AIPS++ Note 172:
Infrastructure Graphics

Paul Shannon, NRAO

December 8th, 1994

1 Introduction

A distinction has developed within AIPS++ between "Infrastructure Graphics" and "Advanced Visualization". The Visualization work takes place at BIMA/UIUC, and is characterized by its use of high-end computer graphics hardware and software, by its arm's length relationship to AIPS++ libraries and classes, and by a focus upon new and novel display techniques for radio astronomy data. The infrastructure graphics, on the other hand, is fully integrated with the AIPS++ libraries and classes; it make almost no assumptions about computer hardware or special-purpose graphics libraries; and it is to be used for a wide variety of display tasks, from the mundane (simple x-y plots) to the demanding (a movie of a spectral cube, volume rendering). This short paper discusses AIPS++ Infrastructrure Graphics. The discussion consists of four sections:

- A short technical summary
- A presentation of the design principles and goals
- A description of several pieces of freely available graphics software which we have adopted which have allowed us to make considerable progress in a short time
- An example that shows the system in action

2 Technical Summary

AIPS++ infrastructure graphics is built upon the X Window system, using the Motif widget set, and considerably enhanced by an "Astronomical Canvas Widget" written by Richard Gooch of the ATNF. These building blocks are used to construct higher-level software components which may be configured and used

in a variety of ways, and in a variety of settings. (Some example Components: a Table browser/editor, an Image display, an x-y plotter.) Each component is derived from the same C++ base class, a "UIComponent" modeled after the work of Douglas Young.[1]

3 Design Principles and Goals

The graphics design is driven by two observations, neither of which is controversial nor especially insightful, but which do have a far-reaching and clarifying effect:

1. There are a small number of graphical tasks which come up again and again in the analysis and exploration of radio astronomy data.

2. These tasks will appear in many different contexts and combinations, some of which we cannot anticipate.

   These observations lead quite directly to the following principle of design: AIPS++ must provide a set of GUI components, which are easy to combine and configure, and which may be extended in the future by subclassing. Each GUI component shall be designed so that it can

   - be called from an otherwise non-graphical program, thus freeing the casual programmer from learning the intricacies of computer graphics
   - assembled, along with other components, into a fancy, integrated application, with many controls and options
   - appear as part of a stand-alone application of narrow focus, for example: the AIPS++ TableEditor allows many tables to be viewed simultaneoulsy, with a small control panel providing centralized control (for iconifying and deiconifying windows, raising windows to the top, and opening new windows)
   - called and manipulated from the glish command line, or from any process connected to the glish "software bus".
   - has (as much as possible) an abstract public interface, which describes what the component does: most window-system-specific details should be hidden from view.

   Some further principles of design:

3. Many of the graphical tasks we want to perform are neither unique to our project nor cheap and easy to build, so we should adopt and adapt existing software into our own system whenever possible.

[1] Object-Oriented Programming With C++ and OSF/Motif, Prentice-Hall, 1992.

4. From the start, we should anticipate the problems that will be posed by "creeping featurism" as the years go by, and as our components are put to highly specialized and presently unforseen use.[2] The best way to prepare for these future uses is by the careful use of inheritance: the latest special purpose plotter component, for example, should be just a subclass of a plotter we've already written.

4 Public Domain and Adopted Software

AIPS++ adopted the Motif widget set after a year's frustrating work with InterViews, and in spite of our admiration for its capabilities and that of its (as yet undelivered) successor, Fresco. Motif is now standard on all commercial workstations (Linux PC's excepted), and a developer's license costs less than a few hundred dollars. It is a mature toolkit and – of immediate benefit to AIPS++ – comes with a large number of truly public domain widgets. We have already made extensive use of XbaeMatrix, a spreadsheet widget from BellCore that forms the heart of the AIPS++ TableBrowser and ArrayEditor.

A short time ago we decided to adopt Richard Gooch's "Karma" graphics library rather than to create an X Windows astronomical canvas from scratch. Karma provides an Xt widget which, among other things,

1. allows the use of astronomical coordinates
2. supports non-linear coordinate systems
3. provides easy colormap manipulation
4. provides postscript output for printing
5. allows for event handlers which return values in astronomical coordinates and intensities
6. permits interactive placement of annotations and labels
7. provides zoom and (soon) pan and scroll
8. any graphical object (axes, markers, text, region-delimiting polygons) may be either drawn in immediate mode, or entered into an overlay list where it may be later manipulated. All objects can be drawn in astronomical or pixel coordinates.

[2] One advanced but not implausible application, which would test the extensibility of our design, is the near real-time analysis and display of data from a telescope, in which the results of preliminary analysis are used to guide decisions about telescope pointing. As an aside, and with a tip of the hat to the Monitor and Control software development team at the Green Bank Telescope (where such an application might have some appeal): both of our projects use glish, and so assembling separately developed software in this way is neither fanciful nor naive.

5 Being Prepared for New Developments in Computer Graphics

The several benefits of our current design and of our adopted software should be clear to see. They come, though, with a possible cost: will AIPS++ be so strongly tied to these choices that we will be unable to pursue compelling or dominant new techniques, tools, and operating systems as they come along? There are at least three areas of concern:

1. A graphics toolkit with embedded interpreter (i.e., the recently popular TCL/TK)
2. New (or newly important) toolkits and graphics libraries: Fresco, OpenGL, OpenInventor
3. Windows NT, Plan 9, Taligent

It is quite possible that some new graphics system, widget toolkit, or operating system will sweep the scientific community in the next 5-10 years, offering so many gains in efficiency and expressiveness that our software will become obsolete. But it is inconceivable that this would happen without there first being some time in which the new and the old systems co-exist, and in which we could mix and match tools according to our best judgement. More specifically:

1. TCL/TK is not yet mature enough to be used for all of our graphics needs.[3] Its primary virtues appears to be its embedded interpreter, and the resulting ease with which casual programmers can construct and configure an interface. If this end-user configurability becomes a requirement for AIPS++, it would be possible to add TCL/TK to our graphics tools, though it is unlikely that we would choose to do so: TCL's only data type is character strings, and as an interpreter, it is much less powerful and elegant than glish. Some limited end-user configurability could be easily accomplished with resourceful use of glish and Motif. Still, TCL/TK *could* be added if there were a compelling reason to do so.
2. In the past, a lot of very interesting user interface programming has been done with InterViews. It's likely that related work will soon be done with Fresco – indeed, Fresco offers many features (graphical embedding, arbitrary geometric transformations) that make it ideal for constructing powerful user interfaces, and it may become very important in years to come.[4]

[3] Actually, it is the TK widget set that is immature – TCL itself is quite well developed and robust.

[4] This author has long believed that an innovative catalog manager could be written in Fresco or OpenGL and that, done right, this would be a real boon to astronomers managing complex and inter-connected data sets. See, for instance, Communications of the ACM, April 93, Volume 36, Number 4, page 57: "Information Visualization Using 3D Interactive Animation", by Robertson, Card and Mackinlay

Fortunately, and even at this early stage, Fresco glyphs and OpenGL canvases can be built inside ordinary Xt widgets. We would have no difficulties adding Fresco to the AIPS++ graphics infrastructure if we had the need to.

3. Even at this early date in the (perhaps oversold?) career of Windows NT, X window servers are plentiful. One can presently buy compatibility toolkits for Windows 3.1, allowing Motif programs to compile and run unmodified. There is every reason to believe that this trend will continue. The same can be said of other new Operating Systems.

The best protection against obsolescence is to keep the implementation details out of the public interface. Client code, then, will not need to change very much when circumstances dictate that (for example) a new graphics library or widget kit should be used. We try to hide the implementation in the C++ classes, but cannot always succeed: see, for example, in the next section the appearance of X widgets in the constructors for the ImageDisplay class. The glish event interface, however, can and will be protected from the implementation details of a particular graphics system.

6 Example: an Image Display Component

This component – a work in progress – is primarily intended for the interactive display of AIPS++ Image class objects[5]. It may be called from the glish command line, from any process on the glish bus, or from a C++ program.

The C++ constructor for this component is many times overloaded, reflecting these different contexts in which it will be used:

```
// for use in a non-X program
ImageDisplay (const Image<T>& image);
ImageDisplay (const String& imageFilename);
ImageDisplay (const String& FITSfilename);
ImageDisplay (const Array<T> array);

// for use in a possibly complex X program
ImageDisplay (const Image<T>& image, Widget parent);
ImageDisplay (const String& imageFilename, Widget parent);
ImageDisplay (const String& FITSfilename, Widget parent);
```

[5]An Image is essentially an array of numbers with coordinate information attached. It is natural, then, for the Image Display Component to also be capable of displaying an array, without coordinates, and using integer axis indices in their stead. Furthermore, since AIPS++ provides converters to and from FITS, the Image Display Component can be used to display FITS images as well, performing conversion behind the scenes.

```
ImageDisplay (const Array<T> array, Widget parent);

// for use in an AIPS++ WindowManagingApp program
// (WindowManagingApp is a class which provides a central,
// scrolling menu of all of the current windows, and a file
// selection widget for opening  new windows.)
ImageDisplay (const Image<T>& image, WindowManagingApp& app);
ImageDisplay (const String& imageFilename,  WindowManagingApp& app);
ImageDisplay (const String& FITSfilename,  WindowManagingApp& app);
ImageDisplay (const Array<T> array, WindowManagingApp& app);
```

ImageDisplay operations: once the display is opened, and the image is on view, there are a number of things that the user might wish to do. These may be initiated from the graphical user interface, from the glish command line, or as the result of glish events that come from another glish client.[6]

Here are several scenarios in which the ImageDisplay will be used:

6.1 Scenario One

An astronomer-programmer is experimenting with a new algorithm, writing application code at a fairly high level of abstraction, and wants to see a plane of his image at every step in the algorithm. By constructing an ImageDisplay from either an AIPS++ array

```
ImageDisplay  view1 (array);  // coordinate information not important
```

or from an Image

```
ImageDisplay  view2 (image);  // coordinate information included
```

he would get a window displaying the image, with appropriate controls, with no more effort than the single lines of source code shown above. The controls would include (for example) a colormap manipulator, intensity and position readouts (which change as the mouse pointer is moved across the image), and a close button.

6.2 Scenario Two

An astronomer wishes to examine a spectral cube. Working from the glish command line, the 0th moment of the cube can be displayed with these simple glish commands:

[6]The author has spent the last several weeks experimenting with and absorbing the details of the Karma graphics library. There has not yet been time to organize this exploratory work into a well thought-out public interface. It is safe to say, however, that the interface will include zooming, panning pixel-picking, region marking, annotation, colormap manipulation, and intensity transformations (linear, log, square root, histogram equalization).

```
image := readImage (‘‘cube’’);
moment0 := calculateMoment (0,image);
displayImage (moment0);
```

Upon examining the moment map, the astronomer decides he's interested in a particular spectrum. There are two ways to get the pixel information back to glish; both methods cause an "ImagePixel" event to be sent from the ImageDisplay component back to glish. The first approach is for the astronomer to invoke a "Select Pixel" menu option available on the graphical user interface of the ImageDisplay component. The second technique is to send a "SelectPixel" event from glish to the ImageDisplay component. Both approaches have the same result: the ImageDisplay slips into a special mode, the cursor changes (for instance, from an arrow to a cross-hair); the special mode ends only when the user clicks on the pixel of interest. The ImageDisplay component then sends a glish "ImagePixel" event back to the glish CLI, which will then extract the appropriate z vector from the spectral cube, and send that vector (with coordinate information) off to another component for display.

6.3 Scenario Three

There will no doubt be a demand for an integrated GUI image application, capable of displaying many images each in their own window, and with a master control panel for opening new windows, and controlling those which are already open (see the WindowManagingApp briefly described above). Our strategy is to expose all of the manipulations provided by a component as member functions in the components public interface. Thus zooming, pixel selection, and colormap manipulation will be available as function calls. These function calls can be invoked through buttons and other widgets in the window of the component, and can also be invoked via glish events. It should be clear that a complex, integrated GUI application for image display and manipulation can be created by wiring together a number of components of different type, each making some calls to the public member functions of the others. To illustrate: our integrated application could make selective use of a TableEditor component to examine and modify keywords stored in the image; the application could run an x-y plotter component to display the sequence of values in one particular array keyword; a spectrum could be plotted (as in scenario two, above) when a pixel is selected in the image; a subimage of interest could be marked, for further exploration, or for masking, using the interactive drawing capabilities of the ImageDisplay component.