NOTE 231 – All About the AIPS++ Display Library

David Barnes

22 May 2000

Contents

1 Overview

The AIPS++ Display Library, or more properly the display package of AIPS++, is a sophisticated class library for static and interactive display of scientific – predominantly astronomical – data. The display package also includes an interface to *Glish* which is loaded on-demand into the `glishtk` client.[1] The highest-level component of the package – the `Viewer tool` – is implemented as a set of *Glish* scripts, and is a genuine AIPS++ `tool` providing display services within the AIPS++ environment for many kinds of data.

The purpose of this document is firstly to give the reader a high-level outline of the Library – an outline not possible in the framework of the AIPS++ class documentation – and secondly to give practical advice on extending the package in a number of ways, for example to work with new display devices, to display new types of data, or to display existing types of data in new ways.

1.1 Design Principles

The initial design of the Display Library was made in 1996, taking ideas from the existing AIPS++ display application `AipsView`, and the ATNF's suite of visualisation tools. The objective was to adopt an object-oriented approach to visualisation so that:

- the many technical and esoteric aspects of data display could be encapsulated and therefore hidden from the application (`tool`) programmer,
- the support of new data types and display paradigms could be implemented in very high-level classes, maximising code re-use and minimising the burden of in-depth knowledge on the part of the application programmer, and
- visualisation applications could be built up in a modular fashion, consistent with the AIPS++ paradigm.

In 1998, the Display Library design was extended to enable the construction of visualisation tools from *Glish*. Instead of using the AIPS++ Distributed Object (DO) system, the extension made use of the *Glish* proxy system, and the dynamic loading capabilities of the one existing *Glish* proxy – `glishtk`. The first *Glish* application created using this system – the

[1] The `glishtk` client is a *Glish* proxy client which provides the Tk widgets to *Glish*.

`viewer` – is, nevertheless, a true AIPS++ tool which can be managed by the AIPS++ Tool Manager.

2 Display Library Components

The Display Library is written in C++, and makes extensive use of the basic object-oriented design concepts of encapsulation, inheritance and polymorphism. The Library is built upon the existing AIPS++ classes, and a sound knowledge of the *Array, Lattice* and *Mathematics* classes in particular is recommended before approaching the following sections.

2.1 PixelCanvasColorTables and Colormaps

Since color plays an important role in visualisation of scientific data, it is important to provide flexible ways to allocate and use colors within the Display Library. To this end, a layered approach to colors has been adopted, wherein one class – the *PixelCanvasColorTable* (for brevity, the *ColorTable*) – is primarily concerned with acquiring and managing blocks of color cells from the underlying hardware, and a second class – the *Colormap* – is mostly concerned with generating sets of colors to be installed in the color cells managed by one or more ColorTable objects. The key to understanding color in the Display Library is to understand the interaction between these two classes, and to more or less ignore the hardware-dependent aspects of the ColorTable class.

The PixelCanvasColorTable class provides dynamic color cell allocation functionality, using primitive routines appropriate to the device in use. For each display device provided by the Display Library, there must be an implementation of the PixelCanvasColorTable class. Upon construction a ColorTable determines the color-specific capabilities of the display device, and where appropriate, acquires control of some number of the device's color cells.[2] The number of color cells controlled by the ColorTable should be dynamic (where possible), and under the control of the programmer and thereby the user.

In use, the role of the ColorTable is to dynamically allocate space out of its own set of color cells to one or more registered Colormaps. A Colormap provides a series of colors, on demand, which can be used to fill a series of

[2] For some devices, the concept of color cells is invalid, and the ColorTable simply makes things "look like" some number of cells has been allocated. Differences like these – between devices – are encapsulated by the PixelCanvasColorTable class.

vacant color cells provided by a ColorTable. Typically, such a set of colors would be used to represent a range of data values, where the minimum data value is mapped to the first color cell, the maximum value to the final color cell, and all other values to cells within the available range.

The interaction between, and division of responsibilities between, ColorTables and Colormaps is important. One or more ColorTables can be constructed for a given display device. For example, a screen display device may provide up to 256 colors, and two ColorTables, each initially acquiring 80 colors, could be created for that screen. The dynamic nature of the ColorTables means that once they have been made, they can be expanded or shrunk, provided enough colors are available. For example, the first ColorTable in this example might then be shrunk to 40 colors, and the second expanded to 120 colors.

In a similar way, one or more Colormaps can be registered with a ColorTable. That is, it is possible to register a greyscale Colormap and a rainbow Colormap on the first of our ColorTables above. Initially, they would each soak up 40 colors (since the ColorTable provides 80 colors), but after the shrink operation described above, they would soak up only 20 colors each. See Figure 1 for a pictorial representation of the hardware color resources, the AIPS++ ColorTable and the AIPS++ Colormap. In order that any images using these Colormaps are still drawn correctly on the screen, a series of callback functions are initiated when ColorTables and Colormaps are resized.

Finally, it is important to stress that in a single application or tool, multiple Colormaps of the same base type can exist. That is, more than one "rainbow" Colormap can be generated, and each can be modified independently. Modification might include altering the brightness or contrast, or "stretching" the colors in the map. The same Colormap can be registered on more than one ColorTable, and furthermore may occupy a different number of color cells in each ColorTable! Designing an interface which allows the end-user complete control over this extensive flexibility has proven to be challenging however, and only a subset of the capabilities are provided at present.

2.2 The PixelCanvas

The *PixelCanvas* is the fundamental drawing canvas of the Display Library. It provides an agreed-upon set of routines for drawing graphic primitives, for controlling the "context" of the drawing operations, for handling interactive events, and for supplying an advanced caching mechanism (where possible).

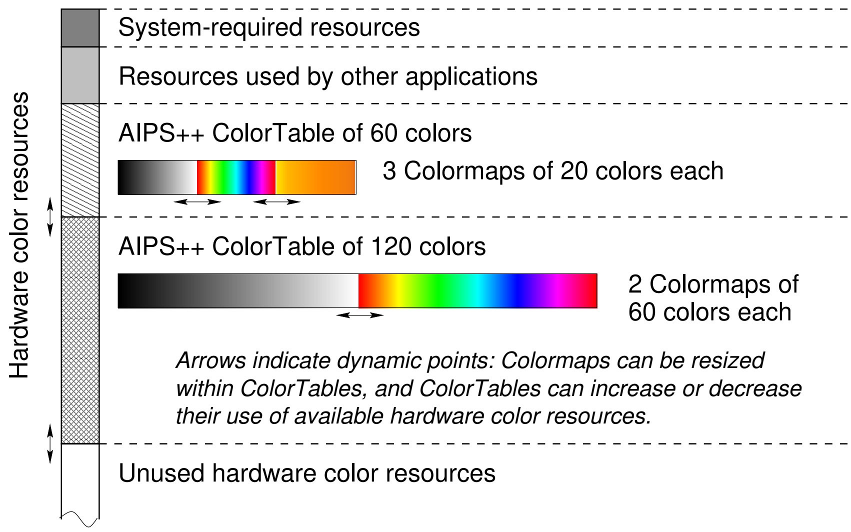

Figure 1: The relationship between hardware color resources, AIPS++ ColorTables and AIPS++ Colormaps.

For each hardware drawing device supported by the Display Library (eg. X-Windows, PostScript) there must be a PixelCanvas class which implements the required routines using low-level, device-specific routines.

The graphic primitives of the PixelCanvas include points, lines, polygons, ellipses, and images. Multiple interfaces are provided where convenient and possible[3], and these can in some instances be implemented with templated member functions in derived classes. All drawing is done in the native units of the device, and in cases where these units are not pixels – for example, pixels are user-defined in PostScript – the constructor for the device-specific PixelCanvas is expected to offer a method for (arbitrarily and artificially) setting the resolution of the device such that PixelCanvas drawing commands retain their meaning across all devices. Again, the PixelCanvas makes use of encapsulation to conceal device differences, and inheritance and polymorphism to enable the same set of drawing commands to be used on any valid PixelCanvas.

2.3 The WorldCanvas

The *WorldCanvas* implements a higher level of drawing capabilities on top of the PixelCanvas class. Multiple WorldCanvas instances can be constructed

[3]Note to developer: a particular recommendation is that this interface be pruned to perhaps provide just `Int` and `Float` interfaces.

on a single PixelCanvas, each drawing on a subset of the entire PixelCanvas. The bulk of the infrastructure provided at the level of the WorldCanvas is concerned with coordinate transformations, so that drawing on a WorldCanvas is accomplished by the programmer in full world coordinates, or in a linear coordinate system which mediates between the pixel coordinate system of the PixelCanvas, and the full world coordinate system of the WorldCanvas. Further facilities include higher-level handling of interactive events, and data resampling and rescaling facilities.

Each WorldCanvas is constructed with reference to a specific instance of a PixelCanvas, and occupies some fraction of the PixelCanvas. The position of the WorldCanvas is dynamically adjustable, and an interface can be imagined in which the user freely positions multiple WorldCanvases across a single PixelCanvas, in order to depict several views of one or more datasets in a form suitable for publication, for example. Overlapping WorldCanvases will ultimately need a "depth specifier", so that drawing order can be controlled and preserved by the programmer and/or user.

In terms of graphic primitives, the WorldCanvas essentially replicates those present in the PixelCanvas, and applies coordinate conversions to input positions as necessary.

2.4 Events

A major part of the "glue" which holds together the many components of the Display Library is the event generation and handling system. At the very lowest level, three fundamental event types are defined: *refresh events, motion events* and *position events*. Each of these has a corresponding event handler type, for example a *motion event handler* is expected to process motion events. Throughout the Display Library, classes and instances of classes can register their interest in events, and implement event handlers which do something in response to a particular event, or pass the event on to another interested party, or both. For each event and event handler (EH), there are classes specific to the PixelCanvas (PC) and WorldCanvas (WC).

2.4.1 Refresh Events

The *refresh event* mechanism is used to synchronise the display of data to the user. For example, when a parameter is changed in a DisplayData, a refresh event should be generated such that all WorldCanvases displaying that DisplayData are redrawn if necessary. Or, if a PixelCanvasColorTable is physically resized, all PixelCanvases using that ColorTable must be cleared and redrawn.

Any class in the Display Library can register its interest in refresh events in one of the following ways:

- it can inherit from the PCRefreshEH or WCRefreshEH classes;
- it can create an instance of a helper class which forwards events to the class itself; or
- it can attach predefined event handlers, or *Tools*, to itself.

Having taken one of the options above, the class need simply then register itself, or its helper class, as an event handler with all potential sources of the event of interest. A specific example is the WorldCanvas class, which wishes to know when its underlying PixelCanvas is resized, so that it can resize itself accordingly. To this end, the WorldCanvas class creates a helper class, called "WCOnPCRefreshEH", an instance of which can be registered as a handler of PixelCanvas refresh events, and whose only function is to add World Coordinate information to the event and pass it on to its parent WorldCanvas. The function called on the WorldCanvas can then look at the type of the refresh, and if it is a size change, will then act accordingly.

There are many possible reasons for a refresh event to be generated, and indeed such an event can be generated by any class which allows other objects to register themselves as handlers of refresh events. The types of refresh event are listed in `DisplayEnums.h`, and include:

- `UserCommand` – this is reserved for explicit refresh requests from the user.
- `ColorTableChange` – the allocation of hardware resources has changed, probably at the level of the PixelCanvasColorTable, and consequently most drawings on PixelCanvases sharing the ColorTable of interest will need to be redrawn.
- `ColormapChange` – some internal change to a single Colormap has occurred, which may mean that data being drawn with the particular Colormap needs to be redrawn.
- `PixelCoordinateChange` – the PixelCanvas has changed shape. All WorldCanvases on top of this PixelCanvas will need to erase themselves and redraw to the new geometry.

- `LinearCoordinateChange` – presently unused, this type of refresh might be required if the WorldCanvas elects to re-negotiate its linear coordinate system.

- `WorldCoordinateChange` – presently unused, this type of refresh could be used if the WorldCanvas elects somehow to re-negotiate its world coordinate system.

- `BackCopiedToFront` – a saved image of the PixelCanvas has been restored to the display, and drawings which are not cached on the (back) buffer may need to be redrawn.

- `ClearPriorToColorChange` – all Pixel and WorldCanvases need to be cleared before a PixelCanvasColorTable change is applied. This can be used to remove drawings before a new Colormap is mapped into the ColorTable, so that invalid images are not left on the screen.

2.4.2 Motion and Position Events

A motion event is deemed to have occured when the user moves the mouse pointer (cursor) over an exposed PixelCanvas. The motion event is described by the position of the pointer. A position event is deemed to have occured when the user presses or releases a keyboard button or mouse button while the input focus is on a PixelCanvas. PixelCanvas position and motion events are generated by the low-level device interface for some, but not all, PixelCanvases. For example, an interactive X-Windows PixelCanvas should generate these events, but a non-interactive PixelCanvas such as a PostScript PixelCanvas will have no need to pass on events to higher-level objects.

Objects and classes in the Display Library can register their interest in motion and position events in the same way as they can for refresh events.

2.5 Tools

In the C++ Display Library, the term *Tool* has a meaning quite different to the standard meaning in AIPS++. A tool in the library is a single entity which can react to – and generate – refresh, motion and position events. A specific example is the zooming tool. This tool listens for position and motion events which indicate the user wishes to zoom in on a part of a WorldCanvas. By reacting in a particular way, this tool allows the user to stretch out a rubberband rectangle, adjust its size, and then double-click inside the rectangle. Upon detecting a double-click, the zoom tool sets some new parameters on the WorldCanvas, and then *sends* a refresh event to the WorldCanvas. In response, the WorldCanvas notices that the coordinates have changed, and redraws itself accordingly. Other tools already implemented include the region tools and Colormap fiddling tools.

2.6 DisplayDatas

The DisplayDatas are a very high-level set of classes in the Display Library, and ultimately will be the place where most programming is done in the Display Library. These objects encapsulate data *and* a drawing method, such that they simply draw their data on demand into a supplied WorldCanvas. The *WorldCanvasHolder* class (see below) manages a set of DisplayDatas and a single WorldCanvas, and coordinates the refresh cycle of the WorldCanvas so that each DisplayData is drawn in turn.

An example DisplayData is the LatticeAsRaster class. It is constructed with reference to an ImageInterface or an Array, which is registered as the source of the data to be displayed. The LatticeAsRaster class can then be called to draw a depiction of the data on any WorldCanvas, and that depiction happens to simply be a false color pixel map of some subset of the data. The LatticeAsRaster object offers various parameters for the user to tune, including such things as what Colormap to use and how to resample data pixels to screen pixels. These parameters are independent of the display device (ie. the PixelCanvas) and so the DisplayData can be simultaneously displayed on more than one device.

The critical function in a DisplayData implementation is the `draw` method. This method is called to get the DisplayData to draw itself on the supplied WorldCanvas. A quite sophisticated caching mechanism is now available such that if this WorldCanvas has previously been drawn on by this DisplayData, each in the same state as they were last time, then a cached drawing can be rapidly drawn to the WorldCanvas. To use this caching mechanism, the programmer need simply derive from the special class `CachingDisplayData`. More on this below.

2.7 The WorldCanvasHolder

The WorldCanvasHolder has been touched upon already. Its main task is to manage a single WorldCanvas, and some number of DisplayDatas who are "registered" to draw on that WorldCanvas. Consequently, the main activity of the WorldCanvasHolder is to register an interest in refresh events occuring on its "contained" WorldCanvas, and upon the occurence of said events to arrange each DisplayData to draw itself in turn. The bulk of the user interface of the WorldCanvasHolder is related to implementing various event handlers for the WorldCanvas, to managing the collection of registered DisplayDatas, and to arranging a set of "restrictions" which are used to assist the DisplayDatas in determining what of their data to draw, and how.

2.8 The MultiWCHolder and PanelDisplay

Over the period of development of the first Display Library application — the *Viewer* (see below) — it became clear that significant demand existed for the type of display where multiple slices from a multi-dimensional dataset are shown in individual panels forming a grid across the visible surface. This type of display, a *PanelDisplay*, is common in multi-frequency radio astronomy packages, and is required in AIPS++. One of the most recent additions to the Display Library then, has been the *MultiWCHolder* and *PanelDisplay* classes. The first of these is essentially a class which is capable of managing a number of WorldCanvasHolders, and which provides functions to register and unregister DisplayDatas across all the managed WorldCanvasHolders: it is a *holder of WorldCanvasHolders*.

The PanelDisplay class wraps all this up in a convenient class which creates a set of WorldCanvases and WorldCanvasHolders on a given PixelCanvas according to specifications given in the constructor, creates a MultiWCHolder to manage these WorldCanvasHolders, and which provides convenience functions for registering and unregistering DisplayDatas across the PanelDisplay.

A new Animation tool will need to be written for the MultiWCHolder to provide smart animation whereby the individual slices of data "march" across the screen from panel to panel. Further tricky issues have arisen in the development of the multi-panel facilities, and relate mostly to event handling in the case of overlapping, or nearly overlapping, WorldCanvases on a single PixelCanvas. The developer in the future will need to decide whether events should be consumed by the first WorldCanvas asked to process the event, or whether they should instead be processed by all WorldCanvases. The latter solution could create havoc for even simple operations, such as zooming, and needs extremely careful thought. However, this is the current mode of operation...

3 The Glish Connection

3.1 Low-level Interface: gDisplay.so

The `glishtk` proxy client has been chosen as the (current) vehicle to provide the Display Library services to the AIPS++ user. This facility has been provided this way so that Display Library canvases could be embedded in the AIPS++ graphical user interface elements which were already supplied by by the `glishtk` proxy client. Additionally, the nature of `glishtk` enables the Display Library services to be loaded *dynamically* into an already-running client. Had the interface been implemented more along the lines of the existing *Glish* distributed-object mechanism, then the canvases of the Display Library would be confined to their own window/s and display tools would have been unable to exploit the graphical user interface elements (eg. buttons, menus) already provided by `glishtk`.

To this end, the major components of the Display Library have been collected into a dynamically loadable Tcl module, known as `gDisplay.so`. At run-time, this module can be loaded into memory and be incorporated into `glishtk`'s execution thread using Tcl's dynamic loading facility. Ideally, a transparent interface could be written to provide constructors, methods and destructors for each class in the Display Library, and then instances of any of the Library classes could be constructed and built from *Glish*, which of course controls the `glishtk` client itself. However, this approach seemed too fine-grained, and would result in a moderately heavy burden on the *Glish* programmer who wanted to produce a simple visualisation application. Therefore, the *agents* which have been implemented in `gDisplay.so` are at an even higher level than the top-level Display Library tools, and thereby provide more convenient but less capable interfaces to the *Glish* programmer. In the future, this design decision should be revisited.

The agents which are presently implemented in `gDisplay.so` include:

- **pixelcanvas**, a basic interface to put a PixelCanvas in a Tk frame,
- **pspixelcanvas**, a very simple interface to create a PostScript PixelCanvas which can be used for saving displays of data,
- **worldcanvas**, which is a compact representation of the union of a WorldCanvas object and a WorldCanvasHolder object, and which can be constructed upon a pixelcanvas or a pspixelcanvas,
- **displaydata**, which is a moderately sophisticated agent capable of creating several distinct implementations of the DisplayData base class,

and performing operations on such,

- **colormap**, which is a compact agent managing a single colormap, which can be edited from *Glish* and attached to displaydata agents,
- **animator**, which is a simple agent for managing animation operations on worldcanvas agents, and
- **drawingdisplaydata**, which is an agent in development for doing simple drawings on a worldcanvas agent from the *Glish* command-line or a *Glish* script.

The source code for the `gDisplay.so` module can be found (at the moment) in `trialdisplay/apps/gDisplay`, and is moderately compact. Typically the source code files in this directory are implementations of the `Proxy` class which is defined in the *Glish* sub-system, and the simplest example to peruse initially is the `GTkColormap` class. The `Proxy` class implementations simply provide construction, destruction and a selection of event interfaces which are accessed from *Glish*.

3.2 High-level Interface: the Viewer

The raw interface to the `gDisplay.so` module loaded into the `glishtk` client is not a particularly convenient one for building up visualisation and display tools at the AIPS++ level. To address this, the proxy agents of `gDisplay.so` have been "wrapped up" with a substantial layer of *Glish* code. This code (where possible) enforces correct use of the proxy agents, and reinforces the object-oriented nature of the display library. It is very important to realise that the Viewer code is merely one possible implementation of the `gDisplay.so` proxy agents, and others which look quite different to the programmer and end-user are easily imaginable.

One very important feature of the Viewer interface is that it unifies pixelcanvas and worldcanvas agents. As the worldcanvas proxy agent combined the WorldCanvas and WorldCanvasHolder classes, the `viewerdisplaypanel` combines the worldcanvas and pixelcanvas agents into one entity. The viewerdisplaypanel is at the time of writing probably the highest-level object in the entire chain of AIPS++ display services. The viewerdisplaypanel is a valid AIPS++ tool, and thus provides method-oriented interface to control its various and many attributes. Construction arguments for the viewerdisplaypanel include width and height settings, foreground and background colors, colortable types ('index', 'rgb', 'hsv'), colortable dimension/s, and flags indicating what user interface elements should be used to adorn the display area itself. Available interface elements include a menu bar, a control bar for selecting and controlling interaction with the display, a button bar, and a tapedeck for controlling animation of the display. Most importantly, the viewerdisplaypanel can be placed in an existing GlishTk frame, thus enabling the embedding of visualisation applications in existing and new AIPS++ tools.

The `viewerdisplaydata` is the AIPS++ tool which can be used to manage the displaydata proxy agents of `gDisplay.so`. It is a moderately clean interface, wherein most methods provided to the user are sent directly on to the proxy agent code in `gDisplay.so`. Viewerdisplaydata tools may be freely registered and unregistered on viewerdisplaypanel tools. A top-level tool, the `Viewer`, is provided to assist in constructing and managing the various viewerdisplaypanels and viewerdisplaydatas, and so a rudimentary, but quite powerful, visualisation application can now be expressed this simply:

```
- include 'viewer.g'
- mv := dv.newdisplaypanel();
- mdd := dv.loaddata('test.im', 'raster');
- mv.register(mdd);
```

4 Programming Practicalities

4.1 Implementing a new PixelCanvas

To support a new output device for AIPS++ display applications or tools, it will be necessary to implement a new PixelCanvas and PixelCanvasColorTable pair. At the time of writing, two major PixelCanvas implementations exist: the X11PixelCanvas for display to X Windows devices, and the PSPixelCanvas for hardcopy output using PostScript. A slightly specialised version of the X11PixelCanvas — TclTkPixelCanvas/GTkPixelCanvas — exists to facilitate the implementation of the Display Library interface in `gDisplay.so`. New PixelCanvases are envisaged for OpenGL and Java devices in the future, but are not yet included in planning documents.

Development of a new PixelCanvas should begin with the design of a PixelCanvasColorTable which manages color allocation strategies on the new device. The PixelCanvasColorTable class is quite sophisticated, but should be adaptable to most display device allocation schemes in the near future. The most important facilities of the PixelCanvasColorTable to implement for new devices are the capability to dynamically resize the entire ColorTable itself, and to dynamically extend and reduce the color usage of the various

Colormaps registered on the ColorTable. Of high importance also is the facility to provide multichannel visuals even on devices whose intrinsic software or hardware does not provide such. For example, the X11PCColorTable provides HSV and RGB facilities even for native X PseudoColor visuals.

With the PixelCanvasColorTable in hand for the new device, attention can turn to the PixelCanvas itself. From the outset, a decision should be made on whether a display list caching facility will be provided. This is highly recommended for interactive devices, such as OpenGL, but is seen as less important for non-interactive devices, such as the existing PSPixelCanvas which has no great need to offer caching services. For interactive devices, code to turn native events (such as mouse movement and key presses) into PC{Motion,Position}Event objects as appropriate must be implemented, and finally the graphics primitives must be implemented to a high level of quality and trustworthiness. Ultimately, it is the intention to provide full test wrappers for all classes in the Display Library, and PixelCanvases being one of the fundamental classes should receive such attention early on in the development process.

4.2 Writing new Tools

The first decision to make when writing a new tool (control) for the Display Library is whether the tool needs to detect events on the PixelCanvas, or instead on the WorldCanvas. Events on the PixelCanvas are appropriate, for example, when the user is going to modify a parameter which might be applied to objects which are not necessarily specific to a WorldCanvas, eg. a Colormap. Events on the WorldCanvas are appropriate when the tool will need real world coordinates to accomplish its task, as is the case for example in zooming. Having made this choice, the programmer should then derive from either WCTool or PCTool in the DisplayEvents module of the display package. In certain cases, deriving from existing higher-level classes such as WCRectTool will be good practise and buy the developer *and user* significant common ground. There are sufficient tools now implemented to give good example code on how to extend the tool beyond the derived capabilities. The majority of this code can be found in `trialdisplay/implement/DisplayEvents`.

4.3 Writing a DisplayData

Writing a new DisplayData still remains the most complex task in developing the Display Library, and no doubt it would be worth spending a considerable amount of time on improving the programmability of DisplayDatas. This complexity is primarily because the DisplayData object is expected to do a lot of work, both in accessing and extracting data from storage, and actually drawing the data to the screen. The complexity also arises from the fraction of development time spent on DisplayDatas to date, compared to the magnitude of the work they are ultimately expected to do. There is no doubt that parts of the DisplayData class interfaces need further work, and this work will need to be carefully coordinated in the future amongst the various Display Library developers. Anyway, on with the show ... Start by examining the code available in `trialdisplay/implement/DisplayDatas`: the primary interfaces that must be understood are the *DisplayData* and *DisplayMethod* classes. These two classes define the interface for the entire DisplayData class hierarchy, and are therefore of utmost importance.

After digesting these two classes, attention should be given to the immediately derived classes *CachingDisplayData* and *CachingDisplayMethod*. These two classes add to the base class interfaces to implement "automagic" caching facilities for DisplayDatas. The idea is that if a DisplayData has previously drawn itself on a WorldCanvas, with a set of given options or attributes or characteristics, then if requested to do so again, and the conditions have not changed, or at least have reverted to the state they were in, then a cached drawing should be available to place rapidly on the viewing surface (ie. the WorldCanvas). The main trick to using these classes correctly is to properly implement the `optionsAsAttributes` function, and to correctly manage the set of `Attributes` on the DisplayData itself. If this is done correctly, as described in the class documentation, then classes derived from CachingDisplayData and CachingDisplayMethod need never worry about implementing the `refreshEH` method of the base DisplayData class.[4] It is the humble suggestion of the author that all future implementa-

[4]For the real details, examine the implementation of `CachingDisplayData::refreshEH` while you read the following notes:

1. First we just find out the WorldCanvas we're dealing with, and likewise the WorldCanvasHolder. Next we get a copy of the restrictions on the WorldCanvasHolder. For our DisplayData 'NGDD' to draw, it will have to match some of these restrictions.
2. To the WorldCanvasHolder restrictions, we add the state of the WorldCanvas - basically we describe its geometry on the underlying PixelCanvas. Then we get a copy of the restrictions on the DisplayData, and to this copy, we add restrictions which describe the options the user has set on the NGDD. This is done with the line: `ddRestrictions->add(optionsAsAttributes());`
3. If caching is **on**, we search our list of previously made CachingDisplayMethods, to see if we can find one whose restrictions match the two buffers created above, and

tions of DisplayDatas be made by deriving ultimately from the CachingDisplay classes.

Other than actually accessing data and drawing it to a WorldCanvas, one very important task still not implemented at the level of the CachingDisplayData class is `DisplayData::sizeControl`. The job of this method is — on request from a WorldCanvas — to set up the linear coordinate system (LCS) of that WorldCanvas. Very roughly, two types of DisplayData exist: those that do implement `sizeControl` and can therefore set up the WorldCanvas LCS, and those that leave a null implementation and return `False` to the caller. The latter type can be termed *passive* DisplayDatas, and they can only ever draw once another DisplayData's sizeControl method has been called for the WorldCanvas/Holder on which they are registered. As such, they are pretty easy to write, but only suitable for use as overlays on other DisplayDatas.

The former type, the *active* DisplayDatas, must have the `sizeControl` function implemented in a non-trivial way, and return a `True` value to indicate to the caller that indeed they did perform a "sizeControl" action. The sizeControl function is expected to set up the LCS of a specified WorldCanvas. It must honour any zoom level that is presently set, and then satisfy any settings on the DisplayData itself regarding the coordinate system. While the sizeControl function is quite complex, it ultimately provides tremendous freedom in terms of the types of DisplayData that can be implemented. For example, in the sizeControl function, a DisplayData could actually mod-

---

if they have in the past been drawn on the WorldCanvas we're supposed to draw on now:

- If we *don't* find such an element (`if (!found) { ... }`) we construct a new one by calling the method `newDisplayMethod`, which *must* be implemented in derived classes, and add the result to our list. We keep a pointer to it though, since we'll draw it below. The `purgeCache` call is made to make sure the cache doesn't get too large.
- If we *do* find such an element, let's do nothing, because `cdMethod` now points to it, so we can ask it to draw just a bit lower down in the code.

4. On the other hand, if caching is **off**, let's just make a new displaymethod using the `newDisplayMethod` function, but we'll delete it after we use it!
5. So now we can delete the restriction buffers we used to compare to our list of already drawn "views", we install a Colormap if required, and just ask the method to draw itself: `cdMethod->draw(ev.reason(), *holder);`
6. Finally, if caching is off, we just delete the `cdMethod` pointer and free the memory used by our drawing.

ify its data based on geometry of the WorldCanvas. A specific example of where this could be used is in some application where the user is allowed to drag a WorldCanvas around a PixelCanvas, and the WorldCanvas acts as a "window" to some larger dataset beneath it — just an idea!

Of course the final task of a DisplayData is to actually draw data to the WorldCanvas upon request. In the CachingDisplayData branch, this is handled by the (already implemented) `refresEH` function constructing a new DisplayMethod if necessary, and then calling the `draw` method within

4.4 Adding Agents to the GlishTk Interface

Adding new *proxy* agents to the GlishTk interface is a fiddly procedure. Aside from making decisions about whether the new agent maps exactly to an existing Display Library class, or instead makes use of multiple classes, there is a moderate amount of coding to be done. Design decisions aside, let us concentrate on the `GTkColormap` class, which contains a Display Library `Colormap` object. This class is an implementation of a `GTkDisplayProxy`, and makes use of some services from the `DisplayOptions` class. A cursory examination of `GTkColormap.h` will reveal that this class mostly provides a brief interface to access the facilities of the `Colormap` class, eg. `char *GTkColormap::setoptions(Value *);` is provided to allow a number of settings to be made from *Glish*. In the way of private data, `GTkColormap` simply stores a pointer to a `Colormap`.

Turning now to the implementation file, `GTkColormap.cc`, most of the interface work is done in the constructor. Each proxy agent must have a unique `agent_ID`, and this tag can be used by other agents to check that a provided agent for some operation is of the correct type, before casting. The contained `Colormap` is constructed, and then a couple of widget commands are installed. Just follow the existing code, which simply sets up a small map of Strings to functions (actually offsets into the class object). At the end of the constructor it is very important to mark the agent as valid, so that destruction can be handled properly.

The next function is a global C-style function, `GTkColormap_Create`. This is actually the function which is registered with Tcl to create `colormap` proxy agents, and after parsing the arguments (`Value *args`), explicitly constructs a `GTkColormap` object. Many things can go wrong here, so a fair component of the code is actually error checking code. This global function is actually registered to create colormap proxy agents by code in the `Gdisplay_Init` function implemented in `gDisplay.cc`. Destruction of the `GTkColormap` agent is straightforward. There follows a simple implementation of the required `IsValid` function, whose return value is examined when the destructor of the base class is called.

Finally, in `GTkColormap.cc`, implementations of the widget commands themselves (in this case `setoptions` and `getoptions` are given. Most of their work is involved in turning arguments coming in to the function in the `Value *args` style into useable data for native Display Library class functions.

Let's now look at the `gDisplay.h` and `gDisplay.cc` files. To "export" the `GTkColormap` class to *Glish*, it is necessary to add protected data to the `TkDisplayProc` class which can store the offset of a widget command in the `GTkColormap` class, and provide a constructor which can set up this protected data member when necessary. In `gDisplay.cc` the `operator()(...)` must be extended to handle the case where the stored offset is one into a `GTkColormap`, and call the method at the stored offset. Additionally, the global function`GTkColormap_Create` must be declared in this file, and then registered with a call to `GlishTk_Register` in the implementation of `Gdisplay_Init`.

It is worth pointing out that the function `Gdisplay_Init` also does a very important thing regarding PGPLOT. It initialises the global pointer `display_library_pgplot_driver` to point to the global external function `wcdriv_`, in order that PGPLOT be able to draw on Display Library World-Canvases. `Gdisplay_Init` is called when the Display Library module `gDisplay.so` is loaded, and so if the Display Library is *not* loaded, the global pointer `display_library_pgplot_driver` remains zero, and is (correctly) never available for use.

The last link in the chain to get `colormap` agents working from *Glish* is to add appropriate code to the *Glish* script `gdisplay.g`. This script actually provides the mechanism from *Glish* for loading the `gDisplay.so` module, and thereby installs global symbols in *Glish* for constructing the various proxy agents. Simply follow the existing code to add a new agent.