Final Report

Lunar Pit Patrol

Levi Watlington, Alden Smith, Caden Tedeschi, and Evan Palmisano

Sponsor: Trent Hare

Mentor: Vahid Nikoonejad Fard

11/05/2024

Lunar Pit Patrol 1
Introduction 3
Process Overview 3
Requirements 4
Architecture and Implementation 5
System Overview 5
Detailed Overview 7
Conclusion 8
Testing 9
Unit Testing 10
Integration Testing 10
Usability Testing 11
Results and Adjustments 11
Project Timeline 12
Future Work 14
Conclusion 14
Glossary 16
Appendix A: Development & Environment Toolchain **17**
Hardware 17
Toolchain 17
Setup 17
Production Cycle 18
Application Open Development 18
Application Closed Development 19

Introduction

When astrogeologists attempt to determine the age of a planetary surface, they often rely on studying impact craters. Specifically, they analyze the number and size of craters, as well as their relationships to one another. Generally, a higher density of craters within a larger crater indicates an older surface. On planetary bodies with erosive processes, like Earth, older craters also tend to show greater signs of weathering and degradation.

To assist with dating surfaces, astrogeologists use a command-line application called craterstats, which generates plots based on input datasets. While craterstats is a powerful tool, its command-line interface (CLI) can be cumbersome and challenging to use, especially for those unfamiliar with command-line operations. This is where our project comes in. We've developed a graphical user interface (GUI) for craterstats, designed to make the application more user-friendly, accessible, and efficient to learn.

Using the standalone CLI can be frustrating for astrogeologists. For instance, entering an incorrect character in a file name requires retyping the entire command. Adjusting dataset parameters or modifying plot settings often involves constructing long, complex commands, and a single error necessitates rewriting the entire input. These inefficiencies make the CLI both time-consuming and error-prone.

Our GUI eliminates these challenges by providing an intuitive interface. craterstats settings are organized into clearly labeled tabs, allowing users to adjust parameters logically and with ease. The generated plots are displayed live within the interface, giving immediate feedback on changes. Additionally, for transparency and educational purposes, the GUI shows the corresponding CLI command required to produce the same graph, enabling users to verify outputs or transition seamlessly between the GUI and CLI if desired.

By simplifying the workflow, our GUI empowers astrogeologists to focus on their research rather than navigating technical hurdles, significantly enhancing their productivity and experience.

Process Overview

We adopted an agile methodology throughout the project's lifecycle, prioritizing frequent iterations, continuous testing, and open communication both within our team and with the client. This approach ensured that our deliverables consistently aligned with client expectations while remaining flexible to adapt to evolving requirements.

For version control and collaboration, we utilized GitHub, which facilitated efficient management of code changes and streamlined teamwork. To maintain productivity and organization, we conducted bi-weekly team meetings to assess progress, address challenges, and allocate tasks effectively.

Our development environment leveraged Conda as the Python environment and package manager, ensuring smooth dependency management and compatibility across team members' systems. To maintain a clear and structured project timeline, we utilized OnlineGantt, a free online Gantt chart tool, that provides a visual representation of milestones and deadlines.

To create a reliable, cross-platform GUI, we implemented the Python-based Flet framework, a distribution of Flutter that ensures compatibility across major operating systems. This choice allowed us to develop a seamless interface while maintaining a consistent user experience. For backend processing, we integrated the CraterStats CLI, ensuring that our application retained all the functionality and options provided by the original command-line tool.

In addition to the software, we deployed a dedicated website hosted on the NAU CEFNS server. The website serves as an introduction to our product, offering essential documentation and resources for users.

Each team member took on specific roles aligned with their expertise, fostering a balanced division of labor. Our structured workflows, supported by collaborative tools and clear communication channels, ensured that the project remained on track and well-organized from start to finish.

Requirements

When we received the project proposal from our sponsor, we were given four key functional requirements. These requirements included the ability to open a plot file from a user’s system in the GUI, the incorporation of every command-line application function into the GUI for user selection, the ability to generate accurate plots using the command-line functions, and the capability to export plots in file formats supported by the application. These requirements were designed to ensure the application would be straightforward and user-friendly for the average astrogeologist.

Using these functional requirements as a foundation, we developed a set of non-functional requirements aimed at enhancing the user experience. These non-functional requirements included ease of use, accessibility, clear navigation, consistency, simplicity, time efficiency, intuitiveness, and a low learning curve. Each of these was carefully crafted to meet our overarching goal of making the application

accessible and efficient for astrogeologists, enabling them to complete their work more quickly and with less frustration.

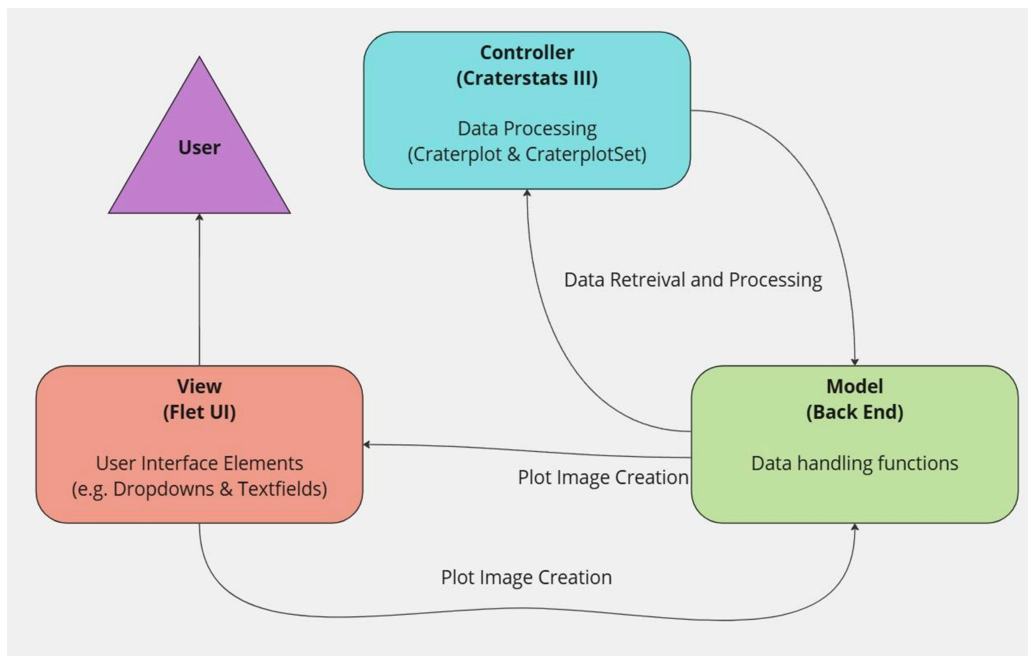
Ease of use ensures astrogeologists can navigate the application without confusion. Accessibility means the application can be easily installed and run on their systems. Clear navigation allows users to quickly find the options they need, while consistency ensures similar options are grouped logically and predictably. Simplicity eliminates unnecessary complexity, making option selection straightforward and transparent. Time efficiency minimizes the steps required to generate a plot. Intuitiveness helps users anticipate where to find the features they need, and a low learning curve ensures they can start using the application effectively with minimal training.

As detailed in the Architecture and Implementation section, these non-functional requirements guided the design of our GUI. They ensured that the application not only satisfied the functional requirements but also delivered an intuitive and user-friendly experience, making it as efficient and accessible as possible for astrogeologists.

Architecture and Implementation

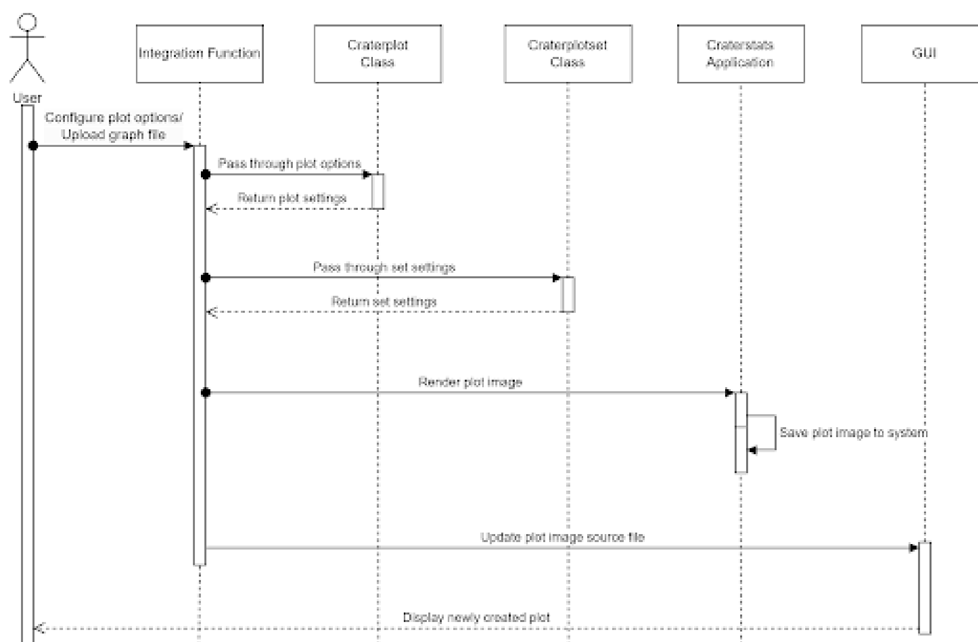
System Overview

Our application is based on two main components that communicate together to provide the user with the plot image they are looking for.



These components are the *Craterstats III* application being used as a library, and our created GUI using Flet. Our GUI gathers all the user information needed to develop the plot image. In the *Craterstats III* application, two major objects are made that are used in the creation of the plot. These objects are the Craterplot object and the CraterplotSet object. Using these objects the application creates a Matplotlib graph that has all the information that a user would need to analyze or research.

In our application, our GUI acts as a data retrieval process to give the *Craterstats III* application the necessary data to create the plot. This data is separated into two separate tabs; Global Settings and Plot Settings, that each correlate either to the Craterplot object or the CraterplotSet object. The Global Settings tab has settings such as the presentation view of the graph, the celestial body of the data being plotted, and the chronology system of the data. The chronology system consists of a chronology function and a production function. There are also settings for an equilibrium function, epoch visibility, isochron visibility and count, axes range options, and legend settings. The Plot Settings tab holds information that is more linked to how the plot looks. These are things like a title and subtitle, the font size, and the print scale as well. On this tab, the user also has the option to create over-plots. These over-plots give the ability to overlay multiple sets of data points, or multiple of the same set of points to compare datasets on the same plot image. Below is a detailed UML Sequence diagram, with UML Class diagrams that show how these components work together to get the final product.



Sequence Diagram

Craterplotset
+ title : String + subtitle : String + presentation : String + style : Int + isochrons : String + show_isochrons : Int + legend_data : String + legend_fit : String + print_dimensions : String + pt_size : Int + ref_diameter : Int + cite_functions : Int + sig_figs : Int + randomness : Int + mu : Int + invert : Int + show_title : Int + show_subtitle : Int
+ CraterPlotSpace() : Void + draw() : Void + plot_isochrons() : Void + autoscale() : Void + create_summary_table() : Void

CraterplotSet Object

Craterplot
+ cratercount : Cratercount + source : String + name : String + range : Int + type : String + error_bars : Int + hide : Int + colour : Int + psym : Int + binning : String + age_left : Int + display_age : Int + resurf: Int + resurf_all : Int + isochron : Int + offset_age : Int
+ calculate_age(Craterplotset) : Void + overplot(Craterplotset) : Void + get_data_range(Craterplotset) : Array

Craterplot Object

Our program was developed and influenced by the Model-View-Controller architecture. This architecture is common in GUI development with the following bases for each aspect. The Model aspect provides the data structures and ability to update the application, the View aspect provides all the user interface aspects of the application, and the Controller aspect provides all the data manipulation and processing, and in regards to our program is what creates the graph.

Detailed Overview

To start the detailed overview section, we'll describe how our application's Controller aspect works. Our controller is the *Craterstats III* CLI application. This application is used by astrogeologists from all around to plot these graphs. From our sponsor Trent Hare, "Crater chronologies are a fundamental tool to assess planetary surfaces' relative and absolute ages when direct radiometric dating is unavailable. Martian crater chronologies are derived from lunar crater spatial densities on terrains with known radiometric ages, and thus they critically depend on the Moon-to-Mars extrapolation". The application works by taking in a series of terminal commands that map to a different setting of the plot that is created. Inside the application is a text document of all the known functions that can be applied to the graph so that when a user uses the program it knows if it has the correct input. There are also numerous objects and functions within the application that are used in the creation of the graph. These include objects like Chronologyfn, Cratercount, Craterpdf, Epochs, and

Productionfn. All of these objects have their functions that are used within the main program to get the final graph. Explaining these objects and functions is outside the scope of this project as well as the team's knowledge.

Furthermore, in the Model aspect of the architecture we have all of our functions that were created to help feed the data that our View aspect gives into our Controller. This also includes functions that help the functionality of the program, like saving and opening configuration files, closing the program, exporting the plot image, creating a summary table of the plot, showing a demo view of the graphs, and having a popup that shows all relevant information to do with the program. Our data-handling functions are mostly to make sure that the data is in the correct data structure required for the Controller. This data structure is a list with two elements, the first one being a dictionary and the second being another list of dictionaries. The first dictionary holds all of the Global Setting information, and the list of dictionaries holds all the information for the overplotting ability of our application. Each dictionary holds the information for each over-plot setting.

Finally, the View aspect of our application is arguably the most important. This aspect is what displays all the UI for the user to interact with. Each UI element has an event handler that updates the graph on any change. This gives a more efficient and accurate plotting experience for the application. The View aspect is also how we get all the data required for the Controller. We used controls like radio buttons, dropdown menus, text fields, and checkboxes to help create our application in a user-friendly way that makes the application easy to use and understand.

Conclusion

To conclude the Architecture and Implementation of our project we will discuss how our processes and application have changed from the beginning to the end of our capstone experience. Our first step through this process was to figure out what technologies we would be using and how we could develop this application. We started with the options of Unity, Windows Forms, Tkinter, DearPyGUI, ReactJS, and ThreeJS. Unity and Windows Forms were nearly immediately taken off as options due to the high overhead that was required to run and build these types of applications. There was also a problem with the compatibility of using and running these systems on Linux. ReactJS and ThreeJS were initially considered as stretch goals for our project to run our application as a web application as well as a desktop application. However, due to timing and other complications, we were only able to create a working desktop application. We started off our prototyping with Tkinter. This was a long-renowned library used in multiple programming languages to create GUI applications. It has also been previously used by a few of our members. After our initial prototype was created

for this we as a team decided that the theming and age of Tkinter didn't adhere to what we were trying to make. Because of this, we decided to create a second prototype using a different library, DearPyGUI. This library was brought to our attention by our sponsor Trent. It was a newer library that allowed for better capabilities with graphing and was easier to use in terms of creating the GUI. A problem was brought to our attention while presenting our prototype with DearPyGUI. The library was primarily developed by one person and it was a newer library so there was a slight chance of the library becoming deprecated which would result in our project being of no use. The final prototype that we ended up with was built with Flet. Flet is a library powered by Flutter, which means that not only did the UI look new and modern, but it was also being developed by a team that would be more dependable than other libraries. Flutter is supported by Google which gave us the theming and dependability that we had been looking for.

There were also a few design and functionality decisions that were made that differed from what we originally planned. Our initial prototypes had been designed to be separated into three tabs, Global Settings, Plot Settings, and Plot. The Global Settings and Plot Settings didn't change from our initial design but from testing and a suggestion from our sponsor, we moved the Plot image to be viewable at all times no matter what tab was selected. Our functionality in terms of how we would integrate the *Craterstats III* application and our GUI was also different than how we planned. Our first iterations of the alpha prototype were created by integrating the two applications through a Python library called subprocess. Using this we would essentially run the CLI command that would create the graph in our application and display the results from that. Our final version integrates differently by instead importing the *Craterstats III* application as a library and using the functions and options within that program to create and display the graphs. This allowed for faster interactions with the application and more efficient updated graphs.

Testing

Effective testing was the cornerstone of our software development process, ensuring that every component functioned as intended and met the end-user's needs. In this section, we delved into the comprehensive testing strategy employed to validate our application. Our approach encompassed unit testing to verify the integrity of individual components, integration testing to ensure seamless interaction between modules, and usability testing to guarantee a user-friendly interface.

By systematically applying these testing methodologies, we aimed to identify and rectify defects at various stages of development. The results of these tests were critical in refining our design and source code, ultimately contributing to a robust and reliable application. This section provides a detailed overview of our testing activities, the

strategies implemented, and the insights gained, highlighting how rigorous testing shaped the quality of our software.

Unit Testing

Our approach to unit testing involved utilizing Python's unit test and pytest libraries to streamline our testing processes. We aimed to determine test-related metrics, specifically focusing on line coverage, which measured how much of our code was executed during tests. For key input options, we ensured that their values matched stored values accurately. For example, we tested that selecting the "Cumulative" radio button correctly stored "cumulative" in the plot configuration. This involved assertion testing to verify input accuracy.

Our focus was primarily on functionalities that impacted user experience, ensuring the critical parts of our system were robust and reliable. We tested UI elements such as radio buttons, dropdown menus, check boxes, and input boxes. Each unit test was designed to run independently, allowing quick diagnosis of issues. Python's testing libraries enabled us to run these tests repeatedly, especially after updates or changes to the code, ensuring consistent functionality.

Tests handled both typical user behavior and edge cases. Text inputs were tested with both valid entries and unexpected inputs, such as special characters or empty fields. Dropdowns and radio buttons were tested for correct default values, ensuring predictable application behavior even if some settings remained unchanged.

Integration Testing

Our integration testing focused on the boundaries where modules interacted significantly. We followed these steps:

First, we identified key integration points by examining the craterstats GUI architecture to pinpoint critical data exchanges and function calls. Specifically, we looked at the GUI-to-CLI interface, where user input from the GUI was passed to the craterstats CLI library, and the data processing and plotting modules, where data retrieved from the CLI library was visualized in the GUI.

Next, we designed test harnesses for each integration point. For the GUI-to-CLI interface, we simulated user commands with mock inputs, verifying correct command transmission. For the data processing module, we provided various data inputs and compared the generated plots against expected outputs.

We then validated our assumptions through data format verification, ensuring that the data structures expected by the plotting module matched the output from the processing module. We also conducted boundary testing, validating interactions under varying input sizes and types, including edge cases, and simulated erroneous inputs to ensure graceful error management.

For each integration point, we applied specific test cases. In the GUI-to-CLI interface testing, we used mock GUI input sequences, checked the data passed to the CLI module, and confirmed that the CLI processed commands without data loss, aligning outputs with GUI expectations. In the data processing and plotting module testing, we used input files with crater data and configurations, visually inspected and programmatically compared output plots against reference images, and ensured consistent plot generation reflecting input data with no visual inconsistencies.

Usability Testing

Our usability testing plan considered the nature of our application and its intended users, primarily astrogeologists with varying levels of CLI experience. Transforming a CLI application into a GUI necessitated comprehensive usability testing to meet user expectations and behaviors.

We employed several usability testing methods. First, we gathered qualitative feedback from astrogeologists by outsourcing testing to the USGS. This allowed us to explore user perceptions and areas of confusion. Second, we performed task-based user studies with screen capture software, noting difficulties or hesitations. Third, we engaged our sponsor to review the interface based on usability principles before the Alpha release. Finally, we conducted acceptance testing with a larger group of potential end-users to validate the application's intuitiveness and expectation alignment.

Results and Adjustments

During our unit testing phase, we achieved a high test coverage of approximately 95%. This high coverage rate indicated that most of our codebase was being executed during tests, helping us identify and fix bugs effectively. For example, our tests confirmed that selecting the "Cumulative" radio button correctly stored the value "cumulative" in the plot configuration, dropdown menus returned correct default values when no selection was made, and text inputs were successfully processed for valid entries, with special characters and empty fields handled gracefully without causing crashes or unexpected behavior.

Our integration testing uncovered several critical issues that were promptly addressed. Initial tests of the GUI-to-CLI interface revealed some inconsistencies in the

data passed from the GUI to the CLI, such as incorrectly formatted commands that led to processing errors. We resolved this by refining our command parsing logic, ensuring all user inputs were accurately transmitted. Early tests of the data processing and plotting modules showed discrepancies between the input data and the generated plots, with labels and markers occasionally failing to match the provided data. By refining our data validation and transformation procedures, we ensured that plot attributes consistently reflected input data, achieving a 100% success rate in automated checks and confirming visual consistency.

Usability testing provided invaluable insights into the user experience, leading to significant improvements. Feedback from astrogeologists highlighted areas of confusion, such as complex navigation paths and unclear button labels. In response, we simplified the user interface, reorganizing menu structures for better accessibility and clarity. Screen capture recordings revealed specific pain points, such as difficulties toggling the demo mode and creating sample files. These insights led to the implementation of clearer instructions and more intuitive controls. Our sponsor identified several usability issues early on, such as inconsistent button sizes and unresponsive elements. Addressing these concerns, we standardized UI components and enhanced responsiveness across all interface elements. Acceptance testing with a broader group of potential end-users validated that the application met their needs. Most participants found the interface intuitive and user-friendly. However, some suggested additional features, such as customizable plot settings. Incorporating this feedback, we added new functionalities to improve user satisfaction and workflow efficiency.

Overall, these comprehensive testing activities and the resulting refinements significantly enhanced the reliability, usability, and functionality of our application, ensuring it met the high standards expected by our target users.

Project Timeline

The development of the CraterStats application followed three distinct phases: the design phase, the prototype phase, and the alpha phase. The first two phases were completed during the first semester of the capstone class, while the alpha phase extended through the entire second semester.

During the design phase, the team’s first task was to analyze the CraterStats CLI code to understand its functionality and determine what needed to be implemented in the final product. Once the code was deciphered and documented, the team began designing the prototype's layout and flow. Multiple designs were proposed, but the team ultimately decided on a multi-tabbed layout that separated global settings, plot settings, and the plot itself into distinct tabs for user navigation. With the design in place, the next

step was to research GUI frameworks capable of supporting the planned layout. The design phase was completed within the first few weeks of the semester, providing a foundation for the prototype phase.

In the prototype phase, team members Caden Tedeschi and Levi Watlington developed simple prototypes using different GUI frameworks to evaluate their usability and functionality. Several frameworks were tested before the team settled on Flet, a Python-based framework, as the best choice for the project. Once Flet was chosen, Caden created a final prototype that served as the starting point for the alpha phase. This marked the conclusion of the prototype phase and the end of the first semester.

The alpha phase began at the start of the second semester, focusing on integrating the CraterStats CLI application with the GUI. The team worked to connect the CLI functions to corresponding options in the GUI, aided by thorough documentation from the design and prototype phases. Caden's well-documented Flet prototype and Levi's detailed comments on the CLI functions streamlined the process, enabling the team to efficiently link the GUI options to their respective backend functions. Evan Palmisano played a critical role in implementing file upload functionality for plot and data files, as well as enabling the exportation of plots.

During initial testing, the team noticed that launching the app required lengthy commands, which was inconvenient. To address this, Alden Smith created a script that allowed the GUI to be launched with a single click. About halfway through the alpha phase, the team presented a semi-functional version of the application to the project sponsor. Based on the sponsor's feedback, the GUI layout was adjusted so that the plot would appear on every tab, rather than being confined to a single tab.

Following these changes, the team completed the integration of the CLI functions and prepared the application for user testing. User testers included the project sponsor, Greg Michael (the creator of CraterStats), and astrogeologists from organizations such as USGS and OpenPlanetary. The feedback from user testing revealed several issues the team had not encountered during development. One significant recommendation from Greg Michael was to switch from using the ".plt" file format to a ".cs" file format for easier overlaying and exporting of plots. This required reworking how the application handled uploaded files to ensure accurate graph generation.

In addition to addressing this major change, the team resolved smaller issues such as range display inaccuracies and redundancies in demo generation. After fixing as many issues as possible within the remaining semester timeline, Ibrahim Hmood packaged the application as a pip-installable module, making it easier to distribute and install.

Future Work

During the development of the CraterStats GUI, our team successfully implemented all the core requirements set by our sponsor. However, we were unable to address the stretch goals outlined in the project proposal due to the need to prioritize releasing the product for user testing. Pursuing these additional goals would have reduced the time available to resolve issues identified by testers, who provided valuable feedback through our GitHub repository.

The stretch goals we were unable to address included integrating the GUI into the QGIS mapping system and developing a Python or web-based interface to enable the GUI's use within a Jupyter Notebook environment. These ambitious objectives, while beyond the scope of our initial deliverables, present exciting opportunities for future development and enhancement of the CraterStats GUI.

Beyond the stretch goals, the CraterStats GUI project (CraterStats III) is open source, offering a platform for continued innovation. This means that astrogeologists, developers, or anyone with coding expertise and creative ideas can contribute by adding new features or improving the existing functionality. Future contributors could focus on achieving the stretch goals or explore original enhancements to make the application even more versatile and user-friendly.

The open-source nature of the project ensures that CraterStats GUI will continue to evolve, empowering astrogeologists with better tools to streamline their work and further the field of planetary science.

Conclusion

The CraterStats GUI was developed to make the powerful plots generated by the CLI application more accessible to the average astrogeologist. Previously, many astrogeologists struggled with the steep learning curve of the CLI application, as they lacked formal training or experience with command-line tools. This learning barrier often made the effort to use the application outweigh its benefits. Our team addressed this issue by transforming the CLI into an intuitive, user-friendly GUI that simplifies the process of uploading data, modifying parameters, and saving plots, all while streamlining workflows and improving time efficiency.

One of the most impactful features we implemented is the real-time updating plot, visible on every tab of the GUI. This allows users to see the immediate effect of their changes, eliminating the need to repeatedly regenerate plots due to incorrect settings.

This feature alone greatly improves the application's usability and efficiency, and when combined with features like functional grouping, clear navigation, and detailed labels, the GUI becomes an indispensable tool for astrogeologists. Now, users with no command-line experience can effortlessly create precise plots by simply following the logical flow of the GUI.

Throughout the development process, our team, the Lunar Pit Patrol, remained focused on our goal: making astrogeologists' work easier and more efficient. We maintained strong communication and collaboration despite balancing multiple other obligations, which allowed us to produce a product we are truly proud of. The experience of working on this capstone project has been invaluable, serving as a stepping stone to our future careers and equipping us with skills we will carry forward into the industry.

In closing, we are honored to have had the opportunity to contribute to the field of astrogeology by creating a tool that will help professionals streamline their research and analysis. We thank you for following our journey, and we hope that our work with the CraterStats GUI will inspire further innovation in this exciting field.

Glossary

Alpha Product: The earliest version of a product that is ready for user testing.

CLI: Command Line Interface - A text-based way to interact with a computer's operating system by entering commands

Craterstats III (A.K.A CraterStats3): A tool to analyze and plot crater count data for planetary surface dating.

File Format: The way a file can be stored on a computer

Framework: A foundation that gives users the tools they need to create a specific type of product.

Flet: A Python GUI library that is powered by Flutter to build multi-platform applications in Python. https://flet.dev/

Flutter: Flutter is an open-source framework for building beautiful, natively compiled, multi-platform applications from a single codebase supported by Google. https://flutter.dev/

Functional Grouping: The grouping of things based on the functionality that they have.

GUI: Graphical User Interface - A visual way to interact with electronic devices, such as computers, smartphones, and tablets.

Jupyter Notebook: The latest web-based interactive development environment for notebooks, code, and data. Its flexible interface allows users to configure and arrange workflows in data science, scientific computing, computational journalism, and machine learning. https://jupyter.org/

Overlaying Plots: A plot made up of multiple plots layered over one another

Python: A high-level general-purpose programming language designed to emphasize code readability.

QGIS: A Geographic information system software that is free and open-source. https://www.qgis.org/

Real-Time Updating Plot: A plot that changes in accordance with the options that the user is choosing.

Windows Forms: a UI framework that creates rich desktop client apps for Windows. https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-9.0

Unity: An application that creates real-time 3D games, apps, and experiences for entertainment, film, automotive, architecture, and more. https://unity.com/

Appendix A: Development & Environment Toolchain

Hardware

Evan Palmisano
- Desktop (Ryzen 7, RTX 2070 Super, 64 GB RAM, Windows 10)
- Microsoft Surface Go 2 (Intel i5, 8 GB RAM, Windows 11 / Ubuntu 20.04)

Ibrahim Hmood
- Laptop: Dell Latitude 3410: 8 GB RAM Windows 11

Alden Smith
- Desktop (Ryzen 9, RTX 4070TI, 32 GB RAM, Windows 11)
- Laptop (Intel i7, RTX 2060, 16 GB RAM, Windows 11/Ubuntu Dual boot)

Caden Tedeschi
- MacBook Pro: Apple M3 Pro Chip, 18 GB RAM
- Windows 11 Desktop (Ryzen 9, 2070 Super, 32 GB RAM)

Levi Watlington
- MacBook Pro: 1.4 GHz Quad-Core Intel Core i5, 8 GB RAM

Minimum Requirements - 4 GB RAM & 8 GB Storage Space

Toolchain

Visual Studio Code
- General programming IDE. Allows for installation of many helpful extensions
- Most used extension: Black (Python formatting extension)

PyCharm
- Python Specific IDE
- Used mostly for testing *Craterstats III* application

Setup

For configuring the project's environment we recommend using the Anaconda package manager for Python. This will allow the user to operate the application in a contained environment. Fortunately, this setup configuration applies to a variety of operating

systems mainly focusing on Windows 10/11, Mac OS, and Linux platforms such as Ubuntu 22.04.

To set up your Python environment, you can download the package manager at https://docs.anaconda.com/anaconda/install/

Upon installation, open your command prompt with the Anaconda base environment activated. The first step is to create a new environment using Python version 3.8. To do that, you can type the following command:

conda create -n <environment name> python=3.8

After typing the command, all a user has to do is install our application using pip as it is conveniently hosted using PyPi. To install the application you can type the following command:

pip install craterstats-gui
Now that you have installed the application, all that is left is to run it. To do so, you can run:

craterstats-gui

After running the command, the application should initialize and run within the configured environment.

Production Cycle

The production cycle for Flet UI development can be done in one of two ways

Application Open Development

Before developing open the application through the terminal by navigating to the directory that main.py is located in and running.

flet run

After the application is running any code changes to the main.py file will automatically update the application and relaunch.

Application Closed Development

This method is more tedious but considered the more proper way to develop. After making any changes to the codebase, go to the directory where main.py is located and run.

flet run

After testing the application close the window and make necessary changes to the codebase.

Repeat.