Lawrence Berkeley National Laboratory
Recent Work

Title
Engineering Insights from a Large, Interactive X Application

Permalink
https://escholarship.org/uc/item/6k6336wz

Author
Courtney, A.A.

Publication Date
1991

**Engineering Insights from a Large, Interactive X Application**

A.A. Courtney

December 1991
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
Engineering Insights from a Large, Interactive X Application

Antony A. Courtney†
Imaging Technologies Group
Lawrence Berkeley Laboratory
University of California
Berkeley, California, 94720

ABSTRACT

This paper describes some of the lessons we learned from implementing a large interactive application for the X Window System. The application, GENIAL, is a tool for analysis of grayscale images of arbitrary size and resolution. However, the software engineering problems we encountered are relevant to any large interactive X application.

The principal lessons we learned from this project are:

- The style of programming enforced by the event-callback mechanism is inadequate for large applications.
- Color applications should share the default colormap to keep the display consistent.
- Code for computation and analysis should be kept separate from the user interface.

December 16, 1991

† Can be reached via USMail at: Lawrence Berkeley Laboratory, MS: 50B-2239, Berkeley, CA 94720. Electronic Mail address is: antony@george.lbl.gov. This work was supported by the Energy Research Division, the Office of Health and Environmental Research, and the Human Genome Program, of the U.S. Department of Energy under Contract No. DE--AC03--76SF0098. Any conclusions or opinions, or implied approval or disapproval of a company or product name are solely those of the authors and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory, or the U.S. Department of Energy.
1. Introduction

GENIAL has grown out of need for an interactive tool for the exploration, quantitative analysis, and annotation of grayscale images of arbitrary size and resolution.

This project began two years ago with the limited goals of providing a front-end to the HIPS [Landy 1984] image processing system. It became readily apparent that something more interactive was needed, and we developed at least three generations of interactive tools to meet the above goals. GENIAL was not a top-down design, nor could it be. Many of the issues addressed in this paper could not be anticipated before the application was put into real use on specific scientific imaging problems. This paper describes our solutions to the software engineering problems presented by this project.

The first issue addressed by this paper is the lack of a centralized control mechanism in X toolkits. Present toolkits force the application to relinquish control to a toolkit supplied dispatch loop. The resulting programming style places exordinant demands on the application programmer, who must go to great lengths to ensure robust program operation and user interface consistency. This problem is easily overcome in small applications. In large, production scale applications, however, this problem becomes unmanageable.

The issue of colormap management is relevant to any application which requires color. A relatively high number of color applications cause colormap “flash”: that is, they temporarily destroy the contents of other windows by use of a private colormap. This is more often due to poor documentation of colormap handling than actual need for a private colormap.

The last issue addressed in this paper is a simple software layering concept that has proven very useful in code management for a large application. In this example, the code for different image analysis functions is kept in separate modules. The user interface is combined with a set of basic services for image management, which together provide an underlying framework for interactive imaging. This underlying framework is augmented by a set of autonomous analysis functions called through a well defined interface. This separation between user interface and analysis functions has proven an effective approach for constructing applications which are both interactive and extensible.

2. User’s Model

While a description of GENIAL’s usage is outside the scope of this paper (for a description of usage and analysis functions, see [Johnston 1990]), the following short description is necessary for a discussion of its features.

Figure 1 shows a screendump of GENIAL operating on a typical image. The window titled “GENIAL Control” in the upper left portion of the screen is the basic control panel for GENIAL. Immediately below the control panel is a scorable display image. In the right half of the screen lie three windows which are the displays from three different analytical functions. Each of these display windows corresponds with one of the numbered “regions” shown in the image window. A region is simply a portion of the image as defined by some graphics primitive (e.g., spline, line, polygon, etc.). The number in the title of each display window indicates the corresponding region in the image display window. For example, the display window titled “Trace: 1” corresponds with the spline labelled “1” in the display image.

The important features of the control panel for GENIAL are as follows: The top line (labelled “Image:”) contains buttons which allow the user to load images, change display options (i.e., perform contrast enhancement on the colormap), and traverse through the frames of a multiple frame image. The next line (labelled “Function:”) has a menu which allows the user to select an analytical or annotation function. As shown here, the presently selected function is “Zoom”. The “eval”, “clear” and “options” buttons allow the user to evaluate the current function, clear the current function, and select a set of options for how the current function will be evaluated. The line labelled “log:” allows the user to select whether or not to automatically store each function in a log of all operations. The next line, “Regions:” allows the user to select the type of region to use as the current region. The present region type is “Box” because the present function, “Zoom”, only works on rectangular portions (“Box”es) of an image. The bottom portion of the control panel is split into a separate area containing “Status:” information. The top lines of the status area display directions to the user. The bottom line
of the status area displays the coordinates and gray value in the original image when the cursor is in the image window. The right side of this line displays information about the geometry and gray scale space of the original image. In this example, the image on the screen is a display representation of a 16 bit ("short") image with dimensions 750x764 and gray values in the range 33-1886.

Each of the display windows is itself interactive. As the user moves in the trace window, a red vertical bar tracks the cursor motion and a red cross corresponding to that point is displayed in the image window. In the histogram window, the user may select portions of the graph corresponding to particular gray value ranges and then select a color from the "Histogram Ctrl" window for display in the region of the original image. In the zoom window, the user may use the mouse to move the zoomed region.

As shown by its log ID number, each of these function invocations has an entry in a permanent log of all operations. This log may be stored in a file along with the original image, and each function will be re-invoked when the image and log are loaded again.

3. Software Architecture

3.1. User Interface

The user interface issues are discussed first because the structure of any X application depends on the user interface. Applications for the X Window System generally use a toolkit to provide the standard elements (widgets) of the user interface (i.e., buttons, scrollbars, panels, sliders, etc.) An application uses the services of a toolkit by creating a widget and attaching a callback or notify procedure to it. A callback is simply a function called when some event occurs (e.g., a button press) in the widget which it is attached to. To manage these user interface objects, the main program must relinquish control to a dispatch loop provided by the toolkit. This dispatch loop reads events from the connection to the X Server, demultiplexes them, decides which objects should be notified of the event, and calls the appropriate callback procedures for those objects. This approach raises a number of problems:

Since control is relinquished to the dispatch loop provided by the toolkit, the application does not directly control when callback procedures are called. For example, consider a simple program which does pixel replication to zoom in on a portion of an image. There is a definite order of operations to such a program. The image is obviously read in from a file before the pixel replication occurs. In conventional applications, this order is imposed by the centralized control of the program, which presumably calls a function (say read_image()) to read in the image before calling the pixel replication function (say pix_rep()) which depends on the data structure for the screen image. The order of operations is easy to implement in a conventional programming environment because input usually comes from only one place (i.e., the standard input or a keyboard). For interactive programs with such an input model (e.g., an interactive debugger or a calculator), a parser provides a centralized control mechanism: The parser maintains the ‘state’ of the program, reads input from a single place, verifies that the input is valid based on the present state, and calls a certain set of routines in a definite order based on the input. The structure of such a program is illustrated in figure 2.
For this example, load and zoom functions would probably be implemented as buttons on a control panel. The callback for the the "load" button would call read_image(), and the callback for the "zoom" button would call pix_rep(). However, since control is relinquished to the toolkit's dispatch loop, the application cannot guarantee that the callbacks for these buttons will be called in the necessary order.

There are two common solutions to this problem. The first solution is to force every callback to check any data structures it depends on for validity and consistency before taking action. (This could be implemented with semaphores, though that is not usually necessary.) The other solution is to use a property of certain toolkits which allows elements of the interface to be "turned off" when the action they represent is not valid. When the dispatch loop of the toolkit receives events for widgets which are turned off, it discards the event and does not invoke the callback procedure associated with the widget. The toolkit usually also "grays out" user interface elements which are turned off, to give the user a visual cue that the element represents an invalid selection. In this example, the program would have to turn off the zoom button before relinquishing control to the dispatch loop. When the file button is pressed, the callback for the file button would turn the zoom button back on. If the file were later closed, the routines which closed the file would have to ensure that the zoom button is turned back off. The structure of such a program is illustrated in figure 3.
While these mechanisms might be adequate for simple applications with simple processing requirements and a relatively low degree of interaction, they have proven inadequate for large, highly interactive applications such as GENIAL. These solutions are really work-arounds for lack of a centralized control mechanism. The work-arounds and problems they create accumulate quickly:

- Every callback for every user interface element must know about all other user interface elements which depend on data structures it modifies, and must turn those elements "on" or "off" appropriately. This effectively defeats the purpose of functional programming -- a function callback for loading a file should load a file and little else.

- Adding new widgets to the user interface becomes difficult and time consuming because code must be added to turn the widget on and off in every function that modifies every data structure on which the widget depends. This system is also highly error prone -- it is very easy to forget to turn off certain widgets when some data structure they depend upon is freed or otherwise invalidated.

- Different widgets may have different implications depending on the state of the program. In the example given above, if an image is already loaded, the load button would probably do some additional processing to free the image memory presently allocated. And since pix_rep() and any associated functions depend on this data structure, the application should notify them that the data structure is no longer valid.

These factors made GENIAL's user interface inconsistent and non-robust in previous implementations. For this latest version, we have implemented a small centralized control mechanism. This control mechanism is implemented as a single function, state_dispatch(), which accepts "tokens" representing each of the user interface events. Each of these tokens is passed to a stack of one or more functions (termed a "dispatch" routine) for processing. Each dispatch routine may either accept a token for processing or pass the token on to the next dispatch routine on the stack. This input scheme is analogous to the input scheme in the STREAM i/o system. [Ritchie 1984]

The main dispatch routine, always at the bottom of the stack of dispatch routines, is a small state machine, with state transitions triggered by user interface events. This routine processes most of the input in GENIAL. There is only a limited set of events that are deemed "state-dependent". Based on these events and the present state, the main dispatch routine will take some clearly defined action and compute the next state. This small amount of centralized control has led to a user interface which is both consistent and robust. The system is consistent because the actions of the widgets are clearly defined in the state transition diagram. It is robust because the centralized control mechanism ensures that the system always returns to a known state. The small centralized control mechanism has turned out to be easy to maintain, debug and modify because all dependencies and state transitions are localized. The resulting structure of the application is shown in figure 4.
Table 1. State Transition Diagram

<table>
<thead>
<tr>
<th>State</th>
<th>Events</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE NOT LOADED</td>
<td>&lt;load&gt; button</td>
<td>load image; initialize current function module; goto IMAGE LOADED;</td>
</tr>
<tr>
<td></td>
<td>&lt;quit&gt; button</td>
<td>goto SHUTDOWN;</td>
</tr>
<tr>
<td>IMAGE LOADED</td>
<td>&lt;load&gt; button</td>
<td>clear all log entries; reset UI; load image; initialize current function module; goto IMAGE LOADED;</td>
</tr>
<tr>
<td></td>
<td>&lt;clear&gt; button</td>
<td>if (current function!=NULL) clear current function;</td>
</tr>
<tr>
<td></td>
<td>&lt;eval&gt; button</td>
<td>if (region type==NONE) { eval current function; reset UI; initialize current function module; }</td>
</tr>
<tr>
<td></td>
<td>&lt;forw&gt; button</td>
<td>advance frame;</td>
</tr>
<tr>
<td></td>
<td>&lt;back&gt; button</td>
<td>reverse frame;</td>
</tr>
<tr>
<td></td>
<td>&lt;function&gt; menu</td>
<td>reset UI; select new function module; initialize current function module;</td>
</tr>
<tr>
<td></td>
<td>&lt;image.button1&gt;</td>
<td>add point to current point list; goto REGION SELECT;</td>
</tr>
<tr>
<td>REGION SELECT</td>
<td>&lt;load&gt; button</td>
<td>clear current point list; clear log; reset UI; load image; initialize current function module; goto IMAGE LOADED;</td>
</tr>
<tr>
<td></td>
<td>&lt;quit&gt; button</td>
<td>goto SHUTDOWN;</td>
</tr>
<tr>
<td></td>
<td>&lt;eval&gt; button</td>
<td>make current region from current point list; evaluate current function using current region; add current function to log; reset UI; initialize current function module; goto IMAGE LOADED;</td>
</tr>
<tr>
<td></td>
<td>&lt;clear&gt; button</td>
<td>clear all points from current point list; goto IMAGE LOADED;</td>
</tr>
<tr>
<td></td>
<td>&lt;image.button1&gt;</td>
<td>add point to current point list</td>
</tr>
<tr>
<td></td>
<td>&lt;image.button2&gt;</td>
<td>move nearest entry in current point list to new position</td>
</tr>
<tr>
<td></td>
<td>&lt;image.button3&gt;</td>
<td>delete nearest entry in current point list; if (current point list is empty) goto IMAGE LOADED;</td>
</tr>
</tbody>
</table>

The state transition diagram is shown in table 1. There is only one general entry point to the control mechanism, state_dispatch(), which has the following form:
/* state dispatch() -- centralized event handling mechanism.  token represents
* the particular user interface event as defined in sm.h, and arg is a pointer
* to an optional argument.
* Returns: the state after processing the event, also defined in sm.h.
*/

int
state_dispatch(token,arg)

    int    token; /* defined in sm.h */
    caddr_t arg; /* pointer to an optional argument */

All of the event callbacks for the state-dependent widgets simply call state_dispatch() to do their event processing. For example, the following is the callback for the load button:

/***/
* Notify callback function for 'load'.
*/
void
load_proc(item, event)
    Panel_item item;
    Event *event;
{
    /* get the file name from the filename text field */
    char *fname=(char *) xv_get(file_win->l_fname, PANEL_VALUE);

    /* call state_dispatch(), passing the file name as an argument. If
     * successful, iconify the file window */
    if (state_dispatch(LOAD, (caddr_t) fname)==IMG_LOADED) {
        xv_set(file_win->window1,
              XV_SHOW, FALSE,
              FRAME_CLOSED, TRUE,
              NULL);
    }
}

Implementing the state transition diagram is relatively straightforward. And the requirements of
the control mechanism directly indicate the entry points needed for the basic services. The state transi­
tion diagram is implemented as a simple two level switch() statement. The outer level does a switch() based on the present state, and the inner switch() is based on the token argument. The actions are
coded directly from the state transition diagram.

Internal to the main dispatch routine, the state is actually changed by calling new_state():

    new_state(ns)
    int ns;
    {
        state=ns;
        panel_state(state);
    }

panel_state() simply turns the appropriate state-dependent user interface elements ‘on’ or ‘off’ based
on the state transition diagram and the present state. So, for example, panel_state() would turn off
the load button if it were called with REG_SEL as an argument.
The "stack" of dispatch routines is used when the programmer wishes to read user interface events directly. For example, the user may prefer to draw lines by clicking the mouse on a start point and "dragging" the mouse to the end point, with a preview line following the mouse. This is implemented by pushing another routine onto the stack of dispatch routines. This routine simply captures all <image.button1> events until the user releases the mouse button. When the mouse button is released, this routine simply pops itself off the dispatch stack. This is the clearest way of expressing what the programmer is trying to do: The program just temporarily captures data from the single input channel.

This centralized control mechanism has proven an effective way to handle the demands of a complex user interface. Actually mapping out the user interface in a state transition diagram proved a worthwhile exercise concisely defining the actions bound to each user interface element. A little time spent reducing the number of states and developing consistent behavior for certain user interface elements went a long way in simplifying the user interface. Further, the state transition diagram maps directly to the code for state_dispatch() and the entry points for other services.

In defense of the event callback mechanism, it could be argued that state could be provided explicitly, and the set of actions associated with each event in state_dispatch() could be coded in the callback based on a check of the current state. Indeed, many of the buttons which are not "state-dependent" do exactly that by calling get_state(), which returns the current state. These are really just two sides of the same coin: A centralized control mechanism is necessary in either case. In practice, coding actions for the most important events in state_dispatch() has proven easier to maintain than explicit state checks in every event callback procedure.

This centralized control mechanism is technically just another work-around for the lack of such a mechanism in available X toolkits. But the state mechanism effectively serves the same purpose. Conventional applications maintain a current status, read input from one place and based on the input call upon a set of services in a definite order. Since the only entry point for input to GENIAL is through state_dispatch(), the structure of the rest of GENIAL is conceptually the same.

3.2. Colormap / XImage management

GENIAL presently runs on 8-bit PsuedoColor framebuffers. However, GENIAL can deal with images which have 8, 16 or 32 bits of depth. GENIAL does this by mapping the range of gray levels in the original image into the number of colormap entries available for display.

The class of framebuffers that GENIAL runs on have 8 bits for each pixel value in the display. Each of these values is an index into a 24-bit wide look-up table (LUT). Each 24-bit entry in this LUT is an RGB triple composed of an 8-bit intensity value for each of the Red, Green and Blue phosphors. A grayscale colormap consists of such a LUT where the intensity values for the red, green and blue members of each entry are all equal to one another, and the entries cover achromatic gray from black to white. Such a colormap is illustrated in figure 5.
The framebuffer we use has only one hardware LUT for all pixels on the display. However, X allows clients to have a virtual colormap associated with each window. A virtual colormap is simply another 8 to 24 bit color LUT private to a particular window. The window manager is responsible for installing virtual colormaps into the hardware colormap when it is appropriate (e.g., when that window has the focus). Many imaging and visualization programs use virtual colormaps in order to obtain the maximum number of possible colors at any time. However, this can cause windows to temporarily disappear from the display. For example, suppose a terminal emulator has a virtual colormap which maps pixel value 0 to an RGB triple for white (0xff, 0xff, 0xff) and maps pixel value 1 to an RGB triple for black (0x00, 0x00, 0x00). And suppose the terminal emulator fills the background of its window with pixel 0 (white in the terminal emulator's virtual colormap) and draws the foreground text in pixel value 1 (black in the terminal emulator's virtual colormap). Now suppose the user moves into the window for an imaging application with the virtual colormap shown in figure 5. The window manager will load this virtual colormap into the hardware colormap, and all pixels on the display with value 0 will be mapped to black and all pixels on the display with value 1 will be mapped to an extremely dark gray (dark enough to appear black). Since our terminal emulator window uses these two pixels for its foreground and background, the terminal emulator window will turn entirely black!

For most users, this is an entirely unacceptable solution. Therefore, we use a property of the root window, the Default Colormap, and use XAllocColorCellsO to allocate some number of color cells for use by GENIAL. XAllocColorCellsO returns an array of indices into the default colormap which may be changed by the client by calling XStoreColorsO with an array of RGB triples. The set of entries in the default colormap returned by XAllocColorCellsO may only be changed by the client making the call. The cells are guaranteed private to our application. However, the number of available colormap entries is substantially less than the range of data values in the original image. This turns out not to be a real problem because our empirical observations indicate that the human eye can not really distinguish more than 64 discrete levels of gray.

The array of RGB triples used in calls to XStoreColorsO comprises an output look-up table (o-LUT). The values in the o-LUT are loaded to cover achromatic gray from black to white. When an image is first loaded, GENIAL does a linear scan through the dataset for the extrema. The min and max from this scan are stored in an image data structure along with the image dataset. GENIAL builds data for an XImage structure by doing another linear scan over the data which passes each data element through a function which performs a linear mapping from a data value in the original image to an index in the o-LUT. This o-LUT index is then converted to a pixel value used in the XImage data. This pixel value (and therefore the XImage data) is immutable for the remainder of time GENIAL is running. Any contrast enhancement is performed by direct changes to the private colormap entries. The source which implements colormaps follows:

```c
/* ... */
#define NCOLORS 64
/* ... */
/* colors[] is an array of pixel values (indexes into the Default Colormap). The
 * particular color associated with each of these pixel values may be changed.
 * This array is first initialized in the call to XAllocColorCellsO in the
 * init_display() routine. It is later passed to build_colormap() as the
 * pixels[] argument.
```
unsigned long colors[NCOLORS];

/*
  * the following are from init_display(). Called once during
  * initialization
  */

/* ... */
/* get the default colormap of the display */
cmap = DefaultColormap(display, DefaultScreen(display));
/* try to allocate a small number of color cells for read/write
  access */
if (XAllocColorCells(display, cmap, False, NULL, 0, colors, NCOLORS) == 0) {
    printf("error trying to alloc color cells.\n");
    exit(0);
}

/*
  * the following routine builds an output look-up table (o-LUT) which covers
  * achromatic gray from black to white in the indexes specified by the pixels
  * argument, with the number of colors specified by nc.
  */

build_colormap(map, pixels, nc)

  Colormap map;
  unsigned long *pixels;
  int nc;

  {
    /* delta_o is the range of intensity values divided by the number of
     * color cells. An index * delta_o gives the corresponding intensity
     * value for building a colormap.
     */
    double delta_o=(double) 255/nc;
    unsigned short v;
    int i;
    XColor *cvec;

    /* create a vector of XColor structures to be filled in with gray
     * values evenly spread between black and white, and passed
     * to XStoreColors to change the colormap.
     */
    if ((cvec=(XColor *) malloc(nc*sizeof(XColor)))==NULL) {
      perror("malloc");
      exit(0);
    }

    /* fill in the cvec array correctly */
    for (i=0; i < nc; i++) {
      cvec[i].pixel=pixels[i];
      v=irint((double) delta_o * i);
      /* Xlib expects the intensity byte to be in the high order...*/
  }
8 bits, so do a left shift. */

v=(v << 8);
cvec[i].red=v;
cvec[i].green=v;
cvec[i].blue=v;
cvec[i].flags=DoRed | DoGreen | DoBlue;
}

/* use XStoreColors to store out the cvec array into our read/write *
cells in the colormap. */
XStoreColors(display, map, cvec, nc);
/* free the cvec vector */
free(cvec);

/* to build an XImage from image data, do a linear conversion of the range *
of data values in the original dataset into the number of color cells in *
o-LUT. Then use this conversion factor as an index into the colors[] *
array to retrieve the corresponding pixel value. */

/* the following is excerpted from mk_x_img(). image is a data structure for *
the original image data. */

/* ... */
XImage *xim;
unsigned char *rast;
unsigned long v;
int u, width, height, x, y;
/* delta_u is the number of colors divided by the range of values in *
the dataset of the original image. provides a linear mapping into *
the colors[] pixel index for creation of the XImage data. */
double delta_u=(double) (NCOLORS-1)/(image->maxv-image->minv);

/* ... */
width=image->width;
height=image->height;
/* allocate space for the XImage data and create an XImage structure */
if (rast=(unsigned char *) malloc(width*height))==NULL) {
    perror("malloc");
    exit(0);
}

xim=XCreateImage(display, (Visual *) winv->visual, winv->depth,
ZPixmap, 0, rast, width, height, 8, 0);
/* ... */
for (y=0; y < height; y++) {
    for (x=0; x < width; x++) {
        /* dval returns the data value from an image dataset *
        * at a particular (x,y) coordinate. */
        v=dval(x,y,image);
        /* convert to an offset into the range of values in *
        * the original image and multiply that by delta_u to *
        * get an index into the colors[] pixel table... */
From this point, all contrast enhancement occurs by changing the color cells in our private region of the colormap and making another call to XStoreColors(). This is presently implemented by allowing the user to do gamma correction to the colormap, which maps indexes to intensity values using a small exponential curve rather than a linear mapping function.

3.3. Regions

When a user performs some analytical function in GENIAL, he usually specifies a region within which the function operates. The user specifies the region by selecting a set of points which unambiguously specify the region (i.e., the two corners of a rectangle). The points selected by the user are kept separate from the rest of the points which make up the region.

The points which the user selected are stored in a doubly linked list called a point list, or plist. A cross is displayed over the image to give the user a visual cue of what points have been selected. The backing store of the image underneath this cross is stored in a cross backing store, or cbstore data structure. A plist entry simply stores the coordinates of the selected point and the cbstore structure for the point.

GENIAL maintains display lists for the graphical objects (lines, splines, text regions, etc.) which lie over an image. Display lists, or dlists, are costly to create and maintain. They would also appear redundant given the functionality of Xlib, which provides a very baroque set of functions for drawing graphical objects and nearly infinite drawing options through modification of the graphics context. There are two basic reasons for using display lists: The first is that the graphical objects represented by a display list are placed over an image on the display, and GENIAL must be able to refresh the pixels underneath the graphical object on the fly. This could be accomplished through overlay planes. However, overlay planes double the amount of colormap space required, and constrain the layout of the colormap. As discussed in section 3.2, colormap space is a precious resource and should be used sparingly. The second reason for using display lists is that many functions need access to the pixels which comprise lines, splines, etc. Xlib does not provide access to this information, so the scan conversion has to be done at some point. Therefore, we decided to implement a general display list mechanism and a set of routines for creating these graphical objects. Display lists, or dlists, are simply doubly linked lists of dynamically sized, dynamically allocated vectors of points.

Regions represent a direct mapping between a plist and a dlist. The mapping is determined by some function (specified by the region type) which does scan conversion from the input points in the plist and creates a dlist. Such a mapping is illustrated in figure 6. (Also note the data structure marked strbs. An strbs, or string backing store, is just another simple data structure for maintaining the backing store of a string.) The form of a region, then, is simply:

```c
/* structure of a region */
struct region {
    int r_type; /* type of region */
    int r_flags; /* special region modes (e.g. least-squares) */
    struct plist *r_plist; /* points which the user supplied */
    struct dlist *r_dlist; /* points which form the region */
    struct strbs *r_sbs; /* strbs is a structure which holds information on the backing store for a string. */
    /* for region type AN_TEXT */
};
```
The region management routines depend upon the current point list maintained by the point list management routines. The region management routines provide simple functions for setting the region type, which determines the function used for creation of a dlist from the current pliSl.

After a region type has been selected and the user selects points for the point list, the next step is to create a region structure based on the current point list and the region type. A single entry point, interp_reg(), is provided. It creates an empty region, flushes the current point list into the region, calls the appropriate mapping function based on the current region type, and returns a pointer to the newly created region. In addition, the usual functions are provided for drawing and refreshing regions, along with free_reg() which frees all the storage associated with a region:

If managed correctly, the display image with region overlays should always reflect the state of the internal data structures. Consistent management routines have proven to be the backbone of good data structure management. This is accomplished by two simple functions for each type of data structure: draw_xxx() and ref_xxx(), both of which take an argument of type struct xxx *. draw_xxx() draws data structure xxx in the standout color on the display. ref_xxx() replaces the original image pixels under data structure xxx. So, for example, if draw_region() is called with a region argument, it will call upon draw_plist(), draw_dlist, and draw_sbs() in turn. draw_plist() will then call upon draw_cbstore(), etc.
3.4. Functional Interface

Each analytical and annotation function is implemented as an autonomous module stored in a separate source file. Control is passed to each module by calling one of the following functions implemented by each function module:

`xxx_init()`
Function called when this module becomes the current function (i.e., when the function is chosen from the “function:” menu). `xxx_init()` should initialize any data structures declared statically in the module, call `reg_setdom()` to set the appropriate region domain, and perhaps makes other changes to the user interface. `xxx_init()` will always be called before any of the following functions are called.

`xxx_eval()`
Function called when the user has selected a region and hit the “eval” button. `curfunc` contains valid information describing the current region for evaluation, log ID, etc. `xxx_eval()` should evaluate the function, perhaps printing some output or illustrating the results of the evaluation in another window (e.g., a graph window for a histogram).

`xxx_clear()`
`curfunc` contains a function which has already been evaluated (through a previous call to `xxx_eval()`), but which should now be cleared away. `xxx_clear()` should free any static data it stored on behalf of this function invocation, free any windows containing output data for this function, etc.

`xxx_opt()`
Function called when the user selects the “options” button on the User Interface. This allows analytical function to pop-up auxiliary control panels for setting options for the function. Many modules do not provide an `xxx_opt()` function because they do not provide an options window.

`xxx_reset()`
Function requesting the module “clean up” any changes it made to the user interface. For example, a module might iconify its options window when `xxx_reset()` is called. Many modules do not provide an `xxx_reset()` function because they do not modify the user interface in any way.

The function interface is analogous to the interface which device drivers present to an Operating System. To implement this interface, there is a file, `functions/conf.c`, which contains an array, `fxnsw[]`, of the following `fxnsw` structure, along with functions to access this array:

```c
/*
 * conf.h -- function configuration definitions
 *
 */

struct fxnsw {
    int (*f_init)();
    int (*f_eval)();
    int (*f_clear)();
    int (*f_reset)();
    int (*f_opt)();
};
/* ... in functions/conf.c: */
extern int trace_init(), trace_eval(), trace_clear(), trace_reset(),
         trace_opt();

extern int histo_init(), histo_eval(), histo_clear(), histo_reset(),
         histo_opt();

struct fxnsw fxnsw[] =
{
```
The following functions provide access to fxnsw[] from the rest of GENIAL:

```c
int fxn_select(fid)
int fid; /* opcode of current function. Also index into fxnsw[]. */
```

```c
int fxn_init()
```

```c
int fxn_eval()
```

```c
int fxn_clear()
```

```c
int fxn_reset()
```

```c
int fxn_opts()
```

When the present function is selected (i.e., through the "function:" menu), fxn_select() is called. fxn_select() simply sets a statically defined integer, cfunc, to be the index into fxnsw[] of the current function. It is expected that fxn_init() will be called immediately after fxn_select(). Selecting the "eval" or "clear" buttons on the user interface would cause fxn_eval() or fxn_clear() to be called. These routines simply call the corresponding function in fxnsw[]. fxn_reset() and fxn_opts() are also called at the appropriate times based on user events and the corresponding actions in the state transition diagram. Both state_dispatch() and read_log() use these routines, which provide the only interface to function modules.

All functions must be re-entrant. That is, the user should be able to have an arbitrary number of separate invocations of any given function active at any time. This becomes somewhat complicated for functions such as histograms which have an independent display window that the user may interact with. Presently, each function must handle this problem independently. Using histograms as an example, the histogram module defines a structure, struct hcontext that contains all the state associated with any invocation of a histogram (such as the display window, graph information, etc.). There is a dynamically sized array of such structures declared statically in the histogram module. When histo_eval() is called, a new hcontext entry is created and set with the appropriate values. Then, when the callback for the histogram display window is called, it retrieves the appropriate hcontext entry by calling hist_by_win() with the window passed to the callback procedure as an argument. hist_by_win() returns the hcontext entry with the corresponding display window. The event callback can then respond to the event appropriately since the hcontext entry contains all information associated with that window. Similarly, histo_clear() removes the appropriate entry from the hcontext table.

Function modules also have access to all of the imaging and display services provided by GENIAL, along with all of Xlib and XView. The display services are particularly useful to functions such as histograms or traces which couple actions in their display window with the corresponding pixel in the display image. At present there is also a limited set of services to aid in the development of interactive user interfaces for analytical functions, in order to reduce the high overhead of using Xlib.
The present functions simply do decade and half-decade scaling for easy display of graphs.

4. Present Status

At present, all the functionality outlined in this paper is implemented and running. There are, technically, a few minor problems with some of the actions in the state transition mechanism. The problem centers around having only one image buffer, which must be flushed before loading a new image. This is a minor problem. Fortunately, such problems can be found and corrected easily by borrowing standard techniques for debugging finite state machines from digital logic design.

5. Conclusions

Practice has shown that a small, centralized control mechanism goes a long way in building a user interface which is both consistent and robust. An interface which allows events to be "read" from a clear set of entry points as in [Pike 1989] would probably serve the needs of large interactive applications more effectively than the current event callback mechanism. However, it is not likely that X toolkits will support such an input paradigm any time in the near future.

In addition, it would be useful if X, Xlib and the available toolkits could provide better tools for colormap handling and access to the pixels used in graphics operations. For this latter problem, a solution is not very likely since rendering of graphics primitives is done in the X server, possibly in hardware, and might not be easily available. In addition, the required extensions to the X protocol for the server to return this information are probably too costly to implement.

We have also found that defining a clean interface between function modules and the underlying framework of GENIAL helps reduce the high overhead of implementing interactive X applications. As we gain more experience with the types of interaction mechanisms that function modules really need, we will be able to provide better tools which aid in the development of function modules. The separation between function modules and an underlying structure could be a useful paradigm for building other interactive X applications. For example, it might be useful to build a drawing program which allows programmers to build their own drawing tools. Or perhaps a window system based debugger could call upon different "tools" for debugging (e.g., traversing a linked list) implemented as separate modules.

6. Future Work

It would also be useful if new analysis tools could be added to the system dynamically. At present, all function modules are linked together with GENIAL and run as a single process. This obviously makes it difficult to add new functions to the system, and if a function module has a bug it can bring down the entire application. For these reasons, it would be useful if functions could be implemented as separate processes, which communicate with GENIAL through shared memory or pipes. Along these same lines, it would also be useful if GENIAL provided some rudimentary functions for calling external image processing filters, such as those provided by the HIPS [Landy 1984] image processing package.

Also under consideration is tying GENIAL to an embedded command language such as John Ousterhout's Tcl. [Ousterhout 1990] Since the analysis functions provided by GENIAL are relatively straightforward, they could be coded quite easily in an interpreted language. One possible advantage of an embedded command language is that the problems of making the tools re-entrant could be handled by having multiple interpreters running simultaneously. In addition, an interpreted command language provides for great flexibility in how the different analysis tools are connected to one another.

7. Acknowledgements

This work is simply an outgrowth of the original research on interactive image analysis by Bill Johnston and Max Rible of LBL's Imaging Technologies Group. Bill Johnston provided the initial design of the analysis functions. Max Rible did several of the early implementations and designed the log header format. The research described in this paper would never had been possible without their efforts.
The LBL Human Genome Center provided the analysis requirements for use of this application on real scientific imaging problems. In addition, Sandro Wallach of RPI and Steve Parker of Unisoft provided valuable feedback on the user interface issues discussed here.

8. References
Figure 1
single input point  
(i.e. open descriptor to keyboard device)

centralized control mechanism  
(i.e. a parser)

ordered sequence of actions

figure 2. control flow in conventional programs
connection to X server

toolkit supplied dispatch loop

unordered calls to:
programmer-supplied callback procedures
locking mechanisms for data structure interdependencies

figure 3. control flow in X programs
connection to X server

toolkit supplied dispatch loop.

unordered calls to:
"stub" callback procedures

single entry point
state_dispatch()

centralized control mechanism
(small state machine)

ordered sequence of actions

figure 4. control flow in GENIAL
<table>
<thead>
<tr>
<th>index</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
</tr>
<tr>
<td>2</td>
<td>0x02</td>
<td>0x02</td>
<td>0x02</td>
</tr>
<tr>
<td>3</td>
<td>0x03</td>
<td>0x03</td>
<td>0x03</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>253</td>
<td>0xfd</td>
<td>0xfd</td>
<td>0xfd</td>
</tr>
<tr>
<td>254</td>
<td>0xFE</td>
<td>0xFE</td>
<td>0xFE</td>
</tr>
<tr>
<td>255</td>
<td>0xFF</td>
<td>0xFF</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

black \} dark gray
\} light gray
white

figure 5. grayscale colormap
struct cbstore:

struct dlist:

struct plist:

prey, pve

Ie

pt (x,y)

prev, next

cb

prev, next

cb

prev, next

cb

prev, next

pt (x,y)

figure 6. Data structures for regions