PHD 2.0 Architectural Description

Bret McKee

bretm@boneheads.us

2013/01

I want to start out making something completely clear -- PHD is the brainchild of Craig Stark – my name appears above because I wrote this document, not because I wrote PHD. Craig has been continually evolving PHD for many years -- new devices get support, algorithms get tweaked, thing get added. It is a fixture of the amateur astronomy community with more than a quarter million downloads. That is all the more impressive because it has always been free, and at some point he also Open Sourced it so that people who wanted a Linux version could have one, while he has mostly maintained PC and Mac versions. I came very late to the party with a seemingly simple quest – to add Adaptive Optics support to PHD. I’m a software developer in my day job, and I’m hoping to give something back to the amateur astronomy community by working on PHD.

When I started looking at the code, it looked just like it was - a system that had evolved not one that had been designed. There is no criticism intended in that remark – PHD is non-new free software that grew from humble beginning to where it is today. The state of the code made it difficult to see how to add AO support, so I started a very delicate email conversation with Craig, where I mentioned the he had built a great thing, but perhaps it was time for some spring cleaning. I was relieved when he not only didn’t take offense, but agreed. After some more encouraging email exchanges, I volunteered to take a shot at refactoring it when “I had some free time” with the new code providing the basis for PHD 2.0. In a surprising turn of events, I actually had some of that elusive “free time” and did some pretty fundamental rearranging of the PHD code base. At this point, I haven’t really added any value to the code, I’ve only added the potential to add value.

Since I just finished the first pass, and it is all still mostly all in my head (and because of the amount of changing I did, not really in anyone else’s) I thought I should write some of it down. I want to be clear, while the code I have created doesn’t look much like the code Craig wrote, one of my highest priorities was to keep the algorithms the same, so that what I finished with works the same way as what I started with. If that turns out to not be the case, the bugs are my fault.

PHD is written in C++ using wxWidgets, so I’m going to introduce the objects that now make up the PHD universe and describe their responsibilities and relationships. I’m also going to walk through the guider state machine, since it is at the heart of the new PHD code base. Hopefully that will be sufficient to allow other people to quickly come up to speed on the new code and allow them to make whatever contributions they would like. As I go through the objects I will try to mention enhancements I was thinking of toe explain design decisions I made. Reasonable people will disagree with some of those decisions – I enjoy a good technical discussion so if you think I’ve done something bone headed, let’s talk about it.

Notes:

* Most of the files have obvious names that won’t be called out in this document (i.e. if the description for Class foo does not mention where the class lives, it means that it is declared in foo.h and implemented in foo.cpp)
* Most Boolean function returns in PHD indicate error status, so true indicates that an error occurred, and false indicates no error occurred (i.e. success – think bError = function()).

It is probably easiest to start off with the classes that PHD uses to abstract physical devices.

Class Camera is the abstraction that PHD uses to talk to imaging devices. As expected the code Camera object lives in Camera.h and Camera.cpp, but it is really the Camera subclasses that support cameras. Those classes live in cam\_<type>.h and cam\_<type>.cpp (e.g. support for SBIG cameras can be found in cam\_sbig.h and cam\_sbig.cpp). Cameras have properties to communicate their interesting hardware properties, such as whether they have gain control, can take subframes, have a shutter, etc. There are methods to connect, disconnect, prepare to capture images and to actually capture an image. Almost half of the PHD code is associated with supporting the ~25 cameras that PHD supports.

Class Mount is the abstraction for things that can move where an image appears on the camera image. Mount’s primary method is Move(), which is a pure virtual function used to cause mechanical movement which ultimately results in pixels showing up in a different place on the next image. The Mount object understands how to transform move requests from camera coordinates to physical device coordinates through a process known as calibration, and based on this transform it issues commands to the physical device it abstracts.

Class Scope:Mount is a class which represents an equatorial telescope mount. It can guide in RA and DEC directions by moving motors, and there is a Guide function that takes a direction and a duration which is used by it’s implementation of Move() It also enforces limits on the durations – it is the scopes responsibility to know how long it is appropriate to run a motor. About 20 percent of the PHD code is associated with supporting the ~15 mounts that PHD supports.

Class GuideAlgorithm is an abstract class which represents an algorithm that affects guiding by potentially modifying the . Its result() represents a classic math function taking a double and returning a double. The input is the error distance, the output is the distance to be used for the move.

Class GuideAlgorithmLowPass:GuideAlgorithm replaces the current error the median of the last 10 errors + a weighted slope modifier. It also enforces a minimum move, returning 0 if the computed value is less than the minimum. See the comments the code – I think that there is an errant division but I’ve left it in to be compatible.

Class GuideAlgorithmLowPass2:GuideAlgorithm replaces the current error with the slope of the last 10 errors if that value is less than the actual error. I did not notice until after I wrote this code that the current version of PHD does not actually offer this algorithm to the user through the brain dialog.

Class GuideAlgorithmResistSwitch is designed for dec guiding. It looks at recent history and disallows the move if it would require switching the direction from the last dec guide unless the majority of recent errors have been in the direction and things are getting worse.

Class PhdApp:wxApp is the wxWidgets entrypoint. This is essentially a wxWidgets boilerplate whose main responsibilities are initialize some state, create a frame object and show the frame.

Class MyFrame:wxFrame is where most of the glue that holds the program together can be found The class implementation is mostly in either myframe.cpp or myframe\_events.cpp. There are a few routines in other places if it made sense (e.g. the routine that handles connecting cameras is in camera.cpp, since it needs all the camera includes.) MyFrame represents the window that comes up when you run PHD, and is responsible for drawing everything in that window except for the big display where the camera images are displayed (that is class Guider). It reponds to events that come from menu items, button clicks and other things. It also creates and stops the worker thread (a new addition that at the moment doesn’t do very much - more on that later). It creates the popup windows that PHD uses as well (the graph, star profile and advanced dialog (aka “The brain screen”).

Class Point reprents a location on a guide camera image. It is not derived from the wxPoint class despite the similar names because that class has very little functionality and making all the base class calls was more trouble than it was worth (I tried). Point has two public data members, X and Y, and a bunch of public arithmetic functions that operate on points – the most interesting being ones that compute the distance and angle between two points.

Class Star:Point representes a star. It is derived from Point, because a fundamental property of PHD stars is where they are in an image. Other properties of the star are its mass (the sum of all the pixels that make it up, and a signal to noise ratio (SNR) that basically indicates how much contrast there is between the star and the sky behind it. There is an overloaded Find() function which determines (and updates) the exact location of a star in an image and a routine to automatically find a star (AutoFind). The star Find() functions return a Boolean indicating success (as opposed to most Boolean routines which return an error indication). There is also a “find state” associated with a with more information about how the find operation went (if there was a low SNR, low Mass, etc…)

Class Guider:wxWindow is an abstract base class that is really at the heart of PHD’s guiding. It is responsible for dealing with new images as they arrive, by:

* Maintaining an internal state machine which controls the guiding process
* Displaying the current guide image in a window
* Drawing overlays (ra/dec lines, etc.)
* Decorating the image window to show the lockpoint and current point on the image window based upon guider state
* Figuring out a “lockpoint”, which is an arbitrary point on an image used as the desired location in the calculation which determines how far the image has moved. The user can assist this process with mouse clicks, or the Guider can select a suitable point automatically.
* Determining a “currentpoint”, which represents the position in the image.
* Making Move() requests to a mount by passing the currentpoint and lockpoint

Notice that the descriptions above do not say anything about stars – one of the things I would like to try is adding mulit-star locking and possibly plate solve based locking to PHD, and enabling these was one of the design motivators. A Guider is an abstract class that has a notion of where “it” is and where “it” wants to be, and it initiates mount Moves() to make that happens.

GuiderOneStar::Guider is the instantiation of Guider that provides the “classic” PHD guide functionalty. The “lockpoint” is the location of a star in an image, and the “currentpoint” is the location of the star in the most recent image.

Worker Thread – There is now a worker thread. My intention was to have it take the exposures and issue the guide commands to increase the responsiveness of the program. In wx, only the main thread is allowed to make GUI calls because the wx routines are not thread safe. It didn’t occur to me that the imaging and guide routines would make GUID calls, so I didn’t look at them until I had the work thread code written and working. I was having a problem, and it turns out that many of the expose and guide routines can make GUI calls – mostly in error case, but they are still in there. I add a virtual function to Mount and Camera that indicates the relevant routines don’t make GUI calls. At present none of Mounts or Cameras set that property. Think of it as an opportunity to improve. In the mean time, imaging requests are passed via a message to MyFrame member routines.

Configuration Data – In PHD 1.X, it was easy to save and restore configuration data because it was stored in global variables, and there was a routine to save them, and another to restore them. In the PHD 2.X code, these variables are dispersed in classes, and the variables are saved and restored in constructors and setter functions.

The Brain Dialog used to be monolithic, and offered settings for options which were not currently active. The same changes that affected configuration data also affected the Brain Dialog, so that had to be changed as well. It now builds the dialog “on the fly” by asking MyFrame, Guider, Camera and Mount for their pieces, which are returned as Sizers. The code was basically “cookie cutter” copied between classes.

Control flow was changed. PHD 1.X was a mix of event driven wx code and imperative code that ran for long times, calling Yield. The new Guider state machine replaces that with event driven code. Images are taken asynchronously, either by the background thread (see the note above), or by a routine in MyFrame that is evoked by the worker thread. When an image is ready, the worker thread sends an event to MyFrame, which dispatches the image to the Guider Class state machine.

Random thoughts:

* Both the Socket Server and the logging file are currently broken in PHD 2.0. It seems like a good time to have a dialog with users of these features in PHD 1.X to understand their use cases and try to modify the interfaces to make them more abstract (i.e. less tied to PHD internals)
  + For the socket server, the current interface exposes the State variable, which is based on an enumerated type that changed between PHD 1.X and 2.X. It seems like it might be more futureproof to expose functions like IsGuiding, IsPaused, etc.
    - I’m not sure if it would be a good idea, but we could consider a more standard “RPC” format. A couple that come to mind are an HTML REST interface, or something like XMLRPC. They would send a bit more data over the wire, but for most languages there are good open source wrapper that might actually make it easier for PHD to implement and consumers to use
  + Logging seems to have two very different potential use cases
    - Users wanting to get a sense that things are OK. For these users, the current format might be OK, or maybe we could make it even easier to determine how things are (i.e. less data might be appreciated here).
    - Software using PHD history data to do things like analyze a mount. For this use case more data might be appreciated, and a format that would make machine parsing less fragile might be better. If we used a format like XML, we could add additional data and consumers would not even notice, which would make it much easier to enhance the data supplied without breaking existing code.
  + There is a mix of tabs vs. spaces and windows vs linux line endings, and we should probably agree on a standard for these things, and have one checkin that normalizes all the files and puts editor hints into them for vim and emacs to help us avoid recreating this issue going forwards. This will be a big checkin that makes following history hard, but the amount of code changes in the PHD 2.0 code base has already done that. Since PHD 2.X currently lives on a branch, I could also “go back in time” and make those changes on top of the PHD 1.X code base as the first PHD 2.X checkin if that seems better.