CMU-CS-86-118

FLAMINGO: Object-Oriented Window Management for Distributed, Heterogeneous Systems

Edward T. Smith and David B. Anderson

Computer Science Department
Carnegie-Mellon University
Pittsburgh
PA 15213

April 1986

Abstract

This report describes the Flamingo User Interface System (Version 15). Flamingo is a system for managing the interface between users and programs that run in large, distributed, heterogeneous computing environments. Using the mechanisms described herein, Flamingo provides a set of user interface features associated with traditional window management.

Flamingo uses an object-oriented structure whose objects can have methods (or "operations") implemented in remote processes. This mechanism differs from the traditional "user/server" relationship that is used to structure many distributed systems. In Flamingo, the system is a central "object manager", while client programs running as remote processes provide the implementations for methods called upon by Flamingo and other clients. Both the clients and Flamingo act as servers and users of each other.

Flamingo is built on the Mach operating system, which provides a UNIX environment plus a message-based Interprocess Communication (IPC) mechanism. Flamingo uses a machine-generated Remote Method Invocation (RMI) mechanism to provide a symmetric interface between it and client programs that wish to call on method implementations located in each. The Remote Method Invocation system itself uses a machine-generated Remote Procedure Call mechanism as a message transport layer.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

Table of Contents

1. Introduction

Flamingo[1] is a system for building user interfaces to programs running within the Spice environment. The Spice environment [CMUSpiceProject 79] consists of a heterogeneous set of machines, typically large personal workstations (along with mainframes, multi-processors, etc.), each of which supports the Interprocess Communication (IPC) message passing model as described in [Rashid 81]. This mechanism provides a transparent, language- and machine-independent means of communication between processes and all system resources they may need (including screens, keyboards, pointing devices, file systems, and other processes). The current implementation of Flamingo is written in C under Berkeley UNIX[2] 4.3 running on a MicroVax[3] II. Our version of this UNIX, called Mach [Baron 85], has been modified at CMU to fully support the IPC mechanism, and includes other features to support an efficient communicating multi-process environment.

Flamingo is primarily designed to support the abstraction of a *virtual graphics device* that we intend to use for developing window systems. Flamingo itself only provides rudimentary features to support traditional window management. The implementation of a virtual graphics device has allowed for ease of porting other window managers and graphics packages to Flamingo so that application programs written in other domains can run without modification in Flamingo. Providing upward compatible user interface support to all our older software was a primary goal with this work. Our computing environment currently consists of about two hundred machines representing several different manufacturers. We have a large and growing base of software written on all these machines that complicates the ability to port software as new machines appear. Often in fact, some software is simply not suitable to port, since some special machine may still perform a service that other machines are not capable of performing. Also, programmers accustomed to using features available only from their favorite graphics package want to continue using these packages to write software and know that they will not be excluded from future changes in architectures or graphics support.

Flamingo provides a mapping between its virtual graphics device and either other virtual graphics devices or various types of hardware. Each virtual graphics device provides an implementation for each of the operations necessary to define the device in terms of the mapping it supports. Flamingo should eventually allow users to sit at any of the computers supported and run programs on any available workstation.

A goal that has guided much of our design has been to simplify the task of implementing and modifying user interfaces and window managers. As we surveyed the needs of the researchers in our group, we found our users wanted to be able to approach a display manager at many different levels, and to modify or replace components of the system without being required to understand the details of an entire system. Our researchers want to take advantage of our computing environment's powerful workstations, message-passing operating system, distributed file system, and large software base, but have been unable to easily modify and extend the existing user interface system to suit their particular needs. In general, buying into any particular window manager also means getting a particular style of human/computer interaction and a particular program-to-window-manager interface.

With Flamingo we intend to define exactly those interfaces for our various systems components in such a way

[1]FLexible, Asynchronous Manager for Interactive Network Graphics Operations!

[2]UNIX is a trademark of AT&T Bell Laboratories.

[3]MicroVax is a trademark of Digital Equipment Corporation.

that we support our researchers more than we support some particular style of interaction. No particular style or set of operations has proven itself capable of answering all the desires of its users in this respect, so we set out to work on this problem for users in our environment.

An ultimate goal of Flamingo is to support research into more intelligent systems that make use of a highly distributed parallel environment. With a system as Flamingo in place providing a user interface for distributed programs (written with a variety of window managers and graphics packages, and able to interact with a variety of machines), we will be prepared to investigate the problems with automatic, intelligent mechanisms for managing distributed applications.

2. Classical Window Management

Window systems and window managers have become a common software tool provided for users of powerful workstations that include large raster display screens, pointing devices, and enough personal computing power to support a medium-size time-shared operating system. Window managers typically provide particular interaction styles and particular mechanisms for handling input and output between client programs and the user. Figure 2-1 shows the typical communication structure between application programs and a particular workstation using a window manager.

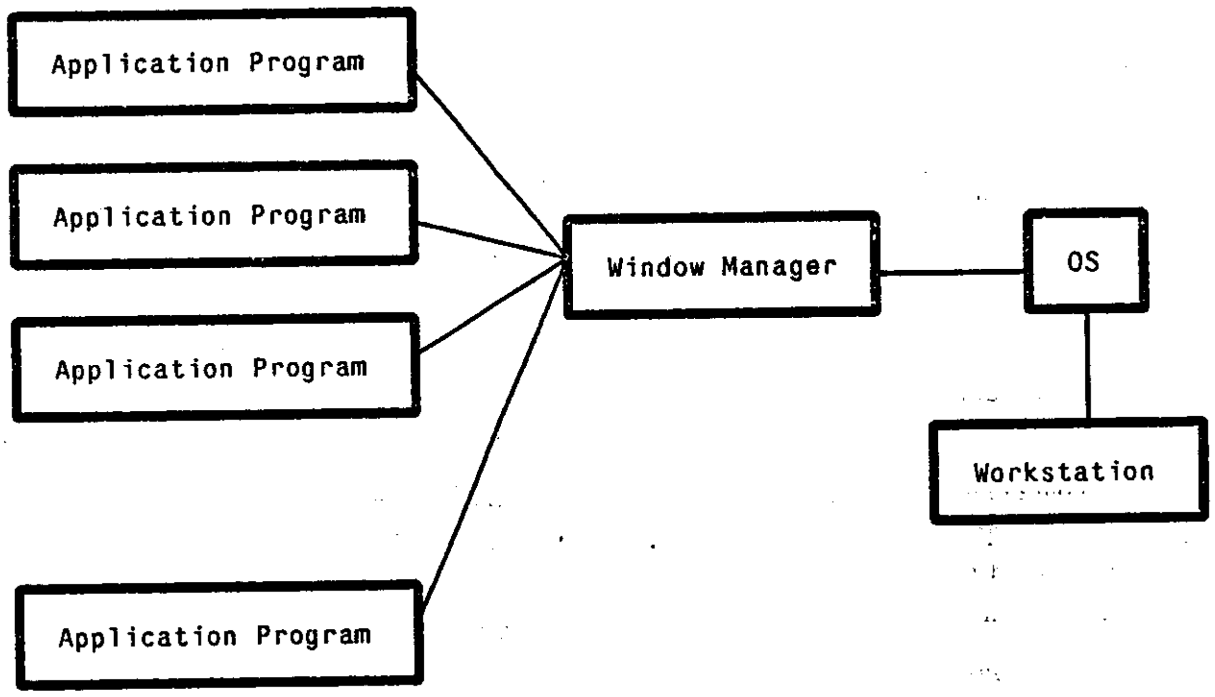

Figure 2-1: Classical Window Management

Window managers such as the Sapphire [Sapphire 84] system from Perq Systems, the Andrew [Gosling 84a, Gosling 84b] window manager from the ITC of CMU, the Macintosh Finder from Apple [Apple 85], and so on, export their own particular style of interaction. Adding features to these systems or changing features for particular instances of window objects or for entire classes of objects is exceedingly difficult even when the source code for such systems is available, and nearly impossible if the source code is unavailable. In particular, these systems tend to be oriented primarily toward people who are end users of programs running under them, and secondly toward those who wish to write programs with them. They rarely (if at all) support those who wish to modify the model of window management presented by the system for all applications.

Project Athena's X [Gettys 86] system separates the implementation of the window manager support software from the window manager itself. This allows for different styles and protocols to be implemented and supported easily within one environment on one machine. Application programs either communicate to the window manager of their choice, directly to X, or to a combination of both.

Output styles vary according to the desires of authors of the systems. For example, overlapping windows have a large following, though some systems, notably Xerox PARC's Cedar System [Teitelman 85] and the Andrew system, have used screen tiling as their primary screen allocation mechanism. Input styles also vary, with systems making different use of keystrokes or mouse movement to direct and control input to processes.

New application programs, new windowing techniques, differing desires of users, and new I/O devices all drive the development of new mechanisms for input and output. Flamingo is intended to support the development of window managers, or more precisely, the development of the systems that provide high-level graphical abstractions to their clients in a machine-independent fashion. This should help support the development of new system input and output styles rather than support the promotion of some particular style.

3. User Interface Management Systems

Many current systems take an approach that makes a clear separation between client programs (that are relatively user-interface-independent) and the user interface itself (the so-called User Interface Management System (UIMS)). This approach has been highly successful in moving code out of ordinary programs that was often redundant with code in other programs to do user interface functions, or (worse) was inconsistent with the user interfaces of many common programs. A UIMS has the advantage that the client writer does not have to write the user interface code of his client, and can take advantage of a package of routines implementing a "standard" user interface.

The simplest form of user interface management is exemplified by the UNIX style of client/user separation. In this model, the UIMS does not really exist at all. Clients and users communicate with streams of characters implemented through thea file system. Another mechanism (called "pipes") allows output files of one process to be connected to the input files of another. Physical terminals connect process output to a screen, and generate process input with characters from a keyboard.

Another simple but non-trivial example of a UIMS is the Virtual Terminal Management System implemented on top of the RIG operating system [Lantz 79]. This system provided a powerful layer of abstraction between programs that communicate with character streams, and the actual input and output of the streams with the user. Many virtual terminals could be created for the user and were managed by the system and controlled by the user with special keyboard commands. Later work by Lantz has extended the power of the local workstations to allow better separation between client processes on remote machines and the user's personal workstation [Lantz 84, Lantz 85]. This latter work emphasizes structured display files to minimize the effective transmission of information between the client programs and their workstations.

The Cousin system [Hayes 85] demonstrates a highly structured approach to communication between the

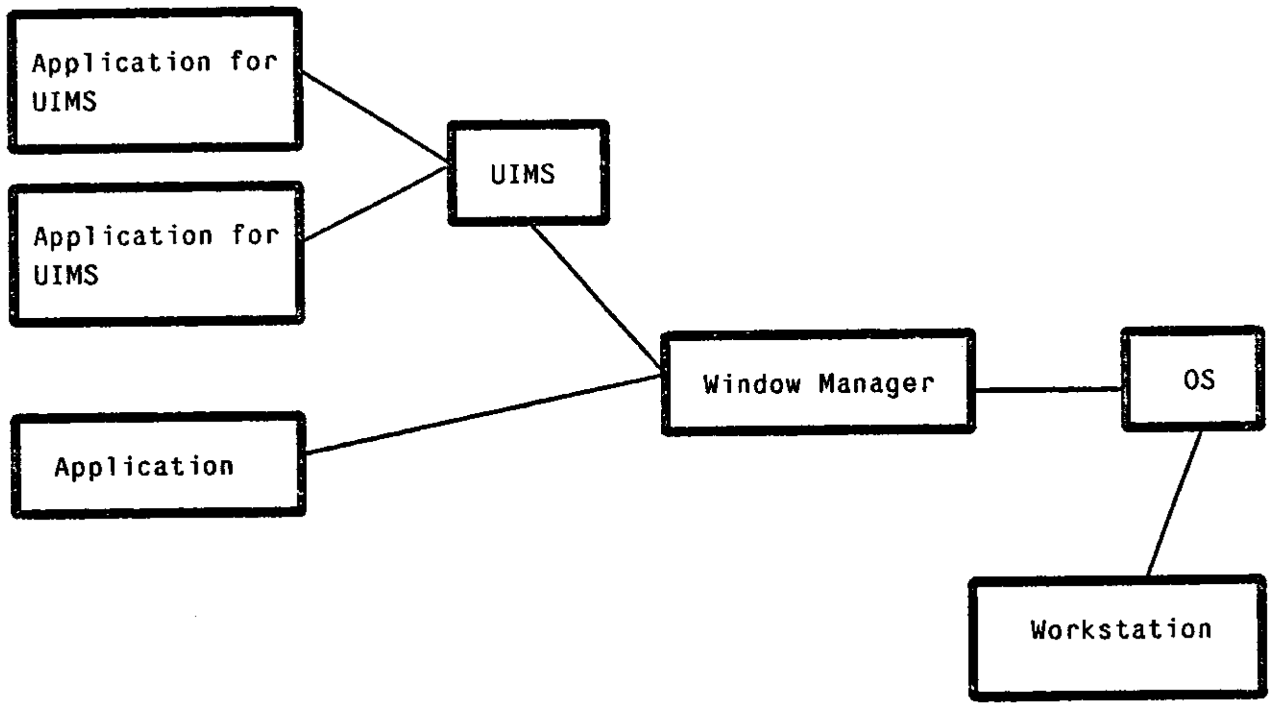

Figure 3-1: User Interface Management System

client program and a UIMS. The Cousin UIMS provides a set of parameterizable routines for mainpulating specific objects such as menus, file names, lists of file names, "buttons" (or "operations" on objects), and so on. A client writer creates a client that communicates with the UIMS via highly structured "forms" or records of information. Clients have a set of forms they can send to and get back from the UIMS. These forms describe the operations available for the client and define the information that must be passed for the interface to the user. The UIMS is responsible for managing the display and interacting with the human user.

The Descartes system [Shaw 83] takes a more formal approach to the separation of the UIMS from the client. (Descartes' roots are in research on strongly-typed programming languages, rather than in user interfaces.) Descartes' view of input and output devices is that they are generators/consumers respectively of data of particular abstract data types. The operations involved in converting these data from the input and output types to types useful in a client program is exactly the type conversion that occurs in a strongly-typed programming language between two variables of different types. Parameters to this conversion process are recognizable in other programming languages as format statements or descriptions. Descartes specifies formally when such conversions take place, where the conversion parameters (formats) can be found to control this conversion, and how the actual input and output of information of the input and output data types occurs.

Other systems have been described in the literature that provide varying levels of abstraction for the client programs and thus varying amounts of separation between the client programs and the user, for instance [Buxton 83, Guest 84, Heffler 82, Kamran 83]. Many of these systems merely provide packages of subroutines for defining such canonical features as windows, menus, icons, mouse cursors, and so on, so that the programmer has less to worry about when trying to imitate a particular style.

When we consider this space of UIMS's, we find that, for the most part, they lie at various points across a spectrum from the relative absence of a UIMS represented by UNIX, through systems which provide a subroutine library of input and display routines, such as the Macintosh[4] [Apple 85], or Smalltalk 80 [Ingalls 80, Goldberg 83], to systems like Cousin and Descartes that separate applications widely from the user interface routines.

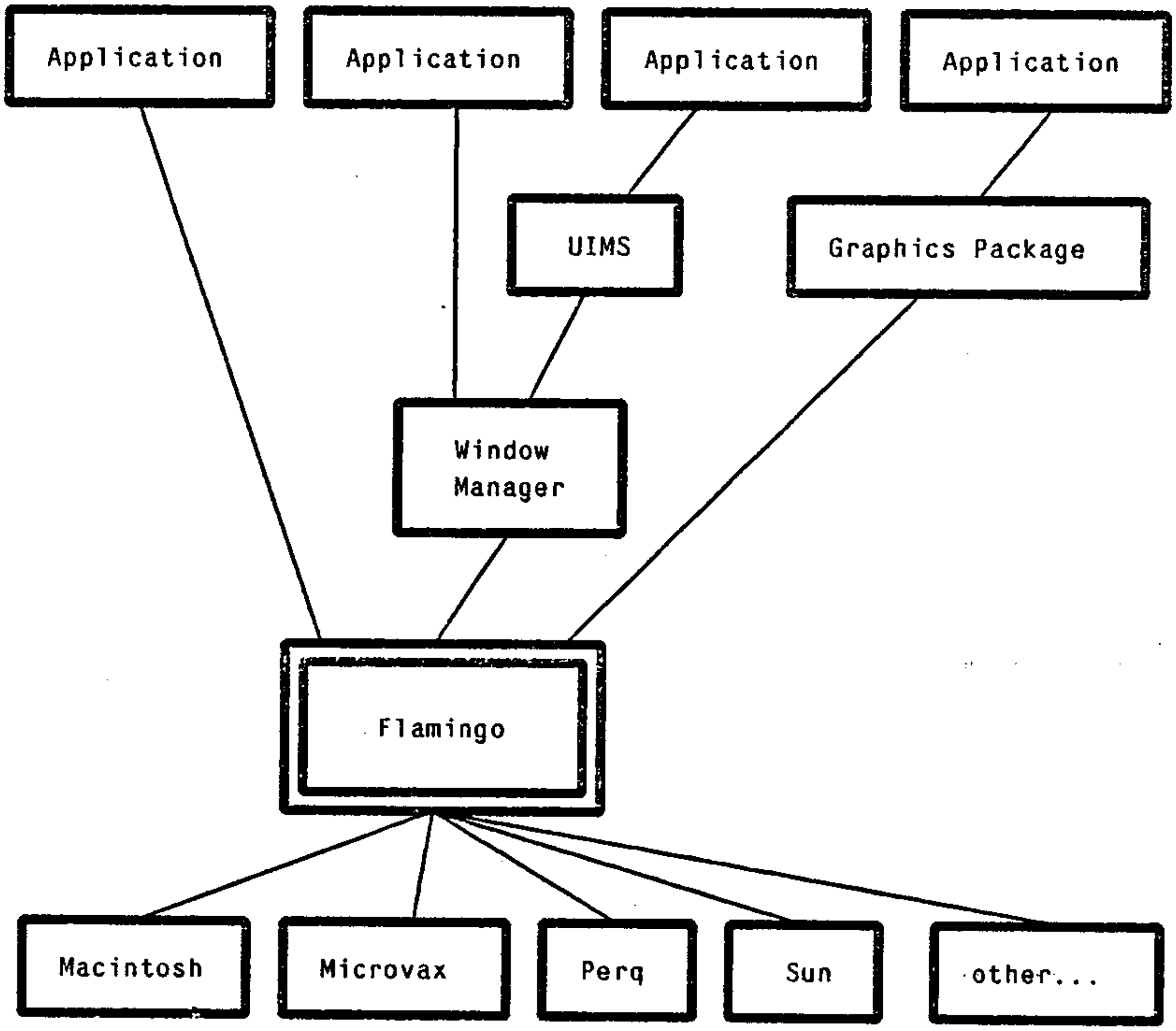

Figure 3-2: Overall View of Flamingo

In contrast to these systems, Flamingo can be considered to lie at several different points at once across this space. Flamingo's objects can be extended via the class mechanism to include as many operations as are necessary to implement a particular type of user interface. New objects of these classes export to applications an interface to the UIMS that is implemented by the routines of the class. Or, on the other end of the spectrum, applications can use operations obtained from the lowest level virtual graphics devices.

[4]Macintosh is a trademark licensed to Apple Computer, Inc.

Flamingo provides low-level virtual graphics support through objects that are mapped to physical screens or other virtual graphics devices. Multiple applications, multiple window managers and multiple graphics packages can coexist within a system that communicates with multiple types of physical hardware. Figure 3-2 shows this view of Flamingo.

4. Flamingo's Object-Oriented Graphics Structures

Flamingo is implemented with a hierarchical, object-oriented design. The Flamingo system exports various objects related to input and output of information between the client program and the user. For input, the primary object is the *input device*, which includes input device state information plus methods for handling input events. For output (by client programs), the exported object is a *pixel array*. A pixel array (PA) is a 2-dimensional region with a shape defined by a *mask*, and a list of *mappings*, which map the pixel array to other PAs, to memory, or to the screen. Several *classes* of pixel arrays are provided. Lower levels of the system export pixel arrays (PAs) with implementations of methods for raster operations, character drawing, and so on, while higher levels export these operations as well as window management functions, input hooks, etc.

A *mask* is an object defined for use by both input and output objects. Masks are used throughout Flamingo wherever there is a need to represent *shape*: the shapes of pixel arrays, mouse sensitive areas, clipping regions, and the uncovered portions of overlapped windows, are all represented with masks.

Each of Flamingo's graphics operations takes one or more masks as parameters to specify the clipping that is to be performed to the source and destination arguments. For example, our bitblt operation, which we call rasterop, has this signature:

```
    mask_rasterop(srcPA, srcMask, sx, sy, x, y, dstPA, dstMask, dx, dy, op)
```

The source bits for this operation are those bits contained within the shape specified by the source mask when it is mapped to the location `(sx, sy)` within the source pixel array. The destination bits are determined in an analogous fashion, and the source bits are mapped to the location `(x, y)` within the space of the destination pixel array, where the raster operation is performed. (One might think of the source and destination masks as 2-dimensional bitmaps that are logically ANDed into the source and destination pixel arrays, where the ones in the masks denote those bits in the source and destination that participate in the operation.)

Masks are similar to the *regions* used in Quickdraw, the Apple Macintosh graphics package [Apple 85], in that they can be used to represent any arbitrary shape, but are implemented in a way that is more portable and scales better to large displays. Internally, masks are represented as a list of rectangles, organized by scanline. This representation permits a compact encoding of the shapes which typically arise from overlapping rectangular windows, and also allows efficient coding of the methods for masks that we have found useful: intersection, union, difference, and the conversion of a mask to a list of rectangles. This latter operation has allowed us to take advantage of much existing graphics code to perform rectangle-clipped rasterops, and line and character drawing, etc., without having to modify this code (often written in assembly language or microcode) to clip to arbitarily shaped regions.

5. Methods, Implementations and Classes

Flamingo operations are implemented as a set of *methods* for each object exported by the system. Each instance of a Flamingo object has a list of pointers to *implementations* for the methods available for that object, and a pointer to the object's parent (or class). Any given method, such as NewPA, RasterOP, or DrawLine, may have several implementations. For example, the window manager has its own implementation of NewPA that creates lower-level pixels arrays for the body, sides and corners of the window, and associates these together in a private structure that represents the window. Figure 5-1 shows how the illusion of a simple-looking window is created by a stack of pixel arrays. Note that the shape off the middle pixel array is not connected. Masks are general enough data structures to represent disconnected shapes. Also note that the mask itself only determines the surface area but not the color of the bits. Shown in figure 5-1 is a background pixel array painted black, the user's pixel array painted white, and a corner pixel array painted white with boundary lines for the user's area.

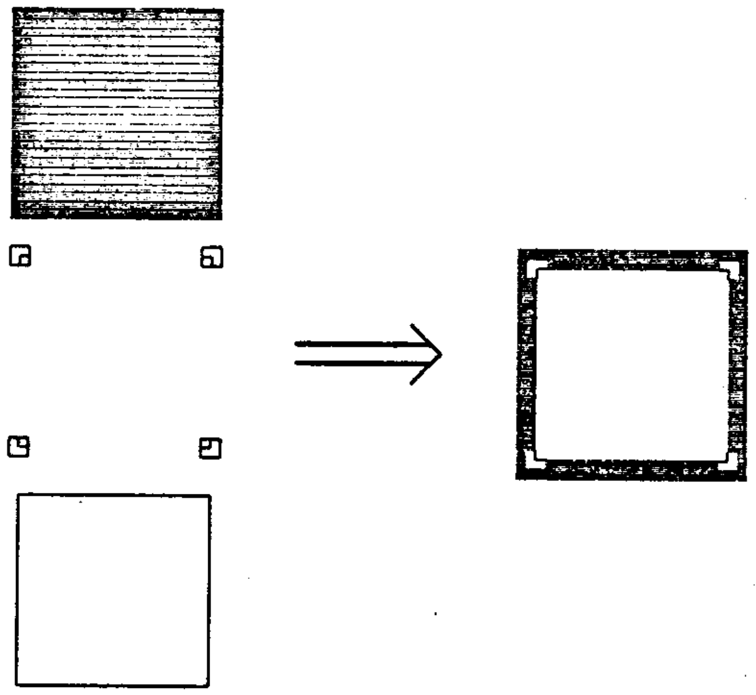

Figure 5-1: Flamingo Windows

The basic data types used throughout Flamingo are shown below. For each type we list the modules that export that type (module name in parentheses), and a description of the method implementations added by each module.

- *Mask* **(mask)**: mask intersection, union, difference, etc.
- *Input Device* **(flim)**: machine dependent input device drivers
- *PA*
    - o **(flam)**: machine dependent output primitives; rectangular raster operations, unclipped string and line drawing
    - o **(fligraph)**: machine independent graphics; raster operations, string and line drawing over

masks

- o (**coverup**): overlapping, mapped pixel arrays
- o (**frawd**): window management functions

New objects are always created through reference to an existing object, called the *parent*. The system is bootstrapped by creating special objects, named `ClassMask`, `ClassInputDevice`, `ClassPA`, `ClassCoverupPA`, and `ClassWindow`, that serve as representative members of their respective classes. Subsequent instances of these classes inherit their methods from these class objects.

Usually, any given object inherits its implementations from several classes, its parent, its parent's parent, and so on. For instance, a pixel array used with a window manager has methods associated with just being a pixel array, with being part of a window, with its ownership by a client process, and with receiving input. Thus each class of pixel arrays is built on top of a more primitive class of pixel arrays, inheriting some of the methods from that lower level, and re-implementing others. In many cases this re-implementation is accomplished through a *super* construct, borrowed from Smalltalk [Ingalls 80, Goldberg 83], whereby a particular method's implementation invokes the parent class' method.

The implementation of the method-calling mechanism is modeled on the **:before** and **:after** daemons found in Flavors [Keene 85]. Before a method is actually called, a set of *before* methods is called, if there are any. Next the method itself (sometimes referred to as the *during* method) is called, followed by the *after* methods, if any. In addition to all the other parameters, *after* methods are passed the value returned by the *during* method, if any. Note that any of the implementations of these *before*, *during* or *after* methods may be local to the calling method's process, or remote and accessed across a process boundary.

A special method called `SetHook` is used to add new methods or to replace old ones in any instance of an object. `SetHook` can be used with the class objects to replace a method for an entire class of objects. Presently our inheritance uses copy semantics: when an object is created, the methods of the parent object are copied into the new object. What this means for replacing class methods is that the only objects that can inherit the new method are those created after the new method has been set.

6. An Example

As an example of how this system structure actually works, we consider what happens when a Flamingo application performs a rasterop operation within one of the application's windows. The user's call has this form:

```
    RasterOP(srcPA, srcMask, sx, sy, x, y, dstPA, dstMask, dx, dy, op)
```

The source bits for this operation are those bits contained within the shape specified by the source mask when it is mapped to the location `(sx, sy)` within the source pixel array. The destination bits are determined in an analogous fashion, and the source bits are mapped to the location `(x, y)` within the space of the destination pixel array, where the rasterop operation is performed. (One might think of the source and destination masks as 2-dimensional bitmaps that are logically ANDed into the source and destination pixel arrays, where the ones in the masks denote those bits in the source and destination that participate in the operation.)

Several different system modules export raster operations: PAs exported by **flam, fligraph, coverup** and **frawd** all have a method for rasterop. For this example, we will assume that the PA inherits (has a copy of) the methods of `ClassWindow`, implemented by **frawd**. The window manager, **frawd**, inherits its rasterop method from **coverup**. Initially, then, when user level code calls `RasterOP` inside a window, it gets the rasterop procedure within **coverup**, providing as arguments the source and destination pixel arrays and masks within those pixel arrays specifying the region to be copied and the destination's clipping region.

Coverup is responsible for maintaining mappings that connect higher-level pixel arrays to lower-level pixel arrays. Through these mappings, **coverup** implements a 2 1/2 dimension space of pixel arrays. These pixel arrays have memory backing them, so that the user can view a **coverup** pixel array simply as a surface for drawing, without any concern for overlap.

The mappings maintained by **coverup** can be as simple or as complex as necessary to implement the particular mapping abstraction at hand. For the current implementation, a set of mappings is defined for the graphics operations that takes into account the *rank* or height of a mapping over other mappings to the same lower-level pixel array. This particular process of mapping a graphics operation from a higher-level overlapped pixel array to a lower-level one involves substituting different destination masks representing the actual shape of the pixel array after the shapes of all the mappings "above" or "covering" the pixel array have been subtracted from the mask.

In our example, for each of the pixel array's mappings, **coverup**'s rasterop method intersects the argument masks with that mapping's uncovered masks, and calls the rasterop method from **fligraph** with these clipped masks. The **fligraph** rasterop decomposes the masks into rectangles, and uses the machine dependent rectangle rasterop procedure in **flam** to actually move the bits.

As a further example that illustrates other aspects of the system design, we will go through the steps that are taken to apply DeletePA (the method used to destroy pixel arrays) to a PA obtained from `ClassWindow`.

Like all of our procedures that implement methods, the procedure named DeletePA performs the method lookup task: it locates the implementation of the delete function for the specified pixel array. The default implementation of `ClassWindow` is provided by **frawd**. So assuming that the method hasn't been replaced, DeletePA will determine that the correct implementation of the deletion method for this PA is frawdDeletePA.

Our window manager creates a window pixel array as a set of three **coverup** pixel arrays, representing the border, corners and body of the window, and keeps track of its windows through a private data structure. What frawdDeletePA does is to free this ancillary window data structure, and call DeletePA to destroy each of the window's component pixel arrays.

At this point those structures which made a window out of this pixel array have been destroyed, but we are not finished. The pixel array that represented the window itself has not yet been deleted, it has merely been cut down a level; all of the structure that it inherited from the **coverup** layer, and below, is still there. Furthermore, its deletion method pointer still refers to frawdDeletePA.

What we want to do is to use the implementation of DeletePA provided by this pixel array's parent class, **coverup**, to delete the remaining parts of this pixel array. DeletePAsuper does exactly that: in this case it chains through the pixel arrays' parent pointers to locate coverupDeletePA. This method implementation frees each of the pixel array's mappings, calls DeletePA on each of the pixel arrays to which the original pixel array was mapped, and then calls DeletePAsuper to destroy a further level of lower-level pixel array structures. This time the super method is flamDeletePA, which frees the lowest level elements of the pixel array, including the 2-dimensional bitmap.

7. Remote Method Invocation

The above mechanism for invoking methods also works across process boundaries using Flamingo's Remote Method Invocation (RMI) system. The RMI system provides Flamingo the ability to execute implementations of methods that live in remote processes. The design assumes that there is only *one* object manager (a running Flamingo program) for a set of clients, and that the location of that object manager is a global value known to all clients. This "location" takes the form of an IPC message port labelled with a name in a global distributed name service.

Each client alternates between being a user of Flamingo and a server of remote calls on methods implemented within its address space. Client code works by invoking methods on objects (as usual). References to global objects are available from the RMI system. References to other objects can be obtained as parameters returned by the invocation of methods on global objects.

Each method is implemented either as a local procedure within Flamingo itself (as is the case with most method implementations) or as a remote procedure. Any method, including the **before**, after or primary methods, can be replaced by either a local procedure address or a remote method implementation structure. A remote method implementation structure contains the IPC port to send the invocation message and a method number (used to indicate to the remote process what procedure should be called for an invocation). The figure 7-1 shows two objects implemented in Flamingo, along with the clients that implement their methods.

In figure 7-1 method `M` of object `O'` is implemented within Flamingo by a routine called `X`, method `M` of object `O` is implemented within `Client1` by a routine called `X`, and so on. If code in `Client1` were to call method `M'` on object `O'`, the routine called `X` in `Client2` would eventually be called with the arguments specified by the call in `Client1`. Any results returned by `X` in `Client2` is returned to the calling code in `Client1` by the RMI system.

The activities of being a user of Flamingo's services or of being a server of Flamingo's users is handled by the clients through the RMI system. This code is symmetric in clients and in Flamingo so that each can be a server of the others requests. Note that *all* objects live inside Flamingo, where all management of their internal state and of their method implementations resides.

Code supporting the RMI system is mostly machine generated by two programs: the method invocation support is generated by `fig` (Flamingo Interface Generator) and the code supporting the underlying procedure call mechanism between processes is written by the remote procedure call generator Matchmaker [Jones 85]. (Some hand-written code to support the software interrupt system for Mach's IPC implementation is included in `flint`, the FLamingo INTerface module.)

We now give a more detailed look at what happens when a method is invoked through the RMI system. Suppose that `Client1` invokes method `M'` on object `O'` like so:

```
M'(O');
```

The code representing the method invocation calls on the underlying transport mechanism to send the

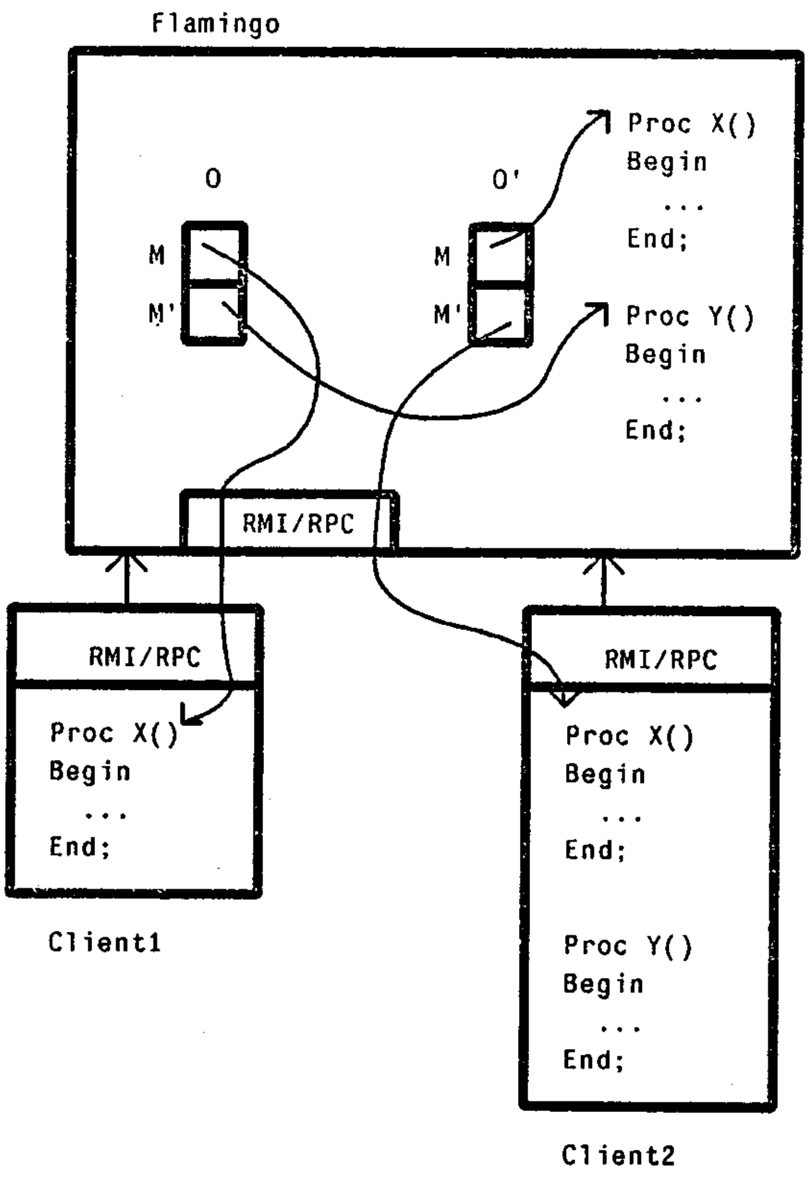

Figure 7-1: Remote Method Invocation System

message to the object manager with this call:

```
uM'(ObjectManagerPort, O');
```

This routine sends all of its arguments to the port indicated via the IPC mechanism. `ObjectManagerPort` is a global value represented in figure 7-1 by the arrow from `Client1` to Flamingo. The RMI system in Flamingo handles the message by calling a service routine for that method:

```
sM'(port, O');
```

The service routine removes the port reference and calls on the actual method within Flamingo:

```
M'(O');
```

Note that this message transmission has occurred since all objects (and all information about the location of methods' implementations) is in Flamingo, the object manager. Now, since the method `M'` in object `O'` is actually implemented remotely, the RMI system is invoked again. The above call now calls its underlying transport mechanism to send the message to the client where the object is implemented:

```
            uM'(O'.M'.port, O'.M'.methodnum, O');
```

Note that the destination port is found within the object `O'`s structure for the method `M'`. The `methodnum` parameter is used in the client to determine which procedure to call for this invocation. The RMI system in `Client2` receives the message to invoke this method, calling on its service routine:

```
            sM'(port, methodnum, O');
```

The RMI code in the service routine of `Client2` now calls on the actual method as determined by the parameter `methodnum`. Since this points to the procedure `X` in `Client2`, we now call `X` with its argument from the original caller:

```
            X(O');
```

This system differs from a pure remote procedure call system (like that generated by Matchmaker) in that the bindings to procedure calls of procedure locations occurs dynamically. Any client with a reference to object `O'` in the above example could easily change the implementation of the method `M'` to point to a procedure in their own address space. From then on, all calls on that method would be sent to this new client. A simplistic RPC mechanism, particularly in a compiled environment, binds these locations at compile time. This binding time is too early and too restrictive for use within an object-based environment.

The current RMI system, based on MACH IPC, can perform about 90 RMI requests per second on the MicroVax II. This has been adequate for many applications; we have done little as yet to optimize the performance. A more portable version of the RMI system that uses UNIX sockets is planned.

8. Masks

Masks are used within the system as a general purpose region descriptor. Our mask structure was originally borrowed from the (undocumented) implementation of Apple Macintosh regions, but it has evolved into something rather different.

The most frequent mask operations are between rectangular masks, such as occurs when a character is copied from a font to an uncovered window. Because of this, the mask structure and mask routines make a distinction between rectangular and non-rectangular masks, and we have optimized most of the code to advantage of this distinction. Another common case is the intersection of the small rectangular mask with a larger non-rectangular mask. This occurs when printing characters in a partially covered window, or when interpreting mouse clicks. Some of the changes that we have made to the mask structure have been to optimize this situation.

The Mask structure is declared as follows:

```
typedef struct stmask
{
        coord w, h;               /* width and height of bounding rect */
        boolean rect;             /* is the mask rectangular? */
        int bodylen;              /* length of the body (in coords) */
        coord *body;              /* the mask data (optional for rect) */
        struct stmask *parent;    /* the parent of this mask */

} aMask, *Mask;
```

w and *h* refer to the smallest bounding rectangle. *rect* is a boolean reflecting whether the mask is rectangular. In the current implementation, rectangular masks do not have bodies. *bodylen* is the length of the mask data (the body) in coords. The *body* is a run-length-encoded description of the mask bitmap as it is traversed from top to bottom. The general format of the body is:

```
<mask-data>  ::=  {<line-data>}* <marker>
<line-data>  ::=  <marker> <start-line> <finish-line> {<run>}*
      <run>  ::=  <left-border-of-run> <right-border-of-run>
   <marker>  ::=  0x7fff
```

NOTE: Masks describe shapes, or regions, and have no explicit origin within any pixel array. Masks used to have x and y coordinates, since many mask operations do not make sense without mapping the mask into some coordinate space. However when we found ourselves constantly remapping masks into different spaces, these coordinates were removed from the mask structure.

Here's an example:

```
                                x

             0  1  2  3  4  5  6  7  8  9 10 11 12 13
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          0 |[]|[]|[]|  |  |  |  |  |  |  |  |[]|[]|  |
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  y       1 |  |  |[]|[]|  |  |  |  |  |  |  |[]|[]|  |
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          2 |  |  |[]|[]|  |  |  |  |  |  |  |[]|[]|  |
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          3 |  |  |[]|[]|  |  |  |  |  |  |  |[]|[]|  |
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
          4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
            +--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The cells filled with [] are those which are inside the mask. This mask is so small that it is unlikely that it would actually be used, but it makes a good example.

The data representation of the body of this mask would be:

```
0x7fff                        ; the starting marker (for going backwards)
0                             ; rows 0 to 0
0
0                             ; cols 0-2 are on
2
11                            ; and cols 11-12
12
0x7fff                        ; start of next group of lines
1                             ; rows 1-3
3
2                             ; cols 2-33
11                            ; cols 11-12
12
0x7fff                        ; start-line
0x7fff                        ; end-of-mask
```

The mask code maintains a special pointer to the empty or zero mask and deletes any intermediate structures that turn out to be the zero mask. Note that the zero mask is considered to be mask of size zero, but a null pointer to a mask is not considered to be a mask at all. Within Flamingo itself, certain routines use masks allocated off the stack to avoid allocating and deallocating memory. This facility is not exported to client programs.

9. Managing Window Managers

As a demonstration of the features of Flamingo, we have implemented an interpreter for calls on the Andrew system [Gosling 84a, Gosling 84b] in a module called **android**. This Andrew interpreter (currently built into Flamingo but soon to be separated into its own process) asserts itself via the appropriate UNIX socket calls as an actual instance of Andrew, and then waits for Andrew calls from Andrew client programs. (Note that we are running Andrew *binaries*: no changes to any of the Andrew programs are necessary. People writing Andrew code are also writing applications for Flamingo!)

We should emphasize here that Flamingo only resembles Andrew at the *program* interface level and not at the *user* interface level. What the user perceives is quite different in our system from running Andrew's window manager. For example, Andrew only provides tiled windows, while Flamingo provides overlapped windows. Andrew shuts down all input and output operations with client programs while menus are displayed, while Flamingo continues to run underneath displayed menus.

Our Andrew interpreter demonstrates all the basic functionality of the Flamingo primitives. It has also made it possible to create an entirely new operating environment for the user with numerous hooks for implementing still more features and functionality, without rendering the system unusable by the mass of software written for previous systems, such as Andrew. (This is of course simply an argument in favor of upward compatibility, but it has proven to be a powerful mechanism to evolve Flamingo from its origin as a graphics package.)

Flamingo's object-oriented architecture provides a flexible mechanism for separating the display and graphics abstractions important to the application process from the lower level abstractions that are important to the system. So, each Andrew process sees only one pixel array, namely the pixel array that represents its window, and more importantly, knows nothing about the details of that pixel array's method implementations. This pixel array could be a standard top-level Flamingo window, or it could just as easily be a sub-window inside of a window running someone else's window manager scheme.

10. Final Remarks

Flamingo was first officially released within our departmental community in October 1985, and has seen two other major releases as of this writing. The system has been used as a basic window manager (with a terminal simulator, our Andrew simulator, and the RMI system) on Microvax workstations. The next major step in our work is to get the basic mechanisms in place for performing distributed process management by hand, so that programable features can be incorporated. This work will progress in two directions: virtual graphics device support for different graphics packages and window managers, and the support of hardware mappings from the virtual graphics devices to the various types of available graphics workstations in our computing environment.

Support for applications will be in the form of graphics packages and window managers to the Flamingo virtual graphics device. This will enable us to continue running already existing programs written for these packages.

The incorporation of mappings from the Flamingo virtual graphics device to various types of graphics hardware (or to other virtual graphics devices) will make it possible to interact with any application running anywhere within the distributed computing environment. This interaction can take place on any hardware supported by the mapping software of Flamingo.

There are an enormous number of performance issues that have yet to be addressed when considering the task of separating a Flamingo system into a set of communicating processes. One major concern is the possibility that the interfaces between these processes will create substantial communications demands. How to structure these interfaces so as to achieve reasonable system performance is an open research problem. This is precisely the kind of issue that Flamingo has been designed to help us investigate, and that we will be looking at.

An interesting use of mappings has been proposed that will allow high-quality images of the screen to be generated. A typical method for getting pictures of a screen is to simply dump the state of the raster memory used to generate the screen bits and display this using either a dot-matrix printer, laser printer, or other suitable, non-alphanumeric device. Such devices as laser printers have a much finer resolution than that of the screen hardware, and the resulting image is often unclear or distorted. A Flamingo mapping could be defined for a PA that would map all rasterop operations to a generator of a file of laser printer commands. Line drawing, character drawing, and all raster operations would all be done in a scale appropriate to that of the printer's capabilities rather than to the scale of a particular screen.

A major goal of this system within the next twelve months is to provide a user interface to a large, heterogeneous distributed computing environment. This system is intended as a vehicle for research into using techniques of Artificial Intelligence to perform large-scale file and process management within this computing domain. Using the Sesame [Jones 82] distributed file system, users of our machines have uniform access to the *file data* located on a large number of distributed machines. We have never had the same kind of uniform access to the *processing power* of those machines.

We are now considering various ideas for distributed resource managment, including the idea of a *butler* [Dannenberg 82] residing on each machine to provide controlled access to the processing power of the machine. Resource management cannot stop there though: dynamic load balancing must also occur to allow users to come and go, and to have adequate resources available to them from their own machine when they need it or from other machines when those are not otherwise in use. Techniques from the domain of Artificial Intelligence, including *planning*, *backtracking*, and *goal-directed activities*, should have a significant impact on our ability to do resource management.

Our experience with Flamingo has been that its universal, built-in object manager is too restrictive, and rather ad hoc. We are now working to separate the object management functions of Flamingo into a more generally useful object manager [Anderson 86]. This object manager will allow objects to kept in many different processes, thus eliminating Flamingo's restriction that all of its objects be kept together. It will also be possible to create new classes of objects while the system is running, and to add user defined methods to existing objects. In doing this well, there are a number of issues that need to addressed, such as protection, garbage collection, error recovery, and object migration, that are the subject of ongoing research.

We gratefully acknowledge the entire Flamingo working group, which has at times included Bob Baron, Rich Cohn, Roger Dannenberg, Dario Giuse, Mark Hjelm, Paul McAvinney, Rob MacLachlan, David Nason, Randy Pausch, Rick Rashid, Walter Smith, Pedro Szekely, Avie Tevanian, and Skef Wholey, for their insights, arguments and ideas. The first running Flamingo system came up on June 5, 1985, and many subsequent versions were written during the summer and fall of 1985 by the authors, with some help from Walter Smith, Avie Tevanian, f i g, and Matchmaker.

11. References

[Anderson 86] D. B. Anderson.
Managing Distributed Objects.
1986.
Doctoral Thesis Proposal.

[Apple 85] Apple Documentation Group.
Inside Mac.
Addison Wesley, 1985.

[Baron 85] R. V. Baron, R. F. Rashid, E. H. Siegel, A. Tevanian, M. W. Young.
MACH-1: A Multiprocessor-Oriented Operating System and Environment.
Technical Report, Carnegie-Mellon University, 1985.

[Buxton 83] W. Buxton, M. R. Lamb, D. Sherman, K. C. Smith.
Towards a Comprehensive User Interface Management System.
Computer Graphics:35-41, 1983.

[CMUSpiceProject 79]
CMU Computer Science Department.
Proposal for a Joint Effort in Personal Scientific Computing.
Internal Document, Department of Computer Science, Carnegie-Mellon University, August, 1979.

[Dannenberg 82] R. B. Dannenberg.
Resource Sharing in a Network of Personal Computers.
PhD thesis, Carnegie-Mellon University, December, 1982.

[Gettys 86] James Gettys.
Problems Implementing Window Systems in UNIX.
In *Proceedings of the Winter 1986 USENIX Conference*, pages 89-97. January, 1986.

[Goldberg 83] A. Goldberg, D. Robson.
Smalltalk-80.
Addison-Wesley, Reading, MA, 1983.

[Gosling 84a] J. A. Gosling, D. S. H. Rosenthal.
A Window Manager for Bitmapped Displays and UNIX(tm).
Technical Report, Information Technology Center, Carnegie-Mellon University, 1984.

[Gosling 84b] J. A. Gosling, D. S. H. Rosenthal.
A Network Window-Manager.
Technical Report, Information Technology Center, Carnegie-Mellon University, 1984.

[Guest 84] S. Guest, E. Edmonds.
Graphical Support in a User Interface Management System.
In *Proceedings of the European Graphics Conference and Exhibit*, pages 339-347. International Federation of Information Processing, Copenhagen, Denmark, September, 1984.

[Hayes 85] P. J. Hayes, P. A. Szekely, R. Lerner.
Design Alternatives for User Interface Management Systems Based on Experience with Cousin.
In *Proceedings of CHI 85.* ACM, April, 1985.

[Heffler 82] M. Heffler.
Description of a Menu Creation and Interpretation System.
Software - Practice & Experience , March, 1982.

[Ingalls 80] D. H. H. Ingalls.
The Smalltalk-76 Programming System Design and Implementation.
Technical Report, Xerox PARC, 1980.

[Jones 82] M. B. Jones, R. F. Rashid, M. Thompson.
Sesame: The Spice File System.
Internal Document, Department of Computer Science, Carnegie-Mellon University, October, 1982.

[Jones 85] M. B. Jones, R. F. Rashid, M. Thompson.
MatchMaker: An Interprocess Specification Language.
In *ACM Conference on Principles of Programming Languages.* ACM, January, 1985.

[Kamran 83] A. Kamran, M. B. Feldman.
Graphics Programming Independent of Interaction Techniques and Styles.
Computer Graphics , January, 1983.

[Keene 85] Sonya E. Keene, David A. Moon.
Flavors: Object-oriented Programming on Symbolics Computers.
In *Common Lisp Conference.* December, 1985.

[Lantz 79] K. A. Lantz, R. F. Rashid.
Virtual Terminal Management in a Multiple Process Environment.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 86-95. ACM, December, 1979.

[Lantz 84] Keith A. Lantz and William I. Nowicki.
Structured Graphics for Distributed Systems.
ACM Transactions on Graphics 3(1), January, 1984.

[Lantz 85] K. A. Lantz.
An Architecture for Configurable User Interfaces.
Technical Report, Stanford University, 1985.

[Rashid 81] Richard F. Rashid and G. G. Robertson.
Accent: A Communication Oriented Network Operating System Kernel.
In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64-75. Pacific Grove, CA, December, 1981.

[Sapphire 84] *User's Guide to the Sapphire Window Manager*
PERQ Systems Corporation, 1984.

[Shaw 83] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, R. Pausch.
Descartes: A Programming-Language Approach to Interactive Display Interfaces.
Proceedings of SIGPLAN 83 Symposium on Programming Language Issues in Software Systems 18(6), June, 1983.

[Teitelman 85] W. Teitelman.
A Tour Through Cedar.
IEEE Transactions on Software Engineering SE-11(3):285-302, March, 1985.

I. Methods for Flamingo Objects

The rest of this document contains a list of all methods used by the current Flamingo implementation for manipulation of its objects and for basic window management. In the actual code there are probably a few more methods defined. These should be treated as experimental and subject to change without notice unless documented.

Each method takes as its first argument one of the basic objects exported by some part of Flamingo, a `Mask`, a `PA` (pixel array) or a `Flim` (input device). The object is expected to contain a reference to some implementation of that method. If the object does not contain any implementation of that method, it does nothing, returning a default "zero" argument when necessary.

I.1. ClassMask

Masks are used within the system as a general purpose region descriptor. Our mask structure was originally borrowed from the (undocumented) implementation of Apple Macintosh regions, but it has evolved into something rather different.

Methods that we provide for manipulating masks include the following:

```
    Mask RectangularMask(m, w, h)
    Mask m;
    coord w, h;
```

`RectangularMask` creates a rectangular mask with width and height `w` and `h`. The mask parameter *m* is ignored. Rectangular masks are more efficient in their allocation and use than more general mask shapes.

```
    DeleteMask(m)
    Mask m;
```

`DeleteMask` deletes the given mask `m`, and that's that.

```
    Mask CopyMask(m)
    Mask m;
```

`CopyMask` creates and returns a copy of the given mask `m`. If the mask has zero width or height, it will return the zero mask.

```
    boolean WithinMask(m, mloc, x, y)
    Mask m;
    Coordinate mloc;
    coord x, y;
```

`WithinMask` returns a boolean value indicating whether the coordinate `(x,y)` is contained within the given mask, with the mask's origin mapped to mloc.

```
    Coordinate GetMaskBounds(m)
    Mask m;
```

`GetMaskBounds` returns a Cooardinate with the `x` and `y` fields containing the width and height of the mask `m`, respectively.

```
ReturnMask MaskDifference(a, aloc, b, bloc)
Mask a, b;
Coordinate aloc, bloc;

ReturnMask MaskIntersect(a, aloc, b, bloc)
Mask a, b;
Coordinate aloc, bloc;

ReturnMask MaskUnion(a, aloc, b, bloc)
Mask a, b;
Coordinate aloc, bloc;

ReturnMask MaskXor(a, aloc, b, bloc)
Mask a, b;
Coordinate aloc, bloc;
```

`MaskDifference`, `MaskIntersect`, `MaskUnion` and `MaskXor` perform the appropriate operation between the two given masks (mapped to the locations specified by `aloc` and `bloc`) and return a structure with a new mask and origin specifying the result. This structure is called a ReturnMask and looks like this:

```
typedef struct streturnmask
{
        struct stmask *mask;
        Coordinate maskloc;
} ReturnMask;
```

Note that these routines always return a new mask even if the result is just a copy of one of the input masks, *unless* the result is the zero mask,.in which case a distinguished `zeroReturnMask` is returned.

```
boolean IsZeroMask(m)
Mask m;
```

`IsZeroMask` returns a boolean value indicating whether the given mask `m` is equivalent to the zero mask or not. To be a zero mask, a mask must have a width or a height of zero.

```
SetHookMask(mask, wh, h, kind, port, hp)
Mask mask;
int wh, h, kind;
Port port;
int (*hp)();
```

`SetHookMask` sets the method `wh` in the mask `mask` to the procedure `hp`. `wh` can take one of the values `BEFORE_HOOK`, `DURING_HOOK` or `AFTER_HOOK`. `h` is a value representing the method being replaced. (This value is constructed by taking `MASK_` and concatenating the name *in capital letters* of the routine to be replaced. To replace the `SetHookMask` routine itself in a mask, one would give for this parameter the value `MASK_SETHOOKMASK`.) The `kind` parameter takes on one of `HOOK_PROC` or `HOOK_IPC` to indicate the kind of new method implementation you are including. If this is `HOOK_IPC`, then the `port` parameter is taken to be the IPC port to send all method calls to. The `hp` parameter is either an actual procedure address, for hooks of type `HOOK_PROC`, or a remote method number otherwise. Client programs need give only the procedure address for the `hp` argument that they want called for this method. Client programs should *always* call `SetHook` with the `kind` parameter set to `HOOK_IPC` and the `port` set to the global variable `FCallPort`.

```
coord *StartRectangle(m, dirbt, dirrl)
Mask m;
boolean dirbt, dirrl
typedef struct stmaskrect
{
        coord *maskbody;
        coord x, y, w, h;
} aMaskRect;

aMaskRect NextRectangle(m, x, y, maskbody, dirbt, dirrl)
Mask m;
coord x, y;
coord *maskbody;
boolean dirbt, dirrl;
```

`StartRectangle` and `NextRectangle` are used together by raster operations to iterate over the various rectanglar regions covered by the mask m. `dirbt` and `dirrl` are true if the mask should be traversed bottom-to-top and/or right-to-left, respectively. `StartRectangle` returns a pointer that is suitable for use as the maskbody parameter for the initial call to `NextRectangle`. `NextRectangle` returns the coordinates of successive rectangles along with an updated maskbody pointer. When there are no further rectangles, the pointer returned will be null.

These methods are presently used by rasterop, and could be used to good advantage by DrawLine and other graphics operations if someone wanted to make them more efficient. (`DrawLine` and `FillTrapezoid` clip to masks by drawing offscreen and then rasteroping from that offscreen pixel array to the screen.) They are currently not exported to client programs.

I.2. ClassPA

Pixel arrays are the primary graphics object of Flamingo. Pixel arrays export a variety of methods for use by different kinds of applications. A client gets a different set of methods for his pixel array depending on the particular pixel array class used to create the it. For example, `ClassPA` only returns a pixel array with the methods to do graphics operations to some "real" location, such as memory or the screen.

The following routines are considered to be simply the ones implemented by the current system, and are not the extent of possible methods for pixel arrays. Note that in a few of these functions the first argument is just needed to locate the method implementation and is in fact ignored.

Each pixel array must implement a few basic operations. These of course could have different implementations depending on which class created the pixel array. The following operations are exported by the lowest graphics level of Flamingo, `ClassPA`. `ClassPA` in fact is a pixel array representing the screen. Eventually, many more graphics operations may be provided, or even better, someone's favorite actual graphics standard or package may be part of Flamingo. These operations are available either by calling them on the `ClassPA` or on any pixel array created by `NewPA` on `ClassPA`.

```
PA NewPA(pa, m, mloc, hasmemory)
PA pa;
Mask m;
Coordinate mloc;
boolean hasmemory;
```

NewPA creates and returns a new pixel array of the shape given by the mask m at offset mloc. The parent of the new pixel array will be set to the parent of the pixel array pa. If hasmemory is TRUE, the pixel array will have a block of memory associated with it for storing the results of graphics operations. Only pixel arrays of class ClassPA really have memory since only their methods are defined to work on real memory.

```
DeletePA(pa)
PA pa;
```

DeletePA deletes the pixel array pa, along with any memory associated with it.

```
int Alter(pa, l, rank, mask, mloc)
PA pa;
long l;
int rank;
Mask mask;
Coordinate mloc;
```

Alter is generally used to alter a pixel array. This changes the shape of the pixel array pa to that of the mask.

Also, on the Microvax, if the pixel array is that of the screen, the rank is used to change the relative location of the mapping of the screen memory to the screen. In other words, this is how to get the screen to "roll" up or down. The screen will not be rolled any further than physically possible. Alter returns an integer representing the actual amount that the screen was rolled so the caller can remap the cursor.

```
PAProps GetPAProps(pa)
PA pa;
```

GetPAProps returns a set of properties about the given pixel array in the following structure:

```
typedef struct stpaprops
{
        Mask mask;
        Coordinate maskloc;
        int boundw, boundh;
        boolean hasmemory;
} PAProps;
```

The mask and maskloc are *copied* from the pixel array. The boundw and boundh parameters are obtained from the mask itself and are intended for client programs that want to know the bounding rectangle of the mask.

```
DeletePAProps(pap)
PAProps pap;
```

DeletePAProps deletes all the structures in the pa properties structure in pap.

```
SetHook(pa, wh, h, kind, port, hp)
PA pa;
int wh, h, kind;
Port port;
int (*hp)();
```

SetHook sets the method wh in the pixel array pa to the procedure hp. (It is not called *SetMethod* for historical reasons.) wh can take one of the values BEFORE_HOOK, DURING_HOOK or AFTER_HOOK. h is a value representing the method being replaced. (This value is constructed by taking PA_ and concatenating the name *in capital letters* of the routine to be replaced. To replace the SetHook routine itself in a pixel array, one would give for this parameter the value PA_SETHOOK.) The kind parameter takes on one of HOOK_PROC or HOOK_IPC to indicate the kind of new method implementation you are including. If this is HOOK_IPC, then the port parameter is taken to be the IPC port to send all method calls to. The hp parameter is either an actual procedure address, for hooks of type HOOK_PROC, or a remote method number otherwise. Client programs need give only the procedure address for the hp argument that they want called for this method. Client programs should *always* call SetHook with the kind parameter set to HOOK_IPC and the port set to the global variable FCallPort.

```
RasterOp(spa, sm, smloc, dpa, dx, dy, dm, dmloc, op)
PA spa;
Mask sm;
Coordinate smloc;
PA dpa;
coord dx, dy;
Mask dm;
Coordinate dmloc;
int op;
```

RasterOp copies the bits designated by the source mask sm at location smloc from the memory of the source pixel array spa (clipped by the source pixel array's mask) to the destination pixel array dpa (as clipped by the destination pixel array's mask) at the coordinate given by dx and dy, clipped by the destination mask dm at location dmloc with the raster operation op. Raster operations can be one of the following integer constants:

```
#define RASTEROP_ZERO           0
#define RASTEROP_NOR            1
#define RASTEROP_AND1           2
#define RASTEROP_NOT1           3
#define RASTEROP_AND2           4
#define RASTEROP_NOT2           5
#define RASTEROP_NOT            RASTEROP_NOT2
#define RASTEROP_XOR            6
#define RASTEROP_NAND           7
#define RASTEROP_AND            8
#define RASTEROP_NXOR           9
#define RASTEROP_2              10
#define RASTEROP_OR1            11
#define RASTEROP_1              12
#define RASTEROP_OR2            13
#define RASTEROP_OR             14
#define RASTEROP_ONES           15
```

Various reasonable defaults exist if either or both of the source or destination masks are null.

```
DrawLine(pa, w, sx, sy, ex, ey, m, mloc, op)
PA pa;
int w;
coord sx, sy, ex, ey;
Mask m;
Coordinate mloc;
int op;
```

DrawLine draws a line into the pixel array pa (as clipped by the pixel array's mask) of width w, starting at coordinate sx and sy, ending at coordinate ex and ey, clipped to the destination mask m at location mloc, with the raster operation op. The mask m may be null, in which case the operation will just be clipped to the mask of the pixel array. BUG: we don't actually do anything with the width paramcter - it should be greater than zero, however.

```
DrawString(pa, st, n, x, y, font, cs, ss, m, mloc, op)
PA pa;
char *st;
int n;
coord x, y;
char *font;
int cs, ss;
Mask m;
Coordinate mloc;
int op;

Coordinate DrawStringEndPoint(pa, st, n, x, y, font,
                    cs, ss, m, mloc, op)
PA pa;
char *st;
int n;
coord x, y;
char *font;
int cs, ss;
Mask m;
Coordinate mloc;
int op;
```

DrawString draws the string of characters pointed at by st (zero-terminated unless n is greater than zero, in which case n is the number of characters to draw), starting at coordinate (x,y), using the font named font, with inter-character spacing defined as cs number of bits and extra spacing for space characters defined as ss number of bits, into the pixel array pa (clipped to this pixel array's mask and) clipped to the destination mask m at location mloc, with the raster operation op. DrawStringEndPoint does the same, but returns the updated coordinate location. The mask m may be null, in which case the operation will just be clipped to the mask of the pixel array.

```
int StringWidth(pa, font, s)
```

StringWidth returns the width in pixels of the string s when drawn with the font named font. The pixel array pa is ignored.

```
FillTrapezoid(pa,x1,y1,w1,x2,y2,w2,font,c,m,o)
PA pa;
int x1, y2, w1, x2, y2, w2;
char *font;
char c;
Mask m;
int op;
```

FillTrapezoid fills the trapezoid defined by the points (x1, y1), (x1+w1, y1), (x2, y2), and (x2+w2, y2) with the character c from the font font in the pixel array pa. An optional mask m and its location mloc in the pixel array can be provided for clipping in the pixel array. The raster operation used to paint the characters is given in op.

```
SetPolarity(pa, b)
PA pa;
boolean b;
```

`SetPolarity` really only works on the screen pixel array, `ClassPA` itself. When set to FALSE, all graphics operations will be normal, ones are white, zeroes are black. When set to FALSE, all grahics operations will be logically modified so that operations that would have yielded ones will now yield zero and vice versa. Note that the operation must take into account whether the source and/or destination has this parameter to determine how to modify the resulting raster operation.

I.3. ClassCoverupPA

Pixel arrays created by class `ClassCoverupPA` can be *mapped* to other pixel arrays. This means that graphic operation(s) to any pixel array of this class will be performed in terms of the same operation(s) on each pixel array it's mapped to, with appropriate coordinate transformations and clipping performed.

Mappings are used to create translations between the graphics operations on one pixel array to graphics operations on another. Each mapping is created with a mask distinct from the mask of the original pixel array and of the destination pixel array. Graphics operations are (at least) clipped to the intersection of these three masks, and possibly to mapping-specific masks. In addition to shifting and clipping, mappings also define *rank* for all pixel arrays mapped to the same destination pixel array. That is, for all pixel arrays mapped to the same destination pixel array, the ones mapped "on top" of others (i.e., with smaller rank) will clip operations that they hide. This implies the existence of an "uncovered" mask for each mapping that is computed by removing from the mapping's mask any areas covered by mappings with smaller rank.

In addition to new methods to handle mappings, all of the methods for class `ClassPA` are available with the same arguments, either from inheritance from `ClassPA` or from reimplementation in the `ClassCoverupPA` module. We note a few routines below that differ significantly in their `ClassCoverupPA` implementation:

`NewPA` works as in `ClassPA`, creating a pixel array of the shape indicated by the mask `m`. It differs in that the mask location is set to `(0, 0)` and the `hasmemory` parameter does not allocate memory directly for the pixel array. Instead, another pixel array, of class `ClassPA` is created, with memory, which is used to keep an image of operations on this pixel array. Note that, until you map this pixel array to someplace real and visible, no operations to it will appear anywhere. And if no memory is assigned to it, no operations will even be performed or remembered.

`DeletePA` deletes the pixel array, the memory pixel array, and all mappings.

`RasterOp`, `DrawLine`, `DrawString` and `FillTrapezoid` all work by performing their graphics operation to each mapping of the given pixel array. Of course, all argments to these methods are the same as described above. RasterOp's copying function (from one pixel array to another) is complicated by the presence or absence of memory. If present, the memory pixel array is used for a source. If absent, an *arbitrary mapping* is chosen as the source for the operation. All graphics operations are shifted by the value of the location of the mapping's mask location, and are clipped to the value of the mapping's (uncovered) mask. The corresponding graphics operation is then performed, with shifted/clipped arguments, on the destination pixel array defined by the mapping.

The following methods are specific to `ClassCoverupPA` pixel arrays:

```
PAMap MapPA(pa, m, mloc, pa2)
PA pa;
Mask m;
Coordinate mloc;
PA pa2;
```

`MapPA` creates and returns a reference to a mapping between `pa` and `pa2`. Mask `m` and its location `mloc` determine the size and location of the mapping on `pa2`. Note that `PAMaps` are *not* objects in the current implementation, but just references to (i.e., integers corresponding to) maps. Mappings are so tightly coupled to pixel arrays that the decision was made to put all of their implementations in objects of class `ClassCoverupPA` rather than make them a separate object.

```
DeleteMap(pa, map)
PA pa;
PAMap map;
```

`DeleteMap` deletes the given mapping. The pixel array `pa` does not have to be the pixel array that was mapped and is in fact ignored.

```
MapProps GetMapProps(pa, map)
PA pa;
PAMap map;
```

`GetMapProps` returns various properties about the mapping in the following structure:

```
typedef struct stmapprops
{
        PA pa;
        Mask mask, uncovered;
        Coordinate maskloc, uncoveredloc;
} MapProps;
```

The pixel array `pa` does not have to be the pixel array that was mapped and is in fact ignored.

```
ComputeUncovered(pa, pamap)
PA pa;
PAMap pamap;
```

`ComputeUncovered` is a maintenance routine used to compute the clipping masks for all mappings in the same set as `pamap`. *This routine should be called after any and all changes to the pixel array mapping structure have been done.* Such changes include any combination of alterations, creations or deletions that involve the destination pixel array of the mapping or of any pixel array(s) mapped to the destination array. This routine will compute the new clipping masks, then call all the necessary refresh methods on the pixel arrays for any newly visible areas on the mapping.

```
PA OnTop(pa, pa2, x, y)
PA pa, pa2;
coord x, y;
```

`OnTop` returns the pixel array that has a mapping defined on the pixel array `pa2` directly under the coordinate defined by `(x, y)`. The pixel array `pa` is ignored.

```
RefreshPA(pa, pa2)
PA pa, pa2;
```

`RefreshPA` causes all refresh methods to be called for all mappings on the pixel array `pa2`. The pixel array pa is ignored.

```
SetTransparency(pa, b)
PA pa;
boolean b;
```

SetTransparency sets a pixel array of ClassCoverupPA to either be normal and opaque (b is FALSE) or to be transparent (b is TRUE). Transparency does not currently affect graphics operations, but does affect the computation of OnTop.

I.4. ClassWindow

Various methods are implemented by a primitive window manager built into Flamingo. A new ClassWindow pixel array is defined by modifying the methods of a ClassCoverupPA pixel array such that all the above calls work with the parameters as stated, but with different implementations in some cases to support the window management scheme. The following methods are modified or added to the pixel array by the window manager:

ComputeUncovered works as described above but with one small difference. If the map passed is zero, a mapping is chosen from some pixel array to the screen. Calling

```
ComputeUncovered(ClassWindow(), 0);
```

will get you the entire screen refreshed with new mappings and everything.

```
WindowProps GetWindowProps(pa)
PA pa;
```

GetWindowProps returns a set of properties of the window associated with this pixel array in the following structure:

```
typedef struct stwindowprops
{
        int rank;
        Mask mask;
        Coordinate maskloc;
        int boundw, boundh;
        PA userpa, borderpa, cornerpa;
} WindowProps;
```

This information is highly specific to the current implementation of the window manger!

```
DeleteWindowProps(wp)
WindowProps wp;
```

DeleteWindowProps deletes all the structures associated with the window properties in wp.

```
TopWindow WindowOnTop(pa, x, y)
PA pa;
coord x, y;
```

WindowOnTop returns a structure describing the top window, the pixel arrays used to implement the window, and various masks. This operation is considered analogous to the OnTop method, but returns a completely different structure:

```
typedef struct sttopwindow
{
        int kind;
        PA userpa,
           borderpa,
           cornerpa;
        boolean blackonwhite;
        Mask cornermask,
             bordermask,
             usermask;
        Coordinate cornermaskloc,
                   bordermaskloc,
                   usermaskloc;
        int rank;
} TopWindow;
```

This information is intended to be used by a user interface routine that wants to know where within the window the user is pointing. Thus, the masks are returned *in the space of the mapping to the screen.* For the sake of expediency, the masks returned in this structure are never copies, so you should copy them before storing or changing them.

```
    ApplicationDied(pa)
    PA pa;
```

`ApplicationDied` is a method implemented by the window manager that should be called by an application when it dies. (Well, at least it *should* be called. Obviously if the application has died then it can hardly call `ApplicationDied`. This routine is in fact used by the terminal simulator program to call on a window when the controlling shell process has died.) One calls this using the pixel array that was being used by the application that died. The window manager will clean up state and the screen.

I.5. Client-Specific Pixel Array Methods

The following methods are set by the client program for his own window. They provide an essential communication protocol between the basic window management functions of Flamingo and the client. Flamingo calls on these methods at specific times during its processing and the client either provides an implementation for these methods for his pixel array(s) or they are ignored. Note that the client program is *providing the implementation* for these routines and will not usually be the caller of them.

```
    Refresh(pa, m, mloc)
    PA pa;
    Mask m;
    Coordinate mloc;
```

`Refresh` method is called on a pixel array with an appropriate mask by anyone who believes that this pixel array must be refreshed. `ComputeUncovered` calls the `Refresh` method for each pixel array that needs to be refreshed. Also `RefreshPA` calls this on each pixel array mapped to a given pixel array. The mask is only given to optimize the operation. It can be ignored by the application program when performing its refreshing operations since clipping will still be performed by other routines.

```
    InitApplication(pa1, pa2, app_type)
    PA pa1, pa2;
    int app_type;
```

`InitApplication` is called by a window manager to start up a "generic" application in a new window. Certainly, applications can come into existence on their own and request windows and such without going through this routine. This routine is provided mostly as a bootstrap mechanism so the window manager does not have to know what to do with a new window by default. The applications types are `APPLICATION_1`, `APPLICATION_2` and `APPLICATION_3`. This is *highly* window-manager-specific. Our current window manager can "start" one of three applications based on which mouse button that was used to drag out the window. Clearly this may change in the future, but in the interim the interpretation of this parameter is up to the method's implementation.

```
StopApplication(pa)
PA pa;
```

`StopApplication` is called by the window manager when it decides to kill a window. The routine should *not* delete the pixel array as this is done by the window manager and in fact may not exist anymore when the call is made.

```
KeyboardEvent(pa, ie)
PA pa;
InputEvent ie;
```

`KeyboardEvent` is called whenever this window has input. The input event structure contains the state of the input device *when the input action occurred* and looks like this:

```
typedef struct stinputevent
{
        int kind;

        int stlen;
        char st[10];
        coord x, y;
        boolean mouse1down, mouse2down, mouse3down;
        int control, shift;
} InputEvent;
```

The parameter `kind` can be one of `KEYBOARD_EVENT`, `MOUSE_BUTTON_EVENT`, or `MOUSE_POSITION_EVENT`. There is currently no way to "mask out" notification of events you are not interested in obtaining. If you are the listener, you get them all. The string parameter `st` and its length `stlen` are only meaningful when `kind` is `KEYBOARD_EVENT`. Again, all other parameters always represent the state of the mouse and keyboard *when the event occurred.*

I.6. ClassInputDevice

The class `ClassInputDevice` provides support for the keyboard and mouse (and potentially any other input device). The following methods are associated with the class `ClassInputDevice`.

```
SampleInput(fl)
Flim fl;
```

`SampleInput` is called periodically by a central loop to poke the input devices and to process any extraneous calls to the system. It is a remnant of an earlier time when the system actually busy-waited on input event device changes. Busy-waiting is sometimes used still when debugging. Clients should not bother calling this.

```
MapInputEvent(fl, ie)
Flim;
InputEvent ie;
```

MapInputEvent is called by the input handler whenever an interesting event has been noticed. This method is actually implemented by the window manager and is responsible for mapping input events to window manager calls or for passing the event on to applications via some notion of a "listener". This method would be an interesting candidate for reimplementation by someone wanting to create a new window manager with a new style for handling input events.

```
FlimProps GetFlimProps(fl)
Flim fl;
FlimProps GetFlimPropsWait(fl)
Flim fl;
```

GetFlimProps and GetFlimPropsWait return the complete state of the input devices in the following structure:

```
typedef struct stflimprops
{
        boolean mouse1down, mouse2down, mouse3down;
        coord x, y;
        int control, shift;
        short oldcursor[16];
} FlimProps;
```

This routine also clears the input queue, so the state returned is always that of the current state of the device. Thus, this routine can be used for polling the input device, and is actually used now by the window manager when doing "rubber-banding" for window creation, moving, and altering. GetFlimPropsWait is identical to GetFlimProps except that it returns only when an actual event is ready. Note that the system is *really hung* when this happens. (Flamingo does not busy wait however so other processing can get done by the rest of the system.)

```
SetCursorPos(fl, x, y)
Flim fl;
coord x, y;
```

This routines moves the cursor to the specified location. This is not a neighborly thing to do, and is only provided for completeness.

```
SetHookFlim(fl, wh, h, kind, port, hp)
Flim fl;
int wh, h, kind;
Port port;
int (*hp)();
```

SetHookFlim sets the given method in the input device to the procedure hp. wh can take one of the values BEFORE_HOOK, DURING_HOOK or AFTER_HOOK. h is a value representing the method being replaced. (This value is constructed by taking FLIM_ and following this by the name *in capital letters* of the routine to be replaced. To replace the SetHookFlim routine itself in a pixel array, one would give for this parameter the value FLIM_SETHOOKFLIM.) The kind parameter takes on one of HOOK_PROC or HOOK_IPC to indicate the kind of new method implementation you are including. If this is HOOK_IPC, then the port parameter is taken to be the IPC port to send all method calls to. The hp parameter is either an actual procedure address, for hooks of type HOOK_PROC, or a method number otherwise. Client programs need give only the procedure address for the hp argument that they want called for this method. Client programs should always call this with the kind parameter set to HOOK_IPC and the port set to the global variable FCallPort.