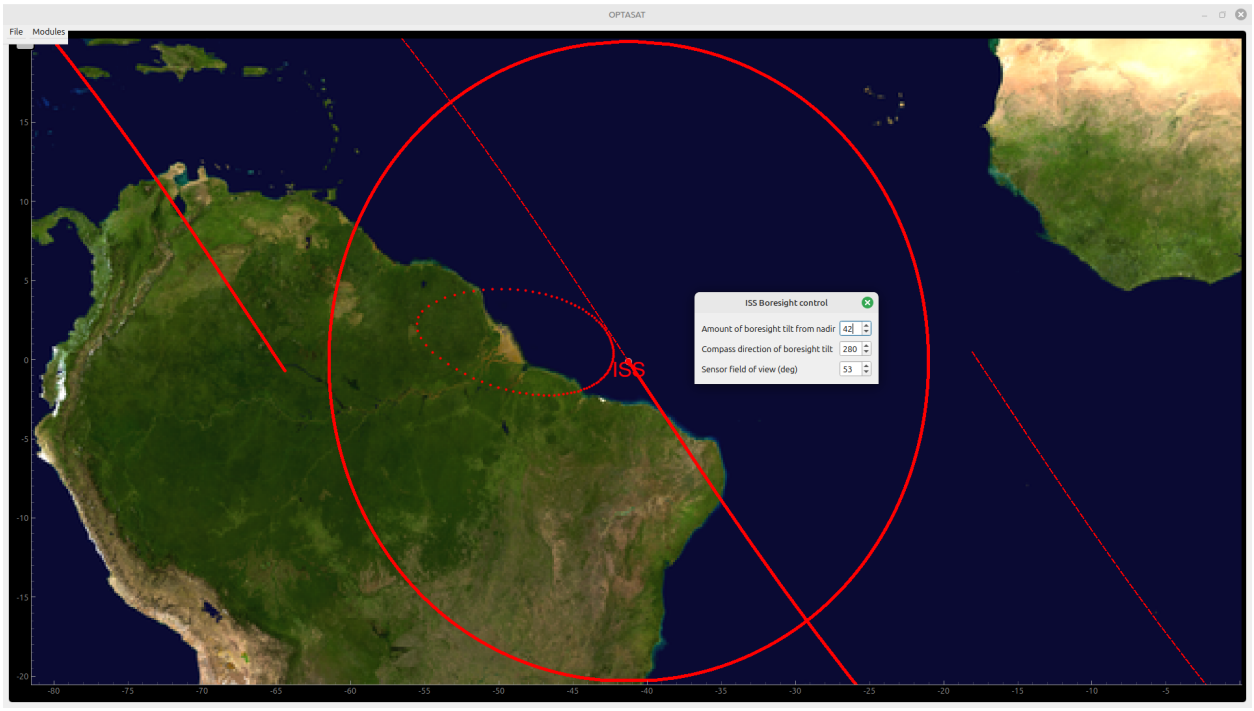

Figure 5.9: Using the image planning capabilities of the mapdot module to align the image sensor to the target

compass direction of 280 degrees (slightly north of due west). We can also see the amount of distortion that we experience due to imaging across the surface of the Earth. By using this modeling to determine the orientation in which to take a photo, an astronaut could work out whether it is feasible to take an image of interest. For non-crewed missions, the angles found through this process of visually confirming the presence of a target within the field of view can be used to drive attitude control commands in order to point a spacecraft's cameras to the desired ground target.

The imaging area simulation is implemented in a relatively simple manner. First, we use simple trigonometry to convert from the selected boresight angles to get a single unit vector pointing to the center of the image. Next, we generate a vector which is rotated by half the field of view angle, which serves as a single vector which will be at the edge of our field of view. Then, we take this single FOV-edge vector, and rotate it about the primary boresight axis in order to generate more FOV-edge vectors, yielding a cone of vectors which lie on the edge of the field of view. Finally, each of these vectors is projected from the location of the spacecraft, until it intersects with the surface of the Earth. Wherever it intersects, a dot is plotted on the map.

Because the boresight control window is opened through the menu bar at the top of the screen, this also serves as a good example to users of how to interact with the menu bar (which is somewhat non-intuitive in the PyQT5 graphics module). For any users wanting to make their own modules, the code for the menu bar interaction here should be a strong starting point.

5.4.2 Time Controller

The next module we will discuss is much simpler than the Mapdot module, but is also more powerful.

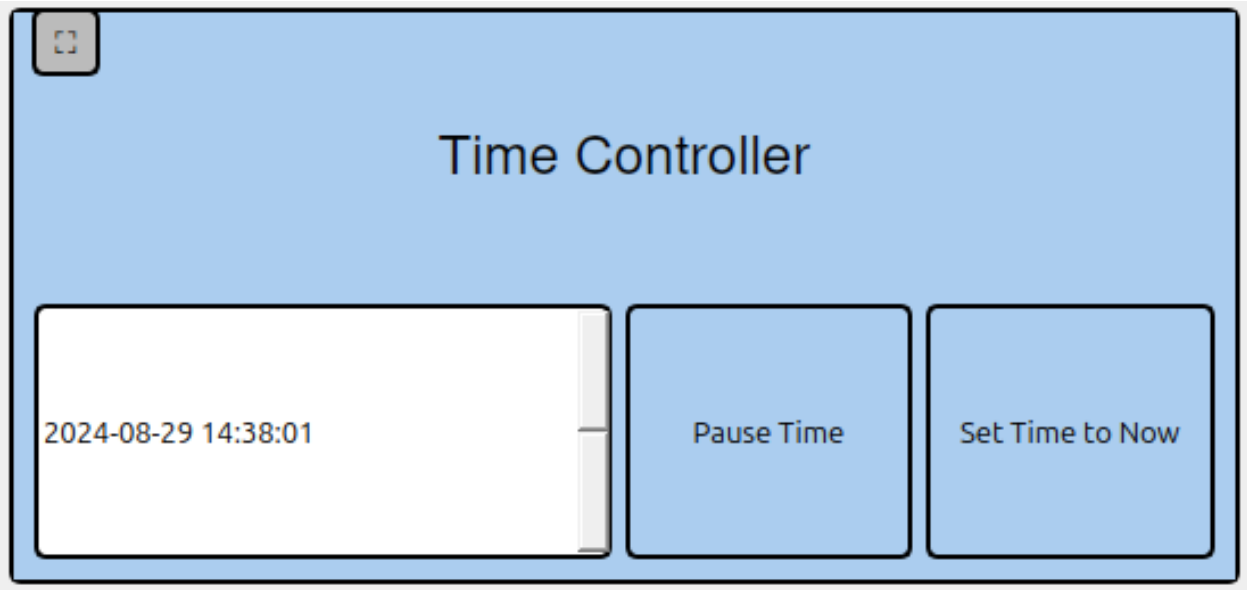

Figure 5.10: The Time Controller module in use

The purpose of the Time Controller module is to control the behavior of all the other modules in an OPTASAT setup. Almost every module incorporates time in some way. In the Mapdot module, the satellites are shown on the map in the locations they occupy at whatever time is specified in the Time Controller. While all modules are optional, almost every OPTASAT configuration will use the Time Controller because of how useful it is for any aspect of mission planning. If the user wants to know where the satellite will be tomorrow (for example, to determine which objects will be available for imaging), the Time Controller makes it easy to simulate tomorrow's conditions. If the user wants to diagnose a fault that happened yesterday, the Time Controller allows simulating where the spacecraft was yesterday.

The interface of the Time Controller is simple. On the left, we have a time and date editing field. It allows the user to independently control the year, month, day, hour, minute, and second of the simulation. These fields can be typed into, or the arrow keys can be used to quickly increment or decrement any of the fields. This is the most direct way to control the time in the simulation. By default, the Time Controller will synchronize to the current time of day, and do a live simulation. The text box actively ticks up seconds as they pass, keeping the simulation up to date. Users can see the movement of satellites as it happens. Setting any past or future time will also cause time to continue passing at a real-time rate. If the user wants to examine the behavior of a satellite at a particular moment in time, without time passing in the simulation, the Pause Time button can be used. Time will no longer pass, until the button (whose text will have changed to "Unpause") is pressed again. Finally, there is a convenient button which will reset the simulation time to re-synchronize

with real-world time, which is useful after either a manual time setting, or at the conclusion of an extended time pause. The Time Controller module is simple, but it unlocks many of the useful capabilities of all the other modules.

5.4.3 Ground View

Our next module works closely in tandem with the Mapdot module, particularly its image planning capabilities. The Ground View module simulates the view of the ground under a satellite, using a specified field of view. It generates a simulation of the image that would be produced if the spacecraft were to point a camera toward nadir and take a photo. The purpose of this module is to aid imaging planning, such that users can simulate the image that they will receive during a future imaging session, in the event that they were to take an image. While the Mapdot shows the area of Earth captured by an image, the Ground View shows what that image would actually look like. An example of a simulated image generated by the Ground View module is shown in Figure 5.11.

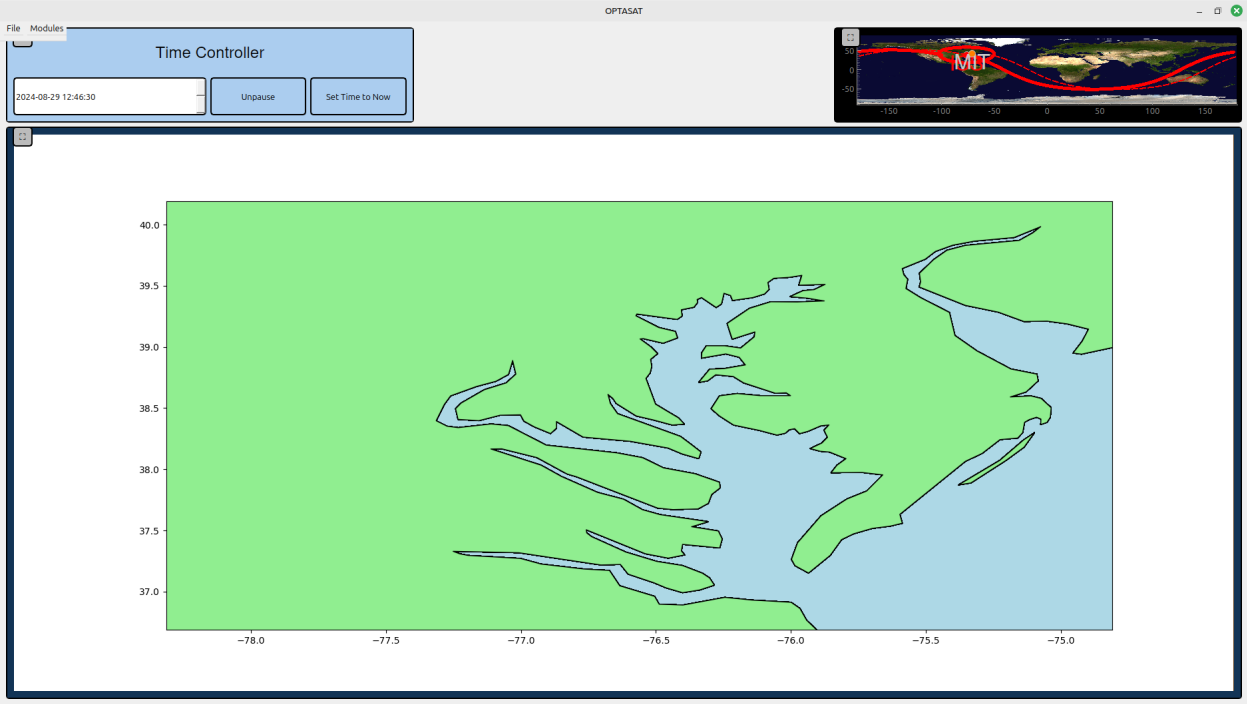

Figure 5.11: The Ground View module, showing the Chesapeake Bay

The Ground View module works through the use of Shapefiles, which are a vector format used for Geographic Information Systems (GIS). These consist of a list of points on Earth and the connections between them, ultimately representing the shapes of coastlines and borders on land. The Ground View module takes a shape file (which can be provided by the user if they have particular needs, or can use the shape file provided along with OPTASAT), and then processes that shapefile to determine which coordinates are within the specified field of view. Once this is determined, the shapefile coordinates are then plotted across the plane of the image, ultimately generating a constructed view of which landforms are present in

the image. Notably, this does not simulate the actual appearance of the land - forest and desert are equally rendered generically as land. Future versions of the Ground View module may become capable of more types of image generation, but for OPTASAT version 1.0, this module only simulates the appearance of the types of borders included in the Shapefiles provided.

As with other modules, the Ground View module is configured in the OPTASAT configuration JSON file. Most importantly, the JSON file specifies the field of view which should be used to simulate the area visible under the spacecraft. With the field of view value, a trigonometric transformation is performed, identifying the view cone of the sensor and ultimately performing a transformation between the cone angle of the sensor and the cone angle of the patch of land imaged by that sensor. The field of view can be set to match whatever imager is used on the spacecraft, thus maximizing accuracy of the generated image. The Ground View module is then able to give a notional impression of what the user can expect to receive from their imaging sensor at the exact specified time.

5.4.4 Pass Finder

Next, we will discuss a few modules related to flight over ground stations. The first is the pass finder, depicted in Figure 5.12.

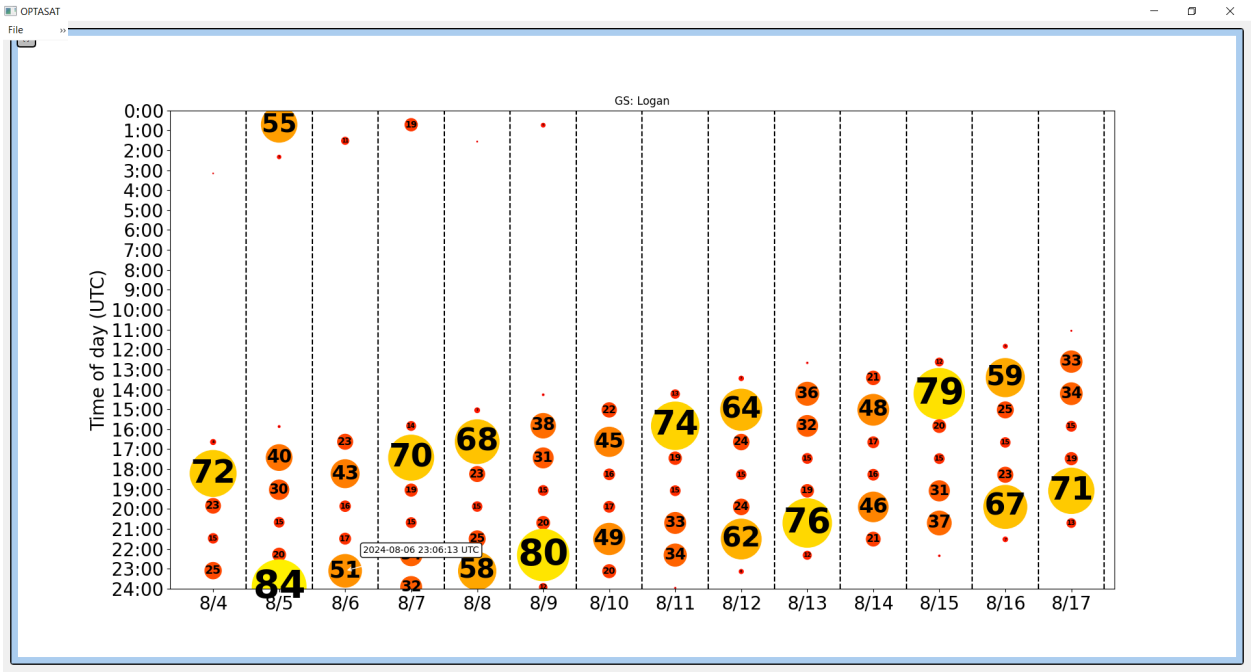

Figure 5.12: The Pass Finder, showing two weeks worth of passes over a ground station

The Pass Finder is adapted from a Python script which was created for the DeMi mission, as discussed in section 4.1.1. In the Pass Finder, a satellite-ground station pair is evaluated to determine when the satellite will be passing over the ground station. Two weeks of passes are predicted. The data is displayed in a manner that is intended to resemble the interface of popular calendar software, such that each day is a column, and the time flows downward

throughout the day, from morning at the top to night at the bottom. Each pass is rendered as a circular bubble in the column. The bubbles are sized, numbered, and colored according to the maximum elevation of the pass above the horizon, which is a strong indicator of the quality of the pass. This visual format for displaying passes acts as a way for users to quickly evaluate when the next passes are, as well as determine trends. In the example in Figure 5.12, there are a few clear trends. One is that it appears that this location gets approximately six total passes per day, of which one or two are strong passes. In fact, the days seem to go through a four-day cycling. We can see this best starting on 8/8 (a convenient day to start since we eliminate the wrapping at midnight). On that day, there is a medium-quality pass around 17:00 at 68 degrees, then a medium quality pass around 23:00 at 58 degrees. Then, on 8/9, the only good pass is a very good one, late at 22:00. The following day is another day with two medium-quality passes, and then 8/11 has a single good pass early at 16:00. The days continue through this cycle: Two medium (one early, one late), one late, two medium, one early. We can also see a trend in that the passes get earlier and earlier with each day. These trends will be useful to have in mind for the purposes of mission planning and identifying operators to staff passes according to availability.

In addition to visualization, the pass finder has functional capabilities that aid in its usefulness in OPTASAT as a whole. First, by mousing over any pass, a callout is created which will give a textual printout of the time of the pass. This capability is important because it eliminates a tradeoff. For the most part, we experience a tradeoff between the precision of data, and the legibility of it. We want to know the exact date and time of some passes, but if we printed all of them, the data would be overwhelming. Instead, we give the notional approximate time which is visible at a glance in the plot, but we also allow the precise time to be viewed, if the circle for a pass is moused over. Further, after mousing over a pass, if the user then clicks on it, the internal simulation time will jump to the time of that pass. This acts as a major convenience feature, because otherwise the user would have to copy the time of the pass into the Time Controller module. But because the Time Controller's control over time is not exclusive, the Pass Finder can also set the time, and specifically set the time to the beginning of a pass. This way, the user can immediately see what the evolution of the simulation across the duration of the pass will look like, and ultimately be able to plan for the pass as it proceeds.

Next we will discuss how we visualize the progress of a pass.

5.4.5 Pass Polar Plot

Once a pass is identified as being of interest, we can plot its progress over time. This module is, like the Mapdot module, inspired by capabilities in GPredict.

Figure 5.13 shows an example of a pass as displayed on the polar plot. Polar plots are a standard view for displaying the location of objects in the sky. The polar plot takes the hemisphere of sky visible to an observer and flattens it into a disc. Thus, the point in the center of the plot represents the point directly above the observer's head. The edge of the circle represents the horizon in every direction from the observer. As a point gets closer to the center of the disc, it gets higher above the horizon. In this implementation, we have lines indicating every 10 degrees of elevation angle above the horizon. The directions around the circle correspond to the directions on a compass. In this example case, we can see the ISS

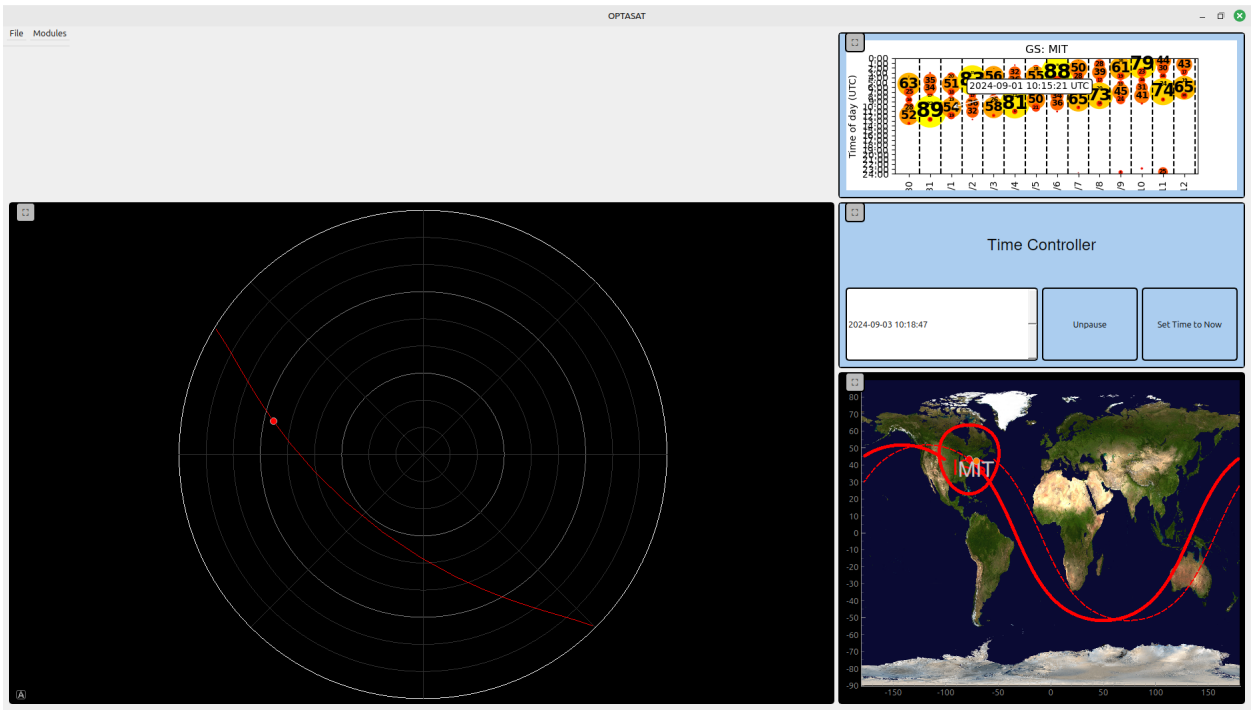

Figure 5.13: Viewing a pass, focusing on the Polar Plot. The Pass Finder, Time Controller, and Mapdot modules are also visible.

passing over the MIT ground station. The start of the pass is in the northwest, at a bearing of approximately 300 degrees. At the moment of this screenshot, the ISS is at an elevation angle of about 34 degrees, and a bearing of 275 degrees. The pass will reach a maximum elevation of almost 60 degrees, before moving back toward the horizon, in the southeast. The views on the map (where the ISS passes from the northwest, down to directly over New York City, and off over the Atlantic Ocean to the southeast) correspond to what we observe in the pass polar plot. The polar plot is useful for any operation that involves tracking the satellite in the sky, including visual observations of the satellite, and more importantly communication with the satellite, as the polar plot directly corresponds with the required aiming commands for any antenna or optical communication telescope. Using the Time Controller with the polar plot module makes it easy to see where the satellite will be in the sky at any given moment. This enables rapid planning for future passes.

5.4.6 Ground Station Access

The next module included with OPTASAT is the Ground Station Access module, which is depicted in Figure 5.14.

The purpose of this module is to display, for a given ground station, all times that any satellites from a chosen set will be visible from that ground station. In this example, we are tracking the times that the Hubble Space Telescope (in the top row) and the International Space Station (in the bottom row) will be visible. There are many uses for this, but one could be planning for communication opportunities. For example, if a commercial ground

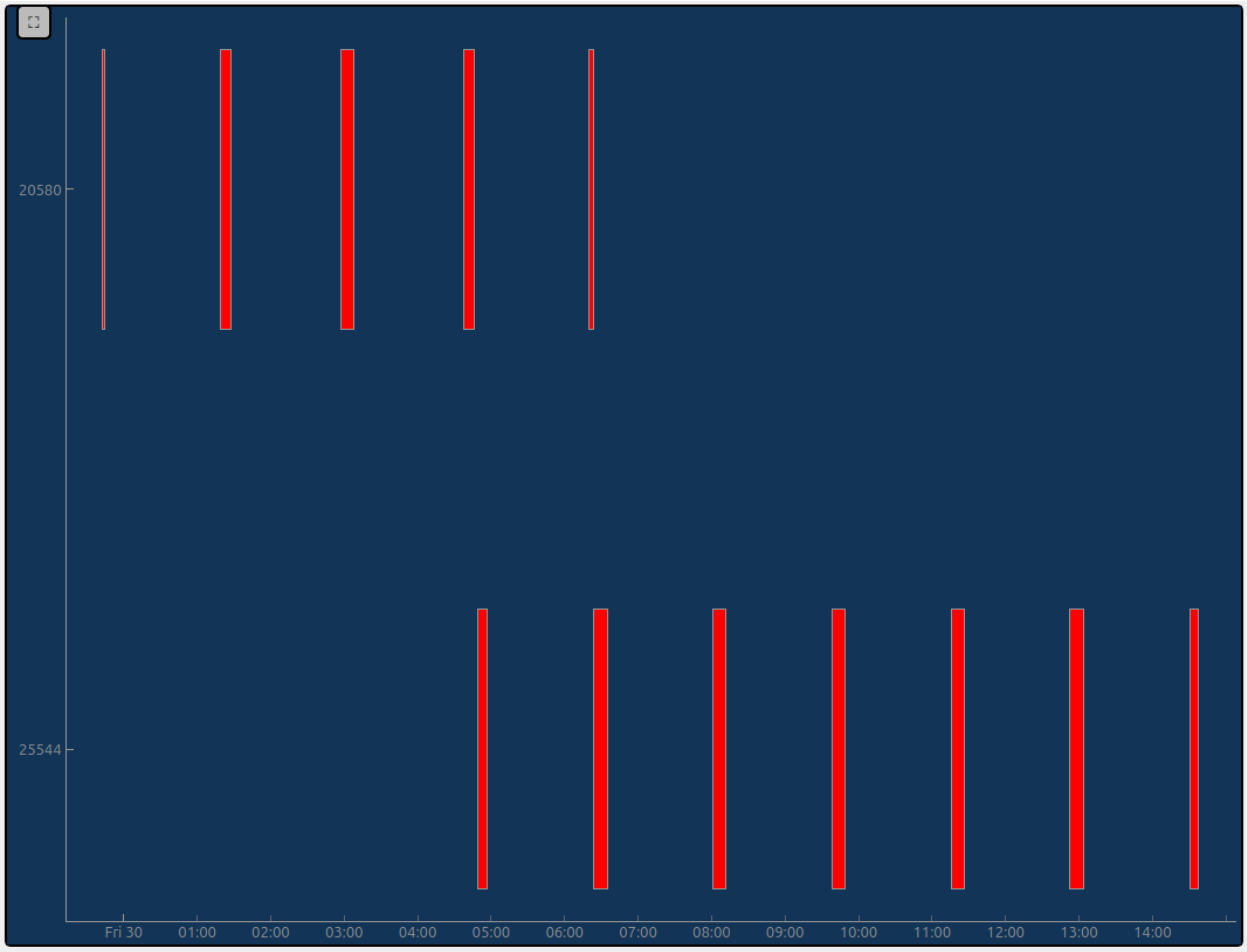

Figure 5.14: An example of the Ground Station Access module, showing passes of the Hubble Space Telescope and the International Space Station over MIT

station operator had 12 different satellites to serve, they could use the Ground Station Access module to determine when each of the 12 were in the sky, and from there, generate a full schedule of contacts. In the Ground Station Access module, each satellite is shown in its own row on the Y axis, and the X axis represents time. Effectively, each satellite displays a timeline of the binary state of whether it is accessible to the ground station. At any given time, the user can zoom in on the X axis to view a vertical slice, which will tell them all of the satellites (from their chosen set) that will be visible at that time. For our imagined ground station operator, zooming in on any time will make it obvious whether there are multiple satellites visible at that time, which would constitute a conflict of scheduling where they may need to choose to communicate with one satellite or another. The Ground Station Access module is inspired by GPredict's "Sky at a Glance" functionality and serves the same purpose of representing the state of satellites in the sky. While the Pass Finder module gave detailed information on the relationship between a ground station and a satellite, the Ground Station Access module gives more surface-level information regarding the ground station's contact opportunities with many satellites.

5.4.7 Eclipse Plotter

The next module, the Eclipse Plotter, uses a very similar display interface to the Ground Station Access module. An example of an Eclipse Plotter applied to the International Space Station is shown in Figure 5.15

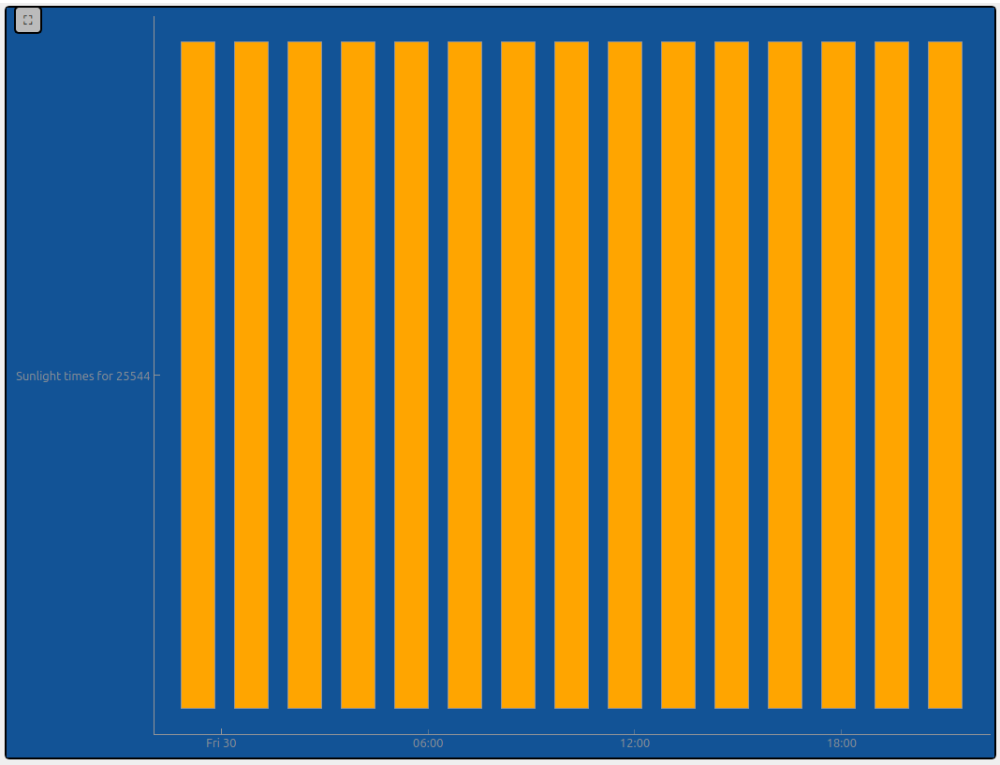

Figure 5.15: An example of the Eclipse Plotter module displaying the times that the International Space Station will transition between being sunlit and being eclipsed

Like the Ground Station Access module, we display a binary state of a spacecraft on a timeline. While the Ground Station Access module's binary state was the ground station's visibility to the spacecraft, the Eclipse Plotter displays the binary state of the spacecraft being in sunlight. The timeline is labeled for the spacecraft where the spacecraft gets a light orange bar whenever it is illuminated by the sun, and no bar when it is eclipsed by the Earth. Knowing the future illumination state of a spacecraft at any given time is useful for planning different operations, particularly in connection with the Power and Thermal subsystems. Since many spacecraft are solar powered with secondary batteries, having access to solar illumination determines what power-consuming instruments may be operated. Additionally, the heat received from the sun when outside eclipse increases the spacecraft temperature, which may restrict the use of heat-sensitive instruments, or the use of heat-generating instruments which may lead to an overheating condition when sunlit. Sunlight may also present a problem for imaging instruments whose sensors may not be able to tolerate sunlight. All these reasons give many justifications for why a mission would want to know whether it is sunlit, and this module fulfills the need to be aware of the spacecraft's illumination status.

5.4.8 Starmap

The Starmap module is one of the modules with the most features. It was initially developed for the DeMi mission as a standalone Python script using Matplotlib for all its graphics, and has now been transformed into an OPTASAT module. Much of the initial code has been preserved; in fact, the current implementation involves a Matplotlib plot embedded within an OPTASAT module. This therefore means that the Starmap module serves as a strong example to users of how to implement a module using Matplotlib, which may be a more familiar interface to them than the native PyQT5 graphics objects. One of the differences between DeMi's version and OPTASAT's is that the DeMi version had its own time controlling interface. Now that the Starmap is integrated into OPTASAT, it can rely on the time chosen by the Time Controller module. Being part of OPTASAT reduces the amount of work that needs to be done by each of the modules.

The Starmap module is intended for use in astronomy missions. A simple example of the module in use is shown in Figure 5.16.

The Starmap shows many objects visible in space; we will discuss each of them one by one. First, the Starmap displays a map of the stars. The stars are imported from the Hipparcos star catalog [59]. The Starmap plots all objects using the same plate carrée projection that the Mapdot module did - astronomical right ascension and declination are plotted as direct X and Y coordinates. This is why the stars appear more concentrated near zero declination - the celestial poles are being stretched across the upper and lower edges of the plotting area. The stars are plotted in such a way that the size of a star is proportional to its visual magnitude as reported in the Hipparcos catalog. This makes it easy to find the brightest stars, which might be the best candidates for an imaging mission. Additionally, the module displays only stars brighter than a chosen magnitude limit. Thus, missions can set the magnitude limit to match the magnitude that their imaging systems can resolve. For the sake of the examples here, we have chosen a magnitude limit of 2.5.

In addition to displaying the stars, the Starmap is helpful for finding them. When any

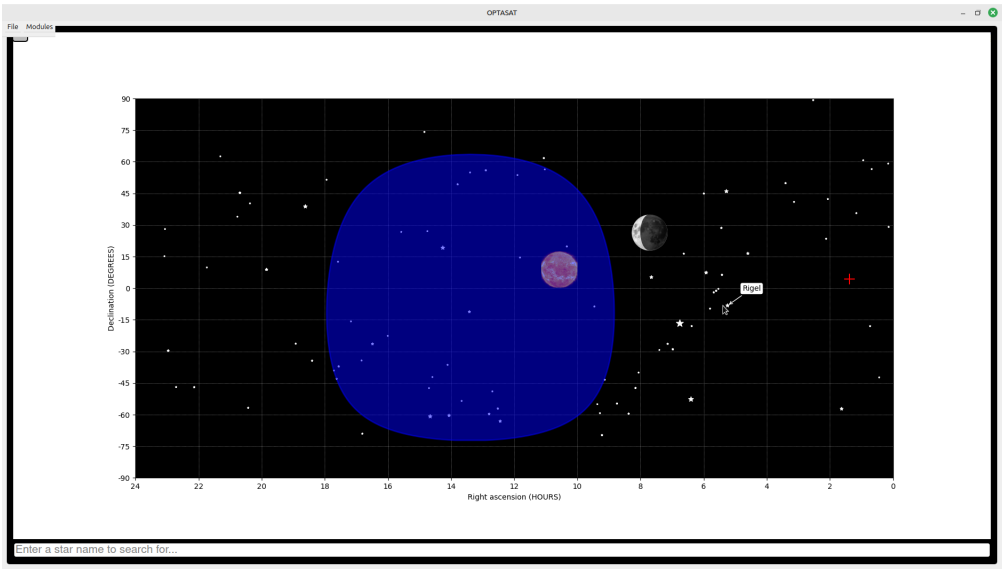

Figure 5.16: A basic example of the Starmap, showing the field of stars, the satellite (the red cross near the right edge), and the Sun, Earth, and Moon.

star is moused over, a callout will appear identifying the star by name. This can be seen in Figure 5.16 where Rigel is currently being moused over, near coordinates (5, -8). Rigel is also notable for being the right foot of the Orion constellation, which can easily be seen on the map. While this functionality is useful for determining the name of a star on the map, the Starmap also provides the reverse functionality. Any star name typed into the box at the bottom of the module will perform a search of the catalog, and if a star matching the name is found, it will be displayed with the same type of callout as is created on mouseover. Thus, if a user wants to image Vega, they can type "Vega" into the box and immediately see where it is on the map.

After the stars, the next simplest object displayed in the Starmap is the Sun. It is displayed at its present location in the sky depending on the date. The right ascension of the sun at the time of the March Equinox is, by definition, zero. Right ascension is, by convention, measured in hours, and this is the unit we use in the Starmap as well. Right ascension increases to the left (thus giving us a flipped X-axis). The sun's right ascension reaches 12 hours at the September Equinox, and therefore because these images are being generated in late August, we see it approaching that 12 hour mark from the right.

Next, we will look at the Moon. The Moon is rendered according to its position in the sky, with an overlay representing its current phase. The Sun and Moon are each implemented as simple imported PNG images. While the Sun's PNG is presented unaltered, the Moon's is given a partially-transparent grey overlay representing the portion which is unilluminated at a given time. This is done by generating a half-ellipse curve which touches the North and South poles of the Moon, and is then stretched horizontally to match the proper current phase. From there, the shape of the shadow mask is expanded to cover the appropriate portion of the Moon. Because the Moon's illumination comes from the Sun, the illuminated portion of the Moon will always point toward the Sun, as we can see in Figure 5.16. It is possible to view this rendered scene of the Sun and Moon as a false three-dimensional effect.

The location of the satellite in the sky is represented with a red cross. Because the celestial coordinate system is centered on the Earth, we can think of the cross as being the location of the satellite as seen from the Earth. Equivalently, the cross is the location in the sky which is, from the satellite's perspective, directly away from the Earth. Therefore, to mitigate the effects of any potential Earthshine, it can be desirable to image objects which are near this cross in the sky.

Finally, we can look at the Earth as represented in the Starmap. Because the Earth is a large object which is usually near the satellite, it is not accurate to represent its location in the sky as a simple single object. Instead, we need to represent it as an extended area masking off a portion of the sky. To do this, we start from the position of the Earth's center in the sky. This is easy to find, because it is the negative of the satellite's position with respect to the Earth (which is the location represented by the red cross). Once we know the center, we take the angular radius of the Earth and apply the Haversine Formula in order to generate points around a circle projected onto the celestial sphere. We can then plot those points and fill in a mask representing a polygon approximating the circular shape. This is the large blue blob visible on the map. When plotting this shape, special care must be taken to handle the edge cases. For example, if the Earth is near the 0/24 hour right ascension point, it will wrap across and will need to be represented with two separate polygons. An example of this effect is shown in Figure 5.17.

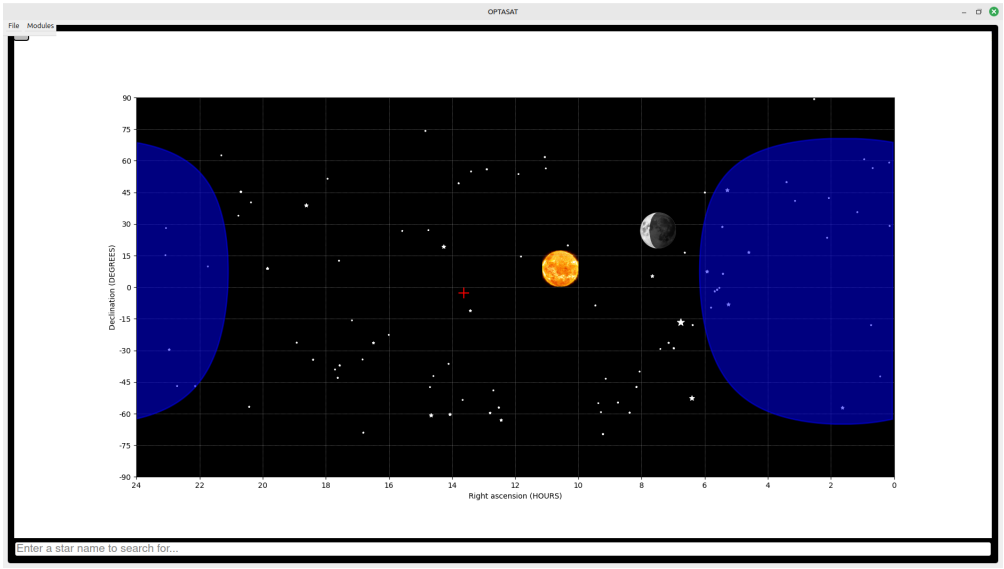

Figure 5.17: An example of the Starmap where the Earth wraps around the Right Ascension axis.

In addition to the possibility of issues when crossing the X axis, we have a different issue when the Earth is covering the celestial poles on the Y axis. Due to the map projection, the North and South pole of the celestial sphere are not represented as individual points. Instead, the pole exists at all points at 90 or -90 degrees of declination. If the circle projected onto the sphere obscures the pole, then when projected onto this map, it will appear to stretch all the way along the X axis. This is shown in Figure 5.18.

Having the Earth visible serves two highly relevant purposes. The first is that any

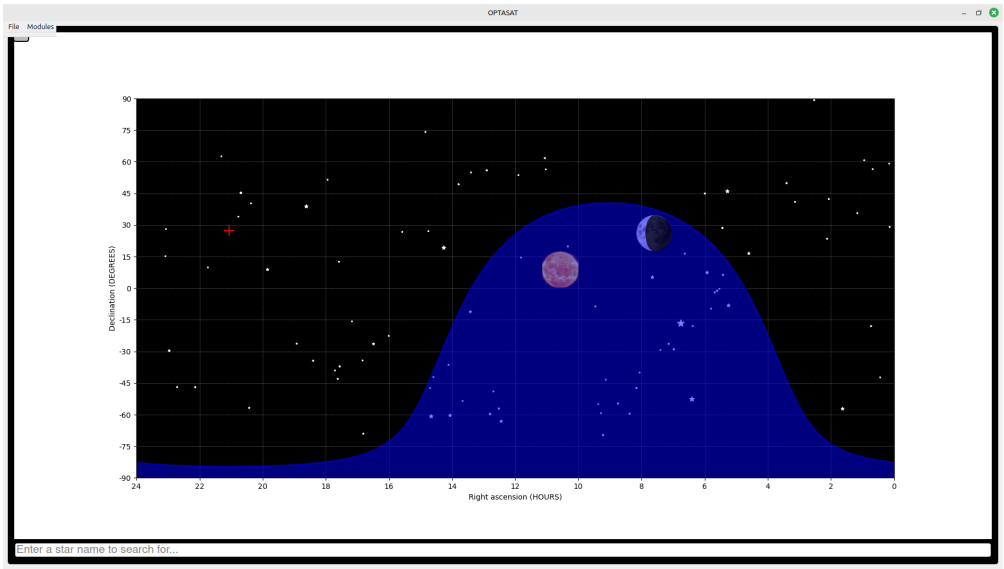

Figure 5.18: An example of the Starmap where the Earth obscures the Celestial South Pole and stretches all the way across the Right Ascension axis

star which is within the blue area representing the Earth will be obscured by the Earth, and therefore will not be available for imaging. When selecting an imaging target, we can immediately rule out any star in the blue region. The other main indication given by the Earth's covered area is that if the Sun or Moon are covered by the Earth, then any stray light coming from these objects will not reach the satellite, meaning that these may be the optimal opportunities for imaging.

One final capability of the Starmap module is the ability to specify keepout zones. These have been left out of all the Starmap examples so far, but are now presented in Figure 5.19.

Keepout zones can be specified in four forms, with each form describing the location where the keepout zone is centered. The first three forms describe a keepout zone centered around the Sun, Moon, or Earth. The final form is a keepout zone placed on a fixed location on the celestial sphere. Each keepout zone, regardless of where it is centered, has a radius in degrees which is specified in the Starmap's JSON configuration. If an instrument needs to be at least 30 degrees away from the Sun to function, the user can specify a 30-degree Sun-centered keepout zone, and that zone will then be drawn on the Starmap to indicate that any star within the zone is unavailable for imaging. In Figure 5.19, we have chosen to specify two keepout zones: A 5-degree magneta keepout zone at (18, -40) fixed coordinates, and a 20-degree grey keepout zone which is set to be Moon-centered. There is no limit to the number of keepout zones which may be specified on a given Starmap instance. The keepout zones are implemented in a way that is very similar to the Earth-drawing routine, as they, in the same way, represent a circular patch projected onto the celestial sphere. We can see that the Moon's 20-degree circular keepout zone gets distorted near its upper edge, as it reaches into the more northern declination coordinates.

Overall, the Starmap presents an intuitive, easy-to-use interface for evaluation of celestial targets, and should be a valuable tool for many types of astronomy missions.

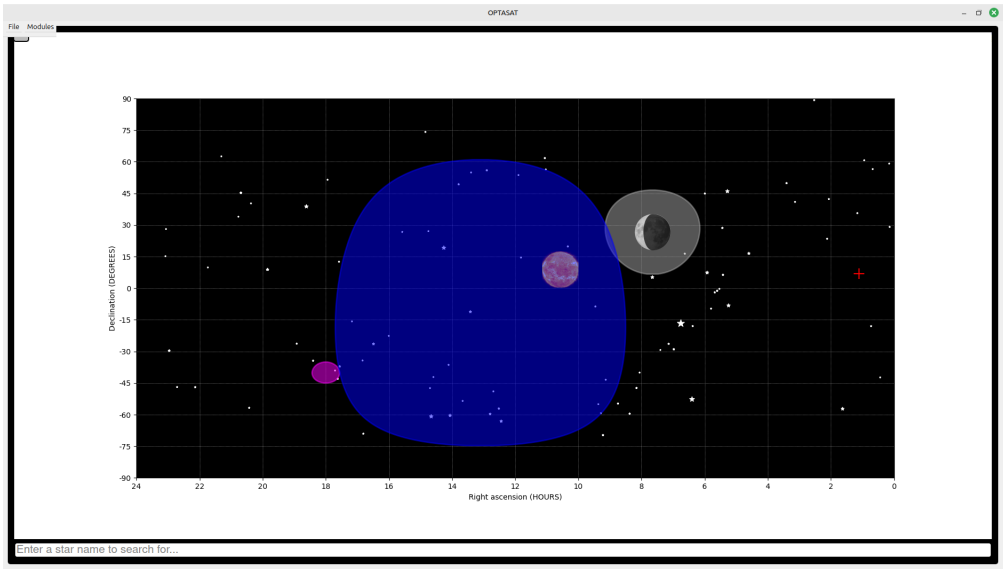

Figure 5.19: An example of the Starmap showing a keepout zone in a fixed location in space (magenta) and fixed around the Moon (grey).

5.4.9 Beta Angle Viewer

We will next examine a unique module: the Beta Angle Viewer. The Beta Angle Viewer is unique because it uses the 3D, OpenGL-based plotting system from PyQTGraph.

When the Beta Angle Viewer is initially loaded, no spacecraft are displayed. The only objects visible are the Sun, the Earth, and a few lines, as well as the background of stars.

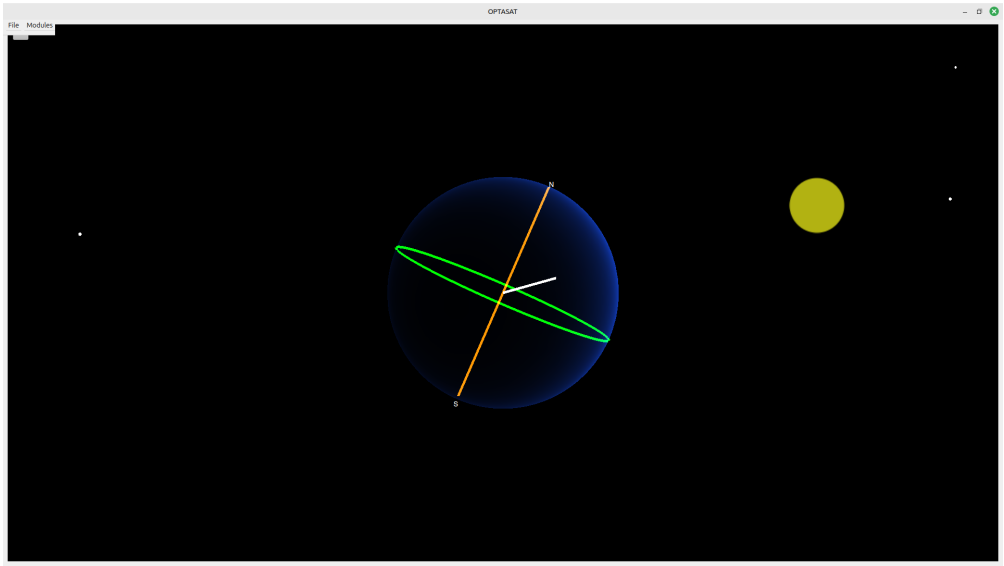

Figure 5.20: The Beta Angle Viewer, showing the startup configuration. This shows the Sun (yellow, right), the Earth (center, shaded blue), the Earth's axis (orange) and equator (green), and the sun-pointing vector from Earth (white).

This configuration is shown in Figure 5.20. The lines shown are each color-coded. In orange we have the axis of the Earth. In green, the equator is shown. Finally, the white vector points directly from the Earth to the Sun. Additionally, the Earth is rendered with a custom shading in order to appear as if it is being illuminated by the Sun. In actuality, the Sun is a simple yellow sphere positioned far away and does not generate light; while we are rendering objects in 3D, we are not using an actual lighting engine as might be seen in game engines like Unity.

In the Beta Angle Viewer, the orbital planes of spacecraft can be visualized. As the name implies, the original purpose of this module was for viewing the solar beta angle of spacecraft orbits. Before viewing spacecraft, the interface displays the Earth as it is located with respect to the Sun. This is a full interactive three-dimensional rendering, including the ability to rotate the camera to any orientation. This functionality is essential for understanding the dynamics and interactions of orbital mechanics, because it allows the user to truly see how the relevant bodies are arranged in space. A two-dimensional view (as on a traditional graph, or a sheet of paper) cannot capture the specifics of planes which may be oriented at arbitrary angles with respect to one another.

The Beta Angle Viewer is controlled from the menu bar of the main OPTASAT window (meaning that this module also works as a demonstration of interacting with menus). The user can select any spacecraft which is loaded into the configuration and view that spacecraft's current orbital situation. The display will show a ring representing the spacecraft orbit, a vector (capped with an arrowhead) showing the normal vector to that orbital plane, and finally the resultant vector that comes from projecting the sun-pointing vector onto the orbital plane. Figure 5.21 shows the same setup, with the International Space Station added

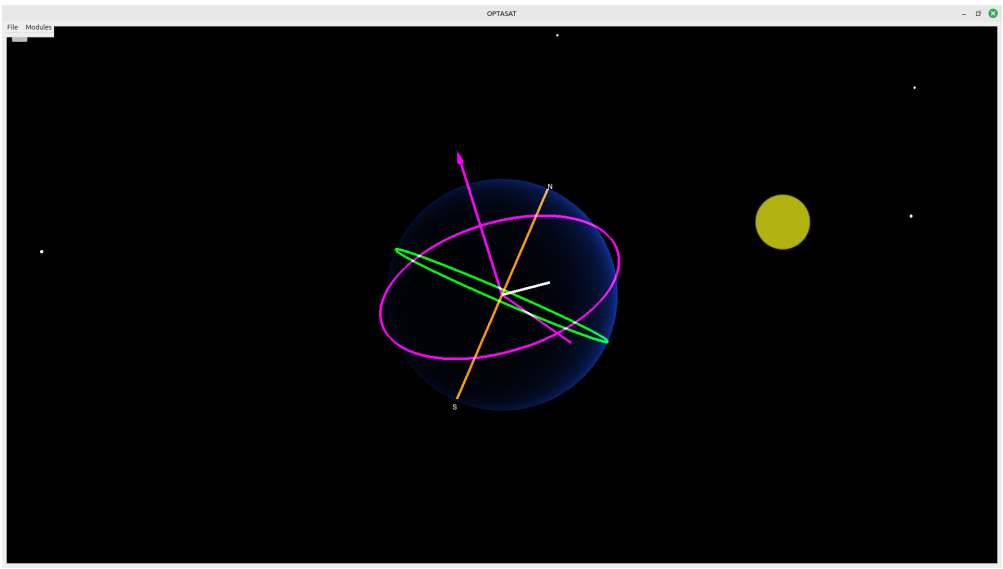

Figure 5.21: The Beta Angle Viewer showing the orbital plane of the spacecraft (the circle), the normal vector to the plane (to the upper left, with an arrowhead), and the projected vector of the Sun-pointing vector (toward the lower right). All spacecraft-focused elements are colored magenta here; if the planes and vectors for more spacecraft were shown, each would have its own color (configured in JSON).

to the view.

As this is a two-dimensional screenshot of a three-dimensional rendering, it is difficult to disambiguate the orientation of a tilted plane. With the ability to pan around, the ambiguity is resolved. In this instance, the angular momentum vector is pointing up and away from the camera, and the camera is viewing the "underside" of the orbital plane. The magneta line pointing in the 4 o'clock direction is the sun-pointing vector (the white line) as projected onto the orbital plane. The angle between these two lines (the white line and this magenta line) represents the solar beta angle. By presenting a tangible example of the way that the various relevant planes may be oriented with respect to one another, the user gets a stronger understanding of the true meaning of their beta angle. This visualization is much more powerful than a numerical statement like "Our beta angle is 65 degrees right now". With the aid of the Time Controller, we can also see how beta angles evolve over time. The tangibility of this kind of data representation greatly aids understanding. Precursors to OPTASAT which rendered the orientation differently have been used in an educational setting to show students how to understand beta angles and their consequences on the dynamics of a mission.

5.4.10 Space Weather Overview

Our next module of interest is the Space Weather Overview. This module will display a simple report of the current space weather conditions, as delivered by the NOAA Space Weather Prediction Center. An example of this module is shown in Figure 5.22.

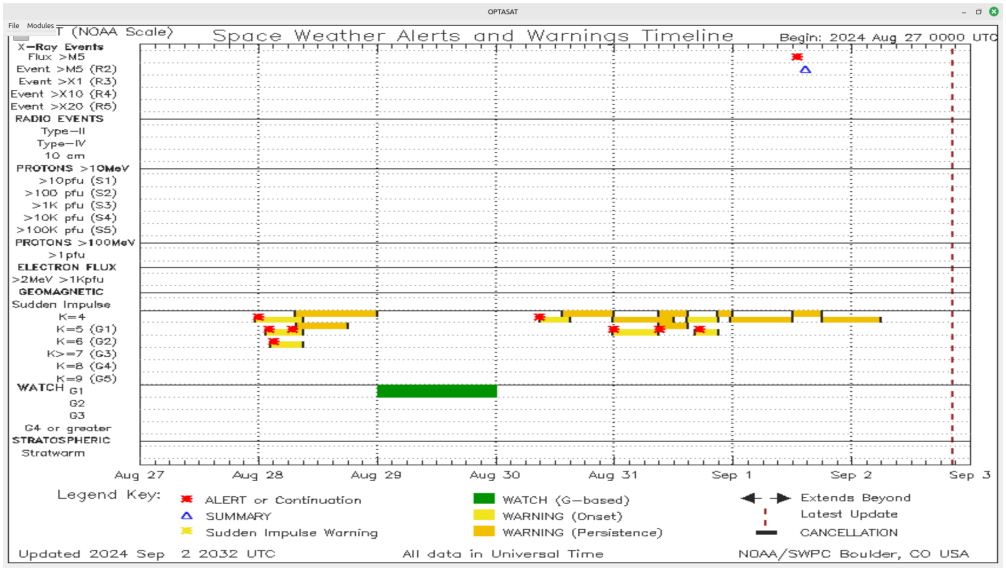

Figure 5.22: The Space Weather Module, displaying the past few days of Space Weather reported by NOAA

In this case, we can see a G1 geomagnetic watch was in effect for the duration of August 29. For several days after, there were also Geomagnetic events, and on September 1 there were two X-Ray events.

This module's internal functionality is much simpler than it appears. The operation of the module consists of a Python urllib request which fetches a data resource from a remote URL. The space weather report is hosted on NOAA's servers [60]. The image is automatically updated by NOAA to show the most recent predictions, so by fetching the image from that URL, we can get the most recent updates. The module accesses this image, embeds it into a PyQT5 QPixMap, and applies that to a QLabel for display on the screen. No actual processing of space weather data is done within this module.

This module acts as a strong demonstration of the ability to embed any image into an OPTASAT layout. If mission operators have any other system which generates images, they can be displayed in the same manner by simply changing the URL. This could even be used to display the last image captured by an imaging mission, or to show GOES imaging data of an upcoming ground target of interest. Because the logic for embedding images is built in a generic manner, any image from any Internet resource can theoretically be fetched and embedded for mission status awareness. The space weather application is demonstrated here as a particularly relevant example, but slight tweaking allows for many more options to suit mission needs.

5.4.11 Telemetry Plotting

Our next module is the Telemetry Plotting module. This module is intended for plotting chosen data points which have been downlinked from a spacecraft. It is intended for use cases where OPTASAT is being run directly as a live operational tool, processing data as a spacecraft communication operation is performed.

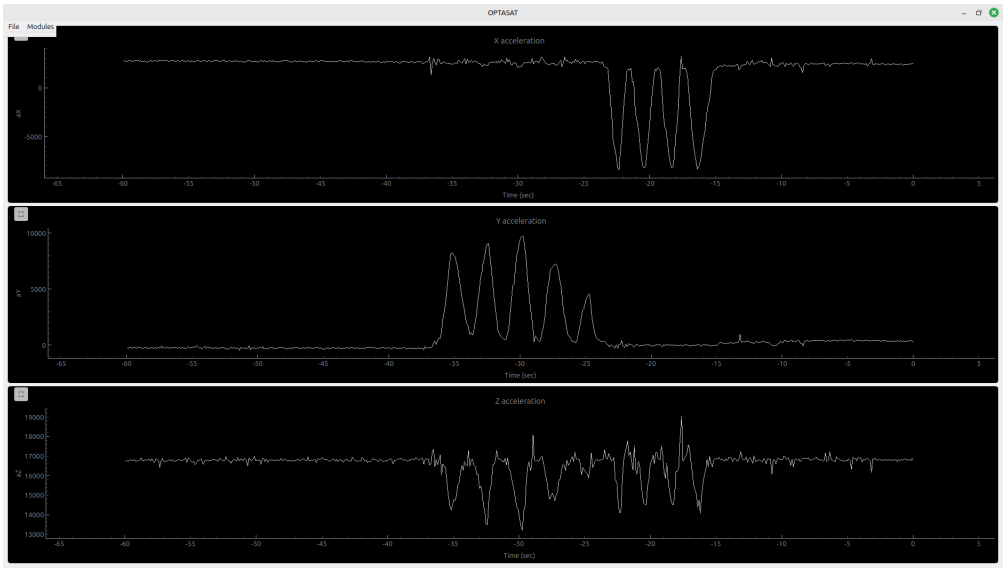

Figure 5.23: Three instances of the Telemetry module, plotting X, Y, and Z accelerations from an accelerometer.

An example of three instances of the Telemetry Plotting module is shown in Figure 5.23. Because there is no convenient operational spacecraft ground station available, an Arduino sharing accelerometer readings is used as a surrogate source of data. In this case, we are

plotting the data from an accelerometer sitting on a desk. The data is plotted for 1 minute, with the three axes each plotted on their own line. The Z axis of the sensor is arranged vertically. Thus, for the majority of the runtime of the screenshot, the Z axis rests at a high acceleration value while X and Y are near zero (ignoring the slight offsets due to imperfect alignment of the axes). After some running at rest, the accelerometer was rotated about its X axis, causing the gravitational vector to become split between the Y and Z axes. The sensor was tipped up and down five times to show the Y acceleration trading off with the Z acceleration. Then, a similar action was performed rotating about the Y axis. The Y acceleration is unaffected while gravity transitions between the X and Z axes.

In operation, these graphs are plotting live. Note that the time on the X axis of the graph is labeled with time relative to the present, such that the current instant of time is zero, and 30 seconds ago is -30. This means that data points are fed in from the right and scroll to the left. This example shows the past 60 seconds of data, but as usual, the amount of time history shown is configurable in the JSON file.

In addition to plotting data from an Arduino, another variant of the Telemetry Plotting module exists, which will plot data fetched from the International Space Station's live data feed, which is provided over the Internet. The intent is that these two variants show the shared features of plotting data in general, in addition to how to implement the specifics of fetching data from multiple sources. Because every mission's data-handling infrastructure is different, the Telemetry Plotting module is intended to be tweaked to accommodate the way that any mission may ingest its data into the ground system.

5.4.12 Follower Satellite

Another module currently contained within the standard OPTASAT release is the Follower Satellite. Unlike most modules, this one does not provide visual output. Instead, it modifies the internal OPTASAT state in a manner not unlike that of the Time Controller module. Hence, other modules can be used to view the results of the Follower Satellite module. In this case, we will show the Follower Satellite module through the use of the Mapdot module. An example of the results from the Follower module is shown in Figure 5.24.

The purpose of the Follower Satellite module is to generate a virtual object for analysis which will follow another chosen object in orbit. This is used for modeling of different scenarios, especially in the design phase of a mission. In this example, we have the International Space Station shown on the map, with our follower (named simply "Follower") configured to follow behind it. In this case, the interface has been used to place the follower at a location that will follow 100 seconds behind the ISS. The interface allows specifying either a time separation or a distance separation. Whichever value is provided, the other field will fill in with the corresponding value. Given that the ISS orbits at 7.65 km/s, we find that a following time of 100 seconds yields a distance of 765 kilometers.

The follower is created by using a TLE which is derived from the leader satellite. In this case, after OPTASAT fetched the ISS TLE and created the ISS satellite, the follower module loaded that TLE, and then uses it to generate a false TLE to drive the orbit of the follower. There are many ways a TLE could be tweaked to create a follower, but in this case, the implementation simply takes the TLE, the specified following distance, and the equations of motion for the orbit, and tweaks the TLE to change the mean anomaly to fit

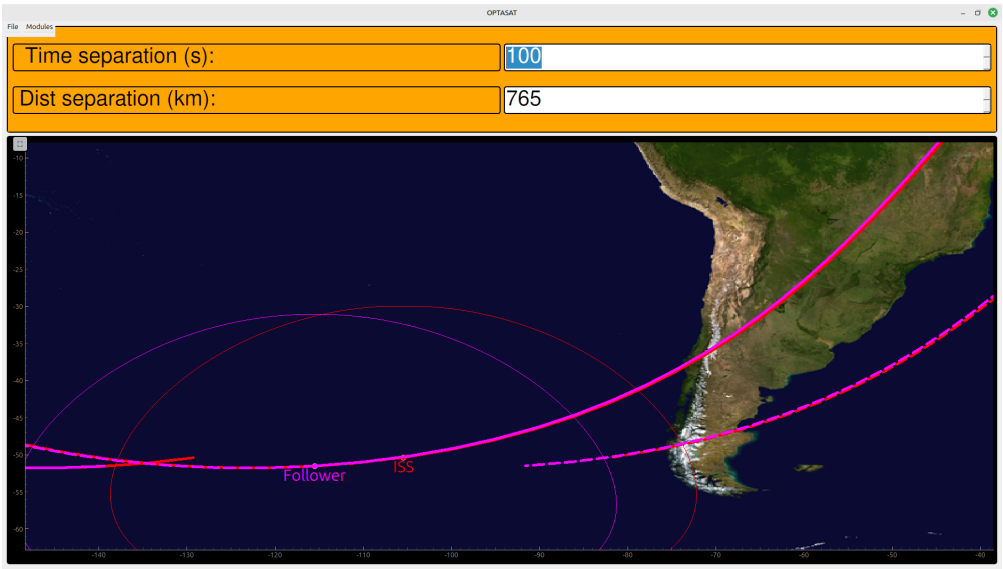

Figure 5.24: The Follower module, shown with its results on the map.

the specified orbit. For the time separation, this is straightforward as the mean anomaly changes by 360 degrees per orbit, at a constant angular rate, and thus we can divide the follower's configured time by the orbital period of the leader satellite to get the fraction of the orbit that the spacing corresponds to. From there, we apply that fraction to the full orbit's 360 degrees to find the angular difference, and offset the TLE value by that much. For the distance separation, we use the orbital velocity to convert to a time, and then apply the same conversion on the amount of time difference.

This method of generating a derived orbit is not perfect, for several reasons. One is that TLEs operate on the mean elements of an orbit. The Keplerian elements of a real object in orbit will always be fluctuating due to various perturbations. By using the same mean elements in a different location in the orbit, we will not be in exactly the same location as the leader satellite. Additionally, there is coupling between orbital elements such that, for example, the ascending node of a "True" follower satellite would not match that of the leader. Therefore, in this implementation, the follower will not occupy the precise point in space where the leader was located 100 seconds ago, but for the sake of a notional prediction, the alignment between the two orbits is sufficient to predict the dynamics between two satellites. The purpose of the Follower Satellite module is for mission architecture evaluation in the early stages of development, and therefore for the sake of understanding the dynamics of a multi-satellite mission, the accuracy should be sufficient.

5.4.13 Rectangle

The final module we will discuss should barely qualify as a module, as it has no functionality. It simply consists of an empty rectangle of a specified color which is inserted into the OPTASAT layout. The purpose of the Rectangle module is to serve as a blank slate for users to create their own modules. The Rectangle module is a minimal example of a module, and contains all the boilerplate code necessary to insert a module into OPTASAT,

including the code which enables the behavior where a module can be expanded to fill the central space of a configuration. The existence of the Rectangle module means that users can quickly create a module by only inserting the features they care about - whatever text fields, buttons, graphics, or other interface elements are desired, the user can begin adding them immediately, without needing to first work out how to make their module display on-screen in OPTASAT. The Rectangle module is the jumping off point for all other modules. It was the first module developed for OPTASAT, and all other modules were created by copying the Rectangle module and adding the required functionality to suit the specific module. The Rectangle module ultimately represents the core goal of OPTASAT to be approachable, extensible, and modifiable.

5.4.14 Modules Conclusion

This concludes the description of OPTASAT's individual included modules. The next chapter will describe the example use cases which were developed to illustrate the functionality that OPTASAT can provide, including a few sample mission concepts with baseline OPTASAT configurations, as well as a real use case where OPTASAT has already been used during development of a real spaceflight mission.

Chapter 6

Example Case Studies

The individual modules of OPTASAT each provide a useful functionality for mission development, planning, and operations. But the real benefits of the software appear when we begin to examine the results of combining the modules to work together in a completed, cohesive configuration. This chapter will explore a few examples of configurations provided with OPTASAT, finishing with a real configuration which has already been used during the development of a soon-to-launch CubeSat mission.

6.1 Subsystem Configurations

We will begin by exploring some configurations of OPTASAT which combine modules to serve the purposes of various subsystems which may apply to any type of mission. Each time OPTASAT is launched, the user has the opportunity to select any of their configurations to explore. For a given mission, there are many different subsystems which may be of interest at different times. Further, different mission personnel may have different specialties and have different information needs. Therefore, it is valuable for any given mission to have multiple configurations available, each tailored to a particular subsystem's requirements. This section will show three examples of OPTASAT layouts which serve particular subsystems. These configurations are all included with OPTASAT.

6.1.1 Power Subsystem

The first example subsystem configuration included with OPTASAT is for the Power subsystem. This configuration is shown in Figure 6.1.

In this configuration, we choose, as usual, the International Space Station. We are displaying the live telemetry feeds for the four Battery Charger Assembly (BCA) units, directly showing the state of battery power being either collected from the Sun or consumed by the Station. The live telemetry from the ISS is obtained through a Lightstreamer data stream, which provides the data through both a dashboard [61] and an API, which is what OPTASAT uses. This data is provided through a NASA feed, and is maintained by the ISS Mimic project [62], a STEM outreach project which involves a 3D-printed model of the ISS which rotates its solar panels to match the real rotations of the ISS solar panels, according

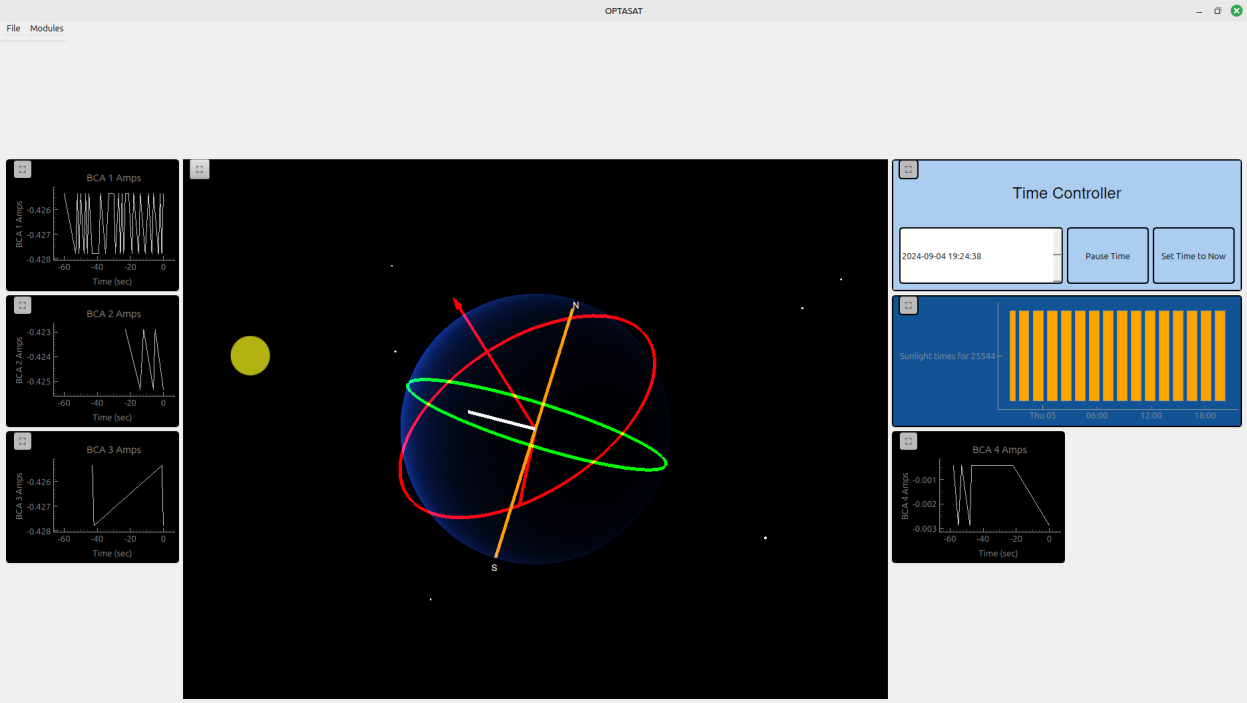

Figure 6.1: The example of the Power Subsystem Configuration.

to its telemetry. OPTASAT can similarly read any of the ISS telemetry values through the same data source. In this case, we are showing the battery charge and discharge power, as these values are highly relevant for this Power configuration.

Moving on to the other modules, we show the Beta Angle Viewer module, which will indicate the relative amount of sunlight we expect to be exposed to over the duration of an orbit. This module is currently in the expanded format for focus. Next, we have a Time Controller, which is broadly useful for almost all OPTASAT layouts. Finally, we have the Eclipse Plotter, showing the moments in time during which the ISS will and will not have sunlight available. Overall, this module gives a strong overview of the ongoing power availability for the spacecraft, at different levels of detail - the second-by-second live data from telemetry, the larger timing of when sunlight will be provided to the power system, and the longer-term view of the beta angles and how they may evolve over time (with the aid of the Time Controller).

6.1.2 Communications Subsystem

Next, we will look at the Communications subsystem view. Most importantly, this configuration is oriented with a strong focus on ground station communications and contact opportunities.

For this example, we are showing two ground stations: One at MIT, and the other in Copenhagen, Denmark. Copenhagen is the default ground station used in Gpredict, meaning that it is easy to use Gpredict to validate OPTASAT's predictions for the same ground stations. For satellites, we choose to use ISS, the Hubble Space Telescope, and GOES West.

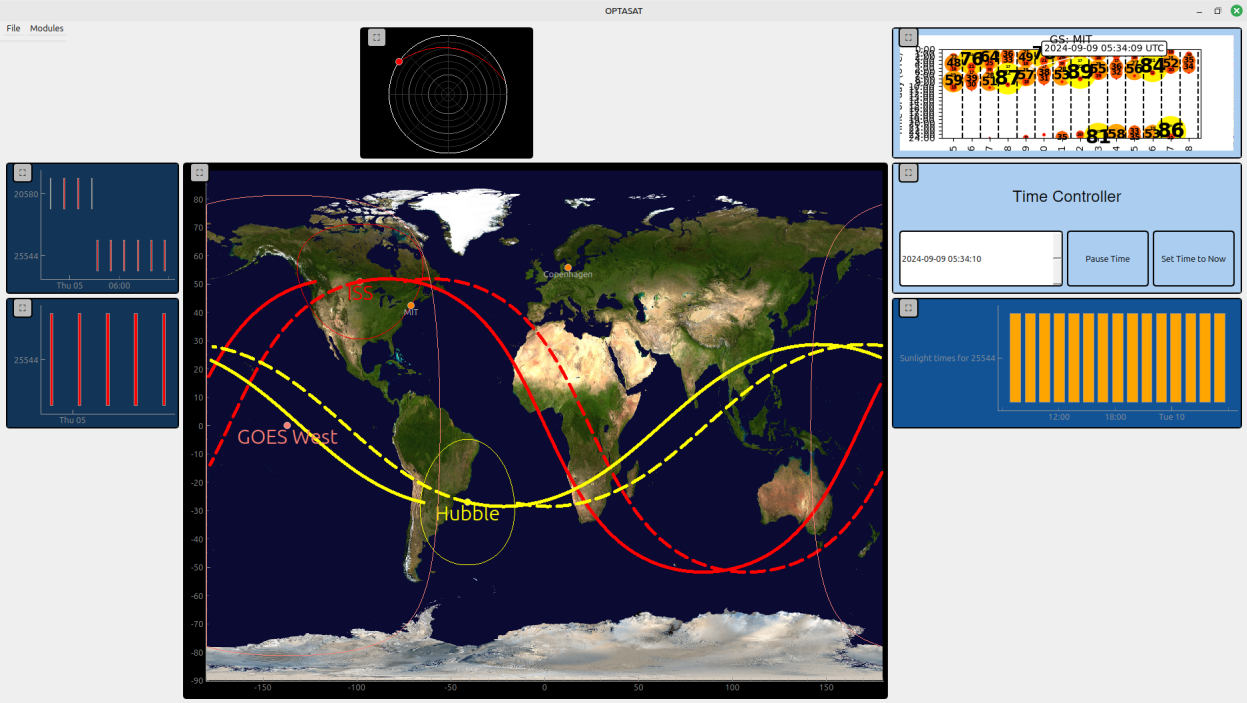

Figure 6.2: The example of the Communications Subsystem Configuration

These are all visible on the Mapdot view, which is displayed in the expanded configuration.

On the left of the configuration, we have the plots for each access opportunity for the two ground stations. Because GOES West is geostationary and does not move with respect to the ground stations, its line of sight does not change over time, so it is not plotted. On the top plot we see the different ISS and Hubble contact opportunities for the MIT ground station. Because the ISS and Hubble are nearly 180 degrees out of phase (the ISS is maximally north at the time Hubble is maximally south), their contact opportunities are currently directly disjoint. Due to Hubble's lower orbital inclination (28.5 degrees, compared to 51.6 for ISS), and Copenhagen's higher latitude than MIT, Copenhagen never gets Hubble visible in its sky, so its ground station access plot is showing only ISS as having contact opportunities.

At the top of the view, we have the Polar Pass Plot, showing the current ISS pass. In the upper right is the Pass Finder, which was used to move the time to show this pass. This Pass Finder is finding MIT-ISS passes, but could be alternatively configured to track any other passes between any satellite-ground station pair. Next we have the ever-present Time Controller, and finally the Eclipse Plotter, which will indicate when the ISS has sunlight, which can be important to contextualize data seen during communications.

Overall, this configuration acts as a cohesive, collected display of the various modules which relate to satellite-to-ground communications, and replicates much of the behavior of Gpredict.

6.1.3 ADCS Subsystem

The final subsystem configuration we will examine is for the ADCS (Attitude Determination and Control Systems) subsystem.

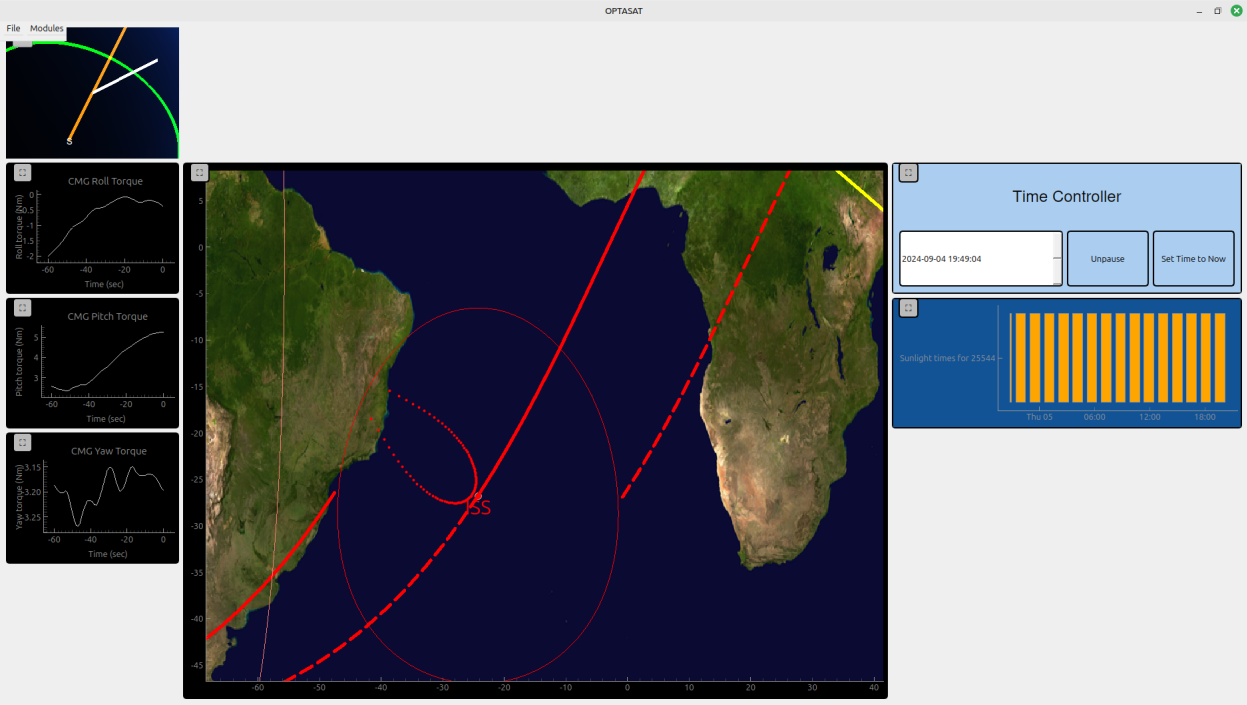

Figure 6.3: The example of the ADCS Subsystem Configuration

In this example, we imagine a spacecraft wanting to slew to view a ground target. For this purpose, we have the Mapdot being used to move a sensor to capture a target. In this case, we are over the South Atlantic Ocean, and tilting in order to view the coast of Brazil, which is just at the edge of our view. Because our viewing area is stretched enough to reach beyond the circle indicating our horizon, this indicates that the sensor's edge does not intersect the Earth, meaning we will be capturing the limb of the Earth. We also show the spacecraft's illumination status, which is important for ADCS, as during sunlit periods it is beneficial to rotate the solar panels to face the sun, while in dark periods we may want to avoid doing too many ADCS maneuvers, as they may consume more power than is available for the duration of an eclipse.

6.2 Mission Configurations

Now that we have seen OPTASAT setups for individual subsystems, we will show a few configurations which share functionality for different types of prototypical missions. These will include an Earth Sensing mission and an Astronomy mission, followed by a real application in the CLICK mission.

6.2.1 Earth Sensing Configuration

A common type of spacecraft mission, especially in the Low-Earth orbits often used for small satellites, is Earth sensing. Broadly speaking, these often consist of spacecraft which are intended to take images of the ground below, for many different purposes. These include agriculture, navigation, disaster recovery, and more emerging use cases. The baseline template OPTASAT configuration for Earth sensing is shown in Figure 6.4.

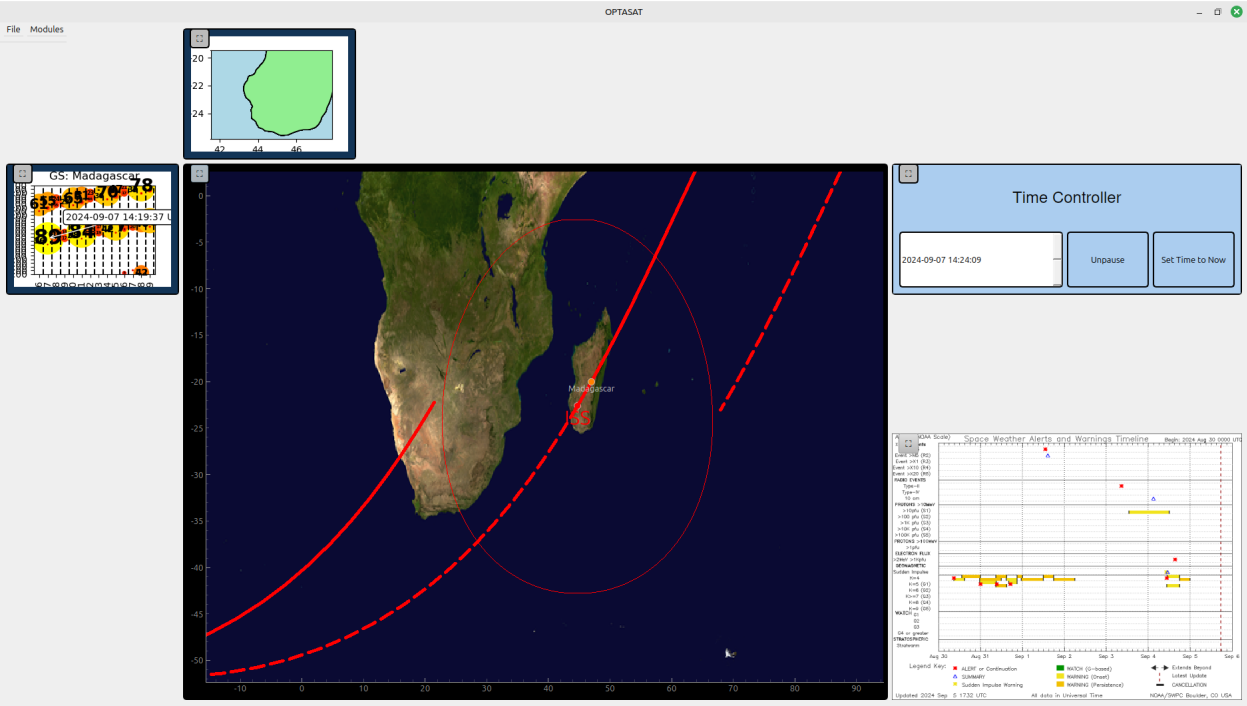

Figure 6.4: The example of the Earth Sensing Mission Configuration.

In this example, we have created a situation where we want to image Madagascar. To do this, we have created a ground station at Madagascar's coordinates (remember that as mentioned in Section 5.2.3, in OPTASAT, a ground station is just a named point on Earth). In the Communications examples, we often used ground stations to indicate the location of a radio system with which our satellite could communicate (which is indeed the normal meaning of the term "ground station"), but in this case, we are using our ground station to represent an interesting location for imaging.

In the screenshot, we see a situation where we have used the Passfinder module (on the left) to select a good pass over the imaging site. In this case, we have found an 89-degree pass, which is nearly a direct overflight. Then, we click the chosen pass to move the simulation time to the time of the pass. This takes us to the beginning of the pass, at which point we can use the Time Controller to shift time forward and back to tweak our exact chosen moment within the time of the pass (since the Passfinder's time will represent our target just approaching on the horizon). Once we are in position, we can use the Groundview module to get an idea of what our imager will see below us at the chosen time. Finally, we have our Space Weather module available to be aware of any space weather conditions which

may be relevant. This may relate to the amount of incoming radiation that the spacecraft will experience while imaging, or could indicate whether we should expect auroras to form. Depending on the type of the mission, auroras could be an important imaging target, or could be a source of noise in an image. This makes space weather an important addition to the Earth sensing configuration example.

This set of modules works well to suit the overall task of imaging locations on Earth.

6.2.2 Astronomy Configuration

Next, we will examine the OPTASAT configuration example for astronomy missions. This is notionally what may have been used on the DeMi mission, if OPTASAT were available at the time that mission operated. An example of the astronomy configuration being used is shown in Figure 6.5.

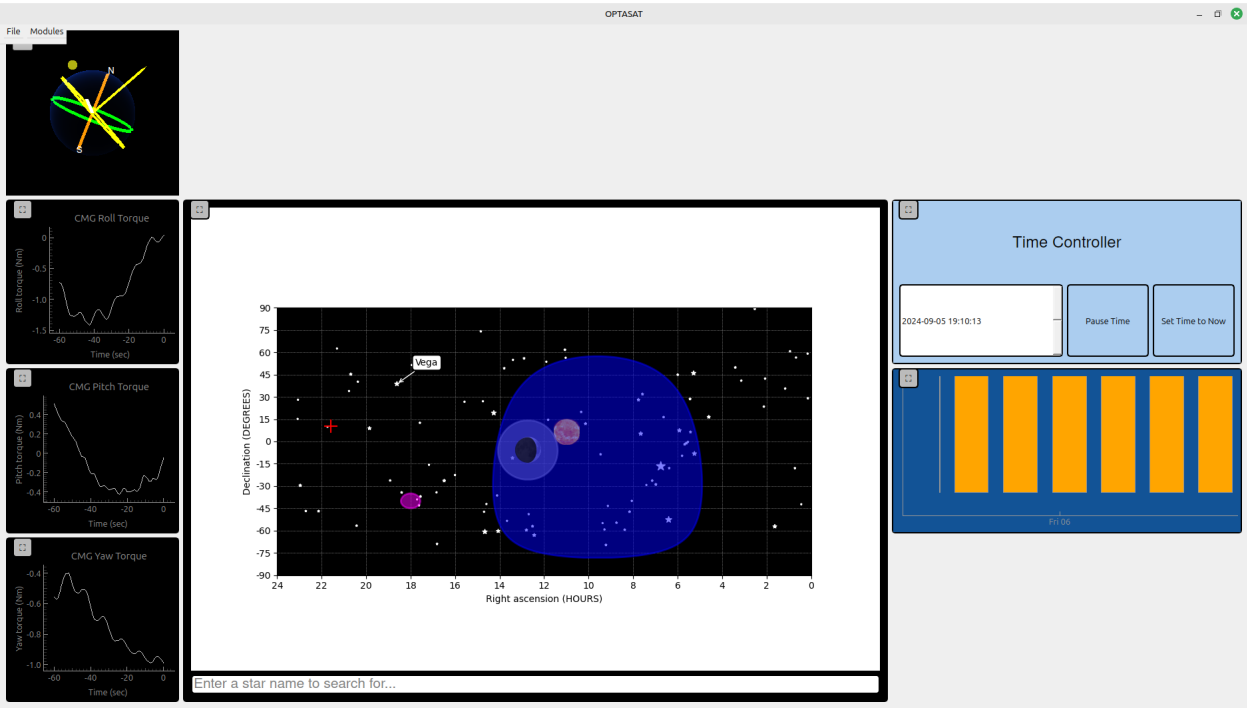

Figure 6.5: The example of the Astronomy Mission Configuration.

The Starmap is the most important module for astronomy missions and is made explicitly for that purpose, and therefore it features most prominently in this configuration. In this case, the spacecraft is in an opportune situation where the Earth is blocking both the Moon and Sun, placing it in maximum darkness. We can see on the Eclipse Plotter on the right that we are in darkness, with further sunlight coming soon. We have selected Vega for imaging, as it is a very bright star, in a good region of sky (near the red cross on the left) to minimize stray light. We also see the attitude control torques plotted on the left, which are critical for slewing to point the telescope at our chosen target. Finally, in the upper left, we have the Beta Angle Viewer, showing us the overall lighting trends throughout the orbit and allowing us to contextualize how much darkness is available for astronomical imaging. With

this configuration, a mission can confidently select its stars to target and begin collecting science data.

6.2.3 Real-Life Mission Configuration: CLICK

For the final example configuration in this chapter, we will look to the CLICK mission. CLICK refers to the CubeSat Laser Infrared Crosslink experiment, which will fly two 3U CubeSats and attempt to demonstrate laser communication between the two. The two spacecraft are currently in assembly, and will be launched to the International Space Station and deployed simultaneously. At this point, the spacecraft will use differential drag to create a separation between the two, creating the range which will be used to demonstrate the crosslink.

Figure 6.6: An artistic rendering of the CLICK mission. [63]

With CLICK's delivery date nearing, a focus has shifted toward planning for operations of the pair of spacecraft. One major question which has arisen is the nature of operating two spacecraft, in a shared orbit, with time separation. Particularly, it is not clear what the contact opportunities with the ground station will look like. If the two spacecraft are flying over the ground station and are near each other in the sky, it may be possible for the ground station's antenna to aim at a point between the two spacecraft, and capture both within its beam pattern. This would have benefits and drawbacks. The benefit is that the ground station could issue commands to the spacecraft at will, sending commands to one and then the other as needed. The drawback is that the antenna would be receiving both spacecraft at the same time, and therefore care would need to be taken in order to ensure that the two spacecraft do not transmit at the same time and pollute each other's signals. If the spacing of the spacecraft is larger, then they will pass over the ground station in sequence - first the leader would pass, and then the follower would pass. It may be the case that the first satellite starts and ends its pass before the second one's pass begins, or there may be some overlap

between the two, in which case the nature of sharing the time becomes a key question. Do we track the leader satellite throughout its pass, and then switch to the follower (which may only have 30 percent of the pass time remaining)? Do we stop communicating with the leader and transition to the follower midway through the pass? These dynamics are uncertain. While simple trigonometry can predict the dynamics of individual cases (by knowing the separation distance of the two spacecraft and their orbital altitude, we can determine their angular separation), the wide diversity in the dynamics of the different passes means that it is hard to get the full picture of this dual-spacecraft operation with simple math. Instead, we constructed an OPTASAT configuration to model the two satellites and analyze how they behave in the sky, from the perspective of the ground station. This configuration is shown in Figure 6.7.

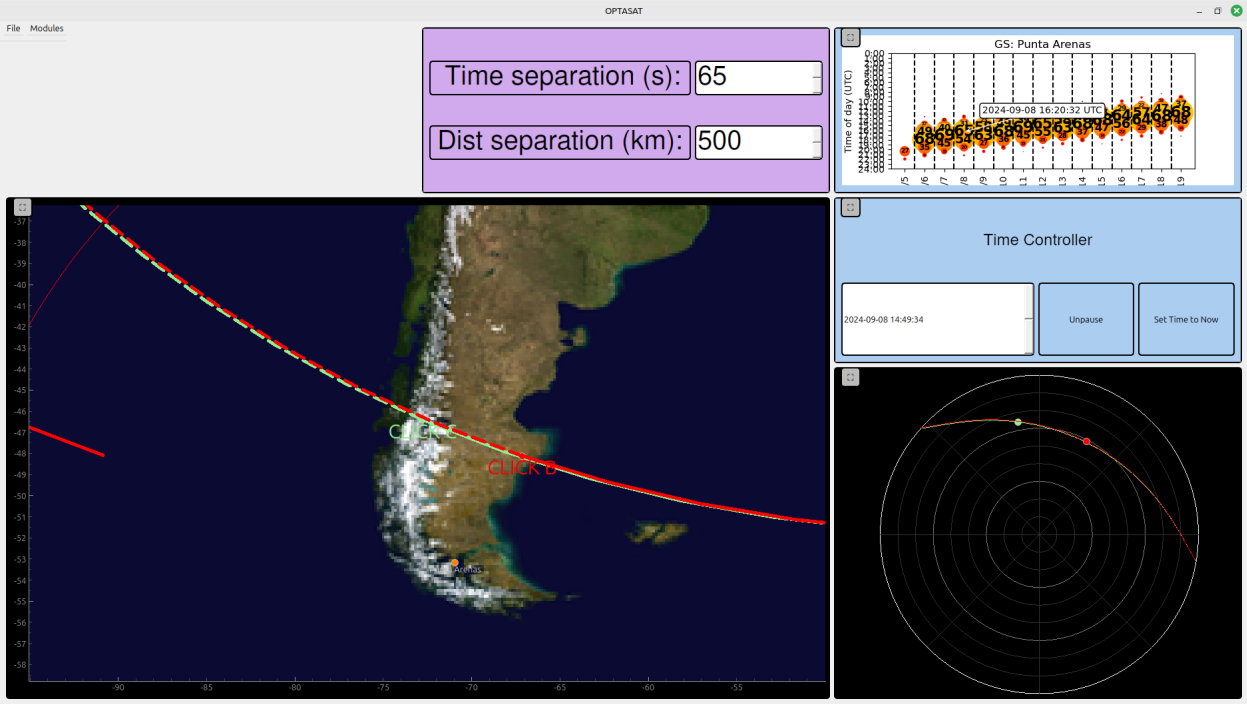

Figure 6.7: A demonstration of the CLICK satellite configuration

In this case, we are only doing mission planning, so there are no spacecraft currently in orbit that we can study directly. Instead, we choose to use the ISS as a surrogate spacecraft. Because the CLICK satellites will be deployed from it, their orbit will be highly similar to its orbit, and therefore its orbit acts as a good approximation for modeling how to expect the satellites to behave. Therefore, we create a Mapdot module where we will take the ISS TLE, but label it as CLICK B (the name shown on the map comes from the JSON file, and therefore it is possible to attach any preferred name to any spacecraft). Then, we use the Follower Satellite module (at the top, in purple) to generate a following satellite. This following satellite is labeled as CLICK C. Once we have the two satellites initialized, we can begin to look at a pass. We set up a ground station representing CLICK's ground station in Punta Arenas, Chile. We can then use the Pass Finder (in the upper right) to select a pass for analysis. Once we are viewing a given pass, the trajectories of the two satellites

are shown in the Polar Pass Plot module in the lower right, along with dots showing the locations of the two satellites at the time chosen by the Time Controller. In this example, we can see that with a 500-kilometer separation, the two satellite are roughly 30 degrees apart in the sky, which means, depending on the exact beam diameter of the ground antenna, it would likely not be possible to communicate with both at the same time.

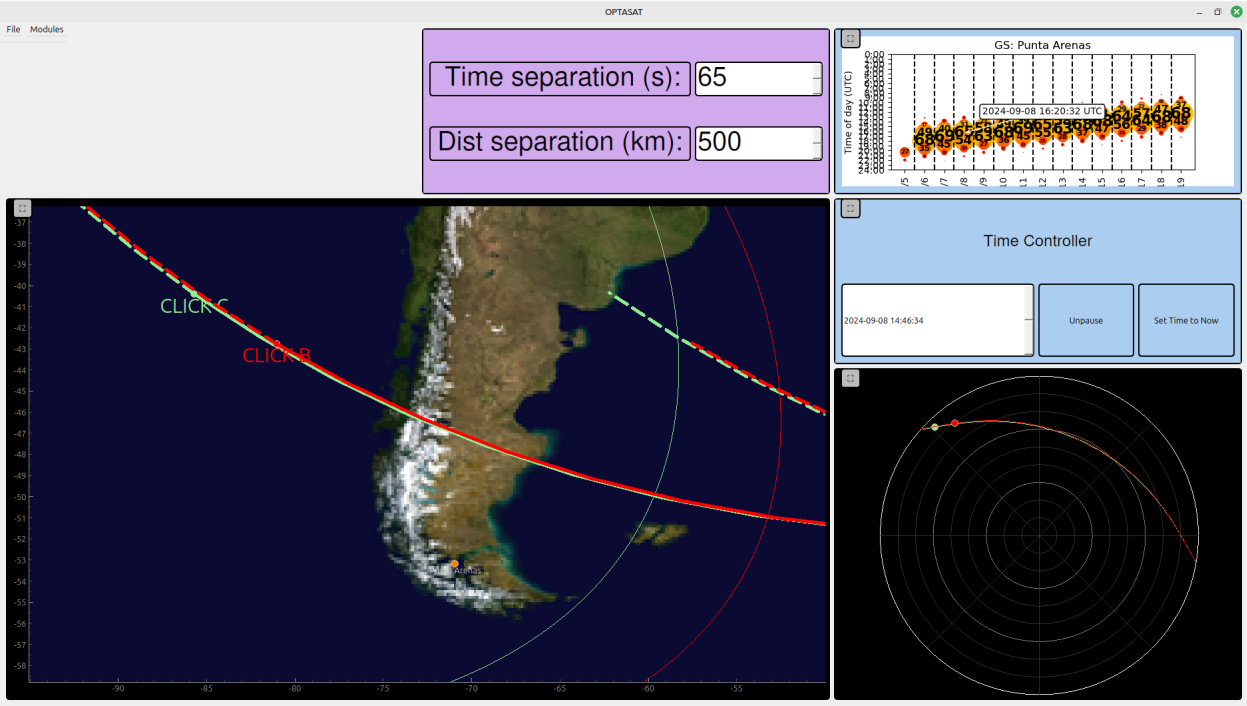

Figure 6.8: The same configuration as above, but now showing the start of the pass.

Interestingly, if we move the time a few minutes earlier, to the start of the pass (as shown in Figure 6.8), we see an emerging effect in the Polar Pass Plot. At the start of the pass, the two spacecraft are much closer to each other in the sky. This indicates that as the two satellites fly past the ground station, the begin close together, then space out, and finally regroup at the end. This means that, at the 500-kilometer separation, we may find that a pass could start by contacting both spacecraft, gathering a health status, and making decisions based on that to prioritize which to communicate with for the remainder of the pass. If one of the two satellites has experienced an anomaly, the remaining pass time can be spent focusing on that satellite and resolving its issues. We will need to commit to one satellite or the other for the middle of the pass, and could potentially switch at some point from leader to follower, if communication with both is desired, but we would not be able to maintain simultaneous communication for the whole pass.

Each separation distance that the satellites may take (which is driven by the needs of the laser demonstration mission, and may change throughout the mission) will have different impacts on the communication situation, and therefore OPTASAT presents a convenient interface to move the satellites around and see how they behave. We examined the 500-kilometer case, but CLICK intends to demonstrate laser communication at ranges from 25 km to 580 km [64] and therefore it is important to study the dynamics of an overpass over

all operating ground stations and at many ranges to evaluate the behavior of each. This set of OPTASAT modules presents a clean, accessible interface to quickly iterate through possibilities and maximize human understanding of how the two spacecraft are interacting with each other and with the ground station.

6.3 Configurations Conclusion

We have now seen six different configurations from OPTASAT and the opportunities they offer for understanding the behavior of different aspects of satellites. We have seen three different subsystem-focused configurations, two prototypical mission examples, and finally a use of OPTASAT in the planning of a real space mission which will be launching soon. Overall, this illustrates the possibilities that arise when the available modules are combined to give an analysis of a space situation from multiple angles. These configurations serve as a strong starting point for any future OPTASAT user who may need to create a configuration to serve a new purpose which has not yet been analyzed.

Chapter 7

Summary and Future Work

OPTASAT has now been fully described. With its state established, we will shift our attention to the future, and finally conclude this dissertation with a discussion of the contributions.

7.1 Future Work

OPTASAT has been released to the public under an MIT License. While the MIT License was originally developed by MIT, it has now moved beyond MIT and is a commonly used license for open source software. Briefly stated, the MIT License allows anyone, anywhere in the world, to take the software and use it as they please, including modification, distribution, or even selling it (though the business case for selling an unchanged version of a free-of-charge software is questionable). The important implication of this is that OPTASAT is available for anyone operating real spaceflight missions to use and to adjust to suit their needs. Most importantly, if they make improvements, it is hoped that they will contribute any improvements back to the original OPTASAT repository. This is a common practice in the open source community and leads to a collaborative environment where every user can benefit from the work of others. This also means that any of the future work described in this section could become part of OPTASAT, either through further work by the author, or from users who choose to contribute.

7.1.1 Historical TLE Integration and Support

Currently, OPTASAT uses a Python script called "load_tle.py" to load TLE files for any spacecraft in the simulation. When the script is asked for a TLE of a spacecraft of a given ID number (for example, 25544 is the number for the ISS), it will first check its local cache to see if it already has a TLE for that satellite. If the local cache does not contain the given TLE, then a query is made to Celestrak [65]. Celestrak is a 501(c)(3) non-profit operated by Dr. T. S. Kelso which operates a website which shares open data pertaining to spacecraft. Most importantly for OPTASAT, Celestrak hosts a catalog for all of the publicly-available TLE files for spacecraft in orbit. When the TLE is received from Celestrak, it is then provided to OPTASAT, and saved to the cache.

The cache is implemented through the simple use of text files. When the TLE for the

ISS is received, it will be saved to a file called "25544.tle". The next time the same TLE is requested, if this file exists, it will be loaded. Then, load_tle will check the age of the TLE file. If a TLE is too old, it is not useful for predicting spacecraft orbits, due to perturbations or maneuvers that the spacecraft may have performed in the meantime between the TLE epoch and the current time. The load_tle script uses 3 days as its default cutoff for TLE age; OPTASAT uses this default value. If the TLE is older than the cutoff, it will be ignored, and a new one will be fetched from Celestrak (the same as if there was no cached TLE at all). The caching system allows OPTASAT to make a single request and continue relying on the same TLE for multiple runs. This is especially useful during development, when a developer may tweak the software many times, testing it repeatedly. If the TLE were requested from Celestrak every time, it may be a waste of Celestrak's networking resources, especially if the same TLE is being sent each time. Users may wish to use a lower TLE age cutoff (for example, 1 day) in order to ensure they are running with the latest available data.

While this fetching and caching system works well, there is one primary limitation: it will currently only fetch the most current TLE for each spacecraft. If a user wants to see the ISS location from 5 years ago, then OPTASAT will dutifully propagate the TLE backward 5 years, even though the results generated will have low accuracy due to the changes in the ISS orbit that have happened since then. In order to properly simulate the orbital behavior from dates in the past, it is necessary to acquire a contemporary TLE which was generated near the time of interest. These historical TLE files are available (especially through space-track.com), but it is more complex to get these files than to get the active current TLEs. As the focus of OPTASAT is live operations, implementation of historical TLE processing has been left as work for the future. The best way to do this would likely be to implement the space-track API, such that OPTASAT can request TLE files from any arbitrary time. Furthermore, we could cache all the historical TLEs for a given object, such that it would be possible to make only a single query, and then keep the full TLE history on the user's computer, meaning that it would only be necessary to make a single request over the network to supply years of history of a satellite of interest. Right now, the caching system only caches the active, most recent TLE. When that TLE gets older than the limit (with a default of 3 days), the new TLE will be fetched and cached, and the prior cached TLE is overwritten. Modifying this system to keep all the TLE history of an object would be a useful evolution of this capability.

7.1.2 Performance Analysis and Improvements

As discussed in Chapter 5, the selection of Python as the programming language for OPTASAT comes with inherent performance trade-offs. This means that it is possible for a given OPTASAT configuration to be too computationally intensive to run on a user's computer. Primary development of OPTASAT was done on a desktop computer running Linux Mint 21, with an Intel i9-7960X CPU. This computer has been able to run all the OPTASAT simulations shown throughout this dissertation, but lower-end hardware may struggle to run some of the configurations, especially those which involve calculating many data points for many objects.

Most modules have ways of making them faster. Modules can show fewer satellites at a time (some modules, such as the Beta Angle Viewer, allow the user to toggle each

satellite in the configuration, so only those which are actively being studied will be rendered). Additionally, modules which update their displayed information automatically have a "self_update_ms" parameter in their JSON configurations; this parameter controls how often the module will recompute the data it is showing. Effectively, this controls the "frame rate" of the module. This is easy to see visually in the Mapdot module, because an update time of 1000 milliseconds leads to the satellite jumping from one position to the next every second, while an update time of 20 milliseconds leads to very smooth motion as the position is updated 50 times per second. The issue comes when the user's computer is not able to keep up with the chosen update time. If it takes the computer 30 milliseconds to compute the new position, then it will take more time than is available to generate a frame, and could run behind. The solution in this case is simply for the user to increase the time available for each frame by increasing the self_update_ms parameter for the module.

Longer-term, the best solution for future work would be to profile each module and identify where the largest amount of time is being used. Then, the logic can be studied to determine if there is a faster way to do any of the computations for a satellite. It may also be possible to identify any duplicated work between modules - the Passfinder module calculates when a given satellite will fly over a given ground station, but the Ground Station Access module also calculates ground station overpasses (importantly, for multiple satellites from a single ground station, while the Passfinder simulates one ground station to one spacecraft). Therefore, there is likely duplicated work happening between these modules. Since all modules are sharing the same computational resources, reducing duplicated work means more resources available to do the unique work for each module.

7.1.3 Graphical Enhancements

OPTASAT uses the PyQT5 library to support all of its graphical capabilities. This library contains all the infrastructure needed to interface with an operating system's graphics infrastructure. It creates the window for the software, as well as the internal contents of the window. This includes interactive buttons, sliders, text boxes, fonts, and more. PyQT5 has served all the core needs of OPTASAT, but it also comes with its limitations.

Initially, OPTASAT was developed to create a window with a fixed size of 1920 by 1080 pixels (a standard 1080p resolution). All modules were given hard-coded pixel coordinates within which to reside. However, upon user testing with different monitors, issues were found. Lower-resolution monitors could not fit the window of this size, and OPTASAT would reach beyond the dimensions of the monitor, meaning the monitor would only show a zoomed-in area of the upper left corner of the window. Higher-resolution monitors had far more pixels than OPTASAT was using, so the window became the size of a postcard, and hard to read. This led to the migration to the current grid-based system, where modules are arranged in a grid which dynamically scales to the size of the window. OPTASAT now works with arbitrary monitor sizes, and additionally can use only a fraction of the user's monitor, if the user prefers. This increases its layout flexibility, but only to a point.

One of the main limits is that the text in OPTASAT does not dynamically scale (depending on the module). A good example of this is in Figure 6.4, in the Passfinder module. The individual circles for the passes are overlapping each other, and the text runs together enough to be a bit of a garbled mess. Furthermore, the mouse-over text is an appropriate

size when the module is in its expanded mode, but in the screenshot it is large and covers a substantial portion of the passes. The X and Y axes are nearly unreadable as well. For future work, it would be highly beneficial to scrub through all the modules and ensure that they can adjust their text to look appropriate at any size.

Another limitation is that OPTASAT currently does not allow runtime modification of the layout, beyond changing which module is in the focused, expanded view. This does have many upsides, including guaranteeing stability in the layout, in that for a given configuration, all the modules will always be in the same places. Standardization can be beneficial during operations. However, users may find that they want to move modules interactively from one place to the next. In the current OPTASAT setup, this is not possible. As a matter of future work, it would be worthwhile to investigate the possibility of adding a drag-and-drop interface to modules. Perhaps in order to preserve the benefits of fixed configurations, there could be a "Lock" mode applied to a layout, which would prevent accidental adjustments from being made. Alternatively, it might be useful to have a mode where OPTASAT could launch a Layout Editor, which is meant strictly for the sake of setting up a layout graphically, rather than by modifying numbers in a JSON file.

Additionally, OPTASAT currently has a fixed single expanded module setup. It may be the case that users would want to have two different modules in an expanded state at once, to look at two important pieces of data side by side. In the future, this could be accomplished perhaps by having a dedicated "Expanded View Module", which would host other modules as needed. This way, there could be multiple Expanded View Modules presented at the same time.

7.1.4 Data Processing Enhancements

When OPTASAT is showing live spacecraft telemetry, the Telemetry modules are currently set up to do the parsing of telemetry bytes and the display of data together. Ultimately, these are two separate tasks, and a future version of OPTASAT should consider the potential for moving these to separate places in the code. This would improve the flexibility of the data analysis, since decoupling the processing and display would allow the user to mix and match their options for each type of data. It would be particularly intriguing if there were a universal data system, which would allow setting up graphs to track any internal data point. For example, the Mapdot module calculates the latitude and longitude of a spacecraft's location, and it could be useful to set up a graphing module that would hook into the Mapdot and plot the latitude as a function of time. More variables could be created which would pass data between modules and allow different modules to affect each other in more ways. When more modules can communicate with each other, more emergent behaviors can be created, and thus enable OPTASAT to serve as a more highly-featured system, beyond the capabilities of its individual modules.

7.1.5 VR/AR visualization

Virtual Reality or Augmented Reality (VR/AR) could be a particularly interesting method of familiarizing spacecraft operators with their operating conditions. In contrast to a traditional data display which appears on a simple two-dimensional computer monitor, these methods

use stereoscopic display techniques to create a three-dimensional display, which can show data to users in unique ways. A simple example of this might be displaying a spacecraft orbit surrounding the Earth. When shown on a screen, a display of an orbit will always have to make concessions for the fact that it appears in two dimensions. For a simple example of this, consider a circular orbit at zero inclination. If we display a spacecraft in this orbit, displayed from a view where the equatorial plane is horizontal, the orbit simply looks like a line. If we display it from an angle, then the orbit, being a circle viewed from an oblique angle, appears like an ellipse. Only from directly normal to the plane do we see a circle, but when normal to the plane it is then impossible to see what locations are beneath the spacecraft. Of course these limits can be mitigated by showing the orbit from multiple angles (as is possible by dragging the view, as we see in the Beta Angle Viewer), but this requires interacting with the module directly, and never truly shows the three dimensions in a single view. Using a natively three-dimensional data display presents interesting capabilities in full mission awareness. It would be valuable to investigate existing systems for presenting data in VR/AR (such as game engines like Unity or Godot) and determine whether it is possible and practicable to pass spacecraft data into these rendering pipelines, followed by identifying which types of data will be most useful to represent in this fashion.

7.1.6 Expanded Documentation

As OPTASAT stands, it has some documented features. The code is commented to describe its functionality, and there are several text files describing how to use the software. Of course, this document itself is a useful resource for anyone wanting to use OPTASAT. While the installation instructions are well-described, the process of using OPTASAT and applying it to new missions could use further development. Given the interactive, dynamic nature of the software, it would likely be best to use video as the format for relaying this information, as the actual process of using the software can be directly displayed, rather than relying on descriptions. Creating instructional usage videos for OPTASAT will be a focused item for future work.

7.1.7 User Testing and Feedback

OPTASAT has now been released to the public via GitHub [66]. This means that users will now be able to interact with the software. Given that this software was developed by one person with only a small amount of feedback from users, it is likely that there are assumptions made in the software which would not be clear to an outsider, and which are hard for an insider to identify. A new pool of users will be able to identify the potential stumbling blocks associated with OPTASAT onboarding, which will ultimately allow these imperfections to be corrected, either through modifying the software to better suit user expectations, or through improving documentation to make the particulars of the software more explicit.

OPTASAT was presented at the Small Satellite Conference in 2024 [67], which may lead to familiarity in the small satellite community and ultimately generate users of the software. Further publication opportunities will be sought in order to expand the reach of OPTASAT awareness. Once external users begin to use OPTASAT, more development can

be done based on their feedback. It may also be worthwhile to perform dedicated user testing sessions, with focused feedback, rather than relying on organic discovery of the software.

As another method of expanding the reach of OPTASAT and reducing the barrier to entry, it would likely be worthwhile to produce educational tutorial videos to assist users in getting started with OPTASAT. A video is able to convey more information than text is, especially for an interactive, dynamic software suite like OPTASAT. Videos also tend to be more digestible than dense technical content, and can make OPTASAT feel more friendly to a new user. Video topics could include how to use the examples included in OPTASAT, how to make new modules, and how to write JSON configuration files. Additionally, these videos would be beneficial for use in educational courses, as they could be used to make a large pool of students familiar with OPTASAT as a tool for understanding spaceflight scenarios. The Beta Angle Viewer module could be particularly useful in education, because of the unintuitive nature of beta angles arising from the arrangement of multiple planes in space. A three-dimensional view as OPTASAT provides can be more educational than a figure in a textbook. By producing videos of OPTASAT, we would expand the userbase and make it even easier for new spacecraft operators to be ready to operate satellites confidently.

7.2 Conclusion

OPTASAT is a complete software toolkit for small spacecraft operators. It represents extensive groundwork toward supporting many of the universal needs for people operating a satellite for the first time. It emphasizes visualization of data in order to make the spacecraft's state easy to understand. The users of OPTASAT are able to operate a spacecraft with it better than they could prior to OPTASAT's development.

We revisit the contributions one more time. Contribution 1 is the development of an open-source tool for the operation of spacecraft. OPTASAT definitively serves this purpose. With its full release now complete, OPTASAT will be able to serve the needs of spacecraft operators who need a low-cost solution that can be modified to fit their needs. OPTASAT will continue to be adapted to have as much capability as possible.

Contribution 2 is the use of existing open-source tools to enhance mission status awareness. OPTASAT fulfills this contribution through its use of Python libraries and public datasets. It uses the Skyfield library [39], discussed in Section 2.3. Skyfield performs all of OPTASAT's internal orbital dynamics, including TLE processing and propagation. We also use the Hipparcos star catalog (which Skyfield itself presents an interface to) as the source of data for the stars used in the Starmap and Beta Angle Viewer modules. For the Ground View module in section 5.4.3 we use open-source data sets for the Shapefiles which describe the geometry of coasts and borders around the world. Other data sources are also included in OPTASAT, including its built-in TLE fetching functionality, and the Space Weather module, which obtains data from NOAA. These modules which present different ways to access external data also act as strong examples to users of how to access their own open-source data of interest.

Finally, Contribution 3 is the ability for OPTASAT to export data to other software for further analysis. This functionality was described in 5.3. Any data relating to the state of the OPTASAT simulation is able to be written to the JSON output, which can then be

processed by external tools. The fact that JSON is an open standard means it is practical for users to take the JSON and convert it into any bespoke format needed for their particular external software. The data exporting utility means that OPTASAT data is not confined to OPTASAT, and can integrate into any required toolchain.

The fulfillment of these contributions confirms that OPTASAT is an operational tool which is ready to serve the needs of small spacecraft operators. OPTASAT's ongoing development will continue to fill the gaps we identified in Chapter 2. The new types of mission operators, who need to perform with limited resources and custom needs, will now have OPTASAT available to fulfill these needs.

OPTASAT was initially announced in August 2024 at the Small Satellite Conference [67], and was released on GitHub at the same time. OPTASAT can be downloaded at https://github.com/bismurphy/OPTASAT.

Appendix A

Appendix A: Examples of JSON configuration files

Here, we present a small selection of JSON configuration files for different OPTASAT layouts. The purpose of this is to give some context for what these files look like, and to share which parts of them vary and which are more constant.

Here is the JSON for the CLICK Following example, as shared in Section 6.2.3.

```
{
    "Spacecraft_IDS": [
        25544
    ],
    "Groundstations": [
        {
            "Name": "Punta Arenas",
            "Lat": -53.166667,
            "Lon": -70.933333
        }
    ],
    "central_geometry": [0, 1, 4, 3],
    "modules": [
        {
            "source_file": "follower_sat",
            "initparams": {
                "name": "Follower",
                "grid_x": 2,
                "grid_y": 0,
                "grid_w": 2,
                "grid_h": 1,
                "color": "#d0aaed",
                "sat_ID": 99999,
                "leader_ID": 25544
            }
        },
        {
            "source_file": "Timecontrol",
            "initparams": {
                "name": "TimeController",
                "grid_x": 4,
                "grid_y": 1,
                "grid_w": 2,
                "grid_h": 1,
```

```
                "color": "#abcdef",
                "self_update_ms": 1000
            }
        },
        {
            "source_file": "mapdot.mapdot",
            "initparams": {
                "name": "Mapview",
                "grid_x": 0,
                "grid_y": 0,
                "grid_w": 2,
                "grid_h": 1,
                "self_update_ms": 1000,
                "SATS": [
                    {
                        "Name": "CLICK C",
                        "ID": 99999,
                        "Color": "lightgreen"
                    },
                    {
                        "Name": "CLICK B",
                        "ID": 25544,
                        "Color": "red"
                    }
                ]
            }
        },
        {
            "source_file": "pass_polar",
            "initparams": {
                "name": "PolarPassPlot",
                "grid_x": 4,
                "grid_y": 2,
                "grid_w": 2,
                "grid_h": 2,
                "self_update_ms": 1000,
                "groundstation": "Punta Arenas",
                "SATS": [
                    {
                        "ID": 99999,
                        "Color": "lightgreen"
                    },
                    {
                        "ID": 25544,
                        "Color": "red"
                    }
                ]
            }
        },
        {
            "source_file": "passfinder",
            "initparams": {
                "name": "Passplot",
                "grid_x": 4,
                "grid_y": 0,
                "grid_w": 2,
                "grid_h": 1,
                "color": "#abcdef",
                "sat_id": 25544,
```

```
                "groundstation": "Punta Arenas"
            }
        }
    ]
}
```

Next, this is the Astronomy Mission Configuration, which was featured in Section 6.2.2. This is particularly useful for seeing both the syntax for the keepout zones, and the way that we can create multiple copies of the same module with different parameters - specifically the Telemetry module.

```
{
"Spacecraft_IDS":[25544,20580,51850],
"central_geometry":[1,1,4,3],
"modules":[
    {
    "source_file":"Timecontrol",
    "initparams":{
        "name":"TimeController",
        "grid_x": 5,
        "grid_y": 1,
        "grid_w": 2,
        "grid_h": 1,
        "color":"#abcdef",
        "self_update_ms":1000
        }
    },
    {
    "source_file":"beta_angle",
    "initparams":{
        "name":"Betaview",
        "grid_x": 0,
        "grid_y": 0,
        "grid_w": 1,
        "grid_h": 1,
        "self_update_ms":1000,
        "SATS":[{
                "ID":25544,
                "Color":"red"
                },
                {
                "ID":20580,
                "Color":"yellow"
                },
                {
                "ID":51850,
                "Color":"salmon"
                }
                ]
        }
    },
    {
    "source_file":"telemetry",
    "initparams":{
        "name":"CMG Roll Torque",
        "telemetry_item":"USLAB000006",
        "ylabel":"Roll torque (Nm)",
        "x_timerange":60,
        "grid_x": 0,
```

```
        "grid_y": 1,
        "grid_w": 1,
        "grid_h": 1,
        "color":"#2F74EE"
        }
    },
    {
    "source_file":"telemetry",
    "initparams":{
        "name":"CMG Pitch Torque",
        "telemetry_item":"USLAB000007",
        "ylabel":"Pitch torque (Nm)",
        "x_timerange":60,
        "grid_x": 0,
        "grid_y": 2,
        "grid_w": 1,
        "grid_h": 1,
        "color":"#E29883"
        }
    },
    {
    "source_file":"telemetry",
    "initparams":{
        "name":"CMG Yaw Torque",
        "telemetry_item":"USLAB000008",
        "ylabel":"Yaw torque (Nm)",
        "x_timerange":60,
        "grid_x": 0,
        "grid_y": 3,
        "grid_w": 1,
        "grid_h": 1,
        "color":"#E29883"
        }
    },
    {
    "source_file":"starmap.starmap",
    "initparams":{
        "name":"starmap",
        "grid_x": 2,
        "grid_y": 0,
        "grid_w": 2,
        "grid_h": 1,
        "color":"#000000",
        "star_mag_limit": 2.5,
        "sat_id":20580,
        "self_update_ms": 1000,
"keepouts": [
                    {
                        "color": "#AAAAAA",
                        "radius": 20,
                        "center": "moon"
                    },
                    {
                        "color": "#FF00FF",
                        "radius": 5,
                        "center": [270,-40]
                    }
                ]
        }
```

```
    },
    {
        "source_file":"eclipse_plot",
        "initparams":{
            "name":"Eclipse",
            "grid_x": 5,
            "grid_y": 2,
            "grid_w": 2,
            "grid_h": 1,
            "color":"#125396",
            "sat_id":25544,
            "self_update_ms":1000
            }
        }
]
}
```

For more examples of configuration files, please download OPTASAT, or browse its Github files. There are more examples of configurations in the "config_files" folder within that project.

Appendix B

Appendix B: Rotated Capability Table

This is a duplicate of Table 2.1, presented on the following page in order to make the relatively large amount of information more readable. The data presented in the table is identical. It is simply presented here rotated and scaled up in order to take advantage of the aspect ratio of the page.

Property	Basilisk	COSMOS	FreeFlyer	GMAT	GPredict	JPL AMMOS	STK	OPTASAT
Simulate simple orbital trajectories around Earth	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Simulate complex trajectories to other bodies	Yes	No	Yes	Yes	No	Yes	Yes	No
Simulate propulsive maneuvers	Yes	No	Yes	Yes	No	Yes	Yes	No
Astronomy Target Simulation	No	No	No	Yes	No	Yes	Yes	Yes
Support for custom code interfacing M = MATLAB, P = Python, R = Ruby	P	P, R	M, P	M, P	No	No	M, P	P (Native)
Cost	$0	$0	$$	$0	$0	$0 (requires license)	$$$$	$0
Open-Source	C/C++	Ruby	No	C++	C	Some parts	No	Python
Live Spacecraft Data Integration	No	Yes	No	No	No	Yes	No	Yes
Export Controlled	No	No	Yes	No	No	Unknown	Yes (Some versions)	No

References

[1] J. Radtke, E. Stoll, H. Lewis, and B. B. Virgili. "The impact of the increase in small satellite launch traffic on the long-term evolution of the space debris environment". In: *Proceedings of 7th European Conference on Space Debris* (ESOC, Darmstadt (Germany), Apr. 18–21, 2017). ESA Space Debris Office, Apr. 2017.

[2] California Polytechnic State University, San Luis Obispo (Cal Poly) CubeSat Systems Engineer Lab. *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*. NASA, Oct. 2017. URL: https://www.nasa.gov/wp-content/uploads/2017/03/nasa_csli_cubesat_101_508.pdf (visited on 07/12/2024).

[3] Y. Wang, Y. Wu, Q. Yang, and J. Zhang. "Anomaly Detection of Spacecraft Telemetry Data Using Temporal Convolution Network". In: *2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. 2021. DOI: 10.1109/I2MTC50364.2021.9459840.

[4] J. He, Z. Cheng, and B. Guo. "Anomaly Detection in Satellite Telemetry Data Using a Sparse Feature-Based Method". In: *Sensors* 22.6358 (2022). DOI: 10.3390/s22176358.

[5] R. K. Schaefer, L. Paxton, C. Selby, B. Ogorzalek, G. Romeo, B. Wolven, and S.-Y. Hsieh. "Observation and modeling of the South Atlantic Anomaly in low Earth orbit using photometric instrument data". In: *Space Weather* 14.5 (May 2016). DOI: 10.1002/2016SW001371.

[6] D. González-Bárcena, J. Bermejo-Ballesteros, I. Pérez-Grande, and Á. Sanz-Andres. "The worst-case thermal environment parameters of small satellites based on Real-Observation Data". In: *50th International Conference on Environmental Systems*. July 2021.

[7] L. Zhou, M. Divakarla, and X. Liu. "An Overview of the Joint Polar Satellite System (JPSS) Science Data Product Calibration and Validation". In: *Remote Sensing* 8.139 (Feb. 2016). DOI: 10.3390/rs8020139.

[8] X. Xiong et al. "VIIRS on-orbit calibration methodology and performance". In: *Journal of Geophysical Research: Atmospheres* 119.9 (Oct. 2013). DOI: 10.1002/2013JD020423.

[9] T. Schneider. *Open Source vs. Open APIs*. Jan. 2007. URL: https://toni.org/2007/01/30/open-source-vs-open-apis/ (visited on 08/13/2024).

[10] Microsoft. *Microsoft Flight Simulator SDK Documentation: Programming APIs*. URL: https://docs.flightsimulator.com/flighting/html/Programming_Tools/Programming_APIs.htm.

[11] M. Heron, V. L. Hanson, and I. Ricketts. "Open source and accessibility: advantages and limitations". In: *Journal of Interaction Science* (Oct. 2013). DOI: 10.1186/2194-0827-1-2.

[12] R. E. Morgan et al. "On-Orbit Operations Summary for the Deformable Mirror Demonstration Mission (DeMi) CubeSat". In: *Adaptive Optics Systems VIII*. 2022.

[13] A. A. Goodenough and S. D. Brown. "DIRSIG5: Next-Generation Remote Sensing Data and Image Simulation Framework". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 10.11 (Nov. 2017). DOI: 10.1109/JSTARS.2017.2758964.

[14] Digital Imaging and Remote Sensing Lab. *DIRSIG: Digital Imaging and Remote Sensing Image Generation*. URL: http://dirsig.org/index.html (visited on 09/11/2024).

[15] Rendered.ai. URL: https://rendered.ai/industries/#earth (visited on 08/13/2024).

[16] MathWorks. *Simscape: Model and simulate multidomain physical systems*. URL: https://www.mathworks.com/products/simscape.html (visited on 09/10/2024).

[17] Ansys. *Ansys STK Software for Digital Mission Engineering and Systems Analysis*. URL: https://www.ansys.com/products/missions/ansys-stk (visited on 08/13/2024).

[18] S. P. Hughes. *General Mission Analysis Tool (GMAT)*. URL: https://ntrs.nasa.gov/citations/20160003520 (visited on 08/13/2024).

[19] L. Maisonobe and V. Pommier-Maurussane. "Orekit : an Open-source Library for Operational Flight Dynamics Applications". In: *4th International Conference on Astrodynamics Tools and Techniques*. May 2010.

[20] P. Hyvönen. *Orekit Map Example*. URL: https://gitlab.orekit.org/orekit-labs/python-wrapper/blob/master/examples/orekit_map.ipynb (visited on 09/11/2024).

[21] A. Csete. *About Gpredict*. URL: https://oz9aec.dk/gpredict/index.php (visited on 08/13/2024).

[22] *AMMOS Model Test Drive*. URL: https://ammos.nasa.gov/aerie-docs/tutorials/mission-modeling/first-model-test/ (visited on 09/11/2024).

[23] NASA. *Advanced Multi-Mission Operations System*. URL: https://ammos.nasa.gov/ (visited on 08/13/2024).

[24] NASA. *Open MCT - Open Source Mission Control Software*. URL: https://nasa.github.io/openmct/ (visited on 08/13/2024).

[25] NASA Ames Research Center. *Open MCT User's Guide*. Mar. 9, 2021. URL: https://nasa.github.io/openmct/static/files/Open_MCT_Users_Guide.pdf (visited on 09/11/2024).

[26] OpenC3. *OpenC3 COSMOS Documentation*. URL: https://docs.openc3.com/docs (visited on 08/13/2024).

[27] OpenC3. *The software for integration, test, and operations*. URL: https://openc3.com/ (visited on 09/11/2024).

[28] Exotrail. *SpaceTower*. URL: https://www.exotrail.com/product/spacetower (visited on 08/13/2024).

[29] Orbit Logic. *Orbit Logic*. URL: http://www.orbitlogic.com/index.html (visited on 08/13/2024).

[30] Orbit Logic. *Collection Planning and Analysis Workstation*. URL: http://www.orbitlogic.com/collection-planning-and-analysis-workstation.html (visited on 09/11/2024).

[31] D. Wright. *Monte Carlo Transport Codes for use in the Space Radiation Environment*. URL: https://three.jsc.nasa.gov/articles/MC-THREE-GEANT4.pdf (visited on 09/11/2024).

[32] CERN. *Geant4, A Simulation Toolkit*. URL: https://geant4.web.cern.ch/ (visited on 08/13/2024).

[33] E. S. Agency. *MULASSIS*. URL: https://essr.esa.int/project/mulassis.

[34] E. S. Agency. *SPENVIS: The Space Environment Information System*. URL: https://www.spenvis.oma.be/.

[35] STScI. *POPPY*. URL: https://poppy-optics.readthedocs.io/en/latest/ (visited on 08/13/2024).

[36] Space Telescope Science Institute. *Example Code and Getting Started*. URL: https://poppy-optics.readthedocs.io/en/latest/examples.html (visited on 09/11/2024).

[37] Cesium. *CesiumJS*. URL: https://cesium.com/platform/cesiumjs/ (visited on 08/13/2024).

[38] OpenStreetMap contributors. *OpenStreetMap*. URL: https://www.openstreetmap.org (visited on 08/13/2024).

[39] B. Rhodes. *Skyfield: High precision research-grade positions for planets and Earth satellites generator*. URL: https://ui.adsabs.harvard.edu/abs/2019ascl.soft07024R/abstract (visited on 08/13/2024).

[40] B. Rhodes. *Example Plots*. URL: https://github.com/skyfielders/python-skyfield/blob/master/documentation/example-plots.rst (visited on 09/11/2024).

[41] M. Holliday. *PyCubed: The Big Picture*. URL: https://rexlab.ri.cmu.edu/projects/pycubed.html (visited on 09/11/2024).

[42] M. Holliday, A. Ramirez, C. Settle, T. Tatum, D. Senesky, and Z. Manchester. "PyCubed: An Open-Source, Radiation-Tested CubeSat Platform Programmable Entirely in Python". In: *33rd Annual AIAA/USU Conference on Small Satellites*. 2019.

[43] P. W. Kenneally, S. Piggott, and H. Schaub. "Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework". In: *Journal of Aerospace Information Systems* 17.9 (Sept. 2020), pp. 496–507. URL: https://hanspeterschaub.info/PapersPrivate/Kenneally2020a.pdf.

[44] Autonomous Vehicle Systems Laboratory. *Hyperbolic Jupiter Arrival Orbit*. URL: http://hanspeterschaub.info/basilisk/examples/scenarioJupiterArrival.html (visited on 09/11/2024).

[45] a.i. solutions. *FreeFlyer Space Mission Design, Analysis, & Operations Software*. URL: https://ai-solutions.com/freeflyer-astrodynamic-software/ (visited on 09/11/2024).

[46] G. van Rossum. *Python 0.9.1 part 01/21*. Feb. 1991. URL: https://www.tuhs.org/Usenet/alt.sources/1991-February/001749.html (visited on 08/23/2024).

[47] A. Crews, K. Cahoy, W. Blackwell, R. V. Leslie, I. Osaretin, M. DiLiberto, A. Milstein, and M. Grant. "Initial Radiance Validation of On-orbit MicroMAS-2A Data". In: *2019 United States National Committee of URSI National Radio Science Meeting (USNC-URSI NRSM)*. 2019, pp. 1–2. DOI: 10.23919/USNC-URSI-NRSM.2019.8713131.

[48] W. J. Blackwell. "The MicroMAS and MiRaTA CubeSat atmospheric profiling missions". In: *2015 IEEE MTT-S International Microwave Symposium*. 2015, pp. 1–3. DOI: 10.1109/MWSYM.2015.7166742.

[49] P. do Vale Pereira et al. "BeaverCube: Coastal Imaging with VIS/LWIR CubeSats". In: *34th Annual AIAA/USU Conference on Small Satellites*. Aug. 2020.

[50] W. Kammerer et al. "CLICK-A: Optical Communication Experiments from a CubeSat Downlink Terminal". In: *37th Annual AIAA/USU Conference on Small Satellites*. Aug. 2023.

[51] S. Cass. *The Top Programming Languages 2024*. URL: https://spectrum.ieee.org/top-programming-languages-2024 (visited on 08/23/2024).

[52] MIT. *MIT Course Catalog Degree Charts: Aerospace Engineering*. URL: https://catalog.mit.edu/degree-charts/aerospace-engineering-course-16/ (visited on 08/23/2024).

[53] Qt Group. *Qt for Python Documentation: QGridLayout*. URL: https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QGridLayout.html (visited on 09/12/2024).

[54] International Organization for Standardization. *ISO/IEC 21778:2017 Information technology — The JSON data interchange syntax*. URL: https://www.iso.org/standard/71616.html (visited on 08/23/2024).

[55] T. S. Kelso. *Frequently Asked Questions: Two-Line Element Set Format*. URL: https://celestrak.org/columns/v04n03/ (visited on 08/23/2024).

[56] NASA. *Blue Marble: Land Surface, Shallow Water, and Shaded Topography*. URL: https://visibleearth.nasa.gov/images/57752/blue-marble-land-surface-shallow-water-and-shaded-topography (visited on 08/23/2024).

[57] Nikon. *Nikon D5 DSLR Camera Tech Specs*. URL: https://www.nikonusa.com/p/d5/1557/overview#tech-specs (visited on 08/23/2024).

[58] NASA. *ISS069-E-91989*. URL: https://eol.jsc.nasa.gov/searchphotos/photo.pl?mission=ISS069&roll=E&frame=91989 (visited on 08/23/2024).

[59] M. A. C. Perryman et al. "The HIPPARCOS Catalogue". In: *Astronomy and Astrophysics* 323 (July 1997), pp. L49–L52.

[60] N. S. W. P. Center. *Notifications Timeline*. URL: https://services.swpc.noaa.gov/images/notifications-timeline.png.

[61] I. M. Project. *ISS Mimic Dashboard*. URL: https://iss-mimic.github.io/Mimic/dashboard.html.

[62] I. Mimic. *ISS Mimic*. URL: https://github.com/ISS-Mimic/Mimic.

[63] R. Morgan. *Nanosatellite Lasercom System*. 2017. URL: https://digitalcommons.usu.edu/smallsat/2017/all2017/122/.

[64] P. Serra, H. Tomio, O. Cierny, W. Kammerer, P. Grenfell, and et al. "CubeSat laser infrared crosslinK mission status". In: *International Conference on Space Optics—ICSO 2020*. 2021.

[65] T. S. Kelso. *Celestrak*. URL: https://celestrak.org/.

[66] Thomas J. Murphy. *OPTASAT*. URL: https://github.com/bismurphy/OPTASAT (visited on 09/12/2024).

[67] T. Murphy and K. Cahoy. "OPTASAT: An Open-Source, Flexible Software Framework for Small Satellite Operations". In: *38th Annual Small Satellite Conference* (Utah State University, Logan, Utah, USA, Aug. 8, 2024). Utah State University, Aug. 2024.