Geolocation of CIPS image pixels

1) Definitions

Camera coordinate system (CAM) has an axis aligned with the boresight of the lens, and the other axes aligned with the CCD edges. No XYZ are assigned to the axes yet. I don't know what is most convenient. It is centered at the center of convergence of the lens system (The exact location of the origin is unimportant, since it can add no more than a couple of centimeters error).

SC body coordinate system (S/C) is aligned to the spacecraft master reference cube edges. X points towards the solar panels, Z is normal to the nadir deck, Y completes the right handed system. It is centered at the spacecraft center of mass (The exact location of the origin is unimportant, since it can add no more than a couple of meters error).

Earth Centered Inertial (ECI) is centered at the center of the earth, X points towards the vernal equinox, Z towards the north celestial pole, Y completes the right handed system. This does not rotate with the earth.

Earth Centered Fixed (ECF) is centered at the center of the earth, X points at the intersection of the prime meridian and the equator, Z points towards the north pole, Y completes the right handed system. This does rotate with the earth.

Global coordinate system is either ECI or ECF, whichever is more convenient. It looks like ECI is more convenient in most cases.

2) Camera model

A camera lens is a system which is designed to map incoming directions to locations on the image plane. Ideally, all rays impinging on the lens from the same direction will be mapped to the same point on the image plane. It does this by refracting all the rays coming in from the same direction such that they all converge on the same point in the image plane. The actual lens system is very complicated, to allow for adequate image quality over a wide field of view, while at the same time gathering sufficient light and satisfying other constraints of the other imaging system elements. See figure 1.

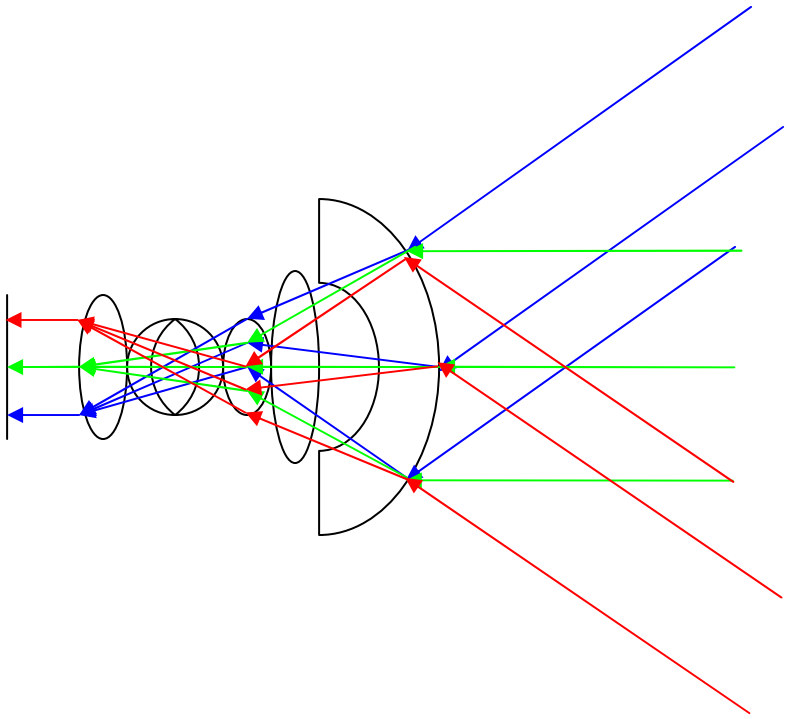

Figure 1. Approximate ray-trace of actual lens system

Conceptually, we can model this camera as a simple pinhole. This performs the same function as a lens (mapping directions to image points) by simply selecting one incoming ray for each direction and masking out all others. See figure 2.

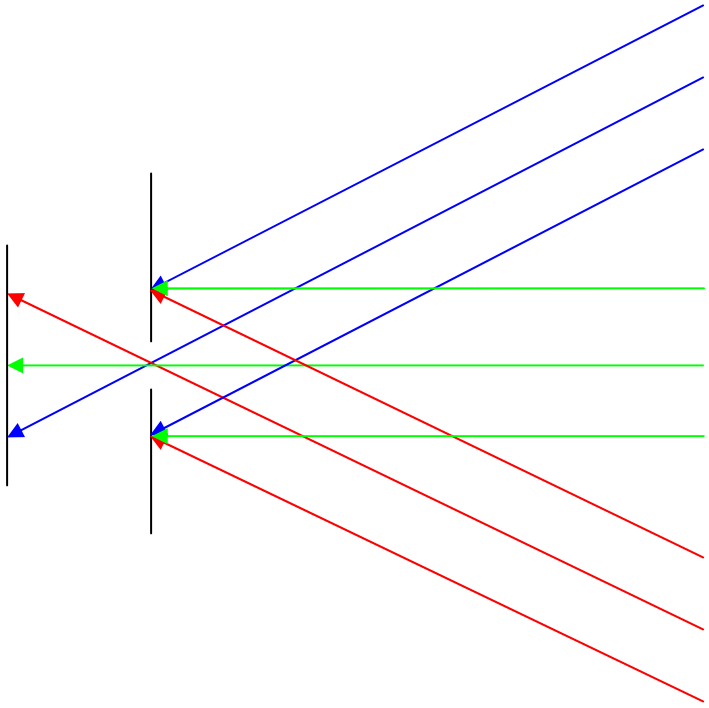

Figure 2. Pinhole model of system. Center of CAM is at pinhole.

The pinhole model is the simplest physically realizable model which captures all the essential aspects of a lens, including the fact that light comes into the lens from the outside and hits the image plane, and that the image is inverted.

It is possible to simplify the system one step further, but only in a mathematical sense. If the system is considered to run in reverse, with light starting at the CAM origin and traveling outward, through the image plane, we achieve the simplest model, the negative focal length pinhole camera. In this system, the rays are projected from the camera, through the image plane, then out into the environment. It's more like a projector than a camera in concept, and this will help out further below. The image which is collected can be thought of as being turned into a slide which is projected back onto the cloud deck.

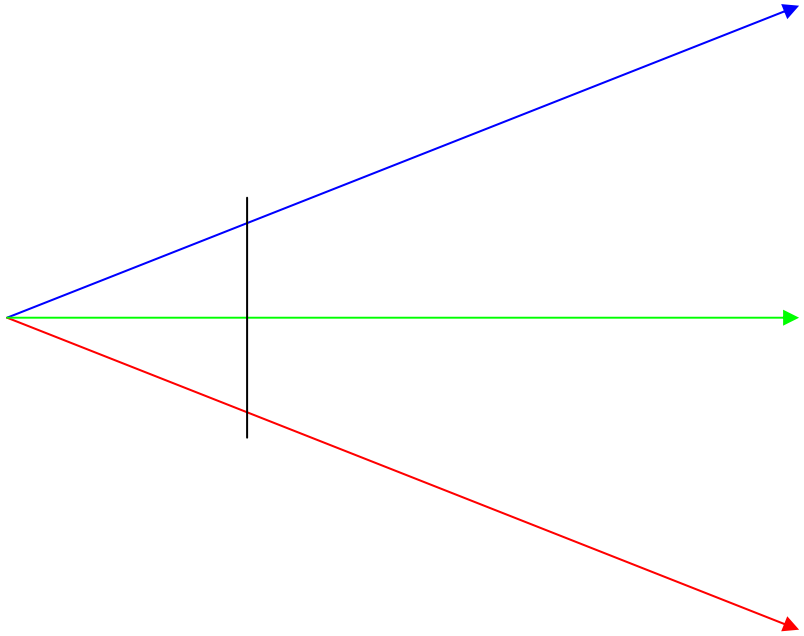

Figure 3. "Negative focal length" pinhole model. CAM is still centered at pinhole, but image plane is in front of pinhole.

The diagrams have all had three directions, colored red green and blue. Each direction corresponds to a single pixel. The real system has thousands of pixels, each of which corresponds to an incoming direction or outgoing ray. A preflight calibration of the lens would measure the exact direction corresponding to each pixel. Without this, the directions can still be modeled by distributing the pixels evenly across the image plane. For the pinhole models, this means that the displacement from the optical axis of an image pixel is proportional to the tangent of the incoming ray angle from the optical axis. The mapping is more complicated for the actual lens.

A ray is modeled mathematically as a parametric equation with the origin and direction as coefficients:

$$\vec{\mathbf{R}} = \vec{\mathbf{R}}_0 + \vec{\mathbf{V}}t$$

R_0 is the position of the ray origin (zero in CAM and S/C frames) and V is a vector pointing in the direction the ray travels. The parameter *t* is proportional to the distance from the origin along the ray, and if the pointing vector is given unit length (1km for instance) the parameter *t* is just the number of km away from the origin.

3) Transforming to global reference frame

This pixel ray is transformed from CAM to S/C by a simple rotation. This rotation can be approximated before launch by knowing the design angles of the camera, and measured after launch by using the results of the stellar calibration. The ray is then transformed to the global frame (Either ECI or ECF, whichever is more convenient) using a rotation derived from the spacecraft attitude data, and then using a translation derived from the spacecraft position. Rotations are applied to the V vector only, and translations are applied to the R vector only.

4) Projection onto the cloud deck

Once we have a ray corresponding to a particular pixel, it can be traced out into the world. This ray will travel a certain distance before intersecting the cloud deck at a definite location, the ozone layer at a different altitude, and finally the ground at altitude zero. We simulate this mathematically by finding the intersection of the pixel ray and the surface of interest. In other words we solve a system of simultaneous equations.

The math is easiest if we model the earth as a perfect sphere. Each surface of constant altitude above the sphere is also a sphere, with a radius greater than that of the earth surface by exactly the altitude. The simultaneous equations are then as follows:

$$\vec{\mathbf{R}} = \vec{\mathbf{R}}_0 + \vec{\mathbf{V}}t$$

$$\left|\vec{\mathbf{R}}\right| = r$$

Where *r* is the radius of the surface of interest. This breaks down into scalar equations for each vector component:

$$x = x_0 + x_{\mathbf{V}}t$$

$$y = y_0 + y_{\mathbf{V}}t$$

$$z = z_0 + z_{\mathbf{V}}t$$

$$x^2 + y^2 + z^2 = r^2$$

which quickly decomposes into a scalar quadratic equation in *t*. Quadratic equations can have zero, one, or two real roots. These correspond to the situation where the ray never touches the surface, grazes the surface at one point, or penetrates the surface at one point and comes back out at another. Also, we are only interested in positive solutions, since negative solutions correspond to the ray going the opposite direction.

As long as the camera is outside the surface, if there are two solutions, the solutions will be either both positive or both negative. The solution we want is the smaller of the two. After we have a value for *t*, we can then plug the value back into the ray equation to get 3D coordinates of the intersection.

The spherical model of the Earth is probably not good enough. The earth is much more accurately modeled as an oblate spheroid, with a certain equatorial and polar radius. A surface of constant altitude above this spheroid is also a spheroid with equatorial and polar radii greater than that of the surface by precisely the altitude.

The math is similar for a spheroid as for a sphere, except it uses the equation for the surface of an spheroid instead of that for a sphere. The solutions for *t* are interpreted in the same way.

5) Latitude and longitude

Transformation from rectangular to spherical coordinates is simple trigonometry. Transformation from rectangular to geodetic coordinates (spheroidal coordinates) is more complicated but still tractable.

6) Breakdown into functions

V=GetPixelRayCAM(Cam, PixX, PixY)

This calculates a pixel ray direction, given a camera number, pixel row and column number. It is in charge of modeling the lens system, either by lens calibration data or by a tangent model or something like that. This function could be run once per pixel in advance, with the results stored in a table.

V'=CAM2SC(Cam, V)

This transforms the ray from CAM to SC coordinates by applying the rotation matrix, either derived from the mechanical design or measured after launch by stellar calibration. It takes a pixel ray direction and camera number as input, and generates the same ray direction rotated into SC coordinates as output.

[R,V'']=SC2Global(V', Time)

This function gets the appropriate TLE and attitude quaternion and uses it to calculate the appropriate rotation and translation, then applies this to a vector in SC coordinates to generate a ray in global coordinates. It takes a time and pixel ray direction in SC coordinates as input, and generates a pixel ray origin and direction in global coordinates as output.

[x,y,z,flag]=RaySurfIntersect(R,V'',Alt)

This function finds the location in global coordinates of the intersection between a ray and a surface of constant given altitude. It takes a ray origin and direction and altitude of the surface as inputs and calculates the Cartesian coordinates of the intersection as output. It reports in flag if anything went wrong, such as if the ray did not in fact intersect the surface.

[lat,lon,alt]=XYZ2LLA(x,y,z,time)
This function calculates latitude and longitude (and altitude as a byproduct) of a Cartesian coordinate set. If ECI is used as the global frame, this function also applies the rotation of the earth. It takes the Cartesian coordinates and time (to calculate earth rotation) and returns latitude and longitude. The spheroid form calculates altitude at the same time, which can either be thrown away or checked against the desired altitude.

All these functions except RaySurfIntersect currently exist in some form in MATLAB in my other projects. They can be readily translated into IDL or any other language with good matrix handling.

[lat,lon,alt,flag]=CAM2LLA(Cam,PixX,PixY,time,alt) is a driver function which calls the other functions in the correct order and solves the geolocation problem. It takes a camera number, pixel row and column numbers, time, and altitude of interest as input, and returns the latitude and longitude (and altitude as a check). This driver function does not exist yet, but can be easily written:

```
[lat,lon,alt,flag]=CAM2LLA(Cam,PixX,PixY,time,alt)
  V=GetPixelRayCAM(Cam, PixX, PixY)
  V'=CAM2SC(Cam, V)
  [R,V'']=SC2Global(V', time)
  [x,y,z,flag]=RaySurfIntersect(R,V'',alt)
  If flag=OK
    [lat,lon,alt]=XYZ2LLA(x,y,z,time)
    return [lat,lon,alt,flag]
  else
    return [NaN,NaN,NaN,flag]
  end
```