ISIS Camera Model Design. J.A. Anderson, U.S. Geological Survey, 2255 N. Gemini Dr., Flagstaff, Arizona, janderson@usgs.gov

Introduction: The Integrated Software for Imagers and Spectrometers (ISIS) provides image processing support for past and current NASA flight missions [1-2]. The ability to geometrically rectify digital images from raw instrument geometry to a map projected image is a critical function of ISIS, and it is supported for a wide variety of cameras including Viking, Voyager, Clementine, Mars Observer Cameras, Mars Odyssey THEMIS, Mars Reconnaissance Orbiter HiRISE, and Cassini VIMS and ISS. Upcoming instruments that will be supported in ISIS include the Messenger MDIS camera and the Lunar Reconnaissance Orbiter camera. Developing a stand-alone camera model for any given instrument allows for the use of a large suite of ISIS application programs (with no modifications required). Adding new instruments to ISIS has become a fairly straightforward exercise and is the main focus of this report.

Background: In late 2001, a modernization of ISIS was undertaken [3]. A primary motivation was to migrate existing code to C++, an object-oriented programming (OOP) language, in order to promote programming efficiency through code reuse. Fortunately, previous experience with redesign of camera models [4] in the ISIS FORTRAN/C environment aided with this conversion.

Methodology: At the lowest level there are a handful of C++ classes that deal with spacecraft position and orientation, target position and orientation, sun position, camera parameters, and instrument timing. Collectively this information is known as SPICE data as coined by the Navigation and Ancillary Information Facility (NAIF) [5]. ISIS wraps small portions of the NAIF toolkit in three C++ classes *(Spice, SpicePosition, and SpiceRotation*) in order to handle SPICE kernel management and requests for spacecraft-specific data at a given ephemeris time.

The next C++ class, *Sensor*, utilizes the OOP "has-a" and "is-a" relationships, class composition and class inheritance, respectively, to build upon the capabilities of the *Spice* classes. The primary function of the *Sensor* class is to set a look direction vector for an instrument and compute its intersection with the target body. A successful intersection results in a ground coordinate, either an (x,y,z) body-fixed coordinate, or a (latitude,longitude,radius) coordinate. Similarly, the class can set a ground coordinate and compute a look direction vector back to the spacecraft. It is important to note that the target body can be a sphere, ellipsoid, or digital elevation model, where the latter allows for orthorectified map projected images. The *Sensor* class also has the ability to compute important viewing information such as phase, incidence, and emission angles.

The foundation for all camera models is the *Camera* class. Its purpose is straightforward; given an image pixel coordinate (sample, line) it will compute the ground coordinate (latitude, longitude) and vice versa. Inheriting from the *Sensor* class implies the major focus is computing a look direction vector at any given pixel. The following coordinate conversions are required for the computation:

Child image pixel coordinate (sample, line)
↕ *AlphaCube*
Parent image pixel coordinate (sample, line)
↕ *CameraDetectorMap*
Detector coordinate (sample, line)
↕ *CameraFocalPlaneMap*
Focal plane coordinate (x,y) in mm
↕ *CameraDistortionMap*
Undistorted focal plane coordinate (x,y) in mm
↕ *CameraGroundMap*
Ground coordinate (latitude, longitude, radius)

These steps are coded as five separate C++ classes: *AlphaCube, CameraDetectorMap, CameraFocalPlaneMap, CameraDistortionMap,* and *CameraGroundMap*, where each class computes the forward and inverse coordinate transform.

AlphaCube Class: When an image is resized or cropped using ISIS programs, metadata is stored in the labels of the file. This information is used to compute the parent pixel coordinate given a child coordinate. This is useful when only a sub-area of the parent image is desired for scientific analysis.

CameraDetectorMap Class: This is used to transform between parent image coordinates and detector coordinates. The base class handles framing camera instruments with on-board electronics that provide summing modes or readouts of detector sub-areas. A derived class, *LineScanCameraDetectorMap,* is used

for line scan cameras where the ephemeris time must change based on the parent line number. On occasions, individual instruments require special derived versions such as the THEMIS VIS camera where the image band number (wavelength) plays a role in detector coordinates.

*CameraFocalPlaneMap Clas***s:** This class is used to transform between detector coordinates and the location on the camera focal plane. The resultant coordinates (x,y) are in millimeters. The detector boresight pixel coordinate maps to (0,0) in the focal plane. Internally, the class uses an affine transform to allow for detectors which have a translation and/or rotation around the boresight.

CameraDistortionMap Class: Distortions due to imperfect optics are transformed using this class. The base class transforms $(x,y)_{distorted}$ to $(ux,uy)_{undistorted}$ using an odd polynomial and the distance from the boresight. Other derived classes include *RadialDistortionMap* and instrument specific distortion maps, for example, Mars Global Surveyor Wide Angle Camera.

CameraGroundMap Class: This final class transforms from undistorted focal plane coordinates to ground coordinates (via the *Sensor* class). For most camera types, the (x,y) to (latitude,longitude) computation is the same. However, the *LineScanCameraGroundMap* derived class, uses an iterative numerical solution to convert from (latitude,longitude) to (x,y) converging on ephemeris time. Future camera types such as the push frame camera will require a new derived ground map class.

.

Writing a New Camera Model: In most cases, developing a new camera model requires minimal coding. The programmer must derive off the *Camera* class and internally choose/create the appropriate transformation classes (e.g. *CameraDetectorMap*) for the camera type, framing or line scan. The majority of work then lies in the acquisition and setting of the camera parameters such as focal length, pixel pitch, boresight coordinate, affine coefficients for focal plane mapping, optical distortion coefficients, etc. In some rare instances, a new transformation class specific to the instrument will need to be derived.

Integrating the Camera Model into ISIS: Adding a new camera model to ISIS applications is nearly automatic one the initial camera model development is complete. Applications programs such as "camstats" and "cam2map" which compute geometric statistics for an image and convert a raw camera image to a map projected image, respectively, do not need to create a specific instance of a camera. For example, creating a C++ object using the *VikingCamera* or *ThemisIRCamera* class is not done internal to the application. Instead, applications which require a camera use the *CameraFactory* class to create a camera instance. The factory looks at metadata in an image header and cross references a text file to determine which type of camera, *VikingCamera* or *ThemisIrCamera*, to create. Adding a new entry to the text file along with the creation of a shared-library allows new camera models to be plugged in to existing ISIS applications without the need to modify the application code. This is a critical element of the design; ISIS users can now apply the same applications when processing any image data, regardless of camera type or capabilities.

Conclusion: The development of camera models will never be a trivial exercise; however, the establishment of true object-oriented C++ classes in ISIS has eased the creation for new instruments. Complex cameras such as MRO HiRISE and push frame cameras such as MRO MARCI and the LRO wide angle camera have and will stress the current architecture design. Over time the system will be refactored to make improvements and possibly support new types of sensors including RADAR imagers.

References: [1] Gaddis, L.J., et al. (1997) An Overview of the Integrated Software for Imaging Spectrometers, *LPS XXVII* 387-388. [2] Torson, J.M., et al., (1997), ISIS – A Software Architecture for Processing Planetary Images, *LPS XXVII,* 1443-1444. [3] Anderson, J.A., et al. (2004) Modernization of the Integrated Software for Imagers and Spectrometers, *LPS XXXV,* Abstract #2039. [4] Anderson, J.A., et al. (2002) Rapid Geometric Software Development for Scientific Analysis and Cartographic Processing of Planetary Images, *LPS XXXIII*, Abstract #1853. [5] Acton, C.H., (1996) Ancillary Data Services of NASA's Navigation and Ancillary Information Facility, *Planet. Space Sci., Vol. 44, No. 1, 65-70.*