A 2-DIMENSIONAL APPROACH TO RENDERING ON- AND OFF-BODY SHAPES AROUND AN ELLIPSOID BODY. J. E. Turner, C. M. O'Shea, F. S. Turner, G. W. Patterson, R. L. Klima. The Johns Hopkins University Applied Physics Laboratory. 11100 Johns Hopkins Road, Laurel, MD, 20723. (Jordi.Turner@jhuaple.edu)

Introduction: There are a plethora of types of planetary data, with some that are easy to visualize on a map-projection of a body, but others that are not, such as off-limb scans. While the development of 3-dimensional visualization technology (including augmented and virtual reality) is rapidly growing, many current scientists do not have the computers or processing power to use them. The authors developed a quasi-3-dimensional rendering approach to visualize and represent different types of planetary data in a fast and interactive manner.

Problem: While visualizing various types of planetary data, we developed a projection-agnostic method of viewing and interacting with shapes on a 2-dimensional map. However, we encountered a problem with some data types that do not easily lend themselves to be visualized on a map. We explored some 3-dimension alternatives, but wanted to maintain the capabilities we had already achieved (such as projection-agnostic shapes, vector-based exporting, etc.). We decided to develop an extension of our current rendering capabilities as we also desired for a seamless integration into our existing tools. Furthermore, a single API would allow for an optimized grow, as opposed to maintaining two different types of visualization, which would be costly to maintain and develop. Our priorities were focused on accuracy and speed, as the solution was meant for integrating in a responsive graphical user interface (GUI) for visual analyses.

Tools: We designed the solution in Java, as our existing 2-dimensional API was written in Java. Java suits our needs well, as it is a multi-platform language that supports similar performance across Mac, Windows, and Linux machines. Additionally, we wanted to leverage various Java libraries developed in our organization over the past 15 years.

In designing our 2-dimensional API, we found an extremely versatile combination of JavaFX and Java AWT. JavaFX is a modern GUI toolkit that is being actively maintained by the open-source community. It has a simple event model that is arguably easier to use than those in other Java GUI toolkits, and provides data binding capabilities that enable synchronization between the GUI and the underlying data. From a developer's standpoint, JavaFX allows faster creation of interactive and attractive GUIs [1]. Java AWT (Abstract Window Toolkit) is an older GUI toolkit, but it includes a highly customizable rendering API. Our GUIs are purely JavaFX, but when we display analysis images, we create them through Java AWT and simply convert them in to JavaFX Images to display.

Visual analyses of planetary bodies with spacecraft instrument data requires high accuracy planetary geometrical outputs. To obtain these outputs we used our organization's Java implementation of JPL NAIF's SPICE toolkit [2]. This allows us to compute a body's ellipsoid, the vector from a view point to a body's center, and the limb ellipse seen from that view point.

Solution: Since planetary bodies are by nature significantly apart from each other, we restricted our solution's scope to only render analysis shapes for a single body at once. This suits our scientific analysis needs as our measurements or observations only focus on one sole body at a time. However, this does not mean we only render a single planetary body, as understanding many analyses often requires contextual information such as relative positioning of other bodies or even stars. Our first step was to develop a rendering algorithm based on proximity to the view point. First, we render abstract visual references (such as a latitude and longitude grids) or objects that are at a practically infinite distance away (such as stars). Then, we render all planetary bodies of interest from furthest to nearest to account for occultation. Rendering planetary bodies includes features such as limbs, names, latitude longitude grids, Sun-illuminated/shaded sections, and texture maps. Finally, we render the analysis shapes for the specific body of interest, which we have named body centered shapes.

Body centered shapes can be anything with a fixed definition relative to a body. They must be either fully on- or off-body. Any partially on-limb shapes are split up into multiple on- and off-body shapes. To explain our solution's detail, we will walk through the creation and rendering of the shapes of a spacecraft instrument's field of view (FOV) that partially intersects a body's limb:

First, by using the FOV data in SPICE, we create a list of unit vectors originating from the instrument that represent the FOV. We treat these as if they were infinitely long and check whether they intersect with the body's ellipsoid. In this case, we have a rectangular FOV that partially intersects with the body ellipsoid, as represented in Figure 1. For any vector that intersects the ellipsoid, we redefine it from the body's center to the intersection point in the body's frame and deem it

on-body. For any vector that does not intersect the ellipsoid, we redefine it from the body's center to the intersection point on the body's limb plane and deem it off-body.

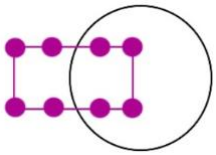

Figure 1: The intersection of an instrument FOV with a body ellipse from the view point of the instrument.

The purple dots represent the vectors that constitute the FOV This leaves us with an open on-body shape and an open off-body shape. To close the shapes, we calculate the exact points that FOV crosses the boundary of the body's limb, and radially connect those points. In this example, we have two limb-crossing points (A and B in Figure 2) that can be radially connected in two ways – either clockwise from A to B, or clockwise from B to A. To determine the correct order, we calculate the midpoints (C and D in Figure 2) and check whether they are contained in the original FOV shape from the instrument's view point..

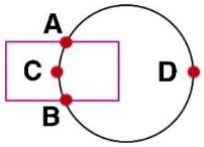

Figure 2: The limb-crossing points (A & B) and their midpoints (C & D).

Connecting the limb-crossing points gives us a fully closed on-body shape and a fully closed off-body shape, shown in Figures 3 and 4. These shapes can now be viewed from any view point around the body.

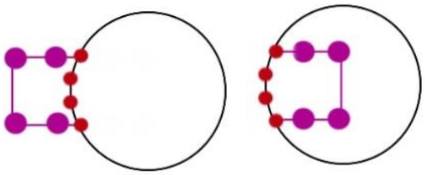

Figure 3: On the left, the final closed off-body centered shape. On the right, the final closed on-body centered shape. The red dots represent the vectors connecting the limb-crossing points to close the shapes.

Given a new view point, we adjust each vector of both shapes to be centered from the view point instead of the body center. We also compute a new body limb ellipse. For each shape, we iterate through every vector and determine its visibility. For on-body shape vectors, we simple check if it is past the limb plane. For off-body shape vectors, we check whether it intersects the body's ellipsoid. If a shape has any invisible part, we divide it to only render the visible parts, in a similar fashion to how we split the FOV shape into two earlier. However, to correctly connect the limb-crossing points, we check whether the midpoints are contained in the on-body shape projected onto a map-projection of the body. Given these new visible shapes, we render them, preferably not fully opaque, so that their interaction is visible to the eye without having to move the view point.

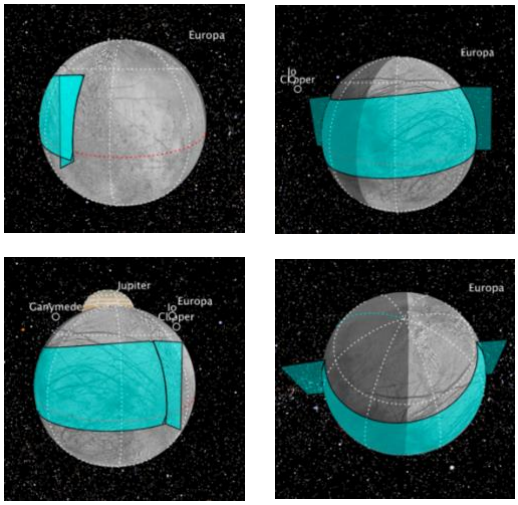

Figure 4: Four different perspectives of body centered shape of an instrument observation on Europa Clipper.

Conclusion: The extension of our 2-dimensional map rendering capabilities allows for flexible and interactive renderings that are quasi-3-dimensional. This solution enables free manipulation of the view point, with options such as being fixed to a spacecraft or rotating around a body at a fixed distance. While our algorithm is complex to explain, it is computationally efficient, and allows interactive visualization of many new types of datasets.

References:

[1] Homeel, S. (2013) *Using JavaFX Properties and Binding*.

[2] https://naif.jpl.nasa.gov/naif/