An Architecture for User Interface Research and Development

A. S. Williams

Informatics Division
Rutherford Appleton Laboratory

1. INTRODUCTION

1.1. Background and the Single User System Programme

The Informatics Division of the Rutherford Appleton Laboratory (RAL) is concerned with research in information technology, and coordination and support for the Alvey Programme of collaborative research.[2] Although the division has recently been formed, the staff in it have a long history of interest in computer graphics, interaction, and user interface design.[15, 8, 12] Since approximately 1981, most of this work has centred on high performance single user workstations, equipped with high speed raster displays and pointing devices (eg mice). The Science and Engineering Research Council (SERC) has adopted a Common Base Policy for support of interactive computing on this type of device.[13] This Policy provides for the central purchase, supply, and support for a common nucleus of hardware and software, to make the best use of the limited resources available to researchers, particularly skilled manpower. The Policy determines a limited range of workstations, currently the ICL Perq, and more recently, SUN machines. The software component is heavily oriented toward standards, and consists of UNIX† along with Pascal, Fortran-77, and the GKS graphics standard. Workstations must be supplied with a window manager. Networking has always been seen as vital if the workstation is to be genuinely useful to the researcher, and the Common Base Policy now requires ethernet to be available on candidate systems.

In 1984, a comparative evaluation of workstations was undertaken, which eventually resulted in the acceptance of Sun into the Common Base. In the course of this evaluation, substantial experience of several different workstations was gained. This experience highlighted the great variety in user interfaces, and in the software interfaces provided to the programmer for implementing them.

Although the man-machine interaction (MMI) research activities are now separate from the Common Base development and support team, we expect to supply SERC- and Alvey-supported researchers with tools and techniques resulting from our research.

1.2. Early Experience in User Interface Design

It is highly desirable to achieve commonality of tools across the range of hardware systems, and this is impeded by the variety of incompatible software interfaces. A discussion of these problems was presented by the author at an Alvey-sponsored workshop on window management, which was organised by RAL. The proceedings of the workshop are published in Hopgood et. al.[14]

Early in the Common Base programme, we undertook the design and implementation of a text editor (called Spy) which exploited the characteristics of workstations, and we took the opportunity to test some of our ideas on user interface design. Spy was originally written on the ICL Perq, and was subsequently ported to the Sun and Whitechapel MG-1 workstations. Spy is supplied as part of the ICL software product for the Perq. Spy is a window-based text editor for UNIX with the goal of making as much as possible visible to the user. Spy is modeless, using graphical selection of command operands, after the style of the Smalltalk editor. Editing commands are almost exclusively invoked using the mouse, with a mixture of

† UNIX is a Trademark of Bell Laboratories.

fixed, pulldown, and pop-up menus. All text, including that of search patterns, filenames and the like are edited in the same way, with the same facilities available to the user. Uniform ways of cancelling actions before completion are provided, as is a single-level "undo" facility (which will undo a global replacement). The keyboard is used for input of new text, but backspace, delete word, and delete line commands are supported on control keys for compatibility with the Unix terminal emulator. Spy supports concurrent editing of multiple files, with each shown in a subwindow. The user moves between files and performs copy and move functions on arbitrary amounts of text with a few swift hand movements, using a postfix command syntax. Unix regular expression searches and substitution are provided. All text, including that of filenames and search/replace patterns, is edited in the same way with the full set of facilities available.

The more unusual characteristics of Spy which are of interest here relate to the questions of feedback and dynamic display. Feedback is given to the user during dragging and drawing actions (i.e. motion of the mouse while a button is held down). For example, as in Smalltalk the range of the selection is highlighted during the draw-through action by the user. Continuous rate-controlled scrolling is supported. During these actions, Spy provides dynamic updating of many graphical representations. Such displays show the visible portion of the file relative to the whole, the position of the selection, the current line number, and the position of markers. Space allocated to subwindows is reapportioned by dragging title bars, which may slide over each other, or "nudge" others along. Given the high speed of RasterOp provided by Perq PNX*, these techniques lead to a comfortable, fast, and effective user interface, by showing the effect of actions to the user before the action is completed.

Spy is therefore a demanding test of the ability of window managers to support highly interactive applications. Some minor points arose during the Spy development, and are worth noting:

- the ability to change window size was added to Perq PNX after the first version of Spy was in use, and Spy was modified to respond appropriately without inordinate effort;
- when ported to the SUN and MG-1 approximately 10% of the source of Spy was window-manager dependent, with the most difficult problems being related to the handling of input;
- the performance characteristics of the host system severely impact the effectiveness of some of the interaction techniques described above;
- implementation of highly interactive applications is extremely onerous in the absence of a library of functions providing interaction techniques.

Figure 1 shows the visual appearance of Spy in use. A paper describing the design process used in the development of Spy is in preparation by the author. A description by a user of Spy has appeared in a Swedish Journal.[18]

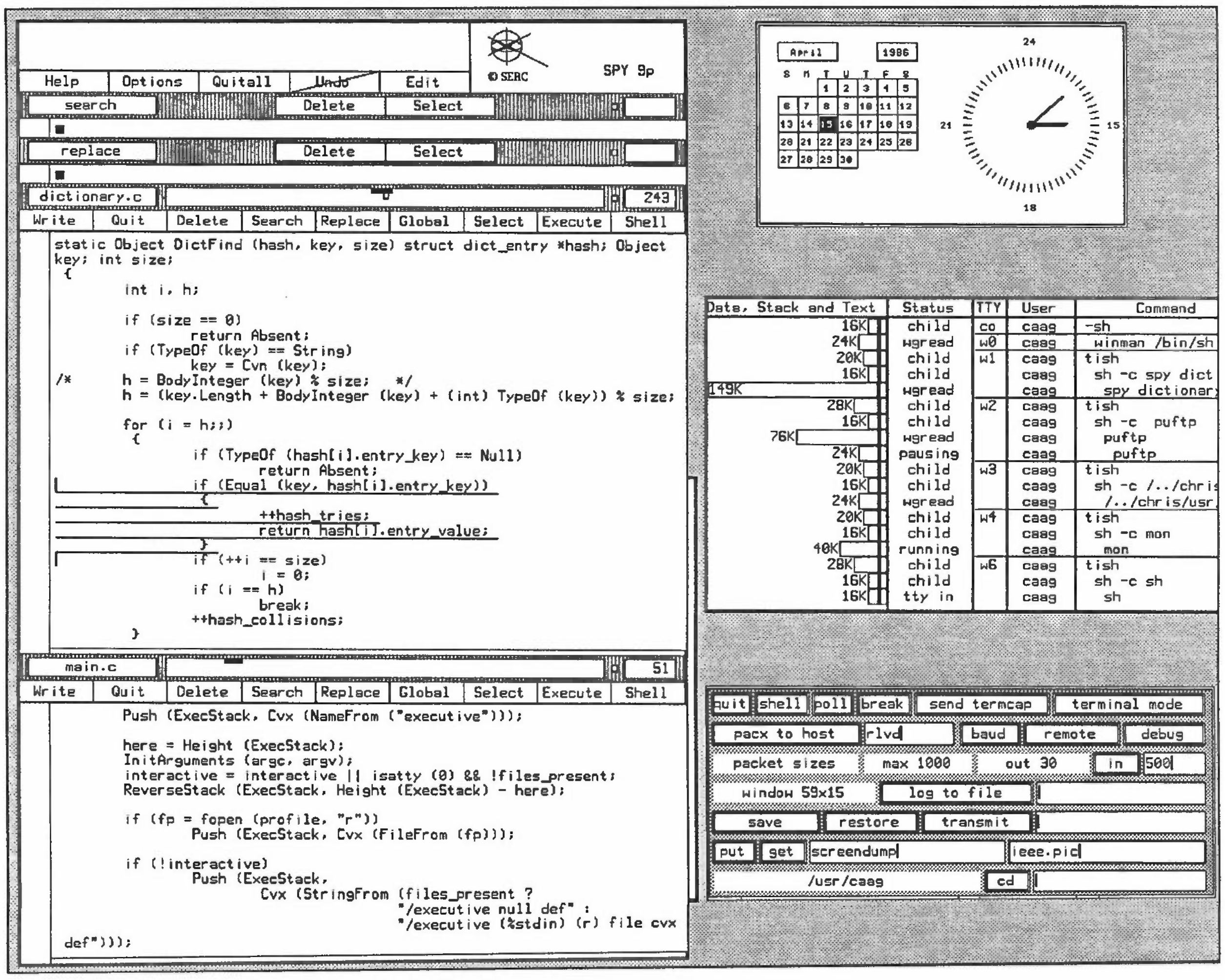
2. OBJECTIVES

2.1. A Vision of the Near Future

Many workers currently view the interaction between a human and a machine as a dialogue, somewhat analogous to a conversation between humans. Substantial research has been done on providing dialogue systems with characteristics which make them more suitable for use, given the (as yet) lack of intelligence in the behaviour of the computer. See for example Hayes et. al.[10] Such dialogues are linear in nature, and composed of a time-ordered sequence of tokens, statements, and/or queries. The time sequencing is used to provide context in which input can be interpreted, by retaining state either over an extended time (in a mode), or over shorter times (by syntax rules). While people form plans of action to achieve some goal, it is not clear that we know enough about the process to identify all of the possible sequences that may be needed to carry out the user's plans. This situation is exacerbated when the goals themselves are ill-defined, for example during exploratory working.

New technologies such as raster displays and pointing devices provide several independent and orthogonal pieces of information simultaneously. Further developments in the industry are extending the capabilities of workstations into providing three-dimensional imaging in real time, with colour, texture, and depth cues.

* ICL's version of Unix for the Perq



```
static Object DictFind (hash, key, size) struct dict_entry *hash; Object
key; int size;
 {
        int i, h;

        if (size == 0)
                return Absent;
        if (TypeOf (key) == String)
                key = Cvn (key);
/*      h = BodyInteger (key) % size;   */
        h = (key.Length + BodyInteger (key) + (int) TypeOf (key)) % size;

        for (i = h;;)
         {
                if (TypeOf (hash[i].entry_key) == Null)
                        return Absent;
                if (Equal (key, hash[i].entry_key))
                 {
                        ++hash_tries;
                        return hash[i].entry_value;
                 }
                if (++i == size)
                        i = 0;
                if (i == h)
                        break;
                ++hash_collisions;
         }
```

```
        Push (ExecStack, Cvx (NameFrom ("executive")));

        here = Height (ExecStack);
        InitArguments (argc, argv);
        interactive = interactive || isatty (0) && !files_present;
        ReverseStack (ExecStack, Height (ExecStack) - here);

        if (fp = fopen (profile, "r"))
                Push (ExecStack, Cvx (FileFrom (fp)));

        if (!interactive)
                Push (ExecStack,
                        Cvx (StringFrom (files_present ?
                                        "/executive null def" :
                                        "/executive (%stdin) (r) file cvx
def")));
```

```
Data, Stack and Text   Status    TTY  User   Command
              16K      child     co   caag   -sh
              24K      wgread    w0   caag    winman /bin/sh
              20K      child     w1   caag   tish
              16K      child          caag    sh -c spy dict
149K                   wgread         caag     spy dictionar
              28K      child     w2   caag   tish
              16K      child          caag    sh -c  puftp
         76K           wgread         caag     puftp
              24K      pausing        caag      puftp
              20K      child     w3   caag   tish
              16K      child          caag    sh -c /../chri
              24K      wgread         caag     /../chris/usr
              20K      child     w4   caag   tish
              16K      child          caag    sh -c mon
           40K         running        caag     mon
              28K      child     w6   caag   tish
              16K      child          caag    sh -c sh
              16K      tty in         caag     sh
```

Figure 1: Spy and other visually oriented tools

At the same time, more flexible input devices are being developed, for example pressure-sensitive touch tablets.[17] Many evolutionary developments fail to exploit this parallelism, by encoding such data as mouse positions as special character sequences embedded in the otherwise linear input stream. More sophisticated developments such as Smalltalk[9] make use of the two-dimensional nature of the display surface and pointing device to provide multiple, concurrent dialogue streams. Each dialogue stream however, often retains much of its time-sequential nature. This approach, while greatly extending the expressive power of the dialogue, serves to add a layer of complexity in the user's mental model of the system. As the scale of the independent dialogues becomes small, and the number of them correspondingly larger, the state of the total dialogue becomes so complex it cannot be wholly remembered or comprehended. Techniques are available to palliate this, such as giving feedback to indicate the state of each dialogue.

In the limit, when the dialogue components become sufficiently small that complete state information can be displayed to the user, these dialogues merge with the application data objects. It becomes unclear whether control objects can and should behave differently to data objects.

By contrast, we are trying to move to a regime whereby the time-sequential dialogues are avoided, to be replaced by spatial encoding of information, in visual representations of the data and control objects supported by the application. Consider the user of a hammer: the user must pick up the hammer by the handle before he can use it to drive a nail. However, one can view this required sequence as an essential property of hammers, rather than a syntax rule describing their use. Imagine the user's dismay on receiving a "syntax error" message if he grasps it by the head.

Our vision of the future is one where the user forms his plans of action, and carries them out in the sequence best suited to him. The system does not impose a syntactic structure on the command language, and thus does not enforce particular time sequences. The user manipulates the objects he can see and their properties, using manipulators which are obvious in their use and effect. We believe this model will extend to cover interaction with dynamic processes as well as simple modification of stored data.

A metamodel of the user interface is helpful to illustrate our point of view. The user carries in his mind models of the application data objects and processes. Similarly, the application code maintains and presents a model of these objects. The user views these objects through a kind of software telescope/microscope, and manipulates them with software "waldos"†. hands Part of the command language serves to control the settings of the telescope and waldos. A large part of the complexity of the user interface arises from the complexity of these controls, rather than from the that of the application objects. A measure of the simplicity and elegance of a user interface is the degree to which the user must be consciously aware of the characteristics and controls of his telescope and waldos. Rob Pike once remarked* referring to the BLIT, that the programs which are used by choice are the ones which the user does not have to think about in order to use. If the mapping between the user's mental models of the objects and the application's representation of them is direct, then the application will be successful. If the control of the telescope and waldos is unconscious and does not require thought, then the user interface will be successful. This is the goal of direct manipulation systems, as yet largely unachieved.

Direct manipulation works well when dealing with the visual representation of objects, but does not cope well with symbolic representations, which are necessary when dealing with large amounts of information, with a complex structure. We do not yet know how to present and manipulate abstract objects such as the class of all objects which match some criteria. Alan Kay has said that deleting three files by moving their icons into the trashcan icon is acceptable, but deleting fifty-three that way is not.[16] The model of objects must not only encompass data and control objects, but also process and agent objects. It must be possible to manipulate these objects both visually and symbolically. The latter capability is essential if we are to enable users to automate routine tasks, by a graphical analogue of command scripts.

2.2. Support for Research and Development

As a route to realising our vision of the future, we are currently building an architecture and a set of tools to support our research. We wish to support research in a number of areas related to user interface design. These include techniques for the presentation of information, ranging from thermometers and dials for numeric values, through forms and property sheets, to multi-media documents potentially incorporating animated sequences and video. Along with presentation, we wish to experiment with techniques for interaction with the information, both in content and in form. We believe that experimentation is particularly important in developing interactive techniques, because the value of a technique cannot be properly appreciated by introspection, but must be experienced. This will require the development of an extensible user interface toolkit, to provide the basic foundation of functions while allowing addition of experimental interaction techniques including novel forms of feedback. Interactive control over the creation and playback of animated sequences is particularly demanding.

Many researchers are currently working on the development of User Interface Management Systems (UIMS), in analogy with Data Base Management Systems.[20,4] In order to support all interaction through a common UIMS, it must support a wide range of application types ranging from draughting systems to

† Waldos are remote manipulators which mimic the motion of an operator's hands, for use in hostile environments (e.g. for handling radio-active material). The name is taken from that of the eponymous hero of a short story by Robert A. Heinlein, who invented the idea.
*private communication.

office automation and publishing quality document production systems. These have very dissimilar requirements, particularly on performance. To date, there has not been a demonstration of a UIMS which is truly general purpose. There has been substantial progress in the use of UIMS in the development of abstractions in user interfaces.[6] Our architecture is intended to support research into the design of UIMS, as a part of the client architecture.

2.3. Exploitation of Technology

As referred to above, technology is progressing rapidly, and making new techniques available. One of our main goals is to put this technology to use, in service of the user interface. Elsewhere, work is already in progress to apply the power of three-dimensional imaging workstations to the visualisation of complex information spaces.[5] Experimentation with new techniques will give rise to the discovery of new forms of presentation and interaction. As language governs thought, so also technology enables discovery.

2.4. Support for Variety of Styles

It follows from the objectives stated above, that our architecture and toolkit should not determine any particular style of user interface, as is done for example on the Apple Macintosh. The lower levels of toolkit, particularly the window manager, must provide sufficient functionality and performance for the upper levels (eg UIMS) to implement whatever techniques they require.

At the same time, providing no support or predetermination presents the applications programmer with a daunting task. He must implement a complete user interface before anything at all will work. Also, the system would have no ''personality'', giving the programmer no feel for its capability. Our solution is to provide an extensible toolkit which supplies a rich set of techniques with a default style of user interface. These techniques may be used at will, or replaced by others piecemeal. In this way, a programmer may take an incremental approach to developing user interfaces.

2.5. An Environment for Experimentation

It is important that any research environment be supportive of the researcher. This means that he requires access to the existing tools and facilities during the process of experimentation. The experimental subsystem must be encapsulated, so that its interaction with its environment is controlled, and the overall system is not destroyed or preemptively locked out by faulty subsystems.

Equally, this encapsulation mechanism must retain the performance of the underlying system, in order that the programmer need not subvert the encapsulation in order to achieve performance equivalent to his intended target environment. Performance requires not only high throughput (for presentation) but also low latency, for responsive interaction and feedback. If dynamic feedback is to be provided to the user, an intimate coupling between the application and the display image is required. Remote access (eg through a server process) increases the response latency, and requires that an encoding of the application's imaging and feedback requirements be defined. Practicalities also call for fast startup of new activities, and shared libraries of interactive techniques (to reduce loading time and paging overhead).

2.6. Open Architecture

The nature of our research aims implies that we do not and cannot know in advance the range of applications to be supported. We want to implement or acquire new applications incrementally, without losing our investment in the older ones. We cannot perform a traditional task analysis to determine the proper functionality to provide. Instead we must provide generic functions so that creative users and programmers can synthesize new techniques.

2.7. Integration

Experience with multiple window systems has shown the value of concurrent use of independent applications. It is important that these applications coexist harmoniously, without detailed mutual knowledge. Instead, they should see each other as well defined functional interfaces or communications protocols. Functionality such as cut-and-paste needs to be provided in a universal way through well defined software interfaces, so that programs participating in the communication need not know the characteristics of the others.

This approach allows an evolutionary development of techniques and the architecture of client programs, without invalidating older software, and retaining the ability to use such software. An important concern to make this strategy work is to implement functionality in the most appropriate system components. A good example of the failure to do this is the implementation of command history in the 4.2BSD C-shell, which prevents interactive applications from making use of it.

3. ARCHITECTURE AND NAMING OF PARTS

This section outlines a generic model of interactive systems, with the intention of identifying the architecture and components of a conceptual system supporting multiple concurrent activities, through multiple windows.

3.1. Concepts supported by the model

This section is primarily intended to define the terms used in this paper: terminology differs considerably according to subject area, for example, the terms 'window' and 'viewport' have quite specific meanings in the area of graphics standards, but are used differently in other areas. Along with these concepts, some issues are listed, which highlight differences between window managers, and interfaces to other components of system software.

Window	a context for display and interaction; normally associated with a visibly distinct portion of the display surface. Issues are: the attributes possessed by a window; mechanisms for window creation and destruction; mechanisms for altering the attributes of a window.
Imaging model	the set of graphic primitives, and the protocol for their display. Issues are: support for multiple levels, as in GKS; the degree of device independence; support for multiple graphics packages (eg GKS, GINO etc); whether the visible appearance of the image is fully defined by the imaging model.
Input model	the set of input primitives and modes, and associated protocol. Issues are: asynchronous vs synchronous communication; dialogue determination; queue management; virtual device emulation (as in GKS); filtering and software generation of input events.
Pane	an independently accessible part of a window, with a defined geometric relationship to the other panes of the window. Sometimes called a subwindow. The main issue is whether these should be supported by the window manager, or some higher layer. Confounding considerations are facilities such as allowing a separate cursors pattern for each pane, allowing different processes to request input from different panes etc.
Bitmap	a component of the implementation of a window; variously called a pixrect, or a panel.
Tty emulator	a mechanism for character stream interaction, provided as a means of communicating with applications written for character terminals; a gateway to history.
Client	an application system, which is a client of the window manager.
Window descriptor	a bundle of attributes of a window.
Sharing	mechanisms to enable many-to-many communication, between tasks and display, and input devices and tasks. Issues are: the degree of transparency of this sharing; and arbitration of control between user and tasks.
Protection	mechanisms to restrict communication, and ensure correct behaviour of tasks with respect to the required transparency of sharing.
Naming	methods of identifying entities such as client applications or windows, and communicating the identity. Both public names (cf Unix file names) and private names (cf Unix file descriptors) are needed.

Mapping	mechanisms to implement sharing and protection. Issues covered here include those related to redrawing of windows following movement, or resizing.
Grouping	manipulation of a collection of objects as a single entity, such as the set of windows associated with a single task. Grouping and subwindowing might be seen as different ways of achieving the same effect.

Icons are deliberately omitted from this list. All of the systems studied which support icons do so differently in terms of visual presentation, application interface, and implementation. There does not appear to be a consensus about the conceptual function of icons. The only common characteristic is that they are small.

3.2. Components of the model

The components described here do not necessarily refer to specific software components. They are identified for purposes of exposition.

WMS	the window management service; a notional software interface provided to client application programs.
LM	the layout manager; which provides the user interface to window management.
GTK	a graphics library and toolkit. This is task-dependent, and provides higher levels of interface, such as menu functions, form filling, and property sheet editing, as well as the normal functions of a graphics package such as GKS.
TTY	notional subsystem implementing tty emulation
PTY	channel for character stream communication between task and tty emulation subsystem, providing a notional software interface to each side.
Client	application task; may comprise one or more processes; a client of the window management service.
Display	notional visible display surface(s).
Mouse	notional set of physical input devices, including the keyboard(s).

3.3. Architecture

Figure 2 shows the above components in schematic form, with the connections between them. The boundaries between components are intended to signify information hiding. For example, the WMS is the only part of the system which has information about the position and rank of all of the windows. Connections represent use of a service provided through a well defined interface.

In this architecture, it might appear that the emphasis is placed on the lower levels of the system. In practice, we realise that most of the work, and particularly the interesting research, lies in the architecture of the client programs, their toolkits, and UIMS. These are covered in a later section.

4. CURRENT PROGRAMME OF WORK

4.1. Window Manager Development

This section describes several activities involved in developing a window management system suitable for our research. The most important current activity is that of defining the Client-Server Interface (CSI) for the window system. The specification of a common CSI is under way in collaboration with UK manufacturers (ICL, Whitechapel, High Level Hardware). This activity will result in a specification of a software interface to be used by clients of the window management system. Additional tasks are to develop interim implementations of the CSI on our current equipment, and to develop a layout manager and terminal emulators (such as ANSI X3.64) as clients.

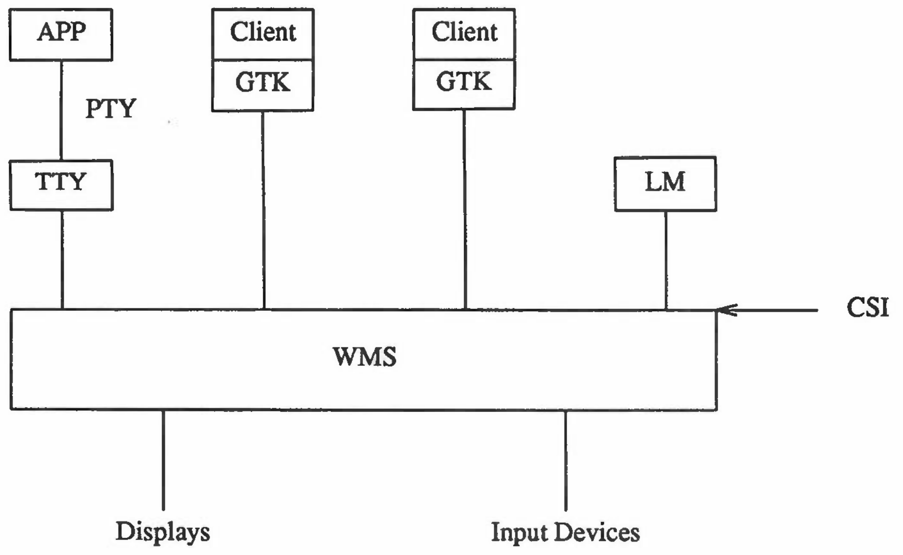


Figure 2: Architectural Model

4.1.1. Rationale for the CSI specification

The specification will not determine aspects of the user interface to the window manager, nor to applications (clients). The graphical output and input will be low level, and the intention is that clients will use an appropriate graphics package (eg GKS) or user interface toolkit or UIMS. The purpose of the CSI is to provide support for the higher layers of software, enabling their development. Commonality of this interface across differing workstation hardware will enable portability of the toolkit, and rapid implementation of a common environment for users and programmers. It is hoped that agreement can be reached, with the UK manufacturers and with the research community, to adopt this specification as an interim standard for window systems. Performance will vary on different workstations, but the choice of hardware will be more independent of the functionality of the associated software.

The CSI will provide functions to control the placement of windows, so that layout management may be implemented by a client of the CSI. Control of the routing of input will also be provided, so that different styles may be specified by clients.

The CSI specification is for window managers for single user UNIX systems with high performance bit-mapped raster displays, at least one pointing device, and a (preferably unencoded) keyboard. This specification has been drawn up in the light of several requirements (some apparently contradictory) and design goals. The most important is that applications with their associated toolkits, and the layout manager, must be able to provide good quality user interfaces, unhindered by the window manager. This has implications for performance, requiring near real-time response by applications in providing feedback. Further, the window manager must not make decisions about the style of user interface or interaction techniques. Support for both tiling and overlapped windows must be provided. Assistance to applications in substructuring their windows is desirable, primarily to simplify input handling, which may differ in different parts of the same window.

Separation of concerns, and simplicity of interface, is very important. Functions concerned with imaging should not be influenced by the existence of multiple windows. Functions concerned with manipulating windows, and specifying their mapping to the display surface, should not make assumptions about the nature of the images in the windows. Protection and sharing between clients are supported, and policies are determined by the layout manager. Clients may consist of multiple processes sharing one window, or several windows may be manipulated by a single process.

The window manager must provide a complete interface to the display(s) and associated input devices: there will be no other routes by which applications may access them. The graphics imaging model must not be tuned for any particular application domain, and must deliver as much of the hardware performance as is feasible, while properly protecting applications from unwanted interference. A low level raster imaging model has therefore been chosen. Imaging algorithms can be made adaptive to different device resolutions and numbers of colours, so these characteristics are not hidden from clients.

The interface specification takes its model from several sources: the Unix model of inheriting capabilities from parent programs, or acquiring them by opening by name is adopted. The client-server model, whereby the client communicates to the server through a unique channel (such as a Unix file descriptor) is maintained. Implementations may be based on remote procedure call if required, or may allow direct access to images. There may be performance implications, but the interface will remain the same. In order to allow for multiprocessing, the window manager minimises modal variables and input modes, but provides sufficient functionality for the GKS input model to be correctly supported without undue programming effort. The interface is sufficient for the hierarchical resource management model of Rosenthal[22] to be realised.

4.1.2. Conceptual model

This section presents the conceptual model developed for the purpose of specifying the Client-Server Interface. This model is a refinement of that presented in section 3.

A *window* refers to a visible component of the display which the user perceives as a single entity, and which he manipulates as a unit. It may have a namestripe, scroll bars, popup menus etc attached, and may have internal substructure. Many of the properties of a window are determined by the layout manager, and are outside the scope of this document. The window manager therefore does not provide windows as a datatype, but provides lower level building blocks for use by the layout manager.

An *image* is a (possibly visible) set of pixels, notionally with associated storage for the pixel values. Image attributes are shape specification, colour depth, and resolution. Normally the colour depth and resolution are limited by the parent image to which an image is (to be) mapped, but unmapped images may have any colour depth and resolution, subject to availability of machine resources. Images are the means by which overlapping is achieved. One example of an image is the frame buffer for a display screen; another example is the cached representation of a popup menu. Images are only a conceptual component of the interface: they are not directly represented in the CSI, and need not necessarily be implemented by allocating memory for pixel arrays. Images are *mapped* to a parent image, using (x,y) positions and a total ordering on visual priority (z position). Typically the parent image is a display frame buffer. Images are not directly represented in the software interface. Clients refer to viewports (q.v.), which result in manipulation of the associated image. There is a master viewport for each image, which controls allocation and deallocation of resources for the image.

A *shape* describes an area which is a generalisation of the normal rectangle. Shapes are positionless, and thus need to be located to provide clipping regions. In the interests of performance, and in order to enable a specification to be produced without further delay, this version only provides functions for creating rectangular shapes, so clipping regions (q.v.), viewports (q.v.), images and rasterOps (q.v.) are all constrained to be rectangular. A later version may generalise them to be arbitrary areas (eg) defined by regions[3] or closed paths as in PostScript.[1,23] Useful algorithms for dealing with such areas have been described by Newell and Sequin.[19]

A *viewport* is a region of an image, totally contained within it. Viewports may overlap in the same image, in which case they have no output priority ordering, and output will interfere. They do have an input priority, however, so mouse and keyboard events will go to the top-most viewport which is accepting events of that class. Viewport attributes include the shape specification. Colour depth and resolution are inherited from the image containing the viewport. Viewports are the unit of discourse between client and server. Functions are provided for allocating viewports, mapping them with defined (x,y,z) relationships, associating input event queues, and generating output primitives. A client may request that the pixels associated with a viewport be mapped into its address space, so that additional output primitives may be implemented efficiently. Clients may control visibility of updates and double buffering, so that animation without flicker is possible. Viewports may be created referring to a new or old image, and/or attached to a new or old input queue (q.v.). Clients may provide hints about the resource requirements and redraw strategy of a viewport, but may not rely on their observance. The intent is that viewports are computationally cheap, and clients may freely allocate viewports (eg as items in a menu).

A *clip area* is a located shape defining the set of destination pixels to be affected by output primitives or raster operations. The window manager forms the intersection of the destination viewport shape, the clip shape, and the shifted source shape if appropriate, when determining the set of pixels to be modified.

Images, viewports, and clip shapes are defined in terms of the coordinate system described by Pike et al,[21] which is also used by the QuickDraw package on the Apple Macintosh. Each image and viewport has its own pixel-relative coordinate system. Coordinate systems of viewports and images related by mapping differ by only translation, not rotation or scale. An image is mapped onto a parent image by specifying the position of the origin of the image's coordinate space in terms of the parent image's coordinates. Viewport coordinates are mapped onto the associated image coordinate space by specifying the origin of the viewport coordinates in terms of the image coordinate space. Output primitives, and any clip location, are specified in terms of the coordinate space of the destination viewport.

The window manager performs the functions of input multiplexing and queuing, but does not provide for input translation (eg to logical device classes as in GKS). Input is therefore inherently device dependent, and clients are expected to implement translation at a low level, to insulate them from variations such as different numbers of mouse buttons, encoded vs unencoded keyboards etc.

An input *queue* is the means by which the server delivers input events (q.v.) to a client. Normally, there is a single queue per client-server pair. Explicit control is given over the creation of queues and their association with viewports. Clients define the conditions under which events are added to a queue by specifying the set of triggers (q.v.). Clients may specify the maximum length of a queue, and may remove events matching a specified set of triggers from the queue without reading them. Clients may request that a Unix signal be generated for certain triggers, in order to perform asynchronous I/O. Clients may generate events with specified triggers for queues to which they have access. It is intended that under most circumstances the input handling (event masks etc) will be changed infrequently. Clients will normally initialise the viewport properties to suitable settings, and will not need to change them to receive the input events they require.

An input *event* is a data structure containing:

- the state of the input devices at the time the event occurred;
- the time at which the event occurred;
- an indication of the trigger which caused the event;
- the identifier of the viewport with which the event is associated;
- the number of events remaining in the queue;
- an indication of whether queue overflow has occurred;
- a client-defined identifier (for client triggers);
- the time at which the event was read from the queue (ie approximately the current time). This datum enables a client to detect its own failure to respond within adequate time, and take appropriate action.

Times are measured in milliseconds from some arbitrary origin for each server. Events from separate servers cannot be reliably serialised without an external synchronisation mechanism. Timer resolution will be at least 50Hz, and the timestamp fields will be large enough to avoid wrap-around within a reasonable time.

A *trigger* may be one of:

- a movement of a pointing device, or change of measure of some other device, by some specified amount; either the complete trajectory or only the most recent position may be obtained;
- an up or down transition of a key on a mouse or keyboard;
- a 'click' of a key on an encoded keyboard;
- the passage of a specified number of milliseconds of elapsed time;
- entry or exit of the focus of attention (eg a cursor) from a viewport;
- a trigger provided by a client (eg the layout manager). Examples of this class are 'size-change', 'iconise', 'client-defined', etc.

4.1.3. Examples of use of the CSI

A tiling layout manager may allocate a single image (the display frame buffer), and subdivide it into non-overlapping viewports, providing these to clients. Clients may then further subdivide these for their own purposes.

By contrast, an overlapping layout manager would allocate an image for each client, perhaps subdividing it into several viewports (namestripe, border controls, etc), and pass on the viewport for the window contents to the client. Input events in viewports associated with border controls would be passed to the layout manager, which would remap the images accordingly, and inform the client(s) of any size change(s).

A session manager providing typescript-based interaction to conventional graphics programs might divide its viewport into two regions, passing one on to a terminal emulator (in a separate process), and the other on to the application. The session manager would arrange for two input queues to be created: keyboard input would be directed to the terminal emulator, whichever viewport contained the cursor position. Mouse input would be directed to the application or the terminal emulator, when in the appropriate viewport.

A complex application might use several viewports, in several different images (perhaps including terminal emulation as above), with a single input queue. Input events are delivered tagged with the appropriate viewport identifier, and mouse position in the appropriate coordinates.

4.2. Toolkit Development

Following the development of Spy, a preliminary toolkit for user interfaces has been developed, partly by extracting some of the interaction techniques from Spy. This toolkit, called *ww*, provides functions for dividing windows into *boxes* and assigning these boxes to different types of presentation and interaction. Basic types such as simple monospaced text, command buttons, and pop-up menus are provided. Facilities are provided for allocating off-screen image store and manipulating images. Ww provides semi-portable ways of defining and using fonts and cursors as well as images.

There were several motivations behind the development of ww. Firstly, we wished to provide a library which implemented our choice of user interface style, so that programmers approaching workstations for the first time would have an example to follow, and a toolkit to help them. Secondly, factoring out these functions in a common library would provide insulation from the variation in window managers, and perhaps provide a starting point for the development of a UIMS. Thirdly, the existence of the toolkit would facilitate rapid development of new applications with good user interfaces. Ww has been in use both internally and externally for some time. The feedback from programmers has been very valuable in determining what has been done right, and what is missing.

A number of small tools have been developed using ww, including: a terminal emulator and file transfer program (using serial lines to other computers) with a control panel; a dynamic Unix process monitor and control program, with graphic representations of process resource usage; a clock/calendar, allowing modification of the time and date using the mouse; a visual file difference program (using Unix *diff*) which presents a side-by-side view of the two files, pointing out differences. Again, ww was developed on the ICL Perq, and has been ported to the Sun and Whitechapel machines. **Figure 1** shows the visual appearance of some of these tools. The present version of ww does not adequately support use of multiple windows, owing to deficiencies in the underlying window managers.

As a result of our experiences with these tools and ww, we are developing several extensions and enhancements to ww. Currently, ww has a fairly rich set of interaction techniques, but is weak on graphical imaging. Many of our users wish to use CAD applications, which require a more complete set of graphic primitives, and for portability reasons would prefer to use the standard, GKS. Although implementations of GKS are available, they tend to be large and slow, and do not provide the level of control over interaction techniques that users want. One of our developments is therefore to provide GKS functionality within ww, so that imaging can be done in a standard way, but programs can surround the image with interaction controls implemented by ww, as illustrated in figure 3.

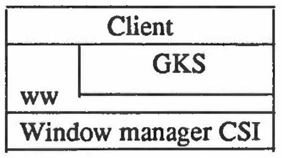


Figure 3: Relationship between ww and GKS

Other candidates under study in addition to GKS are the Macintosh QuickDraw package, and the PostScript imaging model.

The other main area of development of ww is in the provision of facilities to define forms, and the properties of fields therein, along with functions to implement the interaction techniques. Forms can be used to replace many functions of command language dialogues where several parameters are required. See for example Hayes.[11] Other uses are expected to include property sheet for complex objects, and presentation of data structures.

4.3. An Exemplar Program

A more substantial application is in progress, making use of ww. It is an interactive post processor for viewing and manipulating the results of finite element analyses. RAL researchers in finite elements currently use an interactive post processor called Ruthless, which runs on a time-sharing minicomputer and uses a colour raster graphics terminal. Ruthless operates on a variety of data types, including two-dimensional vector fields, allowing the user to view these, and combine them using various operations to produce new data. Ruthless provides monadic and dyadic operations on solutions, and a Basic-like programming language for storing sequences of operations.

Ruthless is not liked by its users, for several reasons. It uses a Forth-like postfix command language, and users find the stack concept unintuitive. It takes a long time and much user interaction to create an image of a solution, which is promptly lost when the next is requested. Users easily lose track of the derivation of particular images.

The speed of presentation of images could be improved simply by running the post processor on a fast workstation with a high bandwidth display. We believe that other problems require that the program be redesigned to provide a different conceptual model. Other motivations for the project are:

- as a learning exercise, in applying our user interface style to larger problems, in a production environment;
- as an exemplar to users and programmers who are unfamiliar with the capabilities of the new technologies;
- as a forcing function for our developments of ww;
- not least, to provide the finite element community with a useful tool.

The present proposal allows the user to have many solutions readily available; some displayed in windows, and others stored in a "filing cabinet" which can be retrieved rapidly. This enables users to compare solutions more easily, and find the solutions of interest. Control of the viewing parameters for a solution will be achieved through a form filling mechanism, with more direct control over zoom and pan functions.

Mathematical manipulation of solutions will be achieved by constructing visual representations of the expressions to be computed. The operators will be selected from menus, and the operands by pointing to the solutions concerned. Evaluation of the expression is activated by clicking on a"do" button in the visual representation. The expressions used to compute a solution will be stored in its header, so that the derivation is always known. Expressions may be stored unevaluated as an item in a menu, for later use.

5. WHERE DO WE GO FROM HERE?

Our forward plans are less well specified, but our goals provide strategic directions. We expect to commence work on the next generation of graphical toolkit, which is likely to be based on object-oriented techniques. The properties of inheritance, and specialisation by extension, seem particularly appropriate ways to build an extensible set of interaction techniques, integrated into a framework of display space allocation and input multiplexing. More automated methods of display allocation will be investigated, based on negotiation between levels of the hierarchy.

More integration between independent clients will be enabled, by developing protocols for cut-and-paste functions for structured information – not merely text.† Since this requires some form of negotiation between the communicating agents, it seems appropriate to investigate the use of an object-oriented representation, where the operations on the information are communicated along with the data. The receiver merely need request the object to (for example) display itself, or handle input events.

Performance is always a potential problem area, particularly in trying to achieve real-time response. An area of study is the downloading of procedures to handle imaging, feedback and input multiplexing and translation, so that application-defined actions can be executed by some agent closer (in computational terms) to the user, such as a server process, intelligent terminal, or interrupt routine. A PostScript interpreter has already been developed, for the purposes of previewing documents. As PostScript is a rich programming language, as well as a good graphics system, it will provide a suitable vehicle for experimentation. Support for handling input devices, and communicating with clients, will be added to the interpreter.

Looking yet further ahead, we hope to build on our experience with toolkits to attempt a truly general purpose User Interface Management System. At the same time, we will work with our user community to test our user interfaces and architecture in a large-scale production environment, involving solution to complex problems. Regarding external users, we hope that the Alvey-sponsored community will make use of our work, for example the Integrated Project Support Environment researchers.[7] Internally, we will work with scientists and engineers to build systems which will support them in all stages of their work, from hypothesis formation and mathematical analysis, through programming and data analysis, to documenting for publication. A goal is that the scientist should only need to specify any given equation once: all other representations will be derived from that.

6. ACKNOWLEDGEMENTS

I wish to thank my colleagues past and present in Informatics Division for the work they have done, and for allowing me to describe it. I regret that the list of names is too long to include here. Any misrepresentations or errors are mine rather than theirs.

7. REFERENCES

1. Adobe, *PostScript Language Manual,* Adobe Systems Inc., 1870 Embarcadero Road, Suite 100, Palo Alto, Ca 94303.
2. J. Alvey, *A Programme for Advanced Information Technology,* HMSO (1982).
3. Apple, *QuickDraw: A Programmer's Guide,* Apple Computer Inc., Cupertino, Ca..
4. W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith, "Towards a Comprehensive User Interface Management System," *Computer Graphics Proc. ACM SIGGRAPH Conference* **17**(3), pp. 35-42 (1983).
5. S. Card and T. Moran, "User Technology - from Pointing to Pondering," *Proc. ACM Conf. on the History of Personal Workstations* (January 1986).
6. Joëlle Coutaz, "Abstractions for User Interface Design," *IEEE Computer* **18**(9), pp. 21-38 (September 1985).
7. Alvey Directorate, *Alvey Programme Annual Report,* November 1984.
8. J. R. Gallop and W. D. Shaw, "PIGS - A Command System for Interactive Graphics," *Proc. EuroComp 75.*
9. Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, Reading, Mass (1984).
10. Phil Hayes, Eugene Ball, and Raj Reddy, "Breaking the Man-Machine Communication Barrier," *IEEE Computer,* pp. 19-30 (March 1981).

† A challenge presented by Colin Prosser of ICL is to cut and paste a bouncing ball, and keep it bouncing.

11. Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner, ''Design Alternatives for User Interface Management Systems Based on Experience with Cousin,'' *Proc. ACM CHI '85 Conference on Human Factors in Computing Systems* (April 1985).

12. F. R. A. Hopgood and D. A. Duce, ''A Production System Approach to Interactive Graphic Program Design,'' in *Methodology of Interaction*, ed. R. A. Guedj et. al., North Holland, Amsterdam (1980).

13. F. R. A. Hopgood and R. W. Witty, ''PERQ and Advanced Raster Graphics Workstations,'' *IEEE Compter Graphics and Applications* (September 1982).

14. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, and A. S. Williams, *Methodology of Window Management*, Springer-Verlag, Berlin (1985). ISBN 3-540-16116-3

15. F. R. A. Hopgood, D. A. Duce, J. R. Gallop, and D. C. Sutcliffe, *Introduction to the Graphical Kernel System (GKS), Second Edition*, Academic Press (1986). ISBN 0-12-355571-X

16. Alan Kay, ''The Dynabook: Past, Present, and Future,'' *ACM Conf. on the History of Personal Workstations*, Banquet speech (January 1986).

17. S. K. Lee, W. Buxton, and K. C. Smith, ''A Multi-Touch Three Dimensional Touch Sensitive Tablet,'' *Proc. ACM CHI '85 Conference on Human Factors in Computing Systems*, pp. 21-26 (April 1985).

18. Jan Leeb-Lundberg, ''Skärmeditor med Möjligheter,'' *Industriell Datateknik* **1985:2**.

19. Martin E Newell and Carlo H Sequin, ''The Inside Story on Self-Intersecting Polygons,'' *Lambda*, pp. 20-24 (Second Quarter, 1980).

20. Gunther Pfaff (Ed.), ''User Interface Management Systems:,'' *Proc. Seeheim Workshop, Nov. 1983*, Berlin, Springer Verlag, ISBN 3-540-13803-X (1985).

21. Rob Pike, Leo Guibas, and Dan Ingalls, ''Bitmap Graphics,'' SIGGRAPH 84 Course Notes, AT&T Bell Laboratories (1984).

22. D S H Rosenthal, ''Managing Graphical Resources,'' *Computer Graphics* **17**(1) (January 1983).

23. J. Warnock and D. K. Wyatt, ''A Device Independent Graphics Imaging Model for use with Raster Devices,'' *Computer Graphics* **16**(3) (1982).