AN INTEGRATED SOLUTION TO DISPLAY LARGE IMAGES AT FULL RESOLUTION. J. E. Turner[1], F. S. Turner[1], C. M. O'Shea[1], and G. W. Patterson[1]. Johns Hopkins University Applied Physics Laboratory, 11100 Johns Hopkins Rd, Laurel, MD 20723 (Jordi.Turner@jhuapl.edu)

Introduction: As technology advances, imaging instruments are becoming more capable and increasingly higher resolution. Consequently, these instruments are recording more data than ever before, and multi-gigabyte files are now common place. Along with images becoming larger and higher resolution, the processes applied to imaging data have also become more complex.

With this advancement in imaging technology comes challenges with loading and displaying these extremely large data sets. Some images are so large that loading them into RAM effectively kills a computer's interactivity. We are working to develop a software framework that can load these large imaging datasets efficiently, apply complex processing algorithms to them, and easily visualize the results. Furthermore, we want to provide the user with the ability to responsively pan and zoom on the image without noticing any delay in the graphical user interface (GUI).

Software Tools: Using out-of-the-box methods to load and analyze these large images is simply not fast enough. We can, however, leverage existing software systems and integrate them in new ways that allow a much-desired performance increase in viewing and manipulating these images. In the following subsections we will describe the software techniques and tools that were used to design an application program interface (API) for efficiently working with such images without overloading our computers.

Java: This API is Java-based for many reasons. Java is a multi-platform language, so we can expect similar performance on any Mac, PC, or Linux machine. Using Java also allows us to leverage existing software that has been developed within our organization over the last 15 years.

Java Advanced Imaging (JAI): JAI is a software suite used to load, analyze, and process images in Java. Released in the late 1990's, JAI was under active development by Sun Microsystems until 2006, and completely stopped once Sun Microsystems was bought by Oracle in 2010 [1]. Although it has not been revised or updated in nearly a decade, JAI is a unique Java imaging tool that offers a pull-based image processing model. Images can be broken up into a collection of uniformly but arbitrarily sized tiles. This allows tile sizes to be selected for a specific application's needs. For instance, visualizing data on the screen typically would require a different tiling than a complex processing of the same image directly on the disk. This tiling method loads only the visible portion of the image into memory, and is the key to loading overbearingly large images.

As the tiles required by the display are loaded from JAI, they are potentially stored into a configurable memory cache. This eliminates the need to recreate some subset of tiles when the display changes, thus increasing rendering speed. Alternatively, disk-based cache implementations also exist. The features of JAI enable complex operational chains to be constructed with little or no computational effort. The API supports multi-threaded computation so the algorithms can take advantage of the parallel capabilities of multi-core processors common in most computers today. There is an inherent 2-gigapixel limit in the more common push-based Java image processing models, which are backed by primitive arrays. The tiling, caching, and multithreading features in JAI provide the means to surpass these limitations.

JavaFX: JavaFX is a modern user interface toolkit that is actively being developed by the open source community. It has a simple event model that is arguably easier to use than those in other Java UI toolkits, and offers basic one-way and two-way data binding that enables synchronization between the GUI and its underlying data [2]. Ultimately, JavaFX is easier to use from a developer viewpoint, and certain features such as the bindings and event handling previously mentioned enable faster creation of interactive and attractive user interfaces [3].

Integrated Solution: Leveraging all of the JAI and JavaFX advantages outlined above, we will present a program to quickly load, analyze, manipulate, and display large images of common and uncommon formats. The structure of this solution is outlined below, and shown in figure 1.

FX Component: This component is effectively the GUI. This is what the user interacts with and it provides the user with the controls necessary to load an image and manipulate the viewable area of an image through mouse reactions such as panning, zooming, and scrolling. It also provides an avenue to create and customize a chained list of complex processes to apply to the image, such as image stretching, color mapping, or blurring.

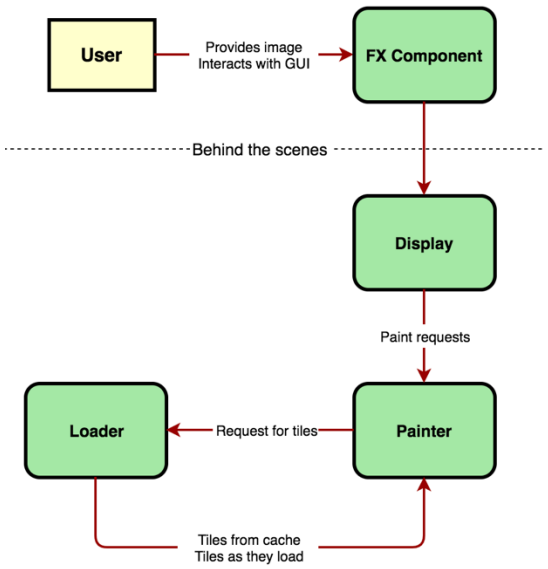

Figure 1: The general structure of the API and how the components interact with each other.

Display: The display is the current view area of the image. This is not necessarily the entire image, especially when the image is large. When viewing the image at full resolution, it likely does not fit on the user's screen, so the display manages what area is viewable. This component will, when necessary, create paint requests to send to the painter. These requests are sent when part of the image becomes visible that was previously not displayed. The painter will take these requests and handle them as appropriate.

Painter: This component receives and processes the paint requests from the display. Due to the interactivity of the GUI, a user can easily create an overwhelming number of paint requests when performing a gradual action such as zooming or panning. Whenever the painter is responding to a paint request, it coalesces any subsequent requests and only reacts to the last request once it finishes its current job. This guarantees speed without sacrificing any visual responsiveness by reducing the minute change requests. Upon receiving a valid paint request for an area of the image, it instantly renders a low-resolution image which is an accurate, though fuzzy, representation of the currently displayed area. With the low-resolution image rendered, the painter determines which of the higher resolution tiles are required. If any are already present in the cache, they are painted on the display immediately. For those tiles that are not currently present on the cache, the painter must send a request to the loader.

Loader: Interfaces with the JAI toolkit to provide methods for requesting tiles via their indices. Any tiles in the cache return an image for immediate rendering. On the other hand, any tiles that must be requested are done so on worker threads with listeners. Once finished, these listeners notify the painter with the images for rendering. This component also provides the painter with the low-resolution image.

Conclusion: While there are many existing tools to load, analyze, manipulate, and display image data, we have not found a comprehensive solution that is capable of handling the larger files that are becoming more and more commonplace in the planetary community. Our solution is able to load 50+ gigabyte files and efficiently display them, while giving the user the capability to modify and chain custom processing algorithms while viewing the image. Figure 2 shows a prototype of the GUI that implements the techniques listed in this abstract. Our GUI provides instant reactions to the user's input, such as scrolling, panning, or reading values from the mouse's location on the image. This API allows immediate and dynamic visual analyses that previously could only be made statically.

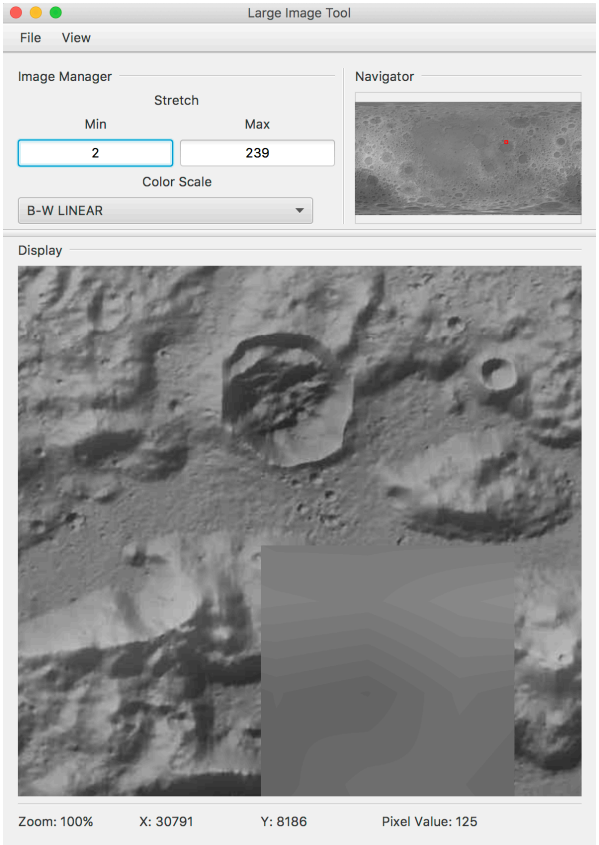

Figure 2: A screenshot of a prototype loading a tiled image of the Moon. An artificial delay was introduced to extend loading time.

References: [1] Tillman, K. (2009) *Oracle Buys Sun.* [2] Hommel, S. (2013) *Using JavaFX Properties and Binding.* [3] Topley, K. (2010) *JavaFX Developer's Guide.*