Jan. 1989

Windows without Events

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Graphics systems, and window systems in particular, are complex to program. Their event-driven input mechanisms lead to intricate structures based on state machines, event masks, and generally idiosyncratic designs. It is hard to build convenient user interfaces with such cumbersome programming models.

The environment of a graphics program can instead be defined entirely in synchronous terms, using synchronous communication and the concurrent composition of synchronous processes. Although these concepts can be put to use in a conventional programming language, they are particularly effective in a concurrent language. Graphics systems themselves can be simplified by decomposing them into synchronous processes. A practical window system can be implemented in a few pages of code in a synchronous concurrent language.

Introduction

> "Most applications simply are event loops: they wait for an event, decide what to do with it, execute some amount of code that results in a change to the display, and then wait for the next event."
>
> *X Windows C Library and Protocol Reference, p. 212.*

This quotation describes a state machine. State machines are capable of universal computation, but universality does not imply convenience. Turing machines are universal, but we use them to prove theorems, not to write software, because they are poor models of programming. Their cousins, state machines, are also powerful, but not for systems programming. Nonetheless, the input interface to most graphics systems is based on events — a press of a key on the keyboard, a change in position of the mouse, a press of a mouse button — and leads to applications structured like state machines.

Output is easier to handle than input because it is synchronous. Whether the output device is an ASCII terminal or a bitmap screen, a single procedure call can change the display. If the input is just a keyboard, then it too may be handled by calling a procedure to fetch the next character. The difficulties arise if the program must not block while waiting for keyboard data, or if it wants to know what the mouse is doing while the keyboard is idle. The step from 'get next character' to 'get next input event' is a logical one to make, but it sweeps aside a number of problems such as describing what an event is or selecting which set of events is currently of interest. Its worst effect, though, is how it influences the design of the program. Imagine we have a program that reads pairs of characters typed on the keyboard:

```
forever{
     c1 = getchar();
     c2 = getchar();
     processpair(c1, c2);
}
```

What happens if we need to read events, only some of which are keyboard characters? The implicit state machine in the loop must be isolated and made explicit:

```
forever{
     e = getevent();
     switch(e.type){
     case KEYBOARD:
          handlekeyboard(e);
     case MOUSEBUTTON:
          handlemousebutton(e);
     /* etc. */
     }
}
handlekeyboard(Event e){
     if(kbdstate == 0){
          c1 = e.char;
          kbdstate = 1;
     }else{    /* kbdstate == 1 */
          processpair(c1, e.char);
          kbdstate = 0;
     }
}
```

This style of programming — switching on the type of event and calling event handlers — is clumsy but familiar to programmers of event-driven interfaces. Explicitly or not, most graphics applications are encouraged to adopt this structure. Several things are wrong with it: all types of events come through the same channel and must be disentangled; the event handlers must relinquish control to the main loop when they run out of events; and the externally imposed definition of events (such as whether depressing a mouse button is the same type of event as releasing one) affects the structure of the overall program.

To make the situation more comfortable, event-driven interfaces typically allow some extra control. For instance, a program displaying a pop-up menu can usually arrange to ask only for mouse events, so the code supporting the menu is not disrupted by keyboard events. Although they help, such details in the interface are just work-arounds for the fundamental difficulties that event-driven programming must ultimately face. The work-arounds accumulate: the X11 windows library, for instance, has 27 standard entry points to handle 33 types of events [Xlib 88]; NeWS has a single general event type but still needs 20 entry points to handle it [Sun 87]. The interface for input in GKS is comparably complex [GKS 84]. It is demonstrably difficult to write simple programs to connect to such intricate interfaces [Rose 88].

Although events from the various inputs may be intermingled and asynchronous, the events from any one device will be well-behaved. We could therefore program each device synchronously and cleanly, as in the first loop above, if we could divide the events into separate streams, one per device, directed at concurrent processes. The resulting program would be a collection of self-contained processes, free of irrelevant bookkeeping, whose execution would be interleaved automatically. (Most systems force the programmer to interleave explicitly.)

That approach was taken in Squeak [Card 85], a concurrent language designed for programming such components of user interfaces as menus and scroll bars. Squeak was a small language, though, and too simple to be useful for writing complete applications. It lacked variables, a type system, channels of anything other than integers, and dynamic creation of processes and channels. To make input manageable in a realistic environment requires a stronger language than Squeak and a more concentrated understanding of graphics applications and the environment in which they run.

I have written an experimental window system, using a concurrent language designed for the job.

The language, called Newsqueak, is documented fully elsewhere [Pike 89]. The window system provides a well-specified environment for its client programs. The environment is not defined in terms of event-producing devices, as it is in Squeak. Instead, it has a synchronous procedural interface for output and structured communication on a small number of synchronous channels to handle input and control. The window system functions by multiplexing its clients' access to its own environment, which has the same structure, allowing the system to be run recursively. The environment is easy to program both from the client's and the window system's points of view. The complete window system is about 250 lines of Newsqueak.

Basics

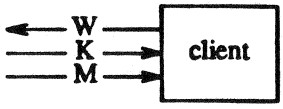
An application or *client* of a window system is an independently executing process whose display activity is confined to a subset of the entire screen controlled by the window system. That subset is the client's *window*. Ignore for the moment the process-specific aspects of the client and assume that it is a procedure written in a high-level programming language. Moreover, assume that there is no implicit global environment for the procedure. Instead, the environment for the procedure must be passed explicitly as parameters. If we can define those parameters and what they signify, we will have completely defined the properties of a client.

One of the parameters will obviously be a variable, say `W`, labeling the window in which the program is running. `W` will be passed to drawing subroutines to place output in the window. Its precise type need not concern us yet, but it must have some geometric component that defines the window's location on the display.

`W` is, loosely, a capability, granted by the window system, enabling the client to use its window. `W` labels a multiplexed component of a larger screen, containing all the windows. Input to the client may be regarded similarly. We need capabilities allowing communication with the multiplexed mouse, keyboard, and perhaps other input devices. Call these `K` and `M`, and pass over their exact specifications. The client's declaration is then, approximately,

```
client: prog(W, K, M)
```

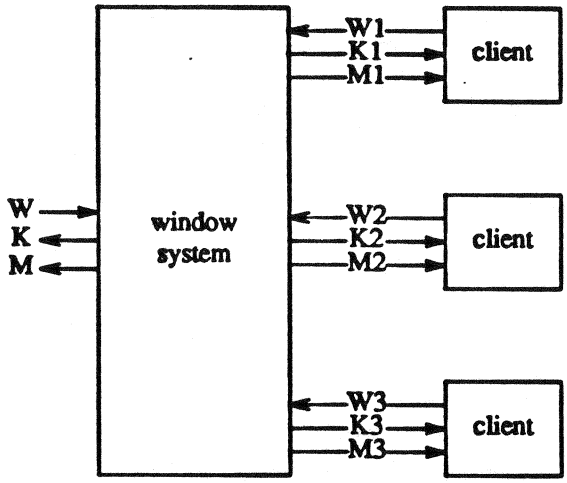
(The syntax used in this paper is based on Newsqueak, but should be mostly self-explanatory. I will explain unusual details.) The client can also be represented pictorially, using arrows to represent the flow of information.



The window system has several independently executing clients, each of which has the same external specification. It multiplexes a screen, mouse, and keyboard for its clients and therefore has a type reminiscent of the clients themselves:

```
windowsys: prog(S, K, M)
```

where `S` is the screen. If we arrange the client's windows to be programmable by the same interface as the full screen (making `S` a `W`), the window system will have the same type as its clients. Pictorially,



If the window system multiplexes clients of its own type, then it may be a client of itself, or a client may pass its environment to a fresh invocation of the window system to do further multiplexing. This recursion allows a client of window system — a text editor, say — to invoke the window system afresh in its window to manage subwindows for multiple views of files being edited.

To fill in this sketchy outline, we need to be more precise about the properties of `W`, `K`, and `M`.

Output

Two problems must be addressed by the output mechanism: how a client can draw in its window and how multiple clients can share the screen harmoniously. The choice of output model — bitmaps and bitblt, PostScript, display lists — is unimportant to the structure of the client, since any model can be implemented by a synchronous, procedural interface, a programming library. Replacing the library will not greatly affect the structure of the client or its environment. The system described here is based on bitmap graphics because that is perhaps the simplest model.

Bitmap graphics has been described before [Guib 82, PLR 85]. Briefly, a two-dimensional portion of memory, possibly but not necessarily visible on the display, is described by a data structure called a `Bitmap`. The data structure may be passed to several graphics operators, of which the best-known is the rectangular operator `bitblt` or `rasterop`, to effect changes to the memory and therefore to the display. Our first guess, then, will make `W` a `Bitmap`. But that ignores the problem of multiple clients with windows on the same screen.

The concept of 'layers' generalizes bitmap graphics so it applies to overlapping bitmap windows sharing a physical display [Pike 83a]. By extending the `Bitmap` type to encompass the properties of layers, the standard operators such as `bitblt` can be applied to partially or wholly obscured windows on the hardware display. Clients of a window system may remain unaware of each other, free to draw in their respective windows regardless of overlap and oblivious to changes in the visibility of their windows.

Freedom from these concerns simplifies the clients, but the layers model has some drawbacks and has not been widely adopted. It is modestly expensive in memory, maintaining off-screen backup memory for invisible portions of windows. But display memory has become much cheaper, and commercial window systems often require the clients to run with several hundred kilobytes of library, so this argument is unconvincing. (Some systems provide 'retained' storage for windows that request it, but that is not the default, for dubious arguments of efficiency and portability.) A more important criticism is that layers require some shared memory, atomicity, and synchronization. The window system must maintain a central data structure describing the configuration of the display, and the clients must not be writing to their windows when that data structure is being modified. That problem is easily overcome, however, using techniques standard in operating systems such as by making `bitblt` a system call, at least when it is operating on a window. Layers remain a viable solution to the windowing problem.

Other systems, their designers having accepted events *ab initio*, push the synchronization issues to their clients by sending them 'damage control' events when the visibility of their windows changes. It is understandable to force the programmer to use input events, but it is difficult to justify events tied to output, an otherwise synchronous, procedural interface. Not only do such events complicate the design, but they require all affected clients to run when the display is rearranged, which can cause considerable paging overhead and delay. The designer of a file system would never consider taking blocks away from a client's file and sending the client a message that the blocks need to be reconstructed. How did that behavior become acceptable in window systems? I think it developed as an inappropriate generalization of an event-based solution to a hard problem: what to do if the user asks a client window to change its size. Layers themselves offer no advice on how to solve this problem, but they also do not force any particular solution. A clean solution exists and will be explained below, in the section on 'Control', after some background has been presented.

In summary, output can be handled by passing the client processes a variable, `W`, of type `Bitmap` (extended to be applicable to overlapping windows) that the client may use to access its window using whatever graphics package is available.

Input

It remains to decide on the types of the variables `K` and `M` that provide the client with access to the multiplexed keyboard and mouse. The type of `K` is easily determined: it can be a synchronous channel that yields integral values identifying the key that has been pressed. It is analogous to a file descriptor on the UNIX system, from which may be read successive input characters. Since `K` will provide only keyboard data, no further specification is necessary; we need not distinguish keyboard data from mouse data, as the mouse information is available only through `M`. `K` does not provide events, it just delivers characters, synchronously, as they become available and are requested. If it is desired to track up and down transitions of the keys, the transitions can still be represented easily as integers. `K` is declared in Newsqueak as

```
K: chan of int;
```

that is, as a channel of integers.

The behavior of the mouse is harder to model. It has one or more buttons that go up and down and two dimensions of translational motion. The usual solution is to represent the mouse's behavior by a series of events: button down, button up, motion, motion while button down, etc. It is simpler instead to track the mouse by a series of synchronous messages reporting the entire state of the mouse.

The state of the mouse is represented by a data structure:

```
type Mouse: struct of{
    buttons: int;
    position: Point;
};
```

The variable `buttons` has a bit set for each button that is depressed, and the position is held in a `Point`, a type that is already part of the graphics library:

```
type Point: struct of{
    x,y: int;
};
```

The actions of the mouse are reported on the channel `M`, defined as

```
M: chan of Mouse.
```

Each time the state of the mouse changes, the full state is made available on the channel `M`. If the mouse is idle, reads from `M` will block.

A programmer accustomed to events might argue that the synchronous way of handling the mouse is awkward. If the program is waiting for some genuine event such as a button transition, then decoding the complete state of the mouse to look for the transition might seem harder than just waiting until the specified event happens. But before making that decision, we should look more closely at how the

mouse is programmed. For simple cases such as waiting for a single button event, it makes no practical difference; the loop

```
do
    e=getevent();
while(e.type!=LEFTBUTTONDOWN)
```

is equivalent to

```
do
    m=readmouse(M);
while(m.buttons&LEFTBUTTON).
```

Imagine, though, that a more complicated condition is to be met, such as the left button being held down while the mouse is inside some rectangle. The proposed interface solves the problem well:

```
do
    m=readmouse(M);
while(not(m.buttons&LEFTBUTTON and pointinrect(m.position, r))).
```

The full programming language may be used to specify the condition. Returning the entire state of the mouse when it changes allows the mouse to be programmed as if it were being polled, which is probably the simplest way to write mouse software. An event-based interface would instead have to be programmed to reconstitute the state so the condition may be tested. Why provide a complex interface when the programming language can already handle all the complexity required? Events add unnecessary complexity to the problem of interpreting the mouse. But we can only avoid that complexity when we can write code to read the mouse channel independently of the code that handles the keyboard; otherwise, the event mechanism is the only way to collect both mouse and keyboard actions. To keep those two tasks separate, we need processes.

Concurrency and multiplexing

Here is the declaration of the type of clients of the window system:

```
type Client: prog(W: Bitmap, K: chan of int, M: chan of Mouse).
```

Any program of type `Client`, including the window system itself, can be run in a window.

A client program is started by the window system by creating a layer and channels for the keyboard and mouse and then by calling the client program as a separate process. In Newsqueak this is written

```
begin client(newS, newK, newM).
```

The window system records the values of the new channels and can then use them to send keyboard and mouse data to the client. The clients execute independently and concurrently and may be (and likely are) themselves composed of concurrent subprocesses. The clients are true processes, not coroutines; their execution is finely interleaved, and not just switched at I/O time as in mpx or NeWS [Pike 83, Sun 87]. No client can completely dominate the system, even if it is in a tight loop without I/O. The user interface of the window system or its clients does not block because a client is busy.

The channels that carry messages between processes in Newsqueak, and hence within the window system, are synchronous, bufferless channels as in CSP or Squeak, but carry objects of a specific type, not just integers [Hoare 78, Card 85]. To receive a message on a channel, say `M`, a process executes

```
mrcv = <-M
```

and blocks until a different process executes

```
M<- = msend
```

for the same channel. When both a sender and a receiver are ready, the value is transferred between the processes (here assigning the value of `msend` to `mrcv`), which then resume execution. Except that the channels have type, this sequence is exactly as in CSP.

The client must obey a simple protocol to function properly in the system. Because all

communication is synchronous, the client must always be ready to receive keyboard and mouse data. If the window system is sending data to a client that is not receiving, the window system will block and other clients will be cut off from their inputs. This belies the statement made earlier about busy clients not dominating the system, but it is easy to arrange for clients to be well-behaved, and in practice the system does not block. The clients are typically written as concurrent processes, one reading the mouse, another the keyboard, and others managing the display. These various processes communicate using internal channels. A complete client that connects an operating system's command interpreter to a window, including all processing of keyboard input such as echoing, typing correction, and so on, takes about 100 lines of Newsqueak, distributed across three processes (keyboard, mouse, and display) and the external command interpreter.

The window system itself is also written in Newsqueak, unlike other language-based systems such as NeWS. The window system is little more than a multiplexer that creates windows and runs clients in them. Its only user interface is that to create, delete, select, and rearrange windows. (How to delete and rearrange windows is explained in the next section.) The main program is a single process that accepts the usual set of parameters `W`, `K`, and `M`, and uses the mouse to select which window receives keyboard and mouse data. When buttons are pressed with the mouse not above any client, the window system activates its own functions. Otherwise it passes keyboard and mouse data to the appropriate client.

The structure of the multiplexing subroutine is straightforward. It maintains an array of data structures describing the environments of its children:

```
type Env:struct of{      # encapsulated world of a window program
    W:  Bitmap;          # screen/window
    M:  chan of Mouse;   # mouse
    K:  chan of int;     # keyboard
};
env: array of Env;
```

The subroutine is a loop that uses Newsqueak's selection control structure, much like the selection operators in CSP and Squeak, to wait for I/O. When keyboard or mouse information is sent, the system decides which client should receive the data, passes the data on, and waits again.

This sounds very much like the loop described in the quotation at the beginning of this paper. The main difference is that none of the software needs to be written as state machines. The problems have been decoupled, and the individual components — window multiplexer, clients, and mouse and keyboard handlers for the individual clients — can execute concurrently, independently, and without explicit state. Simpler software results. The multiplexing subroutine, the heart of the window system, is about 60 lines long. Graphics and layering operations are provided atomically by a built-in Newsqueak library implemented in C.† Another 100 lines or so is used for ancillary functions such as interpreting mouse motion to define the location of new windows. The complete window system, capable of running recursively, including the command interpreter client, is fewer than 300 lines of Newsqueak. With the ability to delete and rearrange windows (which involves more complicated control, discussed below), it is still under 450 lines. It takes more space just to define the standard data structures for the 33 event types in X11 [Xlib 88].

An unusual property of the multiplexer is that, because the interface to the clients is well-defined, it knows nothing about the clients themselves. This allows it to multiplex arbitrary programs that satisfy its protocol, including itself. It may thus be invoked again by a client to do submultiplexing within its window. (A small amount of easily arranged protocol is required to initialize the system with the definitions of the functions to be multiplexed.)

Something conspicuously absent here is the ability to load new clients while the system is running. The major hurdle is linguistic — it is hard to run arbitrary programs in a statically typed language — but soluble. Loadable clients will have little affect on the size or structure of the system, and I expect to have them running when the final version of this paper is prepared.

---

† The layer library comprises 364 lines of C, not counted here.

Control

A client needs to be able to tell the window system that it is exiting. Given communication channels, our solution is obvious; just provide a channel from the client to the system, on which it can announce its departure. For generality, this 'control' channel, C, can hold character strings (arrays of characters in Newsqueak):

```
type Client: prog(W: Bitmap, K: chan of int,
                  M: chan of Mouse, C: chan of array of char).
```

The definition of type `Client` is getting more involved. We can tidy it up a bit by borrowing the `Env` data structure from the main loop of the multiplexer. And we can prepare for the future by providing a pair of control channels, one each way:

```
type String: array of char;
type Env:struct of{     # encapsulated world of a window program
    W:  Bitmap;         # screen/window
    M:  chan of Mouse;  # mouse
    K:  chan of int;    # keyboard
    CI: chan of String; # control messages in
    CO: chan of String; # control messages out
};
type Client: prog(Env);
```

When a client wants to exit, it shuts down internally and as its last action asks the window system to delete its resources:

```
client: prog(e:Env){
    do
        things();
    while(notdone());
    e.CO<- = "Delete";
};
```

Communication is synchronous, so, at the instant the window system receives the client's "Delete" message, it knows the client is gone, and it can shut down connections to the client. A similar argument shows that the system should never block because a client is trying to exit.

What happens when a user of the system wants to delete a window? The usual method is to send an event or, worse, an asynchronous poisonous message (a 'signal' in the UNIX system) to the client, terminating it suddenly. This brutality is avoidable; the system can just notify the client, using the same synchronous methods, that it is being asked to exit. When the client is ready, it can exit by the same method as above. In other words, instead of the client being killed, it can be asked to leave. (Part of the protocol of the client is that it must soon honor the request.) This protocol works well for recursively instantiated windows. When a window system is asked to exit, on its `CI` channel, it turns and asks its clients to exit on their `CI` channels. When they have all gone, it then reports that it is done on `CO`. Initiates of concurrent programming will recognize a potential problem here: there is a communication loop from system to client to system, and that is a source of deadlock. The deadlock can easily be avoided by marking each client so that it is sent only one delete message.

The same logic can be applied to the other control problem, which is how to change the location or size of a window on the screen. Again, the usual solution is to change the client's size and then abruptly to notify it of the change. Instead, as with delete, we can install a protocol so the client may ask the system to change its size for it, then add a message in the other direction so the system can ask a client to request a change. This requires adding a rectangle to the messages on `CI` and `CO` but is otherwise as clean, synchronous, and easy to use as the deletion protocol. Here is the final definition of type `Client`.

```
type String: array of char;
type Msg:struct of{
    s: String;          # "Delete", "Move" or "Resize"
    r: Rectangle;       # ignored for Delete
};
type Env:struct of{     # encapsulated world of a window program
    W:  Bitmap;         # screen/window
    M:  chan of Mouse;  # mouse
    K:  chan of int;    # keyboard
    CI: chan of Msg;    # control messages in
    CO: chan of Msg;    # control messages out
};
type Client: prog(Env);
```

Critics might complain that events have not been banished after all, that the delete and resize messages are events. But the intractability of event-driven interfaces has been overcome: delete and resize messages are synchronous; all events associated with the mouse have been eliminated, replaced by synchronous reads; and events reporting changes of visibility of a window are unnecessary. Most important, though, the structure of the system — a set of concurrent processes communicating on synchronous channels — makes the control of multiple complex inputs decomposable into small, easily understood components whose design is chosen by the programmer, not the interface. Even in a system where events are unavoidable, that approach makes them easier to manage.

Status

The programming language Newsqueak has been fully designed and an interpreter for it written [Pike 89]. Using the interpreter, I have written a window system that demonstrates the viability of the synchronous design. The complete recursive window system, including the definition of one non-trivial client, is just a few hundred straightforward lines of Newsqueak. The window system is usable as a front end to a UNIX system; the functionality it provides is comparable to the standard X window manager or mpx [Xlib 88, Pike 83]. In its ability to run recursively, it offers a unique property. This is no mere trick; it can be used, for example, to run multiple windows on a single connection to a remote CPU server, although that requires some operating system support beyond the scope of this paper. The window system's main lack is that it must be restarted to link in new clients, but that restriction will soon pass.

Although the design is easily implemented in Newsqueak — the language was built for the task — it can be put together in more traditional environments. Any system that allows synchronous message passing and multiplexing can be used to construct a synchronous window system. The interprocess communication tools in most UNIX systems, particularly pipes plus the `select` or `poll` system calls, are sufficient to implement this design [UNIX 4BSD, UNIX SYSV]. Although such a system would involve substantially more code than the Newsqueak version, it would still be conceptually much simpler than an event-driven window system.

Conclusions

Window systems are not inherently complex. They seem complex because we traditionally write them, and their client applications, as single processes controlled by an asynchronous, event-driven interface. We force them into the mold of real-time software. They can be simplified greatly by writing them as multiple processes, each with individual, synchronous communication channels on which it receives data and control information. It is possible to write a complete, useful window system, comparable in basic power to commercial systems, using just a few hundred lines of code in a concurrent language. Even in traditional languages, simplicity can be achieved by replacing event-driven interfaces with synchronous interfaces and some easily-provided multiplexing functions.

References

[Card 85] Cardelli, L., and Pike, R., "Squeak: A Language for Communicating with Mice," *Computer Graphics*, 19(3), pp. 199-204, 1985

[GKS 84] *Draft Proposed American National Standard Graphics Kernel System, Computer Graphics, Special GKS Issue*, Feb. 1984

[Guib 82] Guibas, L. J., and Stolfi, J., "A language for bitmap manipulation," *ACM Trans. on Graph.*, 1(3), pp. 191-214 (1982).

[Hoare 78] Hoare, C.A.R., "Communicating Sequential Processes," *Comm. ACM* 21(8), pp. 666-678, 1978.

[Pike 83] Pike, R., "The Blit: A Multiplexed Graphics Terminal," *AT&T Bell Labs Tech. J.*, 63(8), part 2, pp. 1607-1631, 1983

[Pike 83a] Pike, R., "Graphics in overlapping bitmap layers," *ACM Trans. on Graph.*, 2(2), pp. 135-160, 1983.

[Pike 89] Pike, R., "Newsqueak: A language for communicating with mice," *Computing Science Technical Report 143*, AT&T Bell Laboratories, Murray Hill, New Jersey, 1989

[PLR 85] Pike, R., Locanthi, B., and Reiser, J., "Hardware/software trade-offs for bitmap graphics on the Blit," *Softw. Pract. Exp.*, 15(2), pp. 131-152, 1985.

[Rose 88] Rosenthal, D., "A Simple X11 Client Program," *USENIX Winter Conference Proceedings*, pp. 229-242, Dallas, 1988

[Sun 87] *NeWS 1.1 Manual*, Sun Microsystems Inc., Mountain View, California, 1987

[UNIX 4BSD] *Unix Time-Sharing System Programmer's Manual, 4.3 Berkeley Software Distribution*, University of California, Berkeley, Calif. 1986.

[UNIX SYSV] *System V Interface Definition, Issue 2, Volume III*, pp. 319-321, AT&T, Summit, New Jersey 1986.

[Xlib 88] Scheifler, R. W., Gettys, J., and Newman, R., *X Window System: C Library and Protocol Reference*, Digital Press, Bedford, Massachusetts, 1988.

The window system in action. The grey rectangles are instances of the window system. The white rectangles support command interpreters for the operating system. Several instances of a graphics demo are also visible. The grouping of the windows reflects their structure; the innermost windows are 5 environments deep.

Appendix I

The following is the complete code for a client that copies keyboard characters to its screen, and exits when deleted or resized, or when a mouse button is pressed. Notice that shutdown is the same regardless of how it is begun. The names not defined here are part of the programming environment provided for the client. Their meanings should be self-evident. A plain equals sign = is the assignment operator; := creates and initializes storage.

```
include "windowsys.h"

keyboardslave:=prog(e: Env, done: chan of int){
     a: array of char;
     p:=addpt(e.W.r.min, {5,5});
     for(;;)
          select{
          case <-done:
               break;  # out of loop
          case a=<-e.K:
               p=string(e.W, p, defaultfont, a, SRC);
          }
};
mouseslave:=prog(e: Env, done, telldone: chan of int){
     m: Mouse;
     told:= 0;
     for(;;)
          select{
          case <-done:
               break;  # out of loop
          case m = <-e.M:
               if(m.buttons && told==0){    # tell only once
                    telldone<- = 1;
                    told = 1;
               }
          }
};
client := prog(e:Env){
     Mdone := mk(chan of int);          # to slave
     Mtelldone := mk(chan of int);      # from slave
     begin mouseslave(e, Mdone, Mtelldone);

     Kdone := mk(chan of int);          # to slave
     begin keyboardslave(e, Kdone);

     select{                            # wait for message from above or below
     case <-e.CI:                       # delete or resize; exit either way
          ;
     case <-Mtelldone:                  # mouse slave says exit
          ;
     }

     Mdone<- = 1;                       # tell slave to exit
     Kdone<- = 1;                       # tell slave to exit
     e.CO<- = {"Delete", W.r};          # and tell window system we're exiting
};
```