Architecture and Design of the MGR Window System.

Stephen A. Uhler

Bellcore

ABSTRACT

MGR is a window system for UNIX, currently running on a Sun Workstation. MGR features overlapped, asynchronous windows, and an applications interface that is both machine and network independent. All existing Unix applications can run unchanged under MGR, either on a local, or a remote host. New applications may be written to take full advantage of the graphical capabilities of the bit-mapped display. This paper discusses the implementation and strategy of the window system and demonstrates its applicability in a software development environment consisting of different kinds of Unix machines.

Introduction

What is MGR

MGR (**m**ana**g**e**r**) is a window system for UNIX[1] that runs on Sun Microsystems Workstations. MGR manages asynchronous updates of overlapping windows and provides application support for a heterogeneous network environment with different types of computers connected by various communications media. The application interface allows applications (called client programs) to be written in a variety of programming languages, and run on different operating systems. The client program can take full advantage of the windowing capabilities regardless of the type of connection to the workstation running MGR.

Client programs communicate with MGR via *pseudo-terminals*† over a reliable byte stream. Each client program can create and manipulate one or more windows on the display, with commands and data to the various windows multiplexed over the same connection. MGR provides ASCII terminal emulation and takes responsibility for maintaining the integrity of window contents when parts of windows become hidden by other windows then subsequently uncovered. This permits naive applications to work without modification by providing a default environment that appears to be an ordinary terminal.

In addition to terminal emulation, MGR provides each client window with a variety of features including graphics primitives such as line and circle drawing; facilities for manipulating bitmaps, fonts, icons, and pop-up menus; commands to reshape and position windows; and a message passing facility enabling client programs to rendezvous and exchange messages. MGR also provides for both synchronous and asynchronous event notification as well as a user initiated *cut-and-paste* function that permits a user to sweep out and copy text from any window into a global buffer, then paste it into any other window.

Background of MGR

MGR implementation began in 1984. At that time, there were no window managers available for Sun workstations that functioned on heterogeneous networks of computers, with client programs that used windowing facilities while executing as separate processes on remote hosts. In addition, the computing environment required applications to run on a variety of operating system platforms, connected over different types of interprocess communications networks. The system platforms included

† A *pseudo-terminal* is a pair of devices that together function as a bidirectional pipe, where one side, the slave end has terminal semantics, and the other, or master side, has several control functions as well as the normal read and write capabilities.

both UNIX System V and BSD 4.2, as well as several experimental in-house variants. The computers were connected, not only by ethernet™ but by serial lines and circuit switches as well, with plans for fiber optic token ring networks and perhaps other exotic interconnects to come. It was necessary to carefully separate the networking software from the window management, so arbitrary interconnects could be accessed with no changes required to MGR.

MGR was intended to be a means of providing an application platform in a distributed computing environment. It was appropriate to require compatibility with existing programs, eliminating the need to re-write the existing software base. More to the point, `vi`, `mail`, the `shell`, and the other existing UNIX applications needed to work unchanged, not only because there was neither the time nor resources to rewrite them all, but also to ease the transition for the user community, currently working with ASCII terminal based software.

MGR Architectural Overview

MGR consists of a single user level process. It requires no kernel modifications or special device drivers. Like any other user process it is started from the shell, and may be suspended, restarted, or killed by the user. Figure 1 shows a pictorial representation of the MGR process and its connections to system resources and client programs.

MGR was designed to be portable to a variety of computer systems. To date it runs on Sun Workstations (monochrome and color), Apple Macintoshes and UNIX System V based Dune [2] distributed computers. It is divided into two components, the hardware dependent component, and the hardware-independent, or portable component. The hardware dependent component deals entirely with the physical devices, the mouse (or other locator device), keyboard, and the display. The portable component, the bulk of MGR, contains no hardware dependencies; it calls the hardware dependent routines as required.

The portable component of MGR needs to interact with operating system facilities, such as reading and writing files and, by necessity, contains some operating system dependencies. As with the hardware interfaces these operating system dependencies are well contained and isolated to permit modification for different system interface requirements. This has allowed MGR to be ported to other operating system interfaces quickly, such as a Macintosh-plus in two weeks, and to Dune in a single day.

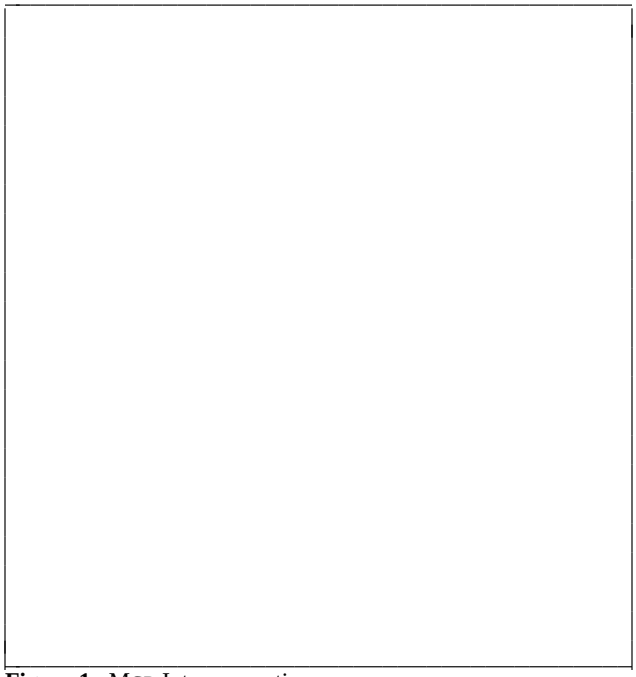

Figure 1. MGR Interconnections

MGR contains five functional components:

(1) **Process Creation and Management.** Process creation deals with the creation of processes (client programs) to run in a window, and the instantiation of the connections between client programs and MGR. Process management deals with the maintenance of those connections over the life of the client programs and their windows.

(2) **I/O Scheduling.** I/O scheduling manages client program output to windows as well as processing and scheduling of keyboard and mouse input.

(3) **Window Management.** The window management component starts with a blank display and creates the *illusion* of overlapping windows. It manages reshaping and repositioning windows, maintaining the display integrity, drawing text and graphics, menus, and tracking the mouse. Everything the user sees on the display is handled by the window management function.

(4) **Command Processing.** The command processing component reads commands and data from a process, interprets the commands, and

issues the appropriate calls to the window management functions. It also provides feedback for the I/O scheduler to maintain smooth operation.

(5) **User Interface.** The user interface takes input from the mouse and keyboard, and processes the system commands issued by the user from these devices.

In addition to those components mentioned above, there is startup and initialization code that reads and executes the startup files, decodes the command line arguments and initializes the display, keyboard, and mouse.

Data Structures for Windows

MGR uses a doubly linked list of objects, called window descriptors, to represent the spatial ordering of windows on the display. For MGR, a window is a bordered region of the display that may be independently moved, reshaped, or changed in spatial position with respect to other windows on the display. Although the actual ordering of windows on the display is not totally ordered, the windows are topologically sorted into the window list to insure their spatial ordering is preservered. The ordering is established by the window creation order, and maintained by the window manipulation algorithms.

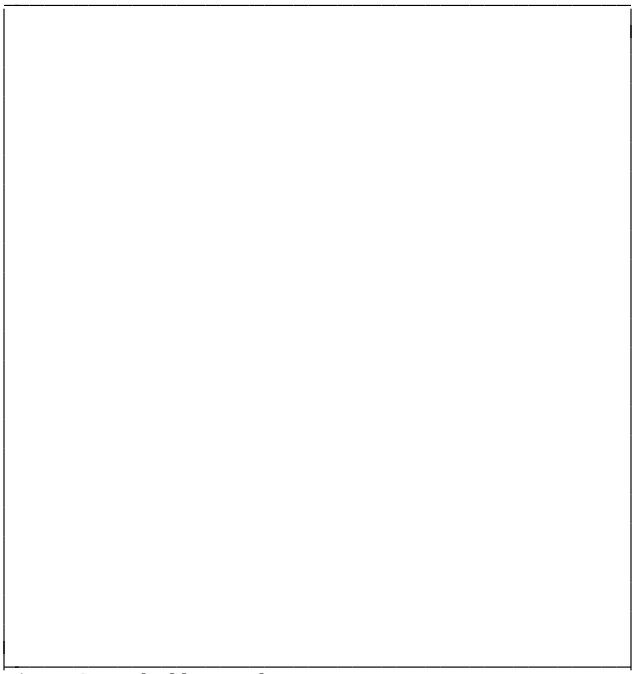

Figure 2. Linked list window structure

Each window descriptor contains a complete state description of its window* The linked list structure permits complete equality among windows: there are no parent - child relationships and no windows are restricted in size or extent by other windows on the display. The order of windows in the list may be altered either by the user or an application program. The use of a list of window descriptors with each object representing the entire state of its window is a simple, effective and easy to manage representation of overlapping windows on the display. Figure 2 shows the relationship between the windows on the display and their window descriptors.

Process Creation and Management

In a networked environment, where client programs running in a window can be either on a local or remote computer, process creation is important. If applications do not migrate from machine to machine over their lifetime (they currently do not), all networking capability must be contained in the process creation and connection phase.

Process Creation

The process creation interface is simple. It is supplied a command, with arguments (e.g. via `exec`) and it returns an *process id* and a *connection*. The *id* is simply the process id or UNIX `pid`, which is used to uniquely identify the process. The *connection* is a file descriptor, which is used along with the `read` and `write` system calls to communicate with the process.

Because of this simple interface, it is easy to plug in different process creation modules to meet the requirements of various operating systems and networks. The process creation routine used by the current Sun implementation operates as follows:

1) A pair of pseudo-terminals (or *ptty*'s for short) is opened. All client program have the terminal semantics implied by the *ptty*'s. 2) MGR `fork`s, creating a child process with standard input and output opened to the slave, or terminal side of the *ptty*. 3) The child process is placed in its own process group, with the *ptty* as its controlling terminal. 4) The ownership of the controlling terminal is

* See the appendix for the complete set of parameters that comprise the window descriptors.

changed to reflect the ownership of the process†. 5) An entry is then made in the */etc/wtmp* file that represents the client program as a user visible to the `who` command. 6) The client program is `exec`ed from the child process.

Internetworking

MGR has no built in notion of networking. Network facilities abound in UNIX systems; there are `rcp`, `rlogin`, `ftp`, `tip`, `kermit`[3], and many other programs whose purpose is to provide network access to remote computers. MGR's role is managing windows and not networks. The MGR strategy is to use existing network access commands to provide networking capability and not to re-implement the networking capability within MGR. Consequently MGR can access any network that is supported by existing programs. New network media and protocols can be used by MGR automatically, with no changes to MGR's code, simply by taking advantage of the already existing networking access routines.

A typical example of networking in MGR is creating a shell on a remote host, running in a window. This may be accomplished by typing: `rlogin` *over_there* if internet access is available, or: `kermit` If the available network is dial up serial lines. In the first case, a shell on the remote host *over_there* is created. The `rlogin` and `rlogind`, networking commands supplied with BSD UNIX, set up the connection and start the shell on the remote computer *over_there*. In the second example, `kermit`, with the appropriate startup file, will dial another computer from a serial port, and start a shell on the dialed-up machine. In both cases, the user is provided with a shell running on a remote machine. MGR is unaware of which network or what mechanism was used to effect this connection.

Implications of the MGR Networking Strategy

Not only is MGR simpler and smaller without any network code built in, but is more portable and more network capable as no new code has to be written or changed in order for MGR to work with a new network interface.

Another important implication of this networking scheme is that MGR has no control over the network. If a client program wants to create a second window on the display, MGR cannot guarantee that another circuit, or connection can be found between MGR and the client program. This is the normal case on serial lines: only one connection is possible per line. Consequently, once a connection is created between an application and MGR, is becomes the only connection between the two for the life of the application. Multiple windows per client program are managed by multiplexing the data to each window over the same channel.

Yet another implication of the networking scheme is that the topology and connectivity of the network is unknown to MGR. If two client programs `A` and `B` are both connected to MGR, it is not always possible for `A` and `B` to open a direct connection between themselves. Consequently, MGR provides an inter-application communications mechanism whereby different applications can rendezvous and communicate with each other using MGR as the go-between. This simple and effective communication mechanism encourages inter client program communication, providing the opportunity for an integrated application environment.

Authentication and Network Security

Since MGR relies on other programs to supply the session level networking control (all network security and authentication issues are resolved by these programs), MGR needs no password authentication or *trusted* host mechanism; it is piggybacked on top of those mechanisms provided by the networking programs. This simplifies the security aspects of MGR considerably.

As a result of MGR's network scheme, the inter-application messaging capability provided by MGR is required to insure applications can communicate with each other. To mitigate the risk of a hostile client program corrupting unsuspecting ones (such as shells) by sending them messages that mimic commands, the message facilities will only deliver messages from applications with the same user permissions as the recipient.

† This is the only place in MGR that setuid-root privileges are needed. It is a design flaw in UNIX that requires root privileges for this operation.

Process Termination.

When the process created with a window and all of its children dies, its window (or windows) should go away, as that process held the only connection to the window. MGR senses the death of a process in a window, either by receiving a `SIGCHILD` indicating its demise, or a permanent error condition on the communication channel. In either case, MGR marks the window as *dead* and buries it at the next convenient opportunity.

Conversely, when the user destroys a client program's window, MGR sends it a hangup signal if possible, severs the MGR side of the communication channel, restores the state of the `ptty`, and modifies the `/etc/wtmp` file to indicate that the client program has logged off.

I/O Scheduling

MGR maintains exclusive control over its system resources. It controls access to the mouse, keyboard, and display. It reads mouse and keyboard data, and dispatches it to the appropriate client program. Similarly, all output from client programs destined for the display is sent to MGR, which schedules and dispatches the data to the appropriate windows.

The method of reading, multiplexing, scheduling and demultiplexing data has the primary impact on the responsiveness, smoothness and throughput of the entire window system. As such, the algorithms chosen to implement this aspect of the window manager determine its real time feel for the user.

MGR's scheduling strategy is to let the users think they are in control (this of course is just an illusion, the user is really a slave to the window system). The mechanism for achieving this goal is to give highest priority for responding to user inputs. When the mouse is being moved, the user is presumably focussed on the mouse and at what it is pointing, not at what output is going to an obscure window in a distant region of the display. MGR has heuristics built in to determine what the user is focussing on, then switches attention to the same thing.

Overall Scheduling Strategy

The main body of code in MGR consists of the main processing scheduling loop. This loop waits in the `select` system call until input arrives. MGR processes each type of input, then goes back to the `select` to wait for more. This input may be from any of the client program connections, the keyboard, or the mouse. Each type of input is handled differently, because the data generation rate and importance of each type of input is different.

Mouse Input

When there are mouse input data is available, they are handled immediately. During periods of low and moderate load, the main processing loop is quick enough that only a small amount of mouse data is ready in the input queue at each iteration, and the mouse is tracked smoothly and uniformly. When the input queue grows, indicating the system is heavily loaded, MGR tracks the mouse in a more efficient fashion by not tracking every mouse position precisely but instead by throwing away intermediate position information. Although this causes a little jerkyness in the mouse tracking, it prevents MGR from getting too far behind, and almost entirely eliminates *mouse-behind*, the dreaded phenomenon where the mouse cursor on the display is so far behind the actual mouse position that the user feedback for mouse positioning is gone, and mouse becomes useless. MGR does away with *mouse-behind*.

Keyboard Input

Unlike the mouse, which spends most of its time idle, then suddenly whizzes about at the rate of 120 characters/second, the keyboard input is slow. MGR reads only one character at a time from the keyboard, processing it as it arrives. It never gets behind, people simply do not type that fast. Just in case, any time MGR receives data from the keyboard, MGR goes back to the `select` at the top of the main processing loop to insure new keyboard input data will be processed quickly on its arrival. In actual use, keyboard response is always immediate.

Client Program Output

When managing the output from client programs destined for a window, it is important to retain the illusion of simultaneity of updating various windows while maintaining a high overall data throughput. Any time there is input from a client program, MGR reads a large chunk of that input and stores it in a queue which is part of the MGR window descriptor for the desired output window.

Reading large chunks (512 bytes) at a time minimizes the system overhead that would be caused by a larger number of small reads. However, dumping the entire queue to the window at once would ruin the illusion of simultaneous updating. The user would see one window update for a while, then the next, then the next. To avoid this, the *chunkiness update problem*, MGR writes only a small part of the queue to the window at once before going on to process data for the next window. As long as the updates to each window are small, and hence quickly completed, all updating appears simultaneous.

For each client program in turn, if there are data to be read, MGR reads a big chunk into the appropriate window descriptor queue. Then, if there are any data in the queue, either because it was just read, or it was left from before, MGR outputs a small amount of the queue to the window See figure 3). When MGR is finished processing data for every window, it goes back to the `select` to wait for more data to arrive. However, if there is any queued, but not yet processed data in any of the window queues, MGR switches in to *polling* mode and out of *waiting* mode, so deadlock is avoided in the situation where there is no data arriving, but still some left in the queues.

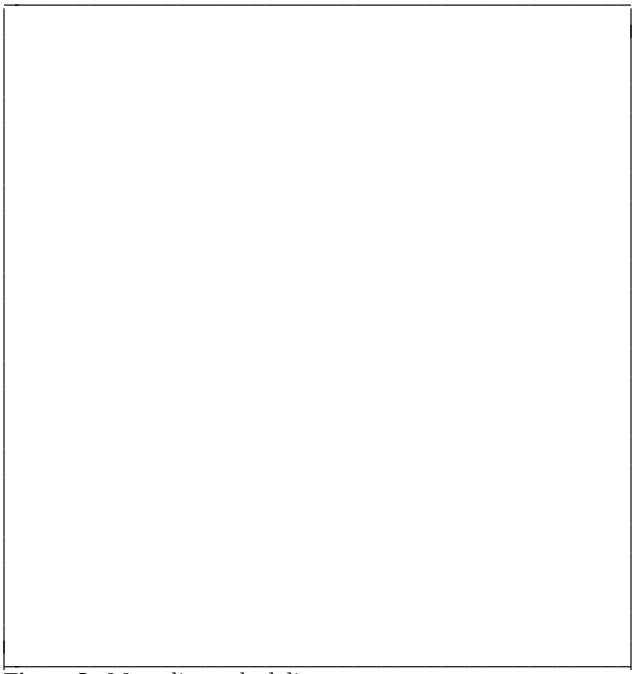

Figure 3. MGR client scheduling

There are two steps involved in determining the amount of data to be sent to a window at one time. The first step sets a maximum number of bytes that may be processed in one pass of the main process loop. The second step calculates the cost of processing the data to insure that the cost does not exceed a predetermined maximum. The cost of an operation is directly related to the amount of processing effort it will take to complete that operation.

The number of bytes to be processed at one time is based upon the window's priority. There are three different window priorities. The *active* window, the one that is currently receiving mouse and keyboard data has the highest priority. Windows that are completely exposed have the medium priority, while windows that are partially or completely obscured have the lowest priority. The different priority levels were chosen to give more attention to what the user is focussing on. High priority windows get twice the maximum process slice as medium priority windows. Low priority windows get half of that. Of course for those users who have no focus at all, there is a round robin scheduler that treats all windows equally. The priority scheduler works well on slow or heavily loaded workstations. For a fast machine, such as a lightly loaded Sun 3/60, it does not make much difference which scheduler is used.

Once the maximum data size is determined, the command processor assigns a cost to each byte (or group of bytes for multi-byte commands) as it is processed. As soon as the processing cost exceeds a preset threshold, processing of the queue is halted, and the command processor returns the number of bytes actually processed. Any unprocessed bytes remain in the queue and are resubmitted to the command processor the next iteration of the main process loop.

The amount of data processed for each window in a single pass of the main loop is the minimum of the number of bytes defined by the window's priority and the number of bytes it takes to exceed the cost threshold.

Although processing exactly one character per window per pass might seem to optimize the simultaneity of window updates, there is a reasonable amount of setup overhead for each call to the command processor. Processing a single character per window would reduce the overall system throughput too much. The amount of data typically processed per call is about of 40 to 80 bytes, or until an expensive operation, such as scrolling the window occurs.

Window Management

The Window Management component of MGR implements all of the algorithms needed to update and maintain the integrity of the display, manage display resources, and handle menus, character fonts, event notifications, and text cut-and-paste.

Window Management Algorithms

Each window descriptor has a flag, the *covered* flag, to indicate if the window is totally visible on the display, or if the window is partially or completely obscured by another window. Any time the arrangement of windows on the display is changed, each window's *covered* flag is updated. If the window has become partially or completely covered by another window, the *covered* flag is turned on. If a previously covered window is completely exposed, the *covered* flag is turned off. When the window becomes covered, an off-display or backup copy of the window is made in main memory. Updates to the window by the command processor are made to the backup copy of the window instead of directly to the display. When a window becomes exposed, the newly exposed portions of the window are copied from the backup copy to the display, then the backup copy is deleted, and the memory used by it freed.

In MGR, only the top window on the display may be moved or reshaped. The window management algorithms could readily handle changes to arbitrary windows, but this would unnecessarily confuse the user interface. The following five window primitives are used by MGR to manipulate windows. All window manipulations, except for becoming the top window, only the top window is subject to window manipulation commands, which are composed of combinations of these five primitives.

A window is created. A new window is always created as the top window. Any windows that are currently exposed, but would be covered by the new window are marked *covered* and their window contents are saved in the backup copy of the window.

A window is destroyed. Only the top window may be destroyed. The background pattern is drawn over the window, obliterating it. Then for each window that intersects the dying window, starting from the bottom window, that portion of the covered window that was obscured by the dying window is copied from the backup copy to the display. If the window was covered only by the dying window, it is marked *uncovered* and dealt with accordingly. Although this technique can result in more display updating than is necessary, it is easy to implement, and the performance is satisfactory.

The top window is pushed to the bottom of the display. If the window intersects any other windows, a backup copy of the the other windows are made, and the intersecting windows, starting from the bottom, are redrawn where they intersect the window in question. Any newly uncovered windows are processed appropriately.

An arbitrary window is brought to the top of the window list. All the windows currently on top of and intersecting the window that will become covered are marked covered, and a *backup* copy of each window is made. The backup copy of the current window is then copied to the display.

The top window is reshaped The process for deleting the top window, then creating a new one are followed in turn. Moving a window is simply a special case of reshaping.

Updating Overlapping Windows

When only a part of a window is visible on the display, because sections of it are covered by other windows, the update manager insures that the visible portions of the window are updated properly. The update manager gets called once by the command processor, just before it completes. The update manager is called whenever the *backup* copy of a covered window has changed, and the visible portions of that window on the display need to be updated, reflecting the current state of the *backup* copy.

The window descriptors maintain no information regarding the position of other windows that may intersect with them. When the update manager is called, it determines which portions of the window are visible and updates them appropriately by copying sections of the *backup* copy of the window to the display. By using this strategy, there is never a performance penalty imposed on a window simply because it *might* have to update while obscured some day. All of the work required to implement background updates is done only when the updates are actually required; there is no information precomputed as windows are rearranged.

The update manager is called with 3 parameters: 1) the window to be updated, 2) the current list of windows, and 3) the bounding rectangle of changes to the *backup* copy of the window, as determined previously by the command processor.

Background updates are accomplished using the following strategy.

(1) The update manager first identifies all of the windows that are on top of the current window.

(2) All of the top and bottom coordinates of the intersecting windows are sorted into one list, whereas the left and right sides are sorted into another.

(3) From these two lists, the set of non overlapping rectangles that cover the window are computed by using adjacent pairs of x and y coordinates to define the sides of each rectangle (see figure 4).

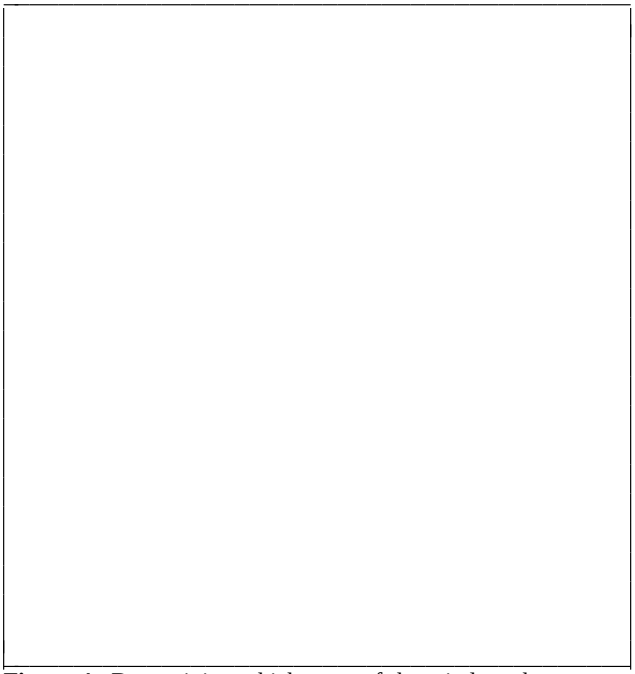

Figure 4. Determining which parts of the window show through to the display

(4) Each rectangle in the set is checked to see if it is covered by some window.

(5) Each rectangle that is visible is put onto a list, the update list, which defines which of the rectangles shows through to the display.

(6) As rectangles are placed on the update list, they are combined with rectangles already on the list, where appropriate, to make fewer large rectangles (figure 5).

(7) Updating the *backup* copy of the window is simply a matter of laying the update list of rectangles on top of the *backup* copy of the window, and transferring the intersection to the display.

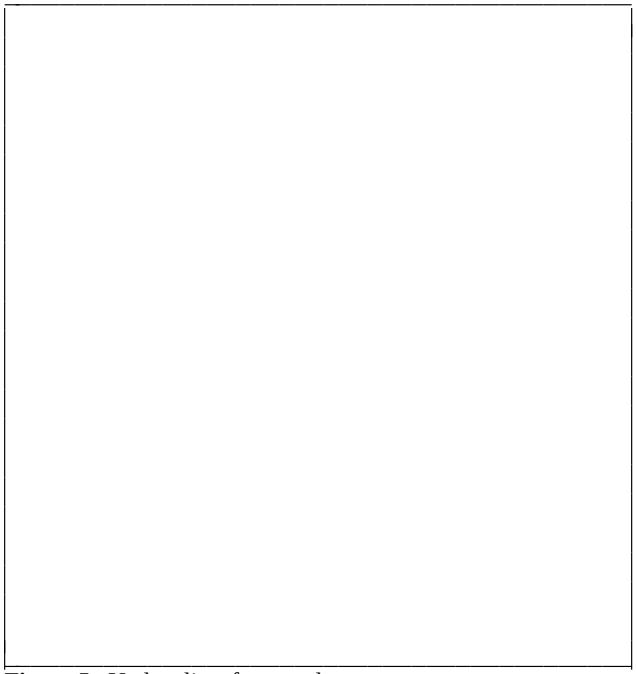

Figure 5. Update list of rectangles

Two additional steps are done to improve the efficiency of the above process. First, the algorithm described above is applied only when some change happens to the shape or position of some window that would result in a change to the update list, and only when a background update has been requested. Second, instead of copying the entire *backup* copy through the update list to the display, only the bounding rectangle of changes is updated, minimizing the copying required.

This mechanism for background updates has several advantages over methods that require each window to keep track of which parts are covered. All information required by the update process is completely contained in the update module, which is less than 300 lines of C code. This also frees all other aspects of the window system from dealing with overlapping updates. No other code anywhere has to deal with which window is covering what and where. Only the covered flag is maintained.

Another advantage of this technique is that of *lazy evaluation.* If background updates are never required, there is no cost penalty associated with the update management.

On the down side, it can take a long time to compute the update list for a window (up to a second), especially if it has pieces of 40 or 50 other windows on top of it. Fortunately, the update list is computed infrequently. In the typical case, the time lag is imperceptible.

Display Optimizations

For many current client programs that run in a window, including the editors, shells and mail, much of the effort expended by the window manager on behalf of the client program involves scrolling the display. On some hardware, including the Sun Workstations, scrolling can be effected more efficiently if the left and right edges of the scrolled region are aligned on byte boundaries. With the `ALIGN` compile option of MGR, windows are always placed so a byte boundary happens somewhere between the outer edge of the window border and the first (or last, for the right edge) pixel of the window. Using MGR standard 5 pixel window borders, MGR will adjust the edges of the window up to 2 pixels in order for the windows to conform to the byte alignment criterion. When the entire window is scrolled, which can happen frequently, a special byte aligned scrolling routine is called to optimize scrolling.

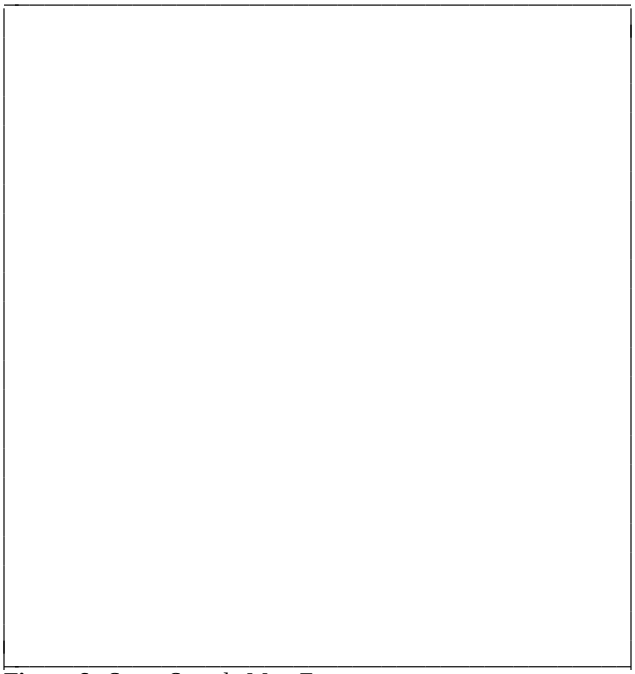

Figure 6. Some Sample MGR Fonts

Fonts

Character fonts in MGR are global resources, they do not belong to a particular window, although a client program may add new fonts to MGR as needed. The character fonts displayed in a window may be changed by a client program, on a character by character basis, and characters may be placed at arbitrary positions on the window. Each font is stored interally as a single bitmap, with the characters of the font strung end to end, in ASCII collating sequence. See figure 6 for some sample MGR fonts.

For some display hardware there exists frame buffer memory that is accessible to MGR, but not mapped onto the display. This *hidden display memory* is often faster in terms of updates to the display than main memory. Consequently fonts, whose characters are copied to the display frequently, are best stored in *hidden display memory*. To best manage this memory the fonts can be cached on a most recently used basis. On computers with no *hidden display memory* main memory is used to cache fonts instead.

Menus

MGR has built-in facilities for managing pop-up menus. A pop-up menu is a list of choices that appears in response to depressing a mouse button. Although pop-up menus can be implemented almost entirely by the client program, as MGR needs them for its own use anyway, it provides complete menu handling services for client programs. Not only does this insure uniformity of the user interface - all menus look and feel the same, but the response time for menus is immediate since they are handled internally. The MGR menu system provides client programs with a wide range of menu capabilities. Position or context sensitive menus, slide-off or hierarchies of menus, and paging or "deck of cards" style menus are all supported. Menus are used by client programs by defining in advance (or downloading) the choices of each menu along with the actions associated with each choice. Actions are ASCII character strings that are send to the client program when a menu item is chosen. Up to a hundred menus may be defined and downloaded by a client program at once, while the selection of which menu pops-up in response to a mouse button hit is selected independently.

As soon as a menu is downloaded, MGR creates an image of the menu in memory. When the menu is to pop-up, as its image is already available, it appears immediately, with never a delay. Thus users are never left in the lurch, holding down a button, wondering if a menu will ever pop-up. Once the menu is on the display, the menu handler takes over from the main process loop, processing mouse input and tracking the mouse exclusively to insuring immediate response to all menu selection activities. As soon as a menu item is selected, the menu is erased from the display, the menu action is sent to the client program, and normal processing resumes.

The menu system is implemented in two layers. The first or bottom level consists of five routines that control a single menu. The top level of the menu system calls the bottom level routines and itself recursively to produce hierarchies of menus. The bottom level routines cooperate by passing a menu state descriptor among themselves, altering the current state of a menu appropriately.

The five bottom level menu management routines operate as follows:

(1) `Menu_define` is given the list of menu items, and the current character font. It sets up a menu descriptor and creates an image of the menu in memory.

(2) `Menu_setup` is called with the menu descriptor provided by `menu_define`, and a location on the display. The part of the display about to be covered by the menu is exchanged with the already created image of the menu, and the appropriate item is highlighted.

(3) `Menu_get` is called with the descriptor of a menu already visible on the display. `Menu_get` reads and tracks the mouse until either the mouse button is released, or the mouse is dragged out of the menu. `Menu_get` then sets a status word in the menu descriptor indicating why it terminated. *Item three was chosen* or *the mouse exited, stage right, off the end of menu item one* are examples of the menu status.

(4) `Menu_remove` is the inverse of `menu_setup`. It exchanges the saved portion of the display with the menu.

(5) `Menu_destroy` is called when a menu is no longer neeeded. It destroys the menu image and menu descriptor, and frees all resources associated with the menu.

The high level menu interface routine, `do_menu`, is called with a list of predefined menus, a set of menu links that determine how the menus are related to each other and the name of the current menu. If the user slides off to the right of a menu that is linked to another menu, `do_menu` calls itself with the name of the next current menu.

Sometimes the choice of which menu to have popped-up is dependent on where the mouse is. In such circumstances, the client program may have any possible menus already downloaded, but have none selected to pop-up. As soon as the mouse button is depressed, the client program may query its position, and while the button is still down, select a menu. The menu will then pop-up immediately.

Although MGR supplies client programs with a comprehensive menu package, there are some instances where client programs need to manage the menus themselves. For these client programs, MGR provides access to the low level menu interfaces routines.

Events

An asynchronous change to the state of the window system is called an event. A mouse button push or a window reshape are examples of events. There are sixteen events currently defined by MGR. They are summarized in table 1.

Table 1: MGR event types

BUTTON 1 up	end mouse button released
BUTTON 1	end mouse button pushed
BUTTON 2 up	middle mouse button released
BUTTON 2	middle mouse button pushed
DEACTIVATE	window deactivated
DESTROY	window destroyed
MOVE	window moved
RESHAPE	window reshaped
ACCEPT	accept messages
ACTIVATE	window activated
COVERED	window covered
NOTIFY	set ID string for other windows
UNCOVERED	window uncovered
REDRAW	display refresh requested
PASTE	text was just pasted
SNARFED	text was put into cut buffer

The first group of events can be sent only to the

currently active window. The next group of events may affect any window, whereas the final group of events affect all of the windows.

For any of the event types, a client program never receives event notification unless the event was requested. Events are requested by specifying an ASCII string (the event string) to be returned to the client program when the event occurs. For most events, `printf` like escapes in the form of **%X** may be imbedded in the event string, where **X** represents a substitution parameter. When an event occurs, the **%X** is replaced by the value of the parameter it represents. An event string specified as:

```
"Hello %p there"
```

might be returned by MGR as

```
"Hello 26 43 there"
```

where the *%p* was replaced by the current mouse position.

Most of the events listed in the table represent obvious changes to the window system state. However, several need a little more explanation. The `ACCEPT` event provides the mechanism for a client program to receive a message from another client program. The event string associated with the event is used to encapsulate the message. Like the `ACCEPT` event, the `NOTIFY` event is used to facilitate communication among client programs. The `NOTIFY` event string, unlike the other events, is not sent to the client program that sets the event string. Instead, the event string is made available to other client programs as a **%X** parameter, normally associated with a button event. This mechanism permits the construction of client programs that act as servers to other client programs without requiring any arrangements to be made in advance. The `NOTIFY` message contains the information needed to utilize the server program's services.

The final use for **%X** parameters in event strings is to instruct MGR to rubber-band (or change the size or position of an object in response to the mouse) an object, such as a line or box for the client program, and report back its coordinates. By having MGR rubber-band the object instead of the client program, not only is the rubber-banding done more smoothly, but is easier for the client program, and more uniform at the user interface.

Managing the Window Environment

Connections to remote computers in MGR are typically created at start-up time, and remain for a long time; many client programs reuse the same connection. To insure a client program receives a known windowing environment when it begins, MGR provides a facility to manipulate the windowing environment, permitting a client program to push some or all of the existing environment on a stack, or selectively inherit the windowing environment from a predecessor. Using this facility, client programs may be stacked with each creating its own environment, yet return to the calling program with the original environment intact.

Cut and Paste Strategies

A desirable capability of any window system is the ability to copy text from one window and paste it into a different window, regardless of the application or remote host in the window. There are three plausible strategies for implementing cut-and-paste in a window system.

The simplest way for a window system to provide this capability is to maintain one or more globally accessible *cut* buffers that client programs may use to share data. This method is satisfactory for new applications, written specifically to use the feature, but the hundreds of existing applications that do not have the window system specific cut-and-paste built into them can not use it.

Another strategy for cut-and-paste involves saving an ASCII interpretation of the data going to the window, which may be recalled as the user sweeps out text to be copied. This approach is feasible if restrictions are placed on the type of data that can be put in the window. This strategy restricts the output to a window to contain a single, fixed width font whose characters are permitted only on a fixed grid, as with a typical CRT. The reason for these restrictions is to permit a rapid conversion from window coordinates, back to the original representation. If arbitrary ASCII combinations of different character fonts, characters at arbitrary pixel locations, and graphic objects, such as lines and circles are permitted on the same window, a carefully sorted list of each object on the window would need to be maintained in some ASCII form to effect the *cut* half of cut-and-paste. Sorting and maintaining this list is too slow, especially as part of the window scrolls

away, which requires the positions of each object in the list to be updated.

The third technique available to perform cut-and-paste is to turn the image on the window back into characters, by recognizing the bit patterns. With this method, the problems of the previous method are avoided, permitting arbitrary graphics in any window, yet still permitting the cutting and pasting of text for terminal based, text-only applications, such as `vi`, `mail` or the shell. The advantages of this method are 1) No effort or cost expended unless the *cut* is actually invoked by the user, 2) No restrictions need to be placed on the type or complexity of objects that may appears in a window. The disadvantages of this technique are 1) Speed. Since the ASCII representation has to be re-created from the bit image, it takes longer that other methods. Finally, as the actual bits on the display are used to recreate the text, what you get is what you see (WYGWYS), which is not always what you want. Notably, tabs, spaces, and nonprinting control characters are indistinguishable.

MGR and Paste Algorithms

MGR uses the two of the cut-and-paste strategies outlined above. A *cut* buffer is available for any client program to write to or read from. The user may request for the contents of the *cut* buffer to be send to a client program.

MGR implements cut-and-paste by translating the image on a window back into ASCII characters. When the user requests a *cut*, by selecting the appropriate option on the command menu, MGR notes the current character font and position and presents a rubber-band box to the user that moves in increments of a character size in response to the mouse. Figure 7 illustrates character text being swept out in preparation for *cutting*. The user releases the mouse to indicate the extent of the swept text. If this is the first time the user has requested a *cut* in this character font, each glyph in the font is hashed into a table. Each character sized region swept by the user is hashed, and compared with the glyphs with the same hash value. If no match is found, the process is repeated for the reverse video version of the character. Once the characters have been found in the hash table, trailing blanks are stripped, and leading spaces changed to tabs as required. The resultant ASCII string then replaces or is appended to the global buffer.

The paste function simply inserts the current contents of the global *cut* buffer into the client program input stream, as if those characters had been typed at the keyboard.

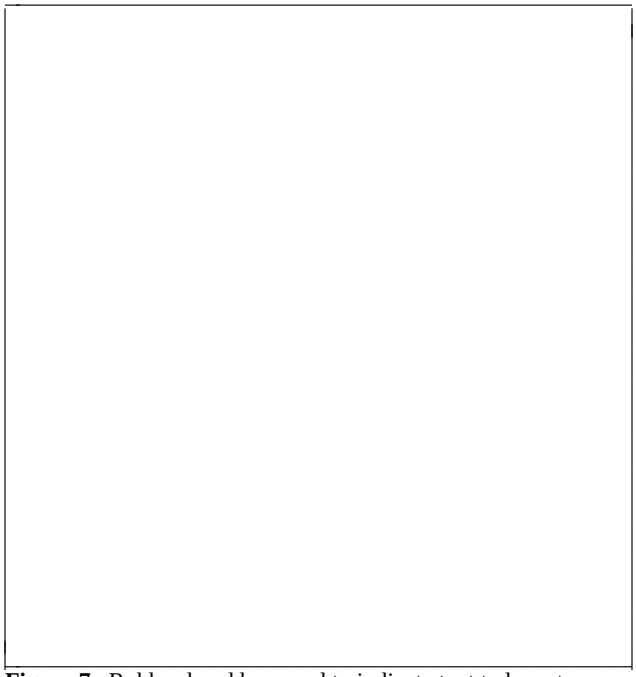

Figure 7. Rubber-band box used to indicate text to be cut.

Command Processing

The command processor interprets the stream of bytes coming from a client program to a window and performs the appropriate actions described by those bytes. The command processor is passed a string of bytes to interpret. It interprets one or more bytes in the string, then returns the number of bytes processed.

An important feature of the command interpreter is its ability to function without any knowledge about the state of the window system as a whole. It simply processes commands for its window as if it was the only window on the display. When the command processor writes a character on the window, for example, it makes know difference whether the window is visible or covered, or if part of that character might be obscured by some other window. This feature greatly simplifies the design and implementation of the command processor.

When the command processor is called, updates affect either the display directly, if the window is completely exposed, or the backup copy when the window is covered. Transferring the

updates from the *backup* copy to the visible portions of the display copy are handled by the update manager, not the command processor.

Windows taking most of the users attention tend to be uncovered; its hard to interact usefully with a window you cannot see. Such uncovered windows operate at full speed, with update going to them directly. Obscured windows pay the performance penalty of having each update done twice, one to the *backup* copy, then again to a visible portion of the window. This performance penalty is more than justified by the resultant simplification of the command processor design.

In addition, this strategy for MGR was originally chosen anticipating graphics display architectures that manage the overlapping windows in hardware, such as in the YAB[4] terminal. Such hardware would take separate memory images of each window, and map them onto the display with the appropriate overlapping and clipping.

When the command processor receives a command that could potentially change the window's relationship with other windows on the display, such as reshaping or moving the window, as soon as that operation is completed, the command processor returns immediately, as it has no way of knowing whether to draw on the display or into memory at this point. Any remaining data to be processed will be dealt with on the next call to the command processor.

To maintain smooth window system operation it is important that updates to one window not take too much time or else window system updates appear bursty. The command processor keeps track of how much time it has spent for one window to insure the window does not get too much time. This is accomplished with a simple strategy. Each possible command is assigned one of two costs, cheap or expensive. Operations such as drawing a single character, or moving the cursor position are examples of cheap operations. Scrolling the entire window is an expensive operation. The command processor permits exactly one expensive operation per call. In practice, this is easy to compute and it works effectively; a more sophisticated mechanism, although possible does not seem necessary.

The last role of the command processor applies only when the window is covered. The smallest rectangle within the window that completely contains all changes made to that window is computed. This rectangle is passed on to the display update routine, permitting updates to the display from the backup copy of the window to be handled more efficiently.

Command Language

The language understood by the command processor is stream based. Unlike X, whose XLIB[5] interface is defined in terms of C language function calls, the MGR command protocol is defined by partitioning the byte stream from a client program to MGR into commands. This protocol is based on the idea that simple things should be simple to do, whereas complex actions might require complex commands. At minimum, a window needs to look like a terminal, so the entire existing pool of terminal based applications work unmodified in a window. Thus the simplest form of the MGR command protocol is <ESC>X, where ESC is the ASCII escape character (octal 033) and *X*, the command character, is any one of the 96 printable ASCII characters. Common terminal like commands, such as *insert line* or *move cursor* are simply <ESC>X for the appropriate character *X*.

Although many commands can be represented as <ESC>X, some require parameters. Such parameters are passed in the form: <ESC> x_1, x_2,... x_nX where x_1 to x_n are comma (or semicolon) separated signed integers. The ASCII representation for integers is used to permit the protocol to work over seven bit channels. In addition, several commands require sending arbitrary data as part of a command, such as downloading a bitmap image to MGR. This is accomplished by a command of the form: <ESC> x_1, x_2,...,<len>X<data>, where <len>, a positive integer, is the count of bytes following the command character, X, to be taken as part of this command.

This command protocol is extensible in the sense that both new command characters may be added, at least up to the 96 printable ASCII characters, and extended parameter lists may be added to existing commands. If the need arizes to send a different type of command, this new command type can be encapsulated in the data field of the third command type.

All commands to MGR are asynchronous, that means the client assumes they will be performed in a reasonable amount of time. A few commands return status or window state information, but they too are

asynchronous; the client does not need to wait for a reply, but can continue to send new commands and data to MGR.

The reply to a command is distinguishable by the client program because the format of the reply is determined by the client program before the command was issued. It is therefore possible to reliably distinguish between a keyboard character and the reply to some previous command. MGR does guarantee that 1) all command replies will be sent in the order that the command was received and 2) a reply will always be sent for every command that expects one.

Any characters sent to MGR between commands, are interpreted as characters to be drawn on the window at the current character cursor position, as one would expect from an ordinary terminal.

User Interface

The user interface was designed to meet two goals: it should be easy to use and very responsive. To keep the interface easy to use and learn, it was kept simple. A single mouse button is dedicated to window manipulation, and the few actions available are fast and consistent. All window manipulations happen to the top, or *active* window, which is always easily identified by its emboldened border. The only action permitted on non-active windows is making them the active window[6]. Although only a small subset of the possible window manipulations is available to the user, in practice it is a sufficient set and easy to master.

Several design decisions were made to achieve fast response in the user interface. As discussed above, keyboard input and mouse tracking have highest priority in the MGR scheduler. All pop-up menus used by MGR, including both the system menus and downloadable applications menus, have their images created and saved in memory as soon as the menus is defined. Consequently, as soon as the mouse button is pushed requesting a menu to pop-up, it appears with no perceptible delay. Ever.

Once the menu has appeared, MGR continues to track the mouse while in the menu, highlighting the various menu items as appropriate, with no application intervention. As long as the user is selecting an item from a menu, only the mouse input is processed, and the rest of the display freezes until the selection is made. The user need not desperately try to select an object with a menu item before it gets away, scrolling off the window. Time stands still for the user to make the selection at their own pace.

Similarly, MGR will rubber-band objects such as lines and boxes on behalf of client programs. The applications do not need to track the mouse, and the user is assured the fasted possible response, even under heavy load.

For advanced users, it is often an annoyance to stop typing, grope for the mouse to select a menu item, then have to relocate hands back onto the keyboard at the proper position. For these users, or anyone else who is not particularly fond of rodents, the pop-up menu activated system functions, as well as several other *shortcut* commands have keyboard equivalents, activated by pressing a special *meta* key on the keyboard along with a mnemonic keyboard command character. On the Sun keyboard, the `left` or `right` keys serve this purpose. Expert MGR users rarely use the mouse for system interaction; it is faster to use the keyboard instead.

Device Interfaces

MGR interacts with three hardware devices, the mouse, the keyboard, and the display. In all cases the interfaces to these devices is well demarkated, resulting in easy portability to different hardware.

For the keyboard, MGR simply uses its standard input stream. Characters with their high order bit on are taken to be *meta* keys that invoke system functions. On the Sun Workstations, keyboard input is confused with the system bell, and the redirection of system messages, so there are special Sun specific keyboard initialization routines to correct for the confoundment. For normal systems the keyboard gets no special treatment.

For the mouse, MGR opens a serial port, normally `/dev/mouse`, and expects to find data in either Mouse Systems mouse format[7], or in Sun's modified Mouse Systems mouse format. For the initial stages of an MGR port, simply plug a mouse systems compatible mouse into a serial port and the job is done. For systems with their own mouse, the mouse interface looks like:

```
button=mouse_get(x_delta,y_delta)
```

where *button* is the state of the mouse buttons, and

(x_delta,y_delta) is the number of units the mouse moved since the last call to `mouse_get`. MGR tracks the mouse on the display. For systems that track the mouse in hardware, MGR provides set of macros for the mouse tracking and mouse cursor interface. These macros may be redefined, or for the Macintosh as an example, set to do nothing, permitting the hardware to track the mouse.

Display Interface Definition

The most complex interface is that to the display hardware. It consists of the seven routines show below:

Display Interface Routines
bit_open
bit_create
bit_alloc
bit_destroy
bit_blit
bit_line
bit point

These routines all reference a data type called a `BITMAP` and a set of access macros that return specific information contained in the `BITMAP` structure.

A bitmap is a rectangular array of pixels, where each pixel is one bit, for monochrome MGR, and one or eight bits for color MGR, along with sufficient information to perform a bitblit[8] on a pair of `BITMAP`s. The MGR interface to the display consists of:

bit_open
: Returns a `BITMAP` pointer to the entire display.

bit_create
: takes an existing bitmap and creates a new one whose pixels are a subset of the one supplied.

bit_alloc
: creates a new `BITMAP` in memory of a specified size. The image data for the bitmap may be supplied, or allocated as needed.

bit_destroy
: destroys a `BITMAP` by freeing all resources associated with it.

bit_blit
: performs a bit_blit between two `BITMAP`s. The bitmaps may be both on the display, both in memory, or any combination of display and memory. For monochrome bitmaps, one of the 16 possible bit-blit functions may be specified. If no source `BITMAP` is provided, then the source is taken to be an infinite supply of *one* bits. For bit_blit calls in which both bitmaps contain eight bit pixels, the bit-blit operation, is performed bit-wise for each bit in the pixel.

  If the source bitmap contains one bit deep pixels, but the destination is eight bits per pixel, then the source `BITMAP` is expanded to eight bits per pixel by assigning to each "on" bin a foreground color, and to each "off" bit a background color. Bitblits in which the destination `BITMAP` has 1 bit per pixel while the source `BITMAP` contains eight bits per pixel are not used by MGR, so their exact operation is undefined.

bit_line
: draws a line on the `BITMAP` Lines may be set, cleared, or inverted.

bit_point
: is functionally equivalent to a bitblit of a single pixel, but is a separate function for potential performance gains. All graphics objects in MGR except lines, such as circles and elliptical arcs are implemented in terms of calls to `bit_point`. Where special hardware is available to warrant it, those graphics routines may be recoded in a hardware dependent way to improve their speed.

All access to the parameters of the `BITMAP` structure are through macro calls that isolate the specifics of the structure from MGR, permitting new different `BITMAP` structures to be implemented without changing MGR.

Summary

MGR was built to be a small, portable, network capable window management system that provides sensible capabilities for an ill defined heterogeneous network environment. The decision to include a feature in MGR, or off-load that capability into the client programs was made to insure uniformity and responsiveness of the user interface while maximizing the networking capabilities of the system.

The algorithms and techniques used to implement MGR were chosen for simplicity and modularity, not necessarily theoretical elegance. They function quickly and reliably, and have ported easily to

several other operating system platforms. The resulting system is fast, fairly small, asn widely used in the local research community.

References

1 D. Ritchie and K. Thompson, *The UNIX timesharing system*, *Bell System Technical Journal,* vol. 57, no. 6, part 2, July-August 1978.

2 J. L. Alberi and M. F. Pucci, *The Dune Distributed Operating System*, Bellcore Technical Memorandum TM-ARCH-010642, 1987, Morristown N.J.

3 Frank da Cruz and Bill Catchings, *Kermit User's Guide*, Columbia University, 6th Edition

4 Jackson, Namon *Yet Another Bitmapped Terminal* Bellcore TM # TM-ARH-000-988 3/25/85

5 Ingalls, D., *The Smalltalk Graphics Kernel*, Byte Magazine, 6(8), August, 1981

6 J. Gettys, R. Newman, T. Della Fera, *Xlib - C Language X Interface January 1986.*

7 S. A. Uhler *MGR users manual* 1987, Bellcore Morristown, NJ.

8 *M-2 Optical Mouse Technical Reference Manual* January 1984, Mouse Systems Corporation Santa Clara Ca.