Mosaicing Report for the Prototype

From: Mark Holdaway
Date: Tue, 24 Mar 92 17:29:14 EST

Hi guys:
Here is a small report on mosaicing in the prototype.
I am leaving for Socorro on Wednesday, I'll be back the night
of March 31. Have fun.

*** The Primary Beam ***

I was planning on using the MathTable class for the primary beam
array (ie, a PB HAS a MathTab, PB suplies Apply method).
However, this turned out to be more trouble than it was worth
for a prototype. Sanjay and I still feel that a general MathTable
class (math function lookup table and associated methods) will be
a useful concept and will save code/time/understanding-effort.

I can apply a primary beam to an image or correct the image for the
primary beam. I've added a pointer to a TelParms object in the
image. TelParms consists of OBSRA, OBSDEC, LAMBDA, and dish diam.
It could grow longer. There is an inheritance heirarchy among
primary beams which is not currently fleshed out:

```
        PB
         |
 --------------------------------------------------
|                                |               |
PB1D                            PB2D             ?
(rotationally symm)        (can rotate on sky)
|    |
|    |
?    ?
```

PB1D objects will have a pointer to a MathTable, and the MathTable could
have a blocked Airy disk squared, Gaussian, or whatever.
The parameters in TelParms allow PB1D.Apply to scale the PB stored in
MathTable appropriately.

*** How Is The primary Beam Known? ***

The details of how an image and a PB are associated are not yet clear.
In principle, it is a similar problem to the associator problem:

Image	Telescope	TN	IM	PB
Im1	Tel1	...	...	Gaussian

However, Im1 may be VERY SHORT LIVED (ie, we might be looping
through 100 pointings, making dirty maps, adding them to an image,
and destroying them, all at a fairly LOW level of the program).
While TelModel and ImagingModel are usually defined at the SURFACE
of a program, we DO NOT want to have to define the PB associated
with a given pointings worth of data at the surface.
Originally, I had envisioned TelParms to carry a pointer to the
PB object to be used. But PB must know about Images, since some
of its methods deal with images. If an Image HAS a pointer to TelParms,
you get a snake eating its tail in include files.

*** The Mosaicing Engine ***

The guts of the mosaicing would reside in a mild-mannered object
called MosAccum, a mosaic accumulator:

```
class FilledImage;
class MosAccum
{
 public:
  MosAccum( const FilledImage &temp);    // constructor
  ~MosAccum() {};                        // destructor
  void AddIm( const FilledImage &Addme, // add image (must have pTelParms
        const PB1D &myPB);               // this use of PB is temporary
```

```
   FilledImage Mosaic();                     // make a mosaic image
   FilledImage Sensitivity();                // make a sensitvity image
  private:
   int Nmaps;                                // how many maps added in so far?
   FilledImage IMsum;                        // \sum PB . Images
   FilledImage PB2sum;                       // \sum PB^2
 };
```

AddIm adds (PB . Addme) to IMsum, (PB.PB) to PB2sum.

To mosaic some dirty maps, code could look like:

```
    PB1D myPB;                                      //construct PB
    IntImagingModel myImMod;                        //set imaging model
    MosAccum MA (TemplateImage);                    //initialize mosaic accumulator
    for (ip = 0; ip < pointings; ip++) {
        FilledImage Dirty = myImMod.InvertYegs (YegSet.Select(ip));
        MA.AddIm (Dirty, myPB);
    }
    FilledImage Mosaic = MA.Mosaic();
    FilledImage Sens = MA.Sens();
```

The details of selecting out the data for pointing ip from YegSet
are unclear, but all data must have the same "TelParms". those
TelParms are copied over to Dirty so PB.Apply knows where the PB
goes. This copying is associated with the construction of the
image coordinates from the astronomical coordiantes contained in
the YegSet (reference position on sky...).
Again, this is the wrong place for myPB to be sticking out.
For nonlinear mosaicing, an overloaded version of AddIm could be
written:

```
     MA.AddIm (Dirty, PSF(ip), myPB);
```

MosAccum would have an additional image, the current model image, and
this overloaded AddIm would accumulate a GRADIENT image constructed
by adding together:

```
        PB . ( Dirty - PSF * [PB . CurrentModel])
```

for each pointing.

Since the YegSet is so primitive and no selection operations can be
done, I have opted for a purely "image plane" implementation
for the mosaic prototype. However, I was stopped short of
a working mosaic prototype by numerous minor problems
such as the low level of support for image arithmetic
and the lack of dynamic memory allocation for FilledImages.

Included below is a list of my current concerns.
Things that I needed which were not there,
things we need to think about for PROTO++....
many of which grew out of my mosaic efforts.

PROBLEMS:

March 24 1992

Need YegSet selection for various pointings

Need to associate TelParms with subsets of YegSets

Need to associate Primary Beams with subsets of YegSets and Images

---

Every time we add something new to image, or FilledImage,
such as pTelParms, we need to add it into the
copy constructor. This is not a good idea!
We should devise a scheme in which EVERYTHING is copied over.
Or at least make it POSSIBLE to copy everything over without
having to know what those things are.

Say we have a class MosAccum:

```
class MosAccum {
  public:
     MosAccum( const FilledImage &template );
     ~MosAccum() {};
     Addim( const FilledImage &Addme );
     FilledImage Mosaic();
  private:
     FilledImage IMsum;
     FilledImage PB2sum;
};
```

the constructor will look like

```
 MosAccum::MosAccum (const FilledImage &temp) {
   FilledImage IMsum (temp);
   FilledImage PB2sum (temp);
}
```

This does not work! There is no dynamic resizing of FilledImage!

We need complex images!

We should think: where do we want to put error checking?
At the top levels, bottom levels, both?
We should try to make a policy decision here.

```
Need stronger support for Image arithmetic:     I1 = I2 . I3
                                                I1 = I2 / I3
                                                I1 = pow (I2, X)
```

Currently, the FilledImage heirarchy has a complete 2-D array and a 0-D pixel. As FilledImage will need to be generalized to N-D, it is clear that we will duplicate effort in the Image Arithmetic methods UNLESS we deal with the concept of an Image Vector or Image Buffer...ie, a 1-D entity which contains all the pixel values. At the vector level, N-D images all look the same. This is also a useful concept for vectorizing code, improving efficiency. (from a discussion with Bob Sault)

We need to arrive at some sort of agreement on data TYPE:
float or double? Or both?
How will we balance the run time efficiency of float images with the fact that all the available math functions work in double?
