Technical Report

UCAM-CL-TR-342
ISSN 1476-2986

Number 342

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

An architecture for distributed user interfaces

Stephen Martin Guy Freeman

July 1994

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted 1994 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

ISSN 1476-2986

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for any other qualification at any other university.

Acknowledgements

There are many people I would like to thank for their help with the technical, literary and emotional aspects of writing this dissertation. These include, in particular: Eric Bier, Matthew Chalmers, Peter Dickman, Paul Dourish, David Evers, Stephen Hailes, Mik Lamming, Gifford Louie, Victor Poznanski, and Pierre Wellner. I also thank my supervisor, Peter Robinson, for accepting me as a student and for his advice and support. Finally, I thank my wife, Paola, for keeping me sane and for putting up with me through the difficult times.

This work was sponsored by a studentship from the Science and Engineering Research Council and included a summer internship at Xerox PARC. I also received support and facilities from Rank Xerox EuroPARC.

Abstract

Computing systems have changed rapidly since the first graphical user interfaces were developed. Hardware has become faster and software architectures have become more flexible and more open; a modern computing system consists of many communicating machines rather than a central host. Understanding of human-computer interaction has also become more sophisticated and places new demands on interactive software; these include, in particular, support for multi-user applications, continuous media, and 'ubiquitous' computing. The layer which binds user requirements and computing systems together, the user interface, has not changed as quickly; few user interface architectures can easily support the new requirements placed on them and few take advantage of the facilities offered by advanced computing systems.

Experiences of implementing systems with unusual user interfaces have shown that current window system models are only a special case of possible user interface architectures. These window systems are too strongly tied to assumptions about how users and computers interact to provide a suitable platform for further evolution. Users and application builders may reasonably expect to be able to use multiple input and output devices as their needs arise. Experimental applications show that flexible user interface architectures, which can support multiple devices and users, can be built without excessive implementation and processing costs.

This dissertation describes *Gemma*, a model for a new generation of interactive systems that are not confined to virtual terminals but allows collections of independent devices to be bound together for the task at hand. It provides mediated shared access to basic devices and higher-level virtual devices so that people can share computational facilities in the real world, rather than in a virtual world. An example window system shows how these features may be exploited to provide a flexible, collaborative and mobile interactive environment.

Contents

Figures

Part I. Introduction

The general acceptance of graphical user interfaces has dramatically changed over the last ten years. Window systems used to be limited to high-performance workstations or unusual personal computers, but now few desktop machines are sold without one and staid corporations are investing in X terminals. The decrease in cost and size of computers, with the related increase in performance, allowed the development of the personal computer in the late 1970's—the computer was relegated to being the servant of the user, rather than vice-versa. Processing power became sufficiently cheap that a computer could be dedicated to a single user, without the need for time-sharing, and a large part of that power could be allocated to providing graphical user interfaces which are more accessible than command line interpreters.

Networked window systems were developed in the 1980s to provide common mechanisms for distributing graphical applications; users could easily run the body of an application on one host and its graphical interface on another. This is now regarded as the normal means of interacting with a workstation, but most implementations embed the assumptions about computing systems and human-computer interaction of that time into their architecture. For example, workstations and displays are seen as large, expensive devices which are installed on a desk with a mouse and keyboard and, while many people may use a display in turn, only one person may do so at a time. Similarly, a common unspoken assumption is that the user interface layer is to help an *individual* interact with a computer system and that people who need to work together may do so *through* the computer, rather than the computer providing part of a larger collaborative environment.

These assumptions may no longer be valid. Lightweight, portable displays with tetherless communications challenge the assumption that displays need to be large and desk-bound, forcing users to fit all their activity onto one or two screens. Faster networks, with predictable message delivery times, and process scheduling which allows more control over end devices mean that devices and processors need not be so closely bound together to provide interactive response times.

A decade of experience has also seen improved understanding of how people work with computers, particularly in groups. The assumption that applications and interfaces need only cater for a single user is now less tenable. Even some explicitly single-user applications, such as the spreadsheet, turn out to be used by groups in practice. Experimental synchronous groupware applications show that single-user assumptions in networked window systems make multi-user facilities, such as tele-pointers, difficult to implement; they also show that group activity is dynamic and so can be hindered by an inflexible infrastructure. While virtual realities demonstrate increasingly complex modes of interaction with a computer, another approach is to distribute computing power around the user's environment—providing many interconnected devices each of which may represent a particular user activity. Thus, people can work together by sharing devices in their physical world rather than the computer's virtual world.

These trends suggest that existing software architectures will no longer be appropriate in the light of more sophisticated user needs and of technical developments which allow the connections between system components to be more flexible. It is necessary and feasible to develop more open user interface architectures which support: dynamic grouping together of interactive devices, dynamic association of users with device groups, devices shared between users, applications which can accept and distinguish between input from many sources, continuous media, and mobile devices.

This dissertation proposes such an architecture, the Graphical Environment for Multiple users and Multiple devices Architecture (*Gemma*), in which all the components of a user interface may be distributed and shared. *Gemma* then provides mechanisms for collecting low-level devices into higher-level abstractions.

1. Dissertation structure

The dissertation is divided into four parts. After this introduction, I present a survey of the current state of user interface software. This shows that new types of application have been stretching the limits of existing architectures, while recent developments in computing systems have not been exploited in the user interface. Squeezed between new demands and a shift in technology, user interface architectures need to become more open and to embed fewer assumptions about how people and computers interact. The virtual reality community has been addressing these issues but its emphasis is on embedding people in a virtual world, whereas *Gemma* is intended to support computing devices embedded and shareable in the *user's* world.

Part III presents several experiments which motivated *Gemma* and contributed to its design. The first chapter describes two separate experiences with the X Window System and shows how X's assumptions about user interfaces make it an unsuitable platform for applications which do not share those assumptions. The next chapter describes the Multi-device Multi-user Multi-editor (MMM) which demonstrated that the hegemony of the notion of the *personal* computer is needlessly restrictive; people can share an interface as well as the data in an application. Finally, the Double Digital Desk (DDD) showed that it is possible, and sometimes necessary, to distribute the components of a user interface. The DDD prototyped various features of *Gemma* such as networked, shareable devices and applications pulling together devices as they need them; it also showed what sort of infrastructure is needed to support such an environment.

Finally, Part IV describes *Gemma* itself. The first chapter presents the motivations for *Gemma*, defining the types of user interaction it is intended to support and the architectural features required to support them, and relating them to the experimental work described in Part III. The next chapter describes low-level *Gemma*, showing how basic user interface components may be made available across a network and shared, how different aspects of a device may be revealed to different clients, and how continuous media may be managed. The sw window system is an example of how low-level *Gemma* may be used to provide a highly flexible interactive system which allows people to interact with dynamic collections of devices, rather than individual workstations. The discussion chapter shows how several example applications, including MMM and DDD, would benefit from being implemented over *Gemma*; the chapter also presents two examples, naming and persistence, of issues which are unresolved in the current design of *Gemma*. The final chapter summarises the conclusions of the dissertation and suggests directions for further work.

Part II. Users and Computers: the Current Status.

User requirements and computer systems have been changing faster than the layer which binds them together—the user interface. Much of this can be attributed to the foresight of those who designed the most popular window systems, some to the efforts of those who found ways of implementing advanced applications over those window systems, and some to the influence of commerce. This part presents a survey of current distributed user interface architectures and describes how new user requirements and new computing infrastructure are challenging the assumptions on which those architectures are based.

Chapter 2 describes some of the user interface facilities that human-computer interaction research shows will be required. Chapter 3 describes approaches to building synchronous multi-user applications and shows how these are hindered by current window system designs. Chapter 4 surveys current trends in networks, operating systems and mobile computing which are not yet being widely used in user interface implementations. Chapter 5 surveys current user interface architectures and discusses the trade-offs made in different designs.

The lessons drawn from this survey are that:

- research into human-computer interaction is generating requirements that current user-interface architectures cannot easily support. These include multi-user applications, continuous media, and "Ubiquitous" computing;
- computer systems are becoming faster, more flexible and more open, but few user interface architectures have as yet taken advantage of these facilities; and,
- the issues in existing window systems are well understood, but work in virtual realities has shown other approaches to building user interfaces.

2. User Interface Requirements

2.1. Introduction

Much of the foundations of current window system architectures was laid by the mid-1980's. As the technology has become established and widespread, researchers have experimented with more demanding applications and acquired a more sophisticated understanding of their use. This, in turn, has led to new requirements that stretch the limits of current window systems, so it is time to look again at window systems and consider what facilities should be included in the next generation. This chapter presents some current work in human-computer interaction which points away from the single-user workstation approach.

2.2. Computer Supported Cooperative Work

After almost 10 years with a distinct title, there are still arguments over a definition of Computer Supported Co-operative Work (CSCW), see Bannon and Schmidt [Bannon91], for example. It is clear, however, that the implementation of any computer system involves social issues. As pointed out in [ANSA89]:

> All of the events and interactions which take place in a distributed system have an effect on other parts of the system and, in particular, on other users. These events and interactions are, therefore, social in nature. Information systems do not simply provide a single user with a piece of technology; they provide a medium for interacting with other people.

Bannon and Schmidt isolate three factors which distinguish CSCW from other systems:

- *articulating the co-operative work:* CSCW systems must benefit *group* activity by making it easier, or more fruitful, for example. This implies that the application must take on some of the burden of administering, in the widest sense, the group. This "administration" may range from the creation of group artefacts and the relations between them (manipulating shared objects), to explicit message filtering and archiving (project management tools).
- *sharing an information space:* it is clear that people must have access to common information if they are to co-operate. It is also clear that, in many cases, people *are* the common information space. An example is the USENET UNIX "Gurus" group where the skill and experience of a software community is available for the cost of the occasional contribution.
- *adapting the technology to the organisation, and vice versa:* CSCW systems cannot exist outside an organisation, however *ad hoc* the organisation or the system. By definition, the design of CSCW systems must include some model of the intended group of users, whether the model is adapted to the users, or the users to the model.

It can be argued that there is a thread in the development of Computer Science which represents an expansion of the sphere of interest: from the focus on hardware in the earliest days, through interest in technical staff which gave rise to "automatic" programming languages, to interest in individual users which produced the direct manipulation interface. As Moran writes [Moran81]:

> A system does not, alas, terminate at its terminals—*users* are attached. The user is one of the critical components determining whether the system as a whole—the human-computer system—works or not.

CSCW can be said to represent a further expansion of interest to include the environment, social and otherwise, in which individual users work. While many CSCW applications do not require advanced user interface technology—messaging systems still work on text terminals—the following sections discuss those which do make greater demands on their infrastructure.

2.3. Single-user/multi-user

An example of how CSCW consists of more than the technology is the shared use of single-user applications. Nardi and Miller [Nardi90] studied spreadsheet use and found that many spreadsheets are developed by more than one person and that people were aware of the co-operative nature of the development process. In addition, the emphasis on end-user programming supports the sharing of both programming and domain expertise. Programming expertise is shared in two ways: first, end-users become the main developers, building the bulk of a spreadsheet to which expert programmers contribute difficult code. Second, less experienced users informally learn from expert programmers by, for example, inquiring about a detail in someone else's spreadsheet. Domain experts no longer specify applications for programmers to build, rather they build their own applications and are *assisted by* programmers who supply training and pieces of complex code.

In an analogous study, Mackay [Mackay90] studied how people in an organisation customised an application and how those customisations were shared. She found three levels of customisation: first, technical staff explore the structure of the software, and make customisations they find interesting available throughout the organisation. A second group of *translators*—end users with more interest than their colleagues in the details of their software—create simplified and more task-specific customisations. Third, most users have not enough time or interest to perform more than the simplest customisations and so copy those provided by the technical experts and translators, even when not entirely suitable, so some customisations may survive for long periods within an organisation despite containing errors or inefficiencies.

The lesson for software providers is that, apart from delivering their products with a well-tested set of defaults, they need to provide tools to help with sharing customisations. One lesson for those developing user interface architectures is that people may need to collaborate closely regardless of the intentions of the software designer.

2.4. Desktop conferencing

2.4.1. Introduction

Connecting remote users together via shared applications seems to have been an obvious application of networked workstations with interactive graphics; examples date back to the Augmented Human Intellect Center in the early 70's [Engelbart88]. How to do it well is less obvious. Ellis et al [Ellis91] list several features which characterise many multi-user applications:

- *shared context*—a set of objects where the objects and changes made to them are visible to a set of users.
- *view*—a representation of some or all of the shared context. Views may differ in their presentation of the same data, or the part of the shared context they represent.
- *group window*—a set of similar windows on different users' displays; changes made to one are reflected in the others.
- *session*—a period of synchronous interaction, such as a meeting or informal interaction.
- *role*—a set of privileges and responsibilities attributed to a person or system module. These may or may not be formally built into the system.
- *tele-pointers*—a cursor which appears in the corresponding position on more than one display and can be used for gesturing. This might also include annotations which allow users to sketch over a shared application or image.

Two approaches were described by Lauwers and Lantz [Lauwers90a]: *collaboration transparent* architectures take single-user applications and filter their input and output to allow them to be shared without alteration; *collaboration aware* architectures are explicitly written for simultaneous use by several people and are able to distinguish between different sources of input and destinations for output. A second issue as to whether the architectures of multi-user applications should be centralised or replicated is discussed in chapter 3.

2.4.2. Collaboration transparency

Investment in existing software, both financial and personal, means that users may wish to share single-user applications. Terminal linking exists in some time-sharing systems, and the Colab [Stefik87] included video switches which allowed users to watch someone else's display, although input was not switched. Distributed window systems, however, provide a natural mechanism for window sharing by tapping into the connection between application and display. Several new components are required when an application is shared: the output must be *multiplexed* to all the clients, which may involve mapping output data for different display server characteristics; there must be some arbitration or *floor control*, however lightweight, between different input sources to avoid race conditions; a *registration manager* is required to allow users to join and leave an existing conference, and to make the address of the conference known to other users; the *current state* of the conference may be held somewhere so that new users can be brought up to date when they join; users may wish to move or resize the conference window independently of the other users or they may wish their views to be tied together, so *workspace management* policies are required; *conference logging* may be useful for providing latecomers with history or recording the session for analysis or legal reasons; and a user interface is needed to allow users to control the conferencing components.

Floor control is an example of the issues that arise from application sharing; policies may range from open access to strict chalk passing. Lauwers and Lantz [Lauwers90a] isolate three characteristic dimensions:

- the number of floors—per conference, per application, or per window;
- the number of people who can hold a floor at the same time; and,
- how the floor is passed between participants.

Some systems have a policy built in: Matrix [Jeffay92] users metaphorically "pass the keyboard" around and VConf [Lauwers90] required the current floor holder to explicitly release control. As Gust points out in [Lauwers90a], however, the most suitable floor control policy may vary with the activity of the group from open-floor "brainstorming" to serial access for record taking. Altenhofen [Altenhofen91] and Greenberg [Greenberg91] separate the setting of floor control policy from its interpretation so that the group can dynamically change the policy.

There are, however, limitations to collaboration-transparent approach, summarised in [Sarin88]: all users must share a single context, such as the insertion point; private views (as against private windows) of shared information are not supported; there may not be suitable support for transfer of application information between the shared and private spaces, such as copy and paste; and access control is limited, users must either be restricted to shadowing the current view, or be allowed to do everything. If these limitations are significant, it is necessary to write a multi-user application which is aware of the participants in a session.

2.4.3. Collaboration awareness

In collaboration-aware systems, the multi-user part of the architecture has access to the contents of the application, whereas collaboration-transparent systems only have access to the input and output streams. Collaboration-aware systems can also distinguish which user performed an action and so manage concurrent streams of input and output. This allows a much finer grain of activity, access *per* data object rather than *per* display, so that different users can safely manipulate parts of the shared context in parallel.

There are a number of real-time shared editors. Grove [Ellis91], for example, is a distributed outline editor which allows users to see and manipulate one or more views of the shared text being worked on. It maintains a distinction between a *view*, a subset of the items in the outline determined by the read access privileges, and a *viewer*, a group window for seeing a contiguous part of a view. Views and viewers may be restricted to one user (private), a set of users (shared), or visible to everyone (public). Group windows provide synchronous shared viewers for a set of users and show a list of currently active users along the lower border of the window so that people can tell who is participating in the session. In use, working in one's own office meant that information access was improved as people could refer to books and papers at hand. The limited range of social cues in a distributed session meant that parallel activity was easier as people could drop out for a while without offending the group but that people had to work harder to maintain the meeting; for example, turn-taking had to be more explicit to avoid confusion. Similarly, the parallelism within the editor meant that sessions could be confusing and unfocused but that co-ordinated activity was more efficient and there was less information loss than if one person alone was entering data. Finally, tutoring became a natural side-product of a shared environment as there was always someone at hand who could see what one was doing.

There are also shared virtual realities. The MultiG Distributed Interactive Virtual Environment (DIVE) [Fahlén93], for example, is investigating the use of presence and proximity in a 3-dimensional space to allow users to interact with tools, services and other users in a distributed environment. Each participant in the space is represented by a 3D icon with an *aura*—a geometric volume around the icon. A user establishes communication with a tool or another user by moving her or his icon until the auras intersect; for example, users can set up a voice connection by moving their icons together. Tools can also be used to mediate between users. The *conference table*, for example, allows users to meet and have a discussion; a user establishes communication with others present at the table by entering the table's aura.

2.4.4. Summary

Desktop conferencing is increasingly being promoted as an aid to remote collaboration, although most current commercial systems support only collaboration-transparent sharing for personal computers. The search for the best group user interface, however, has shown that groupware needs to be flexible, as the dynamics of a group may change constantly during a session. Beck and Bellottii [Beck93], for example, describe how individuals joined, performed multiple tasks and left a team writing a paper together. Computer system infrastructures, including window systems, need to be flexible enough to support dynamic collaborative applications.

2.5. Continuous media

2.5.1. Introduction

Audio and video are becoming commonplace components of interactive user interfaces, although much current video is taken from storage, such as CD-ROM, local to the display machine. Researchers, however, have been developing and finding uses for network video as a means of bringing people and objects together. Many of the experimental systems, such as the Touring Machine [Bellcore93] use analogue video with digital control, but fully digital systems, such as Pandora [King92] and Mermaid [Ohmori92], are becoming more established. This section describes some uses of continuous media, video in particular, and the demands they place on the user interface architecture.

2.5.2. Media spaces

One of the lessons of CSCW is the importance of informal communication for building teams of people, see [Kraut88] for example. Some researchers have been exploiting video and audio to link parts of a team split across sites, a *media space*, where "people can create real-time visual and acoustic environments that span physically separate areas" [Stults86]. A media space node in an office generally consists of a microphone, loudspeaker, video monitor and camera which can be dynamically linked to the corresponding devices on other nodes, so two people who work together might establish a long-term video link to "share" offices even when separated.

Bly et al [Bly93] found a number of uses of their media space. In particular, *awareness*—passively acquired knowledge about who is around and what they are doing—encourages informal interactions, spontaneous connections between people and the development of shared culture. Portholes [Dourish92] is a low bandwidth system which periodically (every 10 minutes) grabs images from offices and public areas into a distributed database. Members of the group can display these images on their workstations, updating the display as the images change. Experience with Portholes suggests that it helps to maintain working relationships within a group split over two sites; communication between remote colleagues increased after the system was installed. The media space supports normal social activities which are difficult at a distance such as chance encounters with colleagues in a third person's office, or locating people by seeing them walk though a public area. The media space also provides flexible support for more formal activities such as video recording and playback of meetings and broadcasting presentations. Social activity over a media space can, of course, be subtly different from the same activity face-to-face; while the media space forms of communication are more limited than direct contact they can also be less intrusive and are less inhibited by physical constraints [Gaver92]. A number of media spaces are summarised in [Bly93] each of which differs in architecture and approach. As with many CSCW projects, each media space evolved out a combination of the culture, experience and technology of the group that commissioned it.

Media space research shows that there is a real use for widespread audio and video, but that the infrastructure to support it needs to be flexible enough to support multiple patterns of use and to support lightweight activity.

2.5.3. Mixing Graphics and Video.

A number of researchers have been experimenting with user interfaces that mix digital graphics and video images to support collaboration. Tang and Minnenman [Tang90] aligned two analogue video systems to experiment with a shared drawing surface. TeamWorkstation [Ishii90] merged digital graphics and analogue video so that a user could, for example, annotate a digital image by drawing on a piece of paper underneath a video camera. A later development, ClearBoard [Ishii92], uses two drawing surfaces with back-projected video and a shared drawing application; each video display shows the head and shoulders of the person at the other drawing surface, aligned so that both people can see what their partner is looking at and where they are working. Hodges et al [Hodges89] allowed users to navigate through a stored video training session by using the mouse to select parts of a screen. For example, a user could phone a character in a drama to teach French by pressing the digits on a telephone in an image of a desk. Tani et al [Tani92] provide real-time "direct" manipulation of remote real-world objects by superimposing digital control objects on a video of real control objects; a user can press a real button on a remote machine by clicking with a mouse on its image in a video view. Feiner et al [Feiner93] went further by using a head-mounted display to superimpose digital information on objects in the real world as the user moves around them. Pagani and Mackay [Pagani93] examined the use of a video connection between design and manufacturing sites and noted that the users were frustrated by the lack of telepointing or annotation when using video to discuss a physical object.

There are difficulties, however, in mixing media with different characteristics, highlighted in [Ishii90]. The different media can be difficult to align and keep aligned should a camera or a drawing surface be moved; this also makes scrolling problematic. Storage requirements also differ: digital drawing packages are unlikely to keep time-based histories of a session, but storing a session only as a video sequence discards information about the structure of the drawing. These problems should diminish as audio and video are increasingly transmitted digitally, and so become more tractable, but they highlight the need for integration between different media. For example, while many are still in use, document processors which do not integrate text and graphics are now regarded as primitive, and even video segments can now be inserted into commercial word processors.

2.5.4. Summary

This section describes two example types of use of continuous media, concentrating on video. Media spaces provide a mechanism for holding dispersed groups together. They demonstrate one role for widespread digital audio and video, providing a flexible mechanism for people to establish connections with each other without special cabling. The rest of the section describes several experiments in using digital overlays to augment the use of video, particularly for interacting with people or objects at a distance. These show the need to be able to integrate and co-ordinate different media within a display system.

2.6. Not Just Workstations

2.6.1. Introduction

There is, of course, much speculation about how people will interact with computers in the near future. Rashid, for example, points out that current children's toys include the most powerful micro-processors of ten years ago, so today's fastest processors may be equally widespread in another ten years [Rashid93]. This implies that everyday devices will have powerful (by current standards) processing abilities to interpret our input and provide feedback. The two articles summarised in this section present visions of how people and computers may interact in the not-too-distant future.

2.6.2. Ubiquitous Computing

Weiser [Weiser91][Weiser93] describes a computing environment made up of many input and output devices, which are portable and range from the hand- to the wall-sized. They will know where they are, and who is in the room, and be able to adjust their displays accordingly. Rather than stacking virtual windows on a desktop metaphor, users will have many cheap and compact display pads which they can spread on their physical desktop (or floor), dedicating pads to particular tasks as paper and folders are at present. Rather than shrinking a window to an icon, users can shrink a window to a palm-sized tab display which can then represent the task which was being displayed in the window. The tab can then be arranged with other tabs or taken to another office where the task can be unpacked again. Wall displays will provide larger surfaces to work on and share discussions (see [Pedersen93]); they can also be used as notice boards which tailor their display to those present in front of it. All these devices will be bound together by ubiquitous wireless communications.

As Weiser points out, this has significant implications for operating and display systems. In particular, the devices in a given room will not remain stable, or always visible to the infrastructure, so the level of software which binds them together must be able to cope with intermittent devices. Similarly, the display system will need to support tasks moving between devices which may be of different sizes and use different communication systems.

2.6.3. Next Generation Interfaces

Nielsen [Nielsen93] proposes that the next generation of user interfaces will involve a wide range of technologies: virtual reality, sound, artificial intelligence, video, pen-based, portable, and so forth. It is unlikely that a single approach will be able to support all of these, unlike the current "Windows, Icons, Menus, Pointer" (WIMP) hegemony. One trend he outlines, however, is a move towards computational environments of data objects which manage their own interactions; so users see documents with embedded components, rather than one application or another. For example, "dumb" file systems should be replaced with a generalised information space containing a large number of data objects. A user's view of the information space depends on the mechanism chosen to gain access to it; for example, Rao et al [Rao92] describe *retrieval-centred workspaces*. Nielsen raises a number of dimensions which may change in future interfaces; three are relevant to the "window-system" level of infrastructure:

- increasing use of multi-threaded input and output, such as a combination of voice and a pointer, which avoids overloading functions onto narrow bandwidth devices like mice and reduces the possibilities for ambiguity; Nigay and Coutaz discuss this further [Nigay93].

- more parallel activity between user and computer, rather than current turn-taking dialogues. An example is the Alternative Reality Kit [Smith87], a physics simulation in which the objects on the screen move subject to their virtual mass and momentum even while the user is engaged in other activity. This issue also arises in multi-user editors where one user's view may be changed by another user's activity.
- moving away from the interactive terminal (textual or graphical) as the primary medium for interaction. Nielsen looks towards mobile devices, location sensors, input by gesture recognition, toy animals as input devices and so forth, all of which break the assumption of the desk-bound computer. One mundane, but practical, example is a supermarket chain which uses wireless bar-code readers, connected to the company's inventory database, to allow its staff to do stock control at the display shelves rather than in an office [Economist93].

2.6.4. The Pad model

An example of how user interfaces might develop away from conventional workstations is the Pad model [Perlin93] in which Pad Objects all inhabit an infinite two-dimensional Pad Surface. Pad Objects are divided into graphics and portals: a *graphic* is any kind of mark such as a bitmap or vector, whereas a *portal* is like a magnifying glass which can roam and peer into parts of the Pad Surface, including other portals, at arbitrary levels of magnification; the user's screen is treated as a special root portal. The Pad model also provides several techniques to allow users to customise their view of the Pad Surface. The significance of Pad for user interface architectures is that it is based on a persistent object space which can be shared between devices and users, unlike conventional window systems where a window or icon only exists on one user's screen. As the authors point out, the Pad model is more suitable than the WIMP approach for an environment containing many display surfaces as it places the user in a information landscape which is not tied to any particular machine.

2.6.5. Summary

Both Nielsen and Weiser see a move to people using many more devices which support a wider range of interactions. In many cases, the distinction between a device and the service it provides will diminish as devices become cheap and powerful enough to dedicate to a particular task. This implies, on one hand, that no single paradigm, such as WIMP, will be sufficient for all these devices; the Pad Model shows an example of an alternative approach. On the other hand, there will be a need for a common infrastructure to bind these devices together as the user's activity requires. These developments also raise interesting questions about the "ownership" of devices—from hand-held portable devices to common wall displays.

2.7. Summary

This chapter discusses some current work in human-computer interaction which suggests that the assumptions on which many current interactive systems were built are beginning to prove inadequate. In particular, the individual user sitting at a terminal driven by an individual machine (real or virtual) is proving to be only a special case of how most people work. Not only are single-user applications turning out to support multiple users in practice, but multi-user applications are becoming increasingly common. We are also seeing a move towards integrating isochronous media, particularly digital video, into user-interfaces to help bring disparate groups together, especially when combined with end-user annotation. Finally, some researchers are looking towards computer intelligence embedded in the world around us, rather than manacled to the desktop. Instead of a single device onto which we overload all our computing activity, we will have many devices for which we will have a wide variety of uses. Some devices will belong to a particular person and task, whereas others are for public use but can be customised for those present at the time. Interactions may become less static as software agents perform tasks for us, reporting to us as results appear, and multiple modes of interaction will widen the bandwidth between user and machine.

The implications for window-system design are that we can make fewer assumptions about how people interact with computers and that application writers need support for more flexible and complex applications, with a less predictable dialogue between person and machine. There are new media and interaction techniques which need to be integrated into user interface architectures, and people will need mechanisms to draw together collections of the proposed myriad of devices.

3. Architectures for Synchronous systems

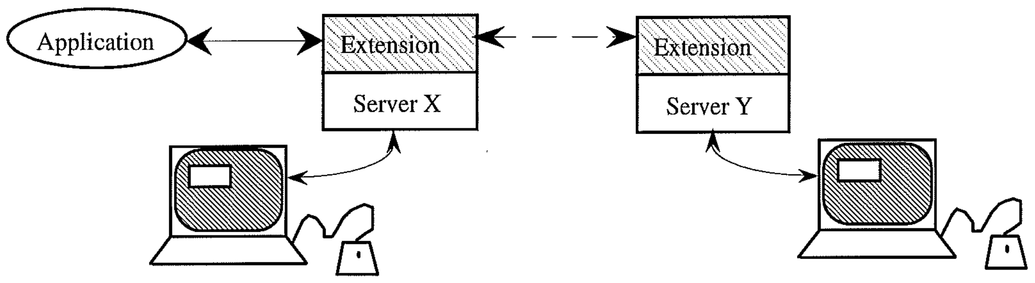
3.1. Introduction

An example of the way current user interface architectures are being stretched is the number of synchronous multi-user applications which have been implemented. Collaboration-transparent and collaboration-aware applications, as defined in the chapter 2, require different implementations. A collaboration-transparent application is not aware of being shared, so the distribution must be done in the display system: *window sharing*. Distribution for collaboration-aware applications, however, takes place within the application—they are *multi-user applications*. This chapter describes some example systems which provide synchronous multi-user applications.

3.2. Window sharing

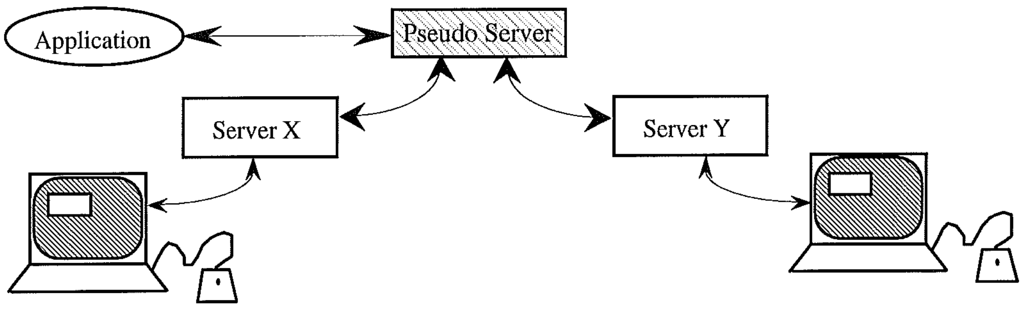
Window sharing can be done in several places. First, the display servers can communicate, as in Shared X [Gust88] (Figure 3-1). The advantages to this approach are that the sharing mechanism has full access to the server state, and that a separate window-sharing protocol extension can be used without altering the core graphics protocol. It is also used for sharing single-user window systems such as Timbuktu for the Apple Macintosh. The disadvantage, however, is that alterations must be made to the display server, which is the most device-dependant component, and its source code may not be available.

Figure 3-1. Window sharing in the display server.



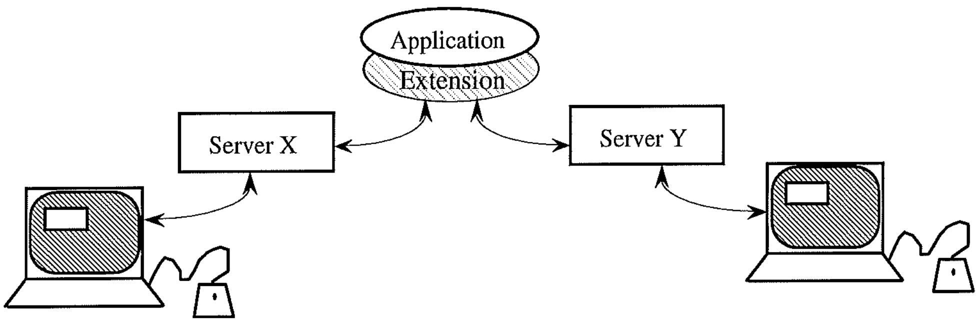
A second approach is to insert a pseudo-server between the application and the actual display server (Figure 3-2). The pseudo-server maintains the state of a virtual display and filters input and output between the client and multiple displays; examples include Matrix [Jeffay92] and XTV [Abdul-Wahab90]. Matrix, in fact, creates a virtual X server above the real X server, so that users may even run, and share, a window manager in their pseudo-server as well as their normal applications. This approach allows the use of unaltered applications but may require the pseudo-server to store much of the state of the application and there may not be enough information in the core protocol to map data such as images across heterogeneous display servers. Furthermore, applications must be started with the pseudo-server as it cannot be introduced later.

Figure 3-2. Pseudo-server between application and display servers.



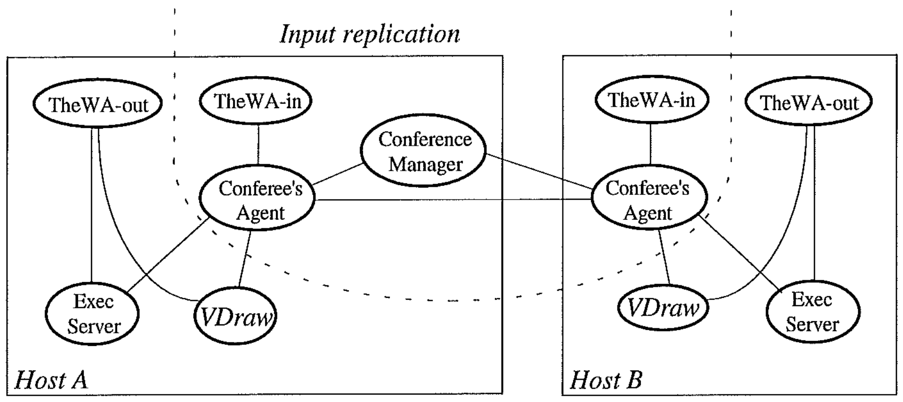
A third approach is to alter the client-side library, the standard low-level programming interface (Figure 3-3). This solves some of the above problems, as the shared library has access to the state of the client, but client applications must be relinked with the new library and the library still has to support common data on heterogeneous displays. Altenhofen's approach [Altenhofen91] is to limit the display to those facilities provided by the least powerful server being used.

Figure 3-3 Integration in the client library.



Finally, a hybrid approach is to replicate the application at each site but distribute the input so that each display looks the same. This was used in VConf [Lauwers90] which took advantage of the separation of input and output streams in the architecture of the *TheWA* window system (Figure 3-4): input devices are managed by a *TheWA-in* process, whereas output is managed by a *TheWA-out* process. Each host has a Conference Agent which routes input under the direction of a common Conference Manager; input is accepted from only one *TheWA-in* at a time and broadcast to all the members of the conference. Only events which affect the state of the application are broadcast, as against events which are local to a particular host; Mermaid [Ohmori92] makes a similar distinction. Each copy of the application, however, communicates directly with its local *TheWA-out*.

Figure 3-4. Replication of input for VDraw in TheWA window system.



Sharing a single-user application by distributing its input and output must require compromise as it is an unanticipated use of the software. The consensus appears to be that distributed window systems use the pseudo-server approach, as this requires the least change to existing software and workstations are now powerful enough to support the extra computation required. Single-machine window systems, such as Timbuktu, tend to distribute events between servers, as the division between application and graphics system is not so easy to intercept.

3.3. Multi-user applications

Patterson [Patterson91] argues that there are three main implementation dimensions in which single-user applications differ from multi-user applications:

- *Concurrency:* single users schedule their actions in predictable ways—they are unlikely to press two buttons at once—and they will tolerate delays resulting from their own actions, but multiple users introduce a new level of unpredictability into the dialogue between user and application and delays introduced by others are seen as system delays. This implies that pre-emptive scheduling is needed in at least part of a shared application to handle the interaction dialogue of each user.
- *Abstraction:* the separation of underlying information structures from their display is a user interface commonplace. This can be ignored in some single-user systems to make development easier but is impossible where several people are to view the same abstract object. On the other hand, the need for multiple views provides a focus for distinguishing data structures from their views and makes the discovery of abstractions easier.
- *Roles:* any multi-user system will need to characterise different users, including their access rights and preferences. Aspects of this characterisation may not be visible to the users and may be changing dynamically—as Beck and Bellotti point out, the membership of a group may be constantly changing—but a shared application may need to tailor users' views of the underlying data to suit their roles.

A synchronous shared application must distribute its input and output but there is a long-standing discussion about how this should be done, particularly over the extent to which the components of the software should be replicated at each site. Replication can reduce network traffic and improve response times by managing output on the user's local processor but centralisation avoids problems of inconsistency between sites.

An extreme case of centralisation is Commune [Bly90] which is driven by a single processor. More commonly, a central application drives windows on several personal computers, as with WScrawl (described in [Greenberg92]). The users' computers provide the window system, but the central process must handle all user interactions: refreshing exposed windows, popping-up a user's private menu, and so on. In addition, a single-threaded application may block everyone if one site suffers delays. The next step is to maintain a central store for the underlying data but manage the user interaction on the users' machines. For example, the Abstraction-Link-View [Hill92] approach for the Rendezvous architecture [Hill93] uses constraints to link the shared abstraction and view components. The *abstraction* provides centralised access to the shared underlying information, whereas a *view* presents the information to the users so they can modify the information or its display. Views of the same information can, of course, differ from each other, the same numbers might be displayed as a table or graph. *Links* are bundles of constraints which maintain consistency between an abstraction and its views, they work in both directions and so provide inter-process communication. Thus, when a user changes a value in a view, the link updates the value in the shared abstraction; the other links to the abstraction, in turn, then update their views.

Bentley *et al* [Bentley92] take this partition a step further. A User Display has three components: a *selection* is a set of entities which are dynamically chosen from the information store according to selection criteria, such as all the aeroplanes in a given sector; a *presentation* is a set of *views* to represent the entities in a selection—the view used depends on presentation criteria, so aeroplanes at different heights may be shown by different symbols; and a *composition* is a set of positions which represent the spatial arrangement of the views in the user display, subject to composition criteria, so a list of aircraft may be ordered by landing time. User Display *agents* keep User Displays consistent with the shared information space they refer to. To present the same User Display on several screens, *Surrogate* User Display agents are available which hold only the current state of the User Display held by a *Master* User Display agent[1]. The shared display state is held by the Master agent, but local customisation may still be possible in the Surrogate agent.

The next option is to distribute the whole application, running a copy of the same application at each site. Colab [Stefik87] maintained copies of the database at each site, multicasting events between them. Colab distinguished three types of operation on a data object: *user inputs* control the user interactions, such as a mouse drag, that specify the changes to the database; *display actions* update the display following changes to the database; and *semantic actions* make the actual changes to the database and are the only operations broadcast. Actions were broadcast optimistically, without a

[1] confirming the remark often attributed to David Wheeler that the solution to any problem in Computer Science is another level of indirection.

centralised controller, on the grounds that clashing operations are rare and can be dealt with socially by the users. The MultiG TelePresence project [Fahlén93] replicates the state of its virtual world at each site and uses *virtual synchrony* [Birman91] to ensure consistency. GroupDesign [Beaudoin92] exploits its knowledge of the application to allow simultaneous operations on the same object to proceed if their effects do not clash—for example, changing the colour and location of a rectangle.

3.4. Summary

This chapter describes a number of approaches for sharing applications. Unfortunately, most current user interface architectures did not consider groupware issues in their designs, and implementers of multi-user applications must often struggle with the window system to provide the functionality they need. The architecture of the window system is particularly important for collaboration-transparent sharing as is shown by the greater difficulty of implementing window sharing over X as against over *TheWA*. Collaboration-aware applications could also benefit from direct window system support, or at least by allowing suitable facilities to co-exist with single-user environments. Navarro et al [Navarro92] propose an open CSCW architecture that would allow users to assemble a working environment out of compatible CSCW components as they require; at least one of those components should be a window system.

4. Computer Systems Trends

4.1. Introduction

Most current window systems are based on models defined a decade ago, but the infrastructures over which they operate are changing. One trend is an "opening up" of computer architectures, particularly operating systems, so that components that once were part of the kernel are being moved into user space and so made easier to alter or replace. Another trend is the ferocious rise in network bandwidth which, combined with new techniques for sharing that bandwidth between media, promises to make continuous media widely available without the need for dedicated hardware or cabling.

A panel session at UIST 1990 [Jones90] discussed the relationship between operating systems and user interfaces. The participants described facilities they would like to have available to support the development of advanced user interfaces, including: better support for real-time scheduling, object-based persistent storage, and interoperability for class libraries and data transfer. Many of these facilities have been available in research operating systems for some time but are only recently becoming more widely used. Another interesting issue is the tension between the need for transparency, to insulate users and applications from the details of a particular operating system or device, and the need to avoid transparency, for time-critical applications and to help resolve performance problems. There have been similar discussions amongst the network community about the difficulties of providing efficient performance when the details of an implementation are hidden in the protocol stack [Clark90].

This chapter discusses current work in networks and distributed systems which is changing the assumptions on which current window system designs were based. It then describes a workstation architecture which exploits such developments and shows how computer infrastructures may change. At present, few existing user interface systems take full advantage of these technologies, which suggests that there is an interesting path to be explored.

4.2. Networks

High speed networks are approaching data rates of several Gigabits per second [Wittie91]. To support time-critical traffic (audio and video) over such networks many groups are experimenting with *Asynchronous Transfer Mode* (ATM) (see [Leslie93]). This technique allows users of the network to make statements to the communication subsystems about the relative importance of various properties for a given connection (latency, jitter, reliability). For example, when sending video data, it is more important that the data arrives in time than that all of it arrives reliably, whereas the contrary is true when sending text.

Until recently, there were two approaches in common use for multiplexing multiple channels over a single network. In Synchronous Transfer Mode (STM) a *frame* is transmitted at regular intervals and each channel is allocated a *time slot* in that frame. This approach is simple and efficient to implement, provides guaranteed bandwidth to each channel and, although the costs of allocating slots are high, the switching time is minimal; it is typically used in telephone networks for digital voice. It suffers, however, from inflexibility and expense as the number of channels is bounded by the hardware and empty slots in one time slot cannot be used by another channel. In Packet Transfer Mode (PTM) each node decides independently when it wishes to transmit and attempts to monopolise the network for a time to do so; there is a protocol for resolving contentions when more than one node tries to transmit at once. PTM networks need minimal set-up time but may have significant switch time, so it is possible for a pair of nodes to use all of the available bandwidth for a time, but there is no guarantee of when a packet will arrive. This mode is suitable for data transfer between general-purpose computers as the traffic tends to be bursty and asynchronous.

ATM represents a compromise between STM and PTM. It is characterised by a fixed cell size, asynchronous access and bounded access time. The fixed cell size allows fast switching in hardware and their small size reduces the grain of multiplexing and contention resolution, which reduces delay and jitter. One major advantage of ATM is that a single cable can carry a wide range of traffic. There is also an advantage over STM in that a channel does not need to retain bandwidth space when silent, reducing the cost to the caller and freeing bandwidth for other users. Better control of access to the medium than PTM means that, when the network is overloaded, packets tend to be queued in the *transmitter*, rather than lost in the receiver or a router; in addition, the reduced jitter makes rate-based flow control easier. On the other hand, the simplicity of the lowest levels makes implementation of the whole protocol stack more difficult.

ATM becomes useful when a network is shared by many channels with different traffic characteristics, so there must be a mechanism for arbitrating between the demands of those channels. Two end-points establish communication by setting up a *virtual circuit*. A route through the network infrastructure is associated with the circuit (although the network management may close a circuit) so packets on a virtual circuit will arrive in the same order as they were sent. Many end-points will need to establish virtual circuits over common resources which, if unconstrained, may lead to congestion and so make traffic guarantees impossible. The solution is to require each end-point to describe the traffic it will offer in a *Quality of Service* (QoS) specification. QoS may be described in terms of bandwidth, burstiness and, for multi-service traffic, maximum delay, jitter and cell loss. The routing mechanism may then determine whether it can honour the connection request or offer one with a lower QoS; it may also break lower priority connections to free network resources. Thus, an application can negotiate with the network over a QoS it would prefer and one it is prepared to accept. Nicolaou [Nicolaou91] describes an application-level language for specifying QoS requirements.

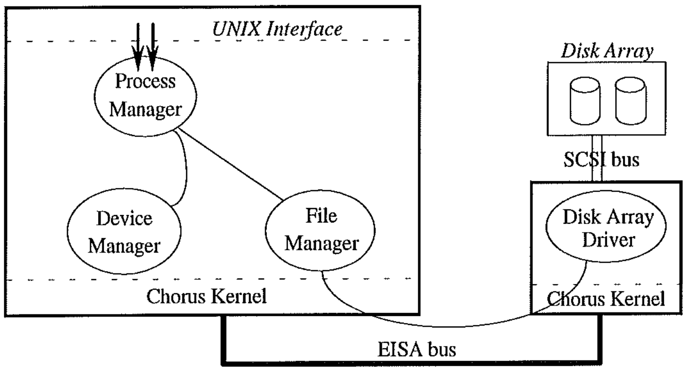
The significance of ATM to application builders and users is that it provides a common infrastructure for multi-medium communication. Users no longer need special cabling for time-based media so individual entry costs are lower, particularly as standard workstations become fast enough to support such media without extra hardware; examples include Pandora [King92] and ViewStation [Tennenhouse92]. In addition, the Broadband Integrated Services Digital Network (B-ISDN) is to be based on ATM, providing integration between local and wide-area networks. For example, MERMAID [Ohmori92], a distributed multi-media desktop conferencing system, has used ISDN to transmit video within Japan and ROCOCO [Clark92], a project to study collaborations between designers, has run a multi-user sketching program between England and Australia. Subject to hardware limitations, the speed of light (see [Kleinrock92]) and the telephone budget, ATM allows a user to say something about when a packet will arrive, even if the source and destination are on different hosts.

4.3. Micro-kernel Operating Systems

The collection of papers in Karshmer and Nehmer [Karshmer91] makes clear the current trend towards micro-kernel (μkernel) based operating systems. As application demands become more complex it is decreasingly likely that they can be satisfied by a general purpose, monolithic operating system. One solution is to build an efficient μkernel which provides processor scheduling, core memory management, and inter-process communication. The efficiency of a well-written μkernel with fast inter-process communication, as against calls to kernel procedures, means that services, such as file systems and high-level communications, can be implemented in user-level processes. For example, to locate a file, a monolithic operating system like UNIX calls a system subroutine which runs within the kernel, whereas an application on a μkernel operating system would send a message to the directory service. In addition, traditional operating systems can be implemented over μkernel systems so that existing software may still be run; for example, Chorus and Mach support UNIX, and L3 [Liedtke91] supports MSDOS. Programmers can use conventional operating system services but also have access to low-level facilities without bypassing the operating system kernel or rewriting parts of it.

This modularity makes maintenance easier and allows flexibility and specialisation when configuring a machine. Armand [Armand91], for example, has experimented with user-level device drivers over the Chorus architecture. The Chorus kernel provides *actors* (which define an address space) that contain *threads* (one or more of which run within an actor) and exchange *messages* through *ports* (destination ports can be on the same host as the source or remote). The storage hardware might be on a different host from the one the user is working on, but moving the filing subsystem to the storage host may create a lot of network traffic for requests which do not access the disk—such as operations on pipes or on cached data. Separating the filing subsystem from the device driver allows the two parts to run on the most appropriate host (Figure 4-1).

Figure 4-1. A file subsystem and device driver on different hosts. Many file system requests can be handled within the File Manager, reducing traffic to the disk array host.



A user-level device driver is also easier to share between clients so that, for example, a UNIX-style file system and a persistent object service on separate processors can have concurrent access to the same hardware without one being superimposing on the other. User-level drivers are easy to change dynamically, new types of service can be added without installing a new kernel and so stopping the processor (as Needham [Needham91] points out, as machine memories grow very large, their state becomes more valuable and rebooting less acceptable). Finally, communicating with a device through an actor allows system servers to run without using privileged instructions, strengthening "firewalls" between components and making possible user-level debugging of subsystems. The implementers of L3 also found that user level device drivers and file systems were efficient and flexible. They could run multiple file systems at the same time and so experiment with different name spaces and access methods or different versions of the same file system.

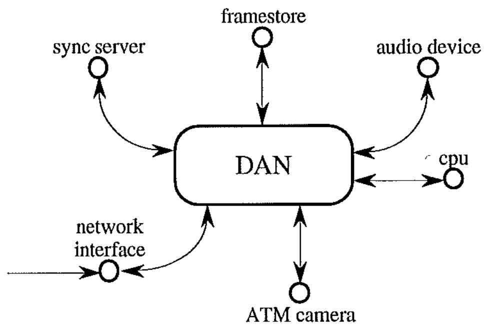
The *Choices* object-oriented operating system [Campbell92] has similar levels of indirection and, hence, flexibility. Hardware devices are accessed through three major components: the *Device* object provides a service for a logical device to other components in the operating system and the *DevicesController* provides DeviceObjects with access to the hardware. Only Device objects may be clients of DevicesControllers; other client types must go through a Device. A Device may control more than one DevicesController, and several Devices may be clients of the same DevicesController. The *DevicesManager* supports changes to devices and controllers. There is only one DevicesManager object per system; it instantiates DevicesControllers when hardware is added and produces new Devices objects on request which are then made accessible via the name server. Other objects find Devices objects through the name server and may then ask for one of a number of interfaces, according to the level of access they need. Devices and DevicesControllers communicate by passing typed Command objects, such as *FlushOutputCommand*, which allows a Device to be used with different DevicesControllers or different versions of the same DevicesController, without changes to the two class hierarchies. Although this makes static type-checking impossible, the interface is internal to the device management subsystem and so causes few problems and is redeemed by its great flexibility. Similarly, Mapp [Mapp92] has implemented an object-oriented virtual memory system in which most of the work is done by a user-level object manager. The kernel need not know everything about the characteristics of the objects in its core memory and paging policies can be changed by substituting different object managers rather than rebuilding the kernel.

This partitioning of operating systems into separate components, however, has not been extended to higher level software. Most current user interface systems are monolithic, assuming a static collection of devices bound into a desktop terminal, with little opportunity to change the devices or how they are managed. Modern operating systems show that very little functionality *must* be embedded in the kernel and that the assumption of a central unit containing all the devices no longer holds. This approach has yet to be applied rigorously to window systems. A useful comparison is with distributed file systems. Few workstation users would accept that they have to restart the machine with a different copy of the operating system in order to mount a disk, yet that is what is now required for the window system to add a new input or output device.

4.4. The Desk Area Network

An example of the application of some of these trends is the Desk Area Network (DAN) [Hayter91], which is an attempt to solve a problem inherent in the traditional interface between workstation and network. The purpose of carrying large amounts of data across a network is to deliver it to a device or processor within the end system, so some thought must be given to the transfer of data between the network interface and the real endpoint of the communication, such as a framebuffer for video data. In a standard workstation, incoming data is demultiplexed according to information held within the packet and sent, via the bus, to its internal destination. This locks up various central resources, such as the bus, during the transfer so nothing else can happen in core parts of the workstation. As Tennenhouse points out [Tennenhouse90], current workstations are monolithic *externally* (a standard set of devices bound to a box which is assigned to an individual user) and *internally* (the internal routing is designed for computation-intensive activity in which only a fraction of the network and storage traffic within the workstation is intended for the display).

Figure 4-2. A Desk Area Network. Several different devices are connected by a small ATM network.



A DAN can be thought of as a small distributed system with devices, such as displays, cameras, and processors, directly connected to an ATM switch. Figure 4-2 shows a DAN with, amongst other devices, a framestore and audio device; a *synchronisation server* (see [Sreenan93]) may be used between the incoming connection and the output devices to ensure “lip-synchrony” for networked digital audio and video. Communication paths within the DAN are considered reliable, so no effort is made to deal with cell loss or broken circuits. A failure of one component is considered to be a failure of the whole system, as is the case with a conventional workstation, which greatly simplifies the protocol software. The small scale of the DAN means that there is not enough traffic to benefit from statistical multiplexing but, in compensation, this allows the controlling software to be aware of all the communications demands being made and arbitrate accordingly.

When a DAN is connected to an ATM LAN, the network interface is reduced, in effect, to a switch which routes packets to the relevant device. The in-band processing, such as header translation and routing, is greatly reduced and may be implemented in hardware—unlike the demultiplexing step in conventional workstations. The network interface is also a security boundary and so may be the best place to provide encryption and decryption services. This approach does not involve passing continuous data via a common bus or central CPU and so avoids blocking other activity within the workstation. One can view the DAN as blurring the boundary between workstation and network so that data connections can be routed, with a user-specified QoS, from end-device to end-device. This is unlike current window systems in which graphics and control data are transmitted over reliable byte streams, with unpredictable delivery times, and continuous data have to be sent by another mechanism (and possibly other hardware) to reach the end-device

4.5. Mobile Computing

Tetherless personal communications have been improving rapidly; Cox [Cox92] describes a range of communication types, from those limited to within the home to portable satellite dishes, and the technologies to support them. A number of researchers have been investigating the application of this technology to mobile computing, Leslie et al [Leslie93], for example, propose a *wriststation*. Some have concentrated on infrastructure, for example by implemented an internet protocol for mobile hosts [Ioannidis93], whereas others have looked at possible new applications.

The Active Badge [Want92] is a small-scale, clip-on tag (weighing 40g) that sends infrared signals to wall-mounted stations distributed around a site. A badge periodically broadcasts an identifying signal which is sent by any local stations to a badge database so that the current position of the badge (and, implicitly, its wearer) is kept up to date. A user may also press a button on the badge to generate an extra signal which can be sent to clients of the badge service. For example, active badges have been used with Khronika, a general event-handling service [Lövstrand91], so that users can have their mail status spoken to them when they click on their badge. The latest version of the badge can also accept input from the wall-stations to play a short tune.

Xerox PARC are building prototype notepad-sized (Pad) and hand-sized (Tab) portables [Weiser91] [Weiser93]; they view these as eventually becoming so cheap and numerous that a user can dedicate them to particular tasks—as cardboard folders are at present (see section 2.6). The Pad uses high-speed radio links, whereas the Tab uses slower infra-red; both are stylus-based. Such devices are expected to become most useful when there are very many of them, so that users inhabit an environment with many "intelligent" components, rather than being restricted to a single user interface. As Cox points out [Cox92], a number of media will be needed to support the range of communication settings (building, campus, region, etc.), and a single network connection to support all these functions has not yet been developed.

Sheng et al [Sheng92] present a design for a radio-based portable multimedia terminal that will support video. Apart from pen input, the terminal is passive; user computations are run on servers connected by a wired network to the radio base-stations, rather than on the terminals. The key to providing video on the terminal is the observation that continuous media have a greater tolerance of bit errors than "normal" data and so can deal with the higher error rates of wireless communication by (more or less) ignoring it; normal data, on the other hand, must still pay the overhead of an error-correcting protocol. This combination of error-free and error-prone communication distinguishes the multimedia terminal from a wireless notebook computer and could be regarded as a simple (and hard-wired) form of QoS specification.

The PenPoint system [Carr91] has a different approach to supporting mobile computing. Long-term storage for documents is held on a network file system, but the network connection to that storage is assumed to be unstable—for example, the portable may be removed without first notifying the system. When the portable is connected, the user's changes are committed immediately, as with any desktop system. When the connection has been broken, however, the user's changes are stored until the portable is reconnected, at which point the changes are automatically uploaded to the permanent host.

The rise in mobile computing signifies another potential direction away from current desktop computing. Most of the portable devices being investigated are either terminals or independent units, whereas the Active Badge is a mobile input device. One can envisage an environment in which most input and output devices are mobile, providing extra information about the location and identity of their users; this approach is developed further in Chapter 11.

4.6. Summary

This chapter describes some trends in computing systems. Faster network hardware, combined with techniques to manage multi-service traffic, promises to make multi-medium communications commonly available, even across wide areas; this includes support for continuous media and synchronisation between media streams. Modern operating systems are based on micro-kernels with much of what was once considered core functionality moved into user-level processes; they are no longer designed to manage a single monolithic host but to tie together many hosts and devices. This allows more flexibility in choosing which components to use and on which machine to run them. These trends are being exploited in the DAN, which supports time-dependant traffic within a workstation without intruding on its other components, and allows events to be scheduled from end-device to end-device. Finally, the chapter discusses some work on mobile computing which shows that hosts can no longer be relied upon always to stay in the same place, and that devices can be associated with users and user tasks.

To summarise, developments in computer infrastructure suggest that we should not rely on current assumptions about how computers are used or how user interface architectures should be structured. In particular, the decomposition of operating systems has not yet extended been into user interface software. Fast networks and efficient micro-kernels mean that interactive response times acceptable to people can be achieved with input devices attached to a local network, rather than built into the display host, so interactive systems can be built which do not assume, or require, a fixed hardware configuration—allowing devices to be attached and detached as the need arises. Previous chapters have provided a motivation for greater flexibility in user interface architectures and this chapter has described the infrastructure which can support such flexibility. The next chapter discusses the current state of user interface architectures.

5. User Interface Architectures

5.1. Introduction

This chapter provides a brief overview of some distributed user interface architectures. With the spectacular exception of NLS/Augment [Engelbart88], which included shared hypertext and computer support for meetings, many graphical user interface systems have been motivated by the notion of the *personal* computer. Kay's notion of a Dynabook [Kay88] was a portable intelligent notebook, for which the Alto [Thacker79] was intended as a interim desktop implementation [Lampson88]. Hence, although Wirth, for example, wrote a multi-user version of the document editor CIL [Swinehart92], distribution was largely at the level of a common file system, electronic mail and remote debugging. As distributed systems, however, became more widespread and networked graphical workstations became widely available, a number of researchers experimented with networked window systems to provide distributed graphics. In the following section, I discuss three influential network window systems and some issues arising from their design and implementation. Another area of research has been the construction of virtual realities, environments in which the graphic objects the user sees respond as if governed by the laws of physics. Many such systems use unusual interaction devices such as data gloves or head-mounted displays which, combined with the need for near real-time animation, place greater demands on the user interface system than window system dialogues. In the next section, I describe some architectures for virtual realities and how they support the demands made on them.

5.2. Networked Window Systems

5.2.1. Introduction

The basic issue when considering distributed graphical applications is choosing which level to distribute. Does one transmit many individual pixels or a shorter description of structures to display? Does one return selection events as points in a plane or with reference to the structure of the image being displayed? This decision also has to take into account the conflicting demands of generality (more and lower-level traffic) and efficiency (less and higher-level traffic).

This section describes three window systems, of which X has come to dominate but the other two had considerable technical merit and are worth studying. VGTS used a high-level protocol which described graphical objects, X uses a low-level protocol which describes drawing operations, and NeWS allows clients to define their own protocol at the level which suits them.

5.2.2. VGTS

Lantz and Nowicki [Lantz84] chose to make the workstation a multifunction component of the larger distributed system, unlike dedicated terminals such as the Blit [Pike83] or personal computers such as the Alto. The use of a distributed operating system, in this case V [Cheriton84], allows the user to run software components on the most appropriate machine—local or remote; the workstation provides a *virtual graphics terminal service* (VGTS) to the user. The main features of the VGTS are:

- The application specifies *what* to draw, not *how* to draw it, the user then specifies *where* to display the result. The server retains a model of the objects to be drawn and provides facilities for viewing them unlike X, for example, where any object models are held in the client.
- The display objects are stored in a hierarchy: an individual object may contain primitives or calls to other objects. The objects are stored in the server in a structured display file.
- The service can be provided by a range of devices. There is a standard interface, the *virtual graphics terminal protocol* (VGTP), between server and client, although servers must be powerful enough to support display files.
- Applications can be distributed over a number of machines. The high-level protocol hides machine dependencies and it may be possible to use the same code wherever the application is running.
- Users can access multiple applications simultaneously.

The system was designed for "mega-workstations": a mega-pixel display, a 1 MIPS processor, and 1 Mbyte of memory. These are connected by a 1 MHz network to a range of dedicated servers (print, storage, compute) and also have access to long-haul networks.

A user can initiate one or more *activities* that may run locally or remotely. Each activity is associated with one or more device-independent *virtual terminals* that may emulate a real terminal, providing text or graphics. The user then maps the virtual terminals to the screen; such mappings are called *views*. A user may create additional views of the same virtual terminal, manipulating each view independently, showing a different pan or zoom, for example. A *view manager* provides a standardised mechanism for the user to initiate activities, and manipulate and move between virtual terminals (and, hence, activities); the view manager also handles authentication when a user wishes to start a session. There are standard editing facilities so that users can copy text or graphics between virtual terminals, and a standard command interpreter shared across applications.

Applications use the terminal-independent *Virtual Graphics Terminal Protocol* (VGTP) to communicate with the VGTS; the VGTP can be thought of as a graphics language that is implemented by the VGTS. The VGTS, in turn, uses the terminal-dependent *Real Terminal Protocol* (RTP) to communicate with the hardware; the RTP can be thought of as a virtual device interface. The VGTS provides, in effect, a translation service between the two protocols.

The designers observed that an interactive program consists of a front-end, for the user to interact with, and a back-end that provides the computation. A VGTS that provides a common editing function also allows some of the interactive response cycle to be short-circuited. In particular:

- the cursor is directly driven by the mouse;

- screen management in the server allows multiple applications to run without interfering with each other;
- applications can wait for significant events, such as selection of an object, rather than tracking the cursor; and
- some editing can be done in the server so that only high-level changes need to be sent to the application.

Such short-circuits provide better interactive response for the user, better use of workstation resources, less network traffic when an application is distributed, and less programming and processing in applications. These optimisations, however, require the server to store a high-level model of the application data, not just its image, so the VGTS stores graphical objects in hierarchical *Structured Display Files* (SDF). Once a graphical object has been defined in a SDF, the user must associate the SDF with a *Virtual Graphics Terminal* (VGT) and then map that VGT to the screen.

The VGTS, however, is not all of the user interface software, it runs with a view manager (described above), an *exec server* and a number of *executives*—instances of the basic command line interpreter. Users communicate with the view manager to manipulate the screen layout and to request new executives which the view manager, in turn, requests from the exec server. Any of these components can be replaced with another implementation, including the VGTS itself.

The authors conducted performance experiments using Sun 8MHz and 10MHz workstations, VAX 11-750 and 11-780 remote hosts, and 3Mbit and 10Mbit Ethernet networks. The most important factor was the speed of the workstation—increases in processor speed produced almost linear increases in the application speed. The next factor was the speed of the remote host where a 100% increase in processor speed produced a 50% speed-up of the application. The third influence was the level of communication protocol—high-level protocols and batching requests together considerably increased the application speed, swamping the effects of increasing raw network bandwidth. Of course, ever faster processors will improve performance until the network becomes swamped with traffic; a faster network will then improve performance, but the authors considered that they had reached the limits of optimising the protocol.

5.2.3. *The X Window System*

Scheilfer and Gettys [Scheifler86] designed the X Window System to be a generic window system, allowing applications to drive windows on displays regardless of hardware or location on the network; it derives from a UNIX port of the W window system, which was built as an alternative to VGTS. X inherited the concept of a network-transparent hierarchical window system but W, like VGTS, required applications to define a window to be one of a fixed set of types, such as text or graphics, which was too limited.

The response was to move many graphics and system functions across the network, requiring the application to store the model of its state. Clients do not download graphical objects into the server but must redraw the image using basic drawing, pixmap and text operations when a window needs refreshing. This imposes fewer restrictions on what can be drawn—the ideal is to provide "mechanism, not policy"—but forces the client to do more work. NeWS (see below), for example, has a device-independent imaging model—images are specified in terms of world co-ordinates and absolute colours, the server must translate these into pixel values on the display. The X imaging model, however, is based on raster operations so clients must get details of the display hardware from the server and perform their own transformations. This processing has been implemented in some client libraries, such as Interviews [Linton89], relieving the application programmer of extra coding and allowing applications with different imaging models to work on the same server.

Desktop management was moved out of the display server into a separate process, a *window manager*. X now defines an *Inter-Client Communication Convention* (ICCC) [Widener90] which provides a standard for clients to pass messages to each other via the server. Applications and the window manager (which is also a client) use this convention to negotiate over access to shared resources, such as screen space. Users can change window management policies, such as input focus, and the interaction mechanisms for controlling them, such as title bars, by running a different window manager rather than changing the server. The ICCC also provides *selections*, so that users can cut and paste between independent clients, and *session management*, which provides a consistent mechanism for starting and stopping clients so that clients can be closed down cleanly.

The major low-level change was from synchronous to asynchronous communication; unlike UNIX (unfortunately), V had very fast networking primitives. Experience with version 10 of X showed that communication costs had a significant effect on interactive performance. In particular, round trip messages were expensive and unpacking messages in the main server loop could be a bottleneck [Gettys90]. Version 11 of X reduced round trip messages by storing more state in the client—the server sends details of its characteristics to the client, that the client is expected to remember, when a connection is first established and allocates to each client a range of resource IDs, so that clients can allocate IDs to resources locally rather than waiting for the server. Version 11 optimised network traffic by allowing variable length messages, the message length is stored in a fixed location in the message so it can be extracted without unpacking the message, and by storing more state in the server. For example, a new resource type, *graphics context* (GC), was introduced to encapsulate the state required for a graphics operation. Once a GC has been defined, the client need send only its ID in a graphics request rather than a complete list of parameters. GCs can be applied to any window so client toolkits, such as Xt [McCormack88], encourage their reuse by providing virtual GCs which map to the same server GC when their attributes are the same.

X is essentially a network protocol which defines a logical display server with a minimum configuration and says as little as possible about the internals of that server. Clients may negotiate with the server over the availability of extra features, such as 3D graphics or extra input devices, but they cannot change it. The low semantic level of the protocol makes it possible for developers to implement new graphic models in clients or libraries and run them on any standard server, so a disparate user community has been able to provide much of the infrastructure.

X, however, was designed for a particular environment and ties us into those assumptions. At one extreme, it cannot cope well with connections which clump blocks of data together (*bursty* connections) rather than transmitting a steady stream [Droms90], there is a round-trip communication every time the user performs an action which changes the display. There are a number of problems with the protocol, described in [Gajewska90], some of which, such as race conditions, derive from mistakes during the design, but others derive from policy decisions. For example, a client cannot retrieve all of a resource's state from the server which makes shared window systems difficult to implement (see section 3.2) The rationale for this is that round-trip queries are too expensive and that clients should remember the information they need—clearly an embedded assumption about how X is to be used. At the other extreme, X cannot take full advantage of new input and output technologies without binding in the extra capabilities as an extension. The only way to add in some local enhancement is to hack the server, a daunting prospect for many sites and not always available. The X server is a mass-production item—reliable (mostly), generic, and not easy to change.

5.2.4. NeWS

The Network Extensible Window System (NeWS) [Gosling89] is a networked window system with some similarities to X in that input and output is provided by a window server which communicates with device-independent client applications over network connections. One major difference is that the protocol is a programming language, PostScript [Adobe85], so clients can download programs into the server. Thus, a client can define its own protocol and send a program to interpret it to the server. For example, if a client sends a rubber-banding procedure to the server, the intermediate activity while a user resizes a display object can all be performed in the server, avoiding network traffic as the client need only be told the final result. This approach is also used for the window manager; rather than running as a separate client, window management routines are loaded into the server and run within it. The extendibility also provides flexibility during the development process. A developer can start by implementing most of the activity in the client then move parts of the implementation into PostScript procedures to be loaded into the server, reducing network traffic and improving local response, as the design stabilises.

The second major difference is that PostScript provides a device-independent graphics model. Clients specify the images they want drawn in absolute co-ordinates and rely on the server to draw them as best it can, so clients know nothing about pixel sizes or colour maps. One consequence is that operations such as scaling and rotating text and graphics, which require much effort in raster-based systems, are implemented in the server at no greater cost than normal graphics operations and are simple to program. NeWS also provides lightweight processes and garbage collection within the server to simplify programming a graphical application with multiple streams of control. Clients of fixed-protocol systems, such as X and VGTS, have no interest in such services as they have little control over the internal processing of the server, whereas NeWS clients may partly be running in the server.

NeWS provides an interprocess communication mechanism based on *events*, which are objects that can be generated by lightweight processes, the server (from input devices, for example) or clients. Processes generate a template event to express interest in a set of events (of a given type on a given canvas, for example) and when an event reaches the front of the server's event queue it is broadcast to all processes whose templates match it. The canvas hierarchy determines the order of the interest search during event distribution.

NeWS provides great flexibility to both the end user and application developer. Clients can be tuned to balance the costs of client and server host processing and network traffic. Program fragments can be developed interactively by sending PostScript text to the server, and the same code that displays an image on the screen can be sent to a printer. This flexibility, however, is not without costs. It is relatively easy for a rogue client to disturb the server; NeWS lightweight processes, for example, cannot be pre-empted, so a function which loops forever may lock the server. This is similar to Apple's System 7 [Apple91] where applications must be "well-behaved" and periodically cede control during long activities to allow multi-tasking on the screen. In addition, the client/server/protocol which a developer implements in one environment may not be appropriate when delivered to the users—even assuming that all the users are in similar environments. Unlike X, where the role of the server is more rigidly defined, system administrators may not be able to predict where the bottlenecks are in arbitrary NeWS applications—the developer may even have built a dynamic protocol to move the work to the least busy host. Of course, all this potential complexity is only necessary if one assumes that network bandwidth and latency are the limiting factors in a distributed environment; protocol optimisations are not an advantage if the application is already delivering data faster than the server can accept it . Finally, major extensions, such as desktop video, still require changes to the basic server, although the generic event mechanism makes easier the addition of input devices.

5.2.5. Summary

This section describes three server-based network window systems. VGTS implemented a graphics service in which the data structures were stored in the server. It provided features such as device-independence and a separate view manager to manage user access to the server, but depended on the (for the time) unusual facilities provided by the V operating system. X, on the other hand, was intended to work on multiple platforms, with standard transport mechanisms and for an unpredictable range of applications, so it uses a lower-level asynchronous protocol. Placing more functionality in the clients has made X very flexible but requires every client to resolve issues such as hardware dependencies. NeWS allows clients to construct their own protocol by downloading code into the server, so clients can adapt the server to their needs but at the risk of disturbing each other and corrupting the server. At present, X is the most common network window system but there are applications it cannot support; in particular, some example virtual realities are described in the next section.

5.3. Virtual Realities

5.3.1. Introduction

A *virtual reality* is an interactive environment where the human-computer interaction attempts to simulate the way we interact with the real world. The environment must accept input and update the display across a range of devices, while preserving the illusion of manipulating a physically-consistent world at a speed which is, to humans, instantaneous. The demands on hardware and software are, consequently, high. The Cognitive Co-processor show how the needs of different asynchronous clients can be resolved in the display service. Both VUE and IBIS show how an interactive system can be (and sometimes must be) made up of components distributed between hosts.

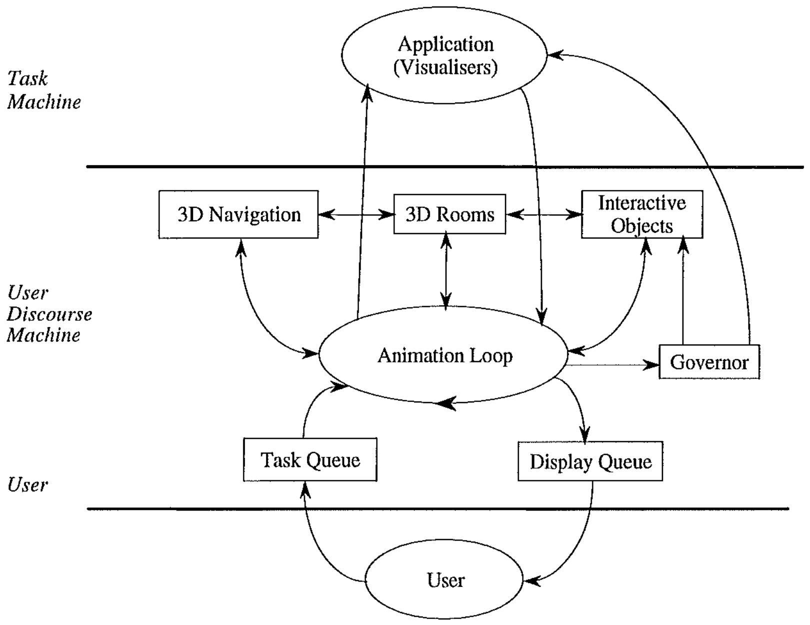
5.3.2. Cognitive Co-processor

The Cognitive Co-processor architecture [Robertson89] was designed to deal with the conflicting requirements of multiple asynchronous agents interacting with a user interface system and smooth animation. An interactive system can be viewed as driven by at least three agents: a *User*, a *Task Machine* (or application) and a *User Discourse Machine*, which mediates between the other two. These agents are, of course, very different, in particular in the speed at which they operate. For example, a search process in an application and the graphical display of its results may be slow compared to the user's ability to understand the results. The user interface needs to provide “impedance matching” between the agents as well as translating between their languages of interaction; that is, it should provide mechanisms for agents to interrupt and redirect each other's work. The conflict is that multiple agents want rapid interaction without being forced to wait for other agents, requiring widely varying computational resources, whereas animation requires steady but guaranteed computation.

The Cognitive Co-processor maintains this three agent division: the user (of course) remains the same, the task machine is an application which contains the state of the virtual world and communicates with components in the user discourse machine (see Figure 5-1). The user discourse machine is based on an inner *Animation Loop* which maintains a *Task Queue*, a *Display Queue* and a *Governor*. The Task Queue contains pending computational tasks from agents; these tasks may spawn heavyweight (operating system) or lightweight (run-time) processes, or directly execute a short procedure. The Display Queue contains mechanisms for painting aspects of the display, in practice it holds all the objects to be redrawn on the next cycle. The Animation Loop must run no slower than some minimum speed to maintain smoothness; the Governor keeps track of how long a cycle is taking so that agents can tailor their activity to maintain constant real-time motion. For example, if an animated ball is supposed to bounce once a second, the distance it is moved for the next animation step depends on how long a cycle is taking. The Application Loop itself is quite simple, it: updates the

Governer's clock, processes pending input from the user or spawned task processes, processes pending tasks on the Task Queue, and updates the display. The user discourse level also contains support for 3D simulated environments, for navigating around those environments and for *Interactive Objects* which provide I/O mechanisms for the user interface.

Figure 5-1. Cognitive Co-processor Interaction Architecture.



While the Cognitive Co-processor is optimised for a single user manipulating data at a workstation, one lesson which is more widely applicable is that user interface architectures need mechanisms to simultaneously handle input streams with very different characteristics.

5.3.3. VUE

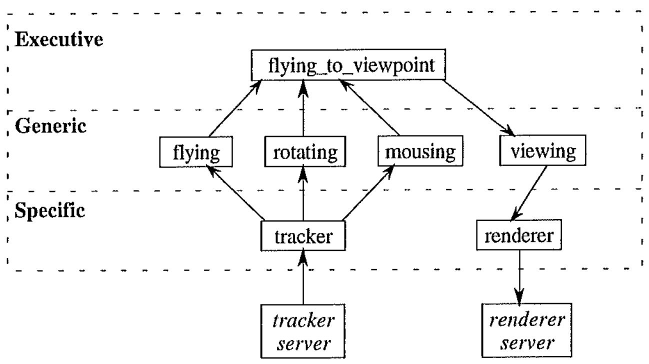
The Veridical User Environment (VUE) [Appino92] consists of devices servers, which manage input and output devices, and application servers, which create the objects in a virtual world that a user perceives through the devices. A central dialogue manager is a client of both devices and applications and binds them all together. It is rule-based, with rules collected into modular subdialogues that define the interaction techniques available in a virtual world. The application server provide the *content* of the virtual world, while the device servers and dialogue manager produce its *style*. All the components use asynchronous message passing to communicate. The authors claim that this division provides increased computational power, as the architecture can easily be distributed over multiple hosts, with relatively low communication bandwidth and natural support for overlapping the processing in the various components.

An application process can have multiple inputs, outputs, and parameters that affect its execution. The parameters are controlled by the devices servers via a mapping established by a set of rules—a *dialogue*; the dialogue is executed by the dialogue manager. Application processes pass parameter values to, and receive values from, the dialogue manager.

The Dialogue Manager coordinates the multiple input and output events in a virtual world; its dialogue is divided into subdialogues—collections of related rules that comprise a small state diagram with shared state. Each rule has the form “INCOMING_EVENT produces RESULT_EVENT” and can perform some state tests to cancel the execution of the rest of the rule. The authors’ experience is that rules should be short and local and that subdialogues should deal with only a single device or interaction technique. Dialogues are divided into three conceptual levels (Figure 5-2):

- at the bottom there are *specific* subdialogues to provide an interface to basic devices. This level interprets low-level details, combining several events into a single message or smoothing jittery input, and passing the event to all the subdialogues above which are interested in such events.
- the *generic* level transforms specific events into more general interactions, independent of a particular device. There is not necessarily a one-to-one mapping between specific and generic subdialogues—a moving hand technique may be composed from tracker and glove events. Similarly, a tracking event may be sent to both flying and rotation subdialogues.
- the third, *executive*, level may then chose between the flying and rotation techniques according to the state of the virtual world and the current interaction modes. This level also handles higher level co-ordination.

Figure 5-2. The flow of tracker events in a VUE dialogue

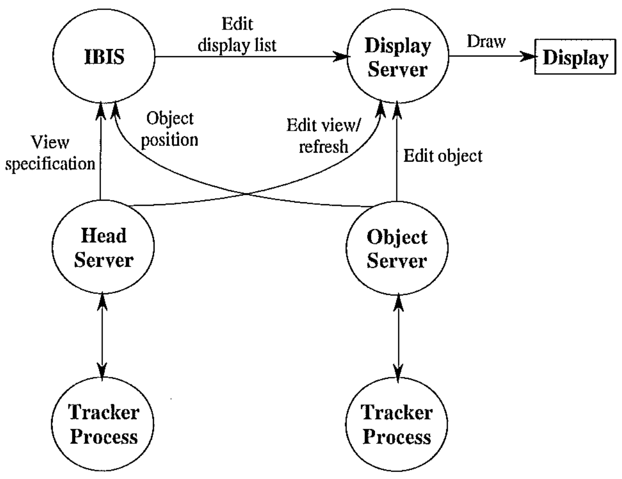


The division between content and style, combined with the use of message-passing to communicate, provides a flexible architecture. Device and application processes can be run on any host on the local area network and added to or dropped from the dialogue manager while it is running. In addition, the division of dialogues into independent subdialogues means that interaction techniques can be dynamically changed by remapping the subdialogues for a set of events. Furthermore, the use of message-passing throughout makes the implementation of multi-user worlds easier by providing each user with a dialogue manager and passing broadcasting events between them [Codella92]. VUE is a large and complex system because it has to maintain the illusion of an entire virtual world, but it demonstrates how a flexible user interface architecture can be composed of communicating processes.

5.3.4. KARMA

A smaller-scale virtual world is the Knowledge-based Augmented Reality for Maintenance Assistance (KARMA) system [Feiner93] (Figure 5-35-3) which uses a “see-through” head-mounted display to overlay a 3D virtual world on the user's view of real-world objects as the user moves around them. Both the user's head and the objects being worked on are tracked with position and orientation sensors so that the virtual and physical worlds can be aligned in the user's view. The example application is to provide dynamic assistance for laser printer maintenance; the user sees parts of the printer highlighted or shown in the position required for the next action while working on it. An Intent-Based Illustration System (IBIS) process uses knowledge-based techniques to decide which virtual objects to display according to specified *communicative goals* (what an illustration is supposed to convey) subject to *design* (what to show) and *style* (how to show it) goals. The head-mounted display is driven by a display server that maintains a 3-D display list. The position and orientation of the user's head and the objects being manipulated are tracked by processes which feed the current positions to head and object servers. The head and object servers then notify the display server of user or object movements so that the user's view can be updated—these updates are atomic to ensure that the display server remains consistent. The servers also notify the IBIS process at the same time so that it can adjust its illustrations. Sending multiple messages allows the processes to operate in parallel (user interactions with the current illustration do not reference the IBIS) and avoids the IBIS becoming a bottleneck or a source of extra latency.

Figure 5-3. Message passing in KARMA.



KARMA provides an example of an user interface architecture which is not based on a virtual terminal approach—there is no straightforward user-machine dialogue, the display is driven by the user performing everyday real-world actions. As with VUE, the KARMA architecture is constructed from a set of distributed processes providing specialised functions.

5.3.5. Summary

This section describes three examples of virtual reality systems. The Cognitive Co-processor shows that different applications may have different scheduling requirements for their output, which current window systems do not support. The VUE architecture shows how a user interface can be built up from multiple components supporting a changing set of devices and the dynamic interpretation of their input. KARMA superimposes a virtual world on the physical world, also building up a user interface system from a set of communicating processes. It also shows another way of interacting with a computer than by “direct manipulation”.

5.4. Summary

This chapter describes a number of distributed user interface architectures designed for a range of applications and operating systems. The X Window System has proved the most popular of the networked window systems, partly because of accidents of history, but largely because its low-level protocol allows each client to implement its own imaging and interaction models. Not every application, however, can run over X; in particular, it provides no support for scheduling between clients or for more complex forms of interaction. In addition, the networked window systems were all designed to manage single-user workstations which, as has been shown in chapters 2 and 3, is a restrictive model; they cannot easily support an interactive environment in which the set of input and output devices changes dynamically to meet the users’ needs.

Some virtual reality architectures have approached these issues and provide mechanisms to support smooth animation, multi-modal interaction, dynamic system configuration and so on. As yet, however, these techniques have not been implemented in a common virtual reality toolkit or folded back into window systems. Furthermore, virtual reality systems only allow users to collaborate in the computational world, rather than supporting collaboration in the real world by allowing people to share physical devices.

6. Summary

This part has surveyed current developments in human-computer interaction, computer systems, and user interface architectures. The first chapter described how user interface requirements are changing. The study of Computer-Supported Co-operative Work has produced a more sophisticated understanding of how people and computers interact, and it is clear that the model of the single-user in dialogue with a single-machine is inadequate for many situations. Some user interfaces incorporate continuous media which become particularly interesting when integrated with existing digital media. Finally, some researchers are looking towards a new generation of user interaction in which there will be large numbers of portable devices and the bandwidth between user and computer can be increased by using a wider range of input and output media and greater parallel activity.

Chapter 3 described strategies for implementing multi-user applications, both collaboration-transparent and collaboration-aware, and notes how support in the user interface system would facilitate their implementation. Future user interface architectures must be based on different assumptions about use than current designs and provide greater flexibility.

Chapter 4 discussed current trends in computer systems. Networks have become much faster and allow processes to specify a minimum Quality of Service for a connection, providing support for networked audio and video. Operating systems have become more modular and open, so that users can assemble a computing environment from a set of host and services rather than relying on a monolithic operating system. The Desk Area Network shows how these trends may be applied throughout a workstation, routing data to the appropriate end-device without overloading central resources. Finally, there is now a range of mobile computing devices, although no common integrated communication medium has yet been developed. Computer systems are becoming faster, more flexible and more open, but few user interface architectures have as yet taken advantage of these facilities.

Chapter 5 surveyed existing user interface architectures. It discussed three networked window systems describing some of the tradeoffs in each design. Work on virtual realities shows how media with different characteristics can be accommodated in the same interface and how user interfaces can be built up from multiple communicating components. Virtual realities also show the limitations of the assumption of the single user at the desktop machine.

This survey suggests that current user interface systems have not followed developments in human-computer interaction or computer system infrastructures. The next part describes some experimental applications, noting their implications for the design of user interface architectures.

Part III. Experiments

The limits of existing user interface architectures become most clear when one tries to use them for unusual purposes. The range of applications which have been built over X, for example, show how successful the original design has been (and how ingenious application writers are), but it does have fundamental limitations. This part presents three sets of experiments which look towards models of interaction other than the standard desktop workstation and discusses their implications for the development of user interface systems.

Chapter 7 discusses experiences with the X Window System and the limitations of its model. Chapter 8 describes the Multi-Device Multi-User Multi-Editor which allows people to collaborate on a shared screen. Chapter 9 describes the addition of multiple networked devices to the DigitalDesk and how this supported the development of an application which provides a shared virtual space.

The lessons drawn from these experiments are that:

- current window systems are only a special case of user interface architectures. Users and application builders may reasonably expect to be able to use multiple input and output devices, changing the devices used as the need arises;
- more flexible user interface architectures, which can distinguish multiple input devices and users, can be built without excessive implementation costs; and,
- current window system models are too strongly tied to their assumptions about user interfaces to provide a suitable platform for further evolution.

7. Experiences with X

7.1. Introduction

At present, any discussion on distributed window systems must include a consideration of the X Window System. It has been a *de facto* standard for some time and is becoming a *de jure* standard. As described in chapter 5, X was intended to manage a wide range of hardware which it achieves by only guaranteeing support for a limited model of a display. Other facilities are managed by extensions to the protocol which an individual X server may or may not provide. While a wide range of applications have been built over X, some have required considerable effort from their designers to implement new features, particularly for groupware, as they are based on a different model of user interaction with computer hardware. We should consider whether there are user interfaces which X cannot, or is highly unsuitable to, support.

This chapter describes two experiments, a collaborative application implemented with X and an extension to X, and discusses the implications for window system design. The second experiment was actually implemented after the MMM system described in the next chapter, but is presented here as part of the discussion of X.

7.2. MultiMed

7.2.1. Introduction

The MultiMed project [Unison90] investigated the use of digital telephony, in particular ISDN, to support medical practice. For example, medical advances have greatly reduced the incidence of some conditions, so clinical staff need to consult experts at other sites as such cases arise; in addition, a larger number of medical staff need to see these cases as part of their training.

My responsibility was to build an example application to show how doctors at different sites could use digital technology—colour workstations connected by ISDN—to discuss medical images without the need to travel. The prototype, however, was only intended to work over a local area network as the ISDN lines would not be available during the project. The design of the user interface was motivated by discussion with and observation of the work of a consultant histo-pathologist in a London teaching hospital. In brief, it allowed the participants to share images grabbed from video cameras, such as microscope slides and pictures of dissections, and to use telepointers to draw attention to areas of interest; the participants were also to use conventional telephones to speak to each other during a conference. The user interface is described in detail in [Unison90].

The central concept of the MultiMed application is the *Session*, which is a discussion started by a user at any MultiMed workstation and controlled by that user—the session *owner*. The owner controls the display and placement of images during a session and the lifetime of the session, which ends when the owner leaves. Other users may join an existing session as passive participants and any changes made by the session owner are broadcast to the other participants and displayed on their workstations.

To help remote users discuss the images, MultiMed provides *telepointers*—labelled arrows which are displayed in the corresponding position on the screen of each participant in a session. Any member of a session, not just the session owner, may create any number of telepointers, and a telepointer belongs to the user who creates it. A telepointer's owner may drag it around the screen to draw attention to items of interest, the telepointer movement is mirrored on the other users' screens.

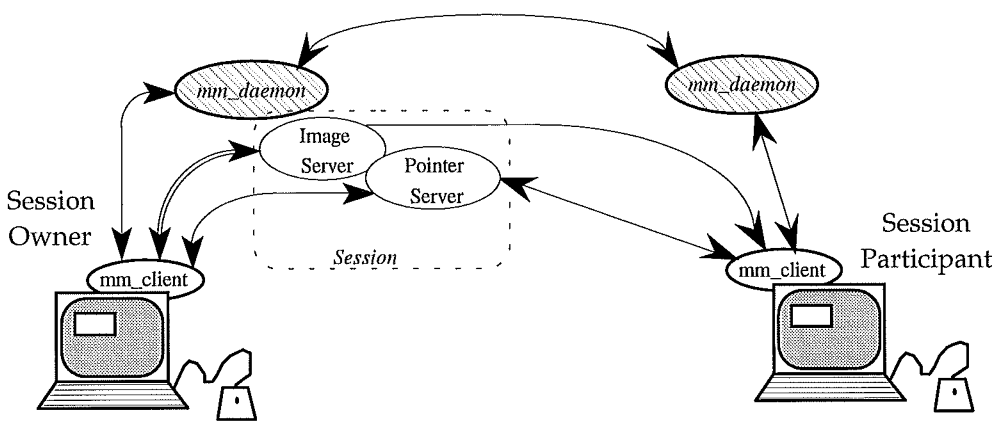
The MultiMed application was implemented using X11.4 and the Athena widget set to ensure compatibility with other parts of the project.

7.2.2. Architecture

MultiMed uses local clients to manage the display, with centralised servers to control the shared data (Figure 7-1). Dæmons on each host keep track of which users are logged on, which sessions are active, and who is a member of each session. To start a session, a user notifies a dæmon which forks the processes that act as servers for the session and then notifies the other dæmons of the new session.

There are two servers per session so that one can handle "lightweight" events, such as telepointer actions, maintaining responsiveness while "heavyweight" events, such as image transmission, take up computing resources. Images are large units of data, which do not change often, and are controlled solely by the session owner. Telepointers involve little data, which may change often, and may be controlled by any session participant. The division of processing into two servers allows telepointer movement and, hence, discussion to continue while a frame is being grabbed and broadcast. Both session servers have a similar structure in that they hold a list of session members and a list of items (telepointers or images), and both receive messages from clients which they broadcast to all the members of the session. Consistency is maintained because all the objects in the session are centrally managed by the servers.

Figure 7-1. The MultiMed architecture. The session owner runs the Image and Pointer servers which session participants can connect to.



User requests (a telepointer motion, for example) are sent by the client to the relevant session server (telepointer or image) which processes the request and then broadcasts the result to all the participants. Thus, any disparity between when the members of a session see an event occur on their screens does not include the delay between the originating user and server, which helps to preserve pointing gestures and to synchronise movement with speech. This is in contrast to systems such as MMConf [Crowley90] or Colab [Stefik87] in which changes are implemented locally before being optimistically broadcast to other participants. MMConf was intended as a user's main working environment and so had to provide good local response and the Colab team reported confusion during meetings due to delays in displaying objects [Tatar91].

To reduce network traffic, the session servers do not directly drive objects on the clients' screens; that is, they are not responsible for drawing or redrawing objects or handling user input. The clients maintain the display and interpret between the user and the session servers, so client and server communicate at the abstract level ("move telepointer 3 to this position" rather than "draw the following set of lines"). This means that as much processing as possible takes place locally, and also makes it easier to adapt the application to work with different implementations of the client. This model is common to many multi-user toolkits, such as Rendezvous [Hill93] or LIZA [Gibbs88]. One can think of the session as a set of objects which are stored in the servers. The clients hold shadows of the objects and define the mapping from the 'true' objects to the screen.

The pointers were implemented as shaped windows and could be hidden by other windows, such as a new image, so they were made "self-raising." That is, when a pointer is obscured by another window it cycles itself back to the top of the stack. To avoid infinite loops, with two pointers repeatedly attempting to raise themselves, pointers are not allowed to overlay each other.

7.2.3. Limitations

A number of issues arose from the implementation of this project, of which three concern the window system: the imaging model, support for overlays, and synchronisation between displays. Some of the other issues raised by Lauwers and Lantz [Lauwers90a], such as floor control or the placement of shared windows, were avoided by restricting the user interface—MultiMed took over the whole screen and was intended for use with similar displays.

X has a pixel-based imaging model—its co-ordinates are specified in terms of pixels and it is up to the client to make any necessary transformations for data specified in absolute measurements. Similarly, each client must implement its own mapping of an image to a colourmap. This is in contrast to the Cedar Imager [Swinehart86] or PostScript [Adobe85] where the output is described in terms of an ideal device and the implementation renders it as best it can. If a shared application shows the same information on multiple screens, pixel-based imaging requires the programmers to implement their own device-independent model to give everyone a consistent view. It is arguable that the standard release of X should have included some support for device-independent graphics so that application writers who do not need per-pixel accuracy would not have to consider different display types—although X's use of bitmapped fonts makes this more difficult.

Much of the effort for MultiMed went into implementing telepointers which would be responsive and work over a colour image (so excluding simple XOR techniques). In the end it was necessary to avoid using part of the toolkit and implement a paint pipeline, with double-buffering, within a basic widget.

Support for overlays could be implemented more efficiently in the server, as happens in NeWS, and would be of particular benefit to those writing groupware applications—as is pointed out in [Lauwers90a].

Finally, X's asynchronous communications, although efficient, mean that it is not possible to know when an event happens on the screen. This can cause inconsistencies between the users' conversation and what each one sees, especially if an X server is occupied with processing a new image rather than showing pointer movements, so applications must implement their own scheduling policies. The Shdr shared drawing tool, for example, [Dourish] waits for acknowledgement of previous telepointer events before sending the current cursor position to avoid events being batched in the network. The ability to assign priorities to graphics requests would make display events more predictable for clients, although the promised multi-threaded X server (in release 11.6 of X) should reduce the problem of a particular action hogging the server.

7.3. An Input Extension Extension

The X Input Extension (XIE) [Patrick89] was designed to allow extra input devices to be used in an X server while staying within the X model. It defines extra protocol messages which allow a client to query the server to see what devices are available and to chose the default pointing and keyboard devices. Multiple devices may also be used simultaneously; more than one message is used for each event to supply the client with the extra information about the source of the device. The main limitation of the XIE, from the viewpoint of this dissertation, is that extra devices must be bound into the server when it is compiled. To plug in an extra mouse, for example, a user must close down the session and restart it with another implementation of the server. While it would be possible to write an implementation of the server which can be prompted to look for particular extra devices, polling a serial port for example, this technique is not general.

To attempt to resolve this problem, I wrote an X Input Extension Extension (XIEE) to provide support for connecting to and disconnecting from remote devices on demand, using sockets to communicate between device servers and the X server. A first step in the description of the XIEE is to understand the structure of the X11 sample server; the implementation described below is that for the Sun SPARC architecture.

7.3.1. Inside the X11 Sample Server

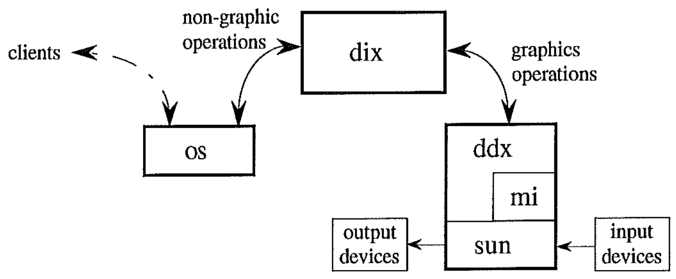
The sample server [Angebranndt88] is divided into several components, (Figure 7-2):

- the *Device Independent X* (DIX) layer contains software which is common across all implementations;
- the *Operating System* (OS) layer contains software which is different between operating systems, but is common to all the devices for a given operating system. It schedules activity in the server, provides communications with the clients, and other utilities such as memory management; and,

- the *Device Dependent X* (DDX) layer contains the software which may vary for each combination of device and operating system. It includes a Machine Independent (MI) library that implements common graphics functions in software.

There is also an Extension Interface that provides a standard mechanism to add features to the server.

Figure 7-2. The structure of the X11 sample server.



Input events are processed in two stages. First, an interrupt handling procedure, `EnqueueEvents`, is called when data arrives at any input device port. The algorithm for this procedure is:

```
LOOP (forever)
  (* get raw data from devices, timestamping them as they arrive *)
  IF pointer.bufferEmpty() AND pointer.inputPending() THEN
   pointer.putEventsInBuffer();
  END;
  IF keyboard.bufferEmpty() AND keyboard.inputPending() THEN
   keyboard.putEventsInBuffer();
  END;
  IF pointer.bufferEmpty() AND keyboard.bufferEmpty() THEN EXIT; END;
  (* put oldest event on DIX input queue *)
  IF Age(pointer.firstEventInBuffer) > Age(keyboard.firstEventInBuffer) THEN
   pointer.enqueueEvent();
  ELSE
   keyboard.enqueueEvent();
  END;
END LOOP
```

The device-dependent part of each device record contains some device state, including a buffer in which to store raw events from the device port, and some procedures to manipulate it. The loop attempts to load new events into any devices whose buffers are empty, then transfers the oldest event from all the devices to the server's input queue, until there are no more events to process. Meanwhile, the main dispatch loop in the server calls `WaitForSomething()`, in the OS layer, which waits until there is a request from a known client, a new client tries to connect, or there is an input event from a device. When `WaitForSomething` returns, the dispatch loop checks for pending input and calls `ProcessInputEvents()` if there is any. `ProcessInputEvents` then dispatches all the events in the queue.

The XIE provides a structure for new devices to be registered so that they are visible to the server and, hence, the clients. The implementer of the new device code must define a procedure (`newDeviceProc`, for example) for the device that responds to four requests: *initialise*, *turn on*, *turn off*, and *close*. The implementer adds an extra call to `AddInputDevice()` for the device in the procedure, `InitInput`, which initialises the X server data structures for input devices. `AddInputDevice` creates a new devices record, held in the DIX, in which it stores `newDeviceProc`. A DIX procedure, `InitAndStartDevices`, later uses these device procedures to initialises all the devices. Finally, the implementer must add the new device to `EnqueueEvents` so that its events will be copied to the main input queue when the device is started.

7.3.2. Adding Networked Devices

I wanted to be able to experiment with an arbitrary set of devices per display, to support an X implementation of MMM (chapter 8) for example, and to be able to move devices between displays, so that several displays on a desk could be driven by a single mouse and keyboard. The first requires that input devices can be added dynamically, rather than compiled into the X server, and both suggest the use of networked input devices so that a client on one machine can connect to a device on another. Unfortunately, `XSendEvent()` cannot be used to transmit basic device events because X events sent this way are marked as such, which some X clients exploit, and because it does not change the state of the device in the X server, which some X clients query. The solution was to provide device servers and extend the XIE to allow X clients to request a connection to such servers.

The XIEE defines an extra request, `XOpenRemoteDevice()`, which allows a client to specify the address of a device server. The X server attempts to open a connection to the device and, if successful, receives a description of the new device and registers it. The client may then gain access to the device using the id returned from `XListDevices()`, which will now return a list including the new device. In the server, the device records are stored in a list, so they can be added and removed dynamically, rather than in a fixed-length array.

7.3.3. Limitations

The main issue arising from the addition of networked devices is how to handle device failures. The previous solution was to stop the X server, but the connection between devices and server has now become more vulnerable. Furthermore, an individual device may no longer be as important to the functioning of the X server as a static device, so stopping the server on the failure of a device may be an overreaction. On the other hand, as Tanner points out [Tanner86], simply ignoring a device when it fails may not be acceptable either; it should be possible to report device failures to the user.

The structure of this implementation of the X server, however, makes such error handling complex as actions to alter device structures originate in a client or in the DIX and are passed down to the DDX, whereas device events are passed up from the DDX to the client. The loss of a network device involves a change to a device structure arising from a device event. Perhaps the best approach would be to define a new event, `X_LostInputDevice`, which would be distributed to the clients of the device as an extension event; clients could then chose whether or not to listen for such events. Cleaning up the data structures for the lost device, however, requires some care, as race conditions may arise if one client is opening a new device while an old device is being cleaned up and both are allocated the same device id. Furthermore, the XIEE should also define server events to notify a client when devices have been added or removed to allow device management tools.

A further point is that the XIE still assumes only one core pointer, although it allows clients to select which input device is used as such. The focus for a second keyboard, for example, may follow the core keyboard or be explicitly set to a particular window, but there is no provision for it to follow an arbitrary pointing device. Similarly, the assumption of a single cursor runs throughout the design of X, although the MI library includes software cursor support which could be extended for multiple cursors. As MMM shows (chapter 8), there are applications and architectures which require more than one.

In the end, the final stages of the XIEE were not implemented as a more practical (in the short term) approach was found; this is described in chapter 9.

7.4. Conclusions

This chapter describes two experiments which highlight the limitations of the X approach to user interface systems. The MultiMed application showed the need for generic support in the server for overlays (to implement multiple cursors, for example), for clients to be able to schedule their requests (to give priority to gestures or events which should be synchronised with other media, for example), and for a common device-independent imaging model (to reduce the effort involved in making clients appear the same on different displays). The Input Extension Extension highlighted the bias towards a particular model of user interaction in the implementation of the sample server, which (not unreasonably) has been optimised for the existing protocol. Extensions to the protocol, such as dynamic devices or multiple cursors are possible but only by breaking conceptual boundaries in the software architecture, increasing the interdependence within and fragility of the implementation.

This is not to argue that X is a flawed design, it is appropriate for the uses for which it was intended (with a few reservations, see [Gajewska90]). The low level of its protocol allows a variety of user interfaces to be implemented over it, but also forces programmers to implement in the client features which would be better implemented in the server. The X Window System provides many benefits in terms of a standard graphics environment for distributed workstations, but two generations of computer hardware have passed since release 10 of X, which suggests that it may be time to consider new approaches. The next two chapters describe experimental applications and discuss the implications for user interface architectures.

8. The Multi-Device Multi-User Multi-Editor

8.1. Introduction

The Multi-Device Multi-User Multi-Editor[2] (MMM) [Bier91] was a project to investigate a software architecture and user interface to aid the development of multi-user editors that are convenient to use. While most previous multi-user editors assume that each user has his or her own networked workstation, MMM concentrated on editors shared on a single display with multiple pointing devices. This restriction allowed us to avoid the problems of co-ordinating multiple workstations and to focus on user interface and internal architecture issues. We also envisioned situations in which collaboration on a single display would be useful in its own right:

- users may wish to collaborate on the same workstation screen. For example, two programmers working together at a workstation may both require input devices to avoid interrupting the discussion by passing a keyboard between them.
- several people can share a computer with a blackboard-sized display by writing directly on the display with a stylus or by controlling it from hand-held computers. In addition, people can share a hand-held computer by passing it back and forth, so a quick mechanism for allocating the stylus to a person would help them tell the application who is currently providing input.
- even when multiple workstations are available, such conference-aware editors will enhance the ability of participants to contribute simultaneously.
- future software applications may provide an "assistant"—a program that acts like another user with which we collaborate on workstation tasks. A conference-aware architecture will make such assistants easier to integrate.

Our architecture also supports a single user with pointing devices in both hands; two-handed interfaces have been motivated by Buxton and Myers [Buxton86]. We believed that large-screen and portable computers will use a stylus as their main input device, so we focused on user interfaces that do not require a keyboard.

MMM consists of a set of simple editors, for rectangles and text, that support simultaneous real-time collaboration. They allow fine-grained sharing, often permitting simultaneous access to the same text string or graphical object. A mouse can be registered with a user and will work with that user's defaults and preferences (such as insertion points, modes, selection colours and mouse parameters)

[2] I spent a summer at Xerox PARC working with Eric Bier on the initial implementation of MMM. The concept and basic structure are Eric's and he implemented, in particular, low-level event handling and the input queue optimisations. Ken Pier added multiple pointers to Cedar. I implemented the bulk of the editors and contributed to the design of the user interface and some of the more specialised structures, in particular, the Home Area editor.

until the device is registered with another user. Registration is sufficiently fast that users can pass devices back and forth during a session. Users can alternately collaborate tightly and work separately with little interference.

The project addressed four user interface problems:

- *registration*: how can an input device be registered quickly with a user?
- *real estate*: how should the screen be managed so that collaboration is practical in limited space?
- *per-user feedback*: how can the system direct feedback to the right user without disturbing others?
- *interference*: how can users engage in separate tasks without interfering with each other?

and three software architecture problems:

- *input handling*: how should input events from multiple devices and/or multiple users be queued and combined to allow fine-grained sharing?
- *replication*: which data structures of a document editor need to be replicated per user, to support per-user modes, selections, and preferences?
- *screen update*: which screen update algorithms will produce an image consistent with the editing activities of all users?

MMM was implemented in the Mesa programming language and runs in the Portable Cedar environment [Swinehart86] on SUN SPARCstations. The workstations were augmented with extra mice to control the additional cursors.

8.2. MMM's User Interface

MMM's user interface consists of three visible components: *Home Areas* provide iconic representations of users; *Editors* allow users to view and modify document media; and *Menus* provide buttons that users can press to invoke commands. This section describes how we use these components to solve our four user interface problems.

Figure 8-1. A typical MMM screen.

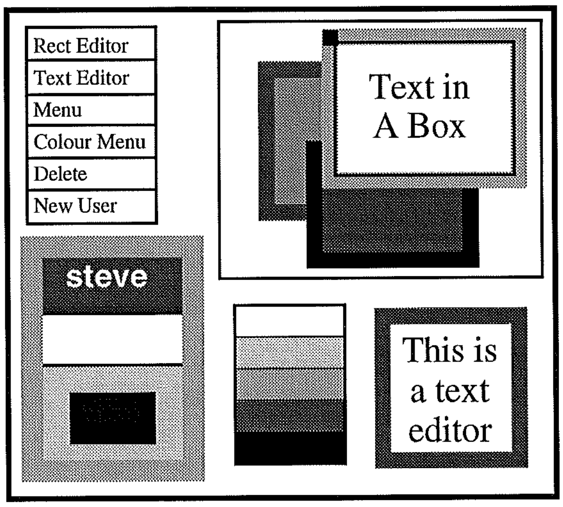
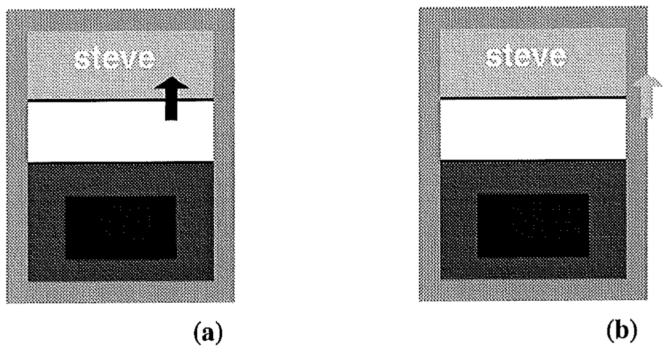


Figure 8-1 shows a typical MMM display with (clockwise from upper left) a command menu, a rectangle editor (with nested text editor), a text editor, a colour menu, and a home area.

8.2.1. *Home Areas*

MMM displays a home area for each user participating in a session. To join a session, a new user types his or her name and clicks the New User button. MMM creates a home area that displays the selected name. To begin using MMM, a user chooses a mouse and uses it to click on the name bar in a home area, as shown in Figure 8-2. As a result, MMM assigns that mouse to that user, and any actions performed with this mouse will take his or her preferences into account. The change of ownership causes the cursor colour to change to that of the home area, as shown in Figure 8-2(b). A user may have more than one home area, and extra home areas allow users to switch back and forth between different sets of preferences. Each home area is said to belong to a different *user instance.*

Figure 8-2. Clicking on a home area to register a mouse.



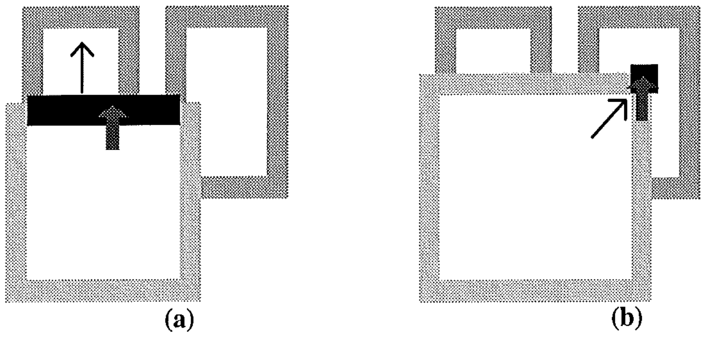
8.2.2. *Two Editors*

There are two main types of editor, for rectangles and text, that are both functionally very simple. The rectangle editor permits users to create solid-coloured rectangles, select and delete groups of them, and change their size, position and colour. The text editor allows users to place a starting point for a line of text, enter characters, choose a font and colour, move a line of text, select a sequence of characters, and change the colour of, or delete, selected characters.

8.2.3. *Spatial Command Choice*

We were interested in keyboardless interfaces, so all editor commands, except text entry, are activated by a pointing device. For many MMM operations, command selection involves pointing to different parts of graphical objects to invoke different operations.

Figure 8-3. Spatial command choice in the rectangle editor. (a) Dragging a rectangle by its edge. (b) resizing a rectangle by its corner. (The black arrows indicate cursor motion; they do not appear on the screen.)



In particular, objects in the rectangle editor are surrounded by a border, a *frame*. The user drags an edge of a frame to move (Figure 8-3(a)), and a corner to resize (Figure 8-3(b)) the rectangle. In each case, the part of the frame being dragged is highlighted. Users create a new rectangle by clicking on the editor background.

Spatial command choice has many advantages in a multi-user context: it reduces the use of persistent modes; it is easy to see—novices can learn from experienced users by watching them, and collaborators can see what others are doing; and, because the cursor need not travel to an off-screen menu, the command motion is unlikely to be misinterpreted as a conversational gesture (see the description of Sketchtool in [Bobrow90]).

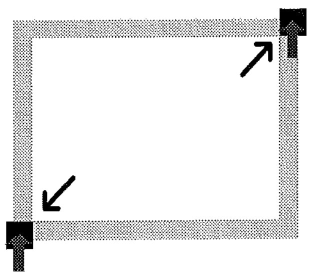
8.2.4. *Per-User Commands, Modes, and Preferences*

Several users can work simultaneously in an editor, performing different operations and using different modes. In the rectangle editor, for example, one user may drag one rectangle while someone else resizes another. The editor keeps track of both operations and updates the screen to show their progress. Similarly, one user may choose blue as the current colour for that editor while another chooses red. The editor stores both users' modes and creates a rectangle of a colour appropriate to the creating user.

8.2.5. *fine-grained Editor Sharing*

Users can work simultaneously on the same object; one user can add characters to a text string while another changes the colour of existing characters. Very fine-grained sharing is also possible; one user can stretch a rectangle by one corner while another stretches its opposite corner, as shown in Figure 8-4. Spatial command choice in the rectangle editor, or one user can type a string of characters while another repositions it. Fine-grained sharing may not often be necessary, but it allows a user to make a change confident that other users will not be locked out, reducing interference among users.

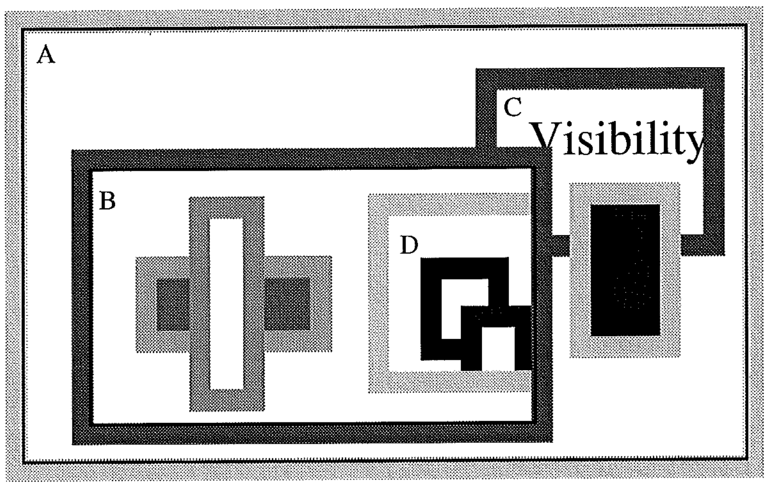
Figure 8-4. Two users stretching a rectangle at the same time.



8.2.6. *Editors as Window Systems*

MMM has no conventional window manager. Instead, the desktop is an instance of our rectangle editor. The rectangle editor allows other editors to be nested inside it. Each editor presents a finite window, or *porthole*, onto an "infinite" plane that contains the objects and child editors that that editor manages, which may overlap. The shape and position of this window is managed by the editor's parent. Editors may be partly or entirely hidden. Figure 8-5 shows several rectangle editors and a text editor. Note that the text editor C is partly hidden by sibling editor B (and by a rectangle), and that rectangle editor D is partly outside the porthole of its parent editor B.

Figure 8-5. The visible parts of four editors. A, B, and D are rectangle editors. C is a text editor. B's highlighted border indicates that a user has selected it.

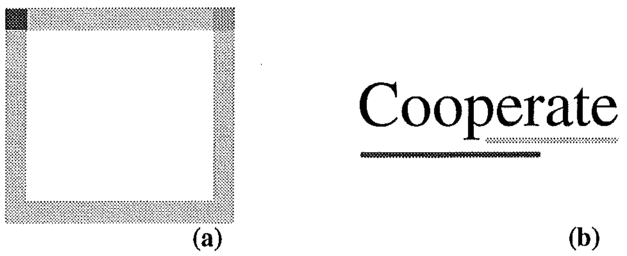


The use of an editor for a window system means that users do not need to learn both window manager and editor operations, unlike systems where selection *of* an editor is different from selection *within* an editor. Users can also place shared tools, such as panels of buttons, in the document in which they are working, as tools need not remain at the top level.

8.2.7. *Per-User Feedback*

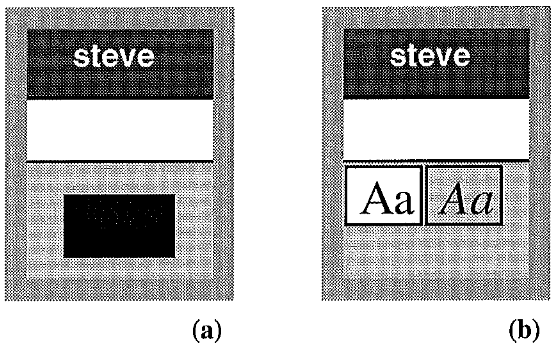
In single-user environments, graphical feedback is used to display aspects of system state, such as the current application, mode, colour, or insertion point; any feedback given is certain to be directed to the correct user as there is only one. This is no longer the case when several people share a screen, so graphical feedback must show all of this information and indicate to which user the feedback applies. We used colour and spatial placement to indicate this correspondence.

Figure 8-6. (a) A doubly-selected rectangle. (b) Doubly-selected text characters.



Each user has a colour to identify his or her cursors, insertion points, and selections. In Figure 8-6(a) two users have selected a rectangle; each user's selection is shown by highlighting a corner of the rectangle frame in that user's colour. Similarly, Figure 8-6(b) shows two selections in a text string, each marked by an underline in one user's colour; the overlapping part of the selection is underlined twice. Likewise, a narrow band in the frame around a user's current editor is set to that user's colour. Where several people are using the same editor, a band is coloured for each user. If too many users have selected an object (rectangle, text or editor) to identify them all in the space available in the frame then only the most recent selections are shown. This is not ideal, but is a reasonable approach for small groups.

Figure 8-7. Mode feedback in the home area. (a) Rectangle creation colour feedback. (b) Current font feedback.



A user's mode within his or her current editor is displayed in the lower half of that user's home area. For example, when a user is working in a rectangle editor, his or her home area displays that user's default colour in that editor (Figure 8-7(a)). Similarly, MMM shows the current font and colours for a text editor (Figure 8-7(b)).

8.2.8. Shared Menus

In many desktop environments, menus are displayed once for each application window or at a unique location on the screen (e.g., the Apple Macintosh pull-down menus are at the top of the screen). For a shared application on a single screen, however, menus displayed once per window take up too much space and menus displayed at a fixed location only allow a single application to be used at once. Instead, menus in MMM can be shared among users and editors and positioned anywhere on the screen, even in documents. For example, the menus in Figure 8-1 can be placed in a rectangle editor, regardless of its nesting level, and then applied to objects in any user's selected editor, regardless of that editor's nesting level. Allowing people and editors to share menus reduces the screen real estate needed for control. Also, menus can be created or positioned where a user is working, avoiding user interference. Finally, users can group menus into a document to use as a customised control panel. Some advantages of such "active documents" have been described by Bier [Bier91a].

8.2.9. Home Area Menus

Shared menus work well for commands, like changing colour or deleting, that apply to several editors. Some functions, however, are specific to a particular editor; for example, only text editors are concerned with fonts. Menus for these functions need only be displayed when such an editor is in use. MMM displays the menus for a user's selected editor in that user's home area. These menus can be combined with feedback that shows current modes in the selected editor. For example, Figure 8-7(b) displays a menu of possible font choices with the user's currently selected font highlighted.

8.2.10. Editing at Different Levels

Users can edit simultaneously at different levels in the editor hierarchy which reduces interference between users; one user can reposition or resize a document while another edits its contents. While some multi-level edits may disturb the user editing at a lower level, others work well. For example, one user may resize a rectangle editor to uncover an object beneath it without disturbing another user who is editing child rectangles that are away from the border being moved.

8.3. MMM's Architecture

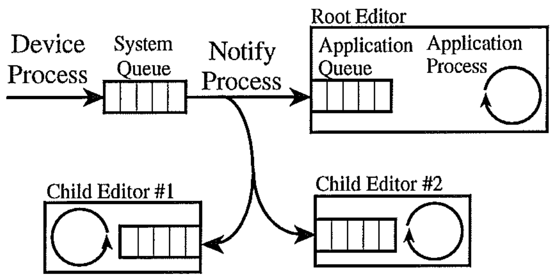
The user interface requires a software architecture that is unusual in several ways:

- it must support a hierarchy of concurrently active editors; it must handle multiple devices and users, and make links between them;
- each input event must be directed to the correct editor, even though other input events may be modifying the editor tree;
- each editor must store state for all of the users who have referenced it; and,
- the editors must co-ordinate their screen updates to produce a consistent image.

This section discusses the editor hierarchy, the preparation of new input events, the routing of events to editors, how editors update documents in response to events, and the algorithms for updating the screen.

8.3.11. The Hierarchy of Editors

Figure 8-8. Input handling processes and queues.



The editors are arranged in a tree; the editor at the root can have any number of child editors, each of which can have any number of children and so on (recall Figure 8-5). Each editor has an *application queue* of input events and its own lightweight *application process* (a thread) to dequeue and execute events (Figure 8-8). When an editor receives an input event, it may act on the event immediately, place it on its queue for sequenced execution, or pass it to a child editor.

At any moment, up to three editors are important to each user. A user's *selected editor* will receive his or her keyboard events and commands generated by clicking on a menu. If a user performs an interactive operation (dragging a rectangle, for example), the editor in which that operation began is the *mouse focus editor.* The mouse focus editor receives mouse motion and mouse button events until the interactive operation completes, so it is possible to drag objects out of view. The most deeply nested visible editor that contains a user's cursor is his or her *containing editor.* If no interactive operation is in progress, the containing editor receives all events which specify a position, such as mouse motion or button events.

8.3.12. Preparing Input Events

Figure 8-9. Constructing an MMM event record

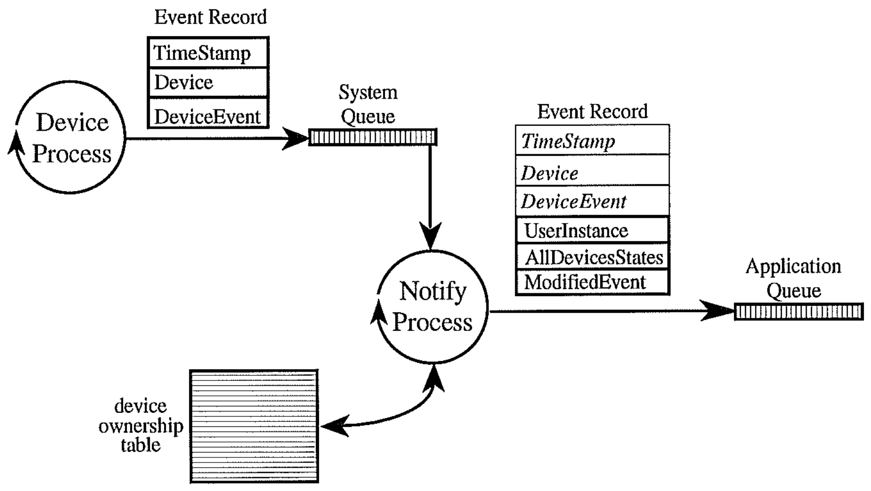
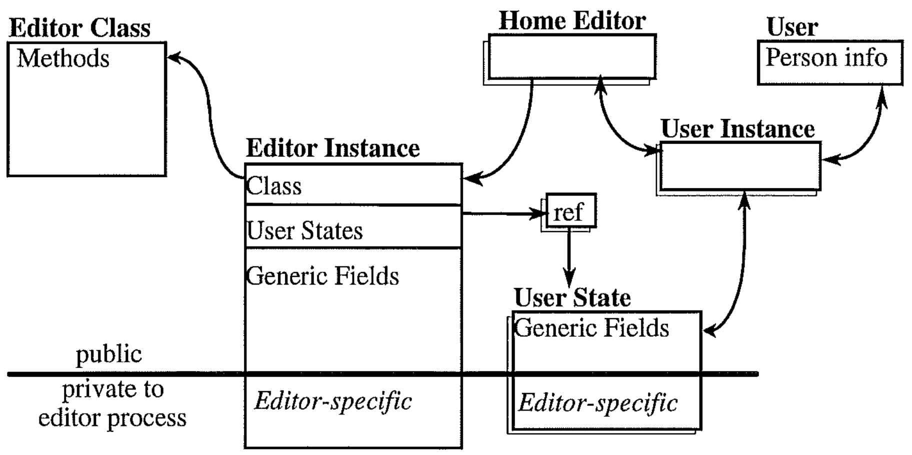


Figure 8-9 summarises the processes and queues used in input handling. When the user manipulates an input device, MMM's *device process* builds an *event record* to represent this event and fills in the time when the event occurred, the device, and the event type (mouse movement or key press, for example). The record is then placed on the system queue. The device process runs at a high priority so that the time-stamps will be close to real-time—to support the interpretation of gestures and double clicks. The *notify process* removes the record from the system queue and fills in the identity of the user (or user instance if a user has several home areas) who generated the event, by looking up the association in the *device ownership* table, and fills in the current state of all devices owned by that user (e.g., whether any mouse buttons are being held down). Only the notify process is allowed to modify the device ownership table, so the table lookup produces a consistent value. Storage is also allocated to hold local information as the event is passed between editors; for instance, x-y co-ordinates will be modified to reflect the local co-ordinate system of each editor.

8.3.13. Editor and user data structures

There are two basic types of object in MMM, users and editors. The behaviour of an editor is defined by its *editor class*, which contains a set of methods, and there may be multiple *editor instances* of an editor class. A *user* record is defined for each person connected to MMM, for which there may be one or more *user instances* (Uinst) records which hold a set of preferences and state for a given user record. Each editor in the tree is represented by an *editor instance* (Einst) record, of which part is editor-specific and accessible only to that editor's application process (internal) and the rest is accessible to the notify process (external). All Einst records include a list of *user-state* records, one for each Uinst which has referenced the editor during the current session. As with the Einst records, the user-state records are split into internal and external components; the external component includes fields for finding the users' current editors (described below). Each Uinst is linked to a Home Area editor that provides graphical access to the contents of the Uinst, such as whether the Uinst has control of a keyboard. The Home Area editor also holds a pointer to the Uinst's current selected editor. These relationships are shown in Figure 8-10.

Figure 8-10. MMM editor and user data structures.



The EditorClass methods are divided into three main categories: the editor acting as a child (messages sent down from root to a leaf); the editor acting as a parent (messages sent up from leaf to root); and communication with a Uinst's Home Area. The "down" methods include initialisation and requests to repaint, handle a resize, and delete the editor contents. The parent may also enquire of a child whether it is prepared to be deleted; this avoids deleting an editor while a descendent is being dragged by another user or deleting the last menu (which would leave MMM without controls). The `notify` method provides a generic mechanism for handling input (discussed in more detail below). Event types are identified by *atoms*, values unique within the process, so new event types can be defined and distributed without changing editors which do not handle them. Thus, a mouse button event and a request to change the colour of the selected object are both managed by the `notify` method. Editors ignore events which they do not recognise and cannot pass on to a child, and so do not fail if they receive a generic event (such as a delete) which does not apply. "Up" methods allow children to call their parents, for example to request a repaint or to get a system-wide value such as the device table. A generic `parentNotify` method allows children to perform actions such as grabbing the input focus. Finally, there are three Home area methods: to request a repaint, to perform a repaint, and to handle user input in the Home area such as the selection of a font button. The Home Area is also implemented as an editor, but its lower half is used to present a graphical view of some of the state of the user's selected editor, so the selected editor drives this region directly. This allows each editor type to implement its own Home Area window, and any controls it might contain, without changes to the Home Area editor type.

8.3.14. Delivering Events to Editors

When an event reaches the head of the system queue, the notify process walks down the tree of editors, beginning at the root, asking each editor either to accept it or to pass it to a child (see Figure 8-8). When an editor accepts the event, the notify process performs any actions that must be done before the next event can be handled (see the next section), dequeues the next event from the system queue and then repeats this procedure.

Each user-state record in an editor describes the child editor that is on the path to the user's selected editor and the child editor that is on the path to the user's mouse focus editor. MMM can find a user's selected editor, for example, by chaining down the path from the root editor. If a user has a mouse focus editor, events are passed to the child editor that lies on the path to that editor; on the way, the event's position information must be translated into that child's co-ordinate system. If the application process of the parent editor is busy, the child may be moving, so the notify process waits for this application process to finish its current event before translating the co-ordinates. Then, if the child editor is the mouse focus editor, this child editor accepts the event, otherwise the notify process continues down the path of editors.

The notify process is a potential bottleneck as the editors will all stop responding to the users if it does not process events fast enough. This is, however, necessary, as it ensures that events are delivered to editors in the proper sequence and maintains the consistency of the devices table and of the chains of pointers for users' selected and mouse focus editors. Timeouts were added to ensure that the notify process does not wait too long. If the notify process needs to wait for more than a few seconds for an application process, it gives up, throws the event away, and marks that process as "sick." Subsequent events that must pass through a sick editor are discarded until its application process is idle, whereupon it is once again marked as "healthy." While events are sometimes lost by this scheme, this seems an acceptable price to pay for guaranteed responsiveness; ideally, synchronous collaborations should proceed at a conversational pace.

If there is no mouse focus editor, the notify process passes mouse events to the containing editor. As the event is passed down the tree, the notify process waits for each editor's application process to be idle before translating event co-ordinates and passing the event to a child editor. Keyboard events do not contain co-ordinates so they are passed down the tree to the selected editor without any waiting.

8.3.15. Editor Input Handling

When an editor receives a user event, it looks in the editor-specific fields of its user-state record for that user to discover his or her current modes and preferences. Our rectangle editor, for example, stores the user's default colour for creating rectangles, the user's interactive command (dragging versus resizing a rectangle) and the last co-ordinates of the user's cursor that were processed during dragging. In addition, each rectangle records a list of all users who have selected it.

Editors respond to events in two phases. First, the editor may request that it become the mouse focus or selected editor for the user who originated the event. This is done by the notify process to ensure that the chains of pointers in the editor hierarchy that represent the path to the mouse focus and selected editors are updated atomically. The editor may then place the event on its application queue. In the second phase, if any, the application process chooses an action from its queue and executes it. To choose an action, it inspects all the queued actions. It performs the oldest mandatory action or, if there are no mandatory actions, it executes the most recent optional actions (e.g., motions during rubber-banding are optional), skipping older ones. This helps the editor handle large volumes of input. Editors place incoming events from all users on the same queue, so it is necessary to decide which user's event to respond to first. Mandatory actions are handled in the order in which they are received, regardless of user. For optional actions, the editor selects the user who has been least recently served, so the editor feels equally responsive to all.

8.3.16. Screen Update

Any application process can make changes that require the screen to be refreshed. Objects and editors can overlap so refreshing the screen for one editor may require the co-operation of both parent and sibling editors. If each application process were allowed to walk the entire editor tree as needed to update the screen after a change, one process might read editor data while another is modifying it. Instead, when an editor, E, wishes to paint, it creates a paint request record which includes (in the co-ordinates of E) a rectangle that bounds the region to be repainted. The editor notifies MMM that it would like to paint and then turns off its application process and waits. The system asks all application processes to pause and waits until they all have stopped, then the system paint process walks the tree of editors to determine a rectangle or rectangles bounding all the regions that have changed. It then walks the tree again asking each editor in turn to repaint itself within these rectangles. The resulting paint actions are double-buffered so that they appear to the user all at once, enhancing the perception of simultaneous motion. Once all editors have finished painting, their application processes are reactivated.

8.4. Conclusions

MMM proved valuable for testing ideas about the architecture and user interface of multi-user editors and for demonstrating that fine-grained editing with per-user customisation can be achieved with reasonable performance and only a modest increase in data structure complexity. As a simple testbed, MMM does not have a large community of users, but many of the ideas embodied in MMM have been of value in other projects, such as GroupKit [Roseman92] and Tivoli [Pedersen93]. We drew the following conclusions about the user interface of shared editors:

- the home area is effective as a quick way to register a pointing device with a user and as a place in which to display per-user feedback;
- selecting operations by pointing at specific object positions works well because it is easy to learn and hard to misinterpret as a gesture;
- when editors are used as window systems, users can work at different document levels simultaneously and can copy control objects, such as menus, into editors for convenient access;
- using colour to associate feedback with particular users allows the use of traditional shapes and sizes of feedback, such as underlines in the text editor and control point highlighting in the rectangle editor, although this technique is less effective on black and white displays or with colour-blind users; and,
- allowing menus to be shared among users and among editors simplifies the interface and conserves screen space.

To support this user interface, we provided four elements of system architecture:

- when an input event arrives, we identify which user it comes from using a table of user-device associations;
- events with co-ordinates, when destined for a nested editor, wait until that editor has a well-defined position relative to its parent before they are delivered to the nested editor;
- many editor data structures, including those representing the current modes, operations, insertion point, and selected objects, are replicated once for each active user; and,
- all editors stop manipulating their document data structures during screen refresh to produce a consistent view of the screen and a smooth view of simultaneous motions.

The main lesson from MMM for this dissertation is that the pointer/keyboard/display configuration of the standard personal computer is only a special case of a user interface. It is both feasible and useful for people to share input and output devices, rather than the “advanced glass teletype” approach which many user interface architectures implement. Furthermore, MMM devices are shared in the user’s world (on their desk, in their office) unlike shared virtual realities in which people use private input and output devices to share virtual objects in a digital world.

9. Multiple Input on a DigitalDesk

9.1. Introduction

The Double DigitalDesk[3] (DDD) was a project to investigate the extension for multiple users of an unusual user interface. The design and implementation of the DDD required the solution of a number of technical problems which highlighted issues of relevance to the design of user interface architectures:

- the DDD is an example of a user interface for which some of the component devices must be distributed. We implemented virtual devices to hide the location of physical devices from the application so that the same code could be used for local and networked devices;
- a DDD session is started by sharing the input from two individual DigitalDesks and we also wanted to support collaboration on the same desk, so the DDD has to allow input devices to be dynamically attached and detached. We achieved this by modifying the toolkit in which the DDD was implemented. The changes also allow input devices to be attached at any level in the window hierarchy; and,
- the sharing of DigitalDesks by distributing input means that connections between input devices and applications are unpredictable, so input devices are managed by servers rather than being tied to a virtual terminal. This means that any application can acquire input from a device, and devices can be shared between multiple clients.

The DDD provided a motivation for a user interface architecture which allows applications temporarily to assemble the devices they need for the task in hand, rather than having the devices bound into the display server. It shows that such an architecture can, in principle, be implemented over conventional computer systems even without the advantages of the improved technologies described in Chapter 4. There are, however, some issues unresolved in this implementation. For example, there is no mechanism to arbitrate between input sources; we had to throttle back the stream of pen events at the device, rather than being able to do so in the application. Furthermore, we did not have time to investigate how to handle race conditions in our highly replicated approach, except to note that they occurred rarely in practice.

This chapter first describes the original DigitalDesk and our motivations for extending it. The next section describes the shared desk application which provides digital copy and paste on paper for multiple users. The chapter then describes the distributed approach taken with the implementation

[3] The Double DigitalDesk was developed by Pierre Wellner and myself. The original DigitalDesk was developed by Pierre Wellner who implemented a single-user version of digital copy and paste. In particular he developed the thresholding and self-calibration, although I optimised part of it for this application. I worked on the distributed aspects of this project, including the decomposition into component parts, virtual devices, and handling multiple input sources.

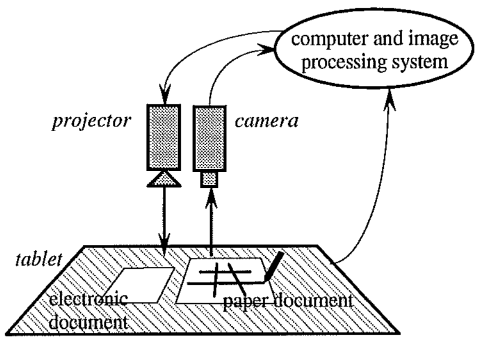
and how images and events are aligned on multiple desks. The next sections describe how multiple input is handled within a DigitalDesk and how input devices, particularly the frame grabber, are distributed. Finally, the chapter summarises our experiences and presents some unresolved issues.

9.2. The DigitalDesk

The world of the electronic desktop and the world of the physical desk top are normally very separate. The desktop metaphor has been very successful at helping users understand the electronic world by making it similar to a real desk. Studies of work in real world environments, however, have revealed that the use of paper documents persists after the introduction of computers [Luff92]. It appears that paper supports well both individual and shared activity, which suggests the potential value of a system that could merge paper and digital activity. An alternative explored by the DigitalDesk [Wellner93] is to keep the familiar physical desktop and enhance it with some of the features of electronic desktops.

The DigitalDesk (Figure 9-1) consists of a normal office desk with a computer-controlled projector and video camera above it. The computer display is projected onto the desk, superimposing electronic objects onto the paper objects lying on the desk. The user can interact with the system using the normal workstation mouse, a digitising tablet and stylus, or by pointing with a finger that is tracked through the camera with an image processing system. Text that is printed on paper can be recognised and used by the system.

Figure 9-1. A DigitalDesk.



Experience with the Digital Desk has shown the value of the concept, [Newman92] for example, but existing implementations were still bound by workstation limitations; in particular, they were limited to one location input device and one display. Where an unusual input device, such as following the user's finger, was used the locations were translated into normal workstation mouse events. When two people are at a real desktop they do not normally have to pass a token around to be allowed to move items on it—I can set down my coffee cup while you wield a pencil—yet this is how applications on the DigitalDesk were controlled (the token being a mouse or pen); our experience was that this felt unnatural and constrained.

Pierre Wellner and I wanted to relax these boundaries and experiment with multiple input devices; we also wanted to advance the technology to allow people to draw on the same "virtual" piece of paper at a distance. To this end, we combined two DigitalDesks into a Double DigitalDesk by multiplexing pointer and image input to both sites. There were other motivations for distributing parts of a single

DigitalDesk between hosts, as some components of the system, such as image processing, make heavy computational demands on the host. We expected that we could improve performance and user responsiveness by distributing the system between more than one machine. Furthermore, some input devices needed for a single client are attached to different hosts and so are inherently distributed.

9.3. The Double DigitalDesk

The DDD is a pair of DigitalDesks that allows users in two separate locations to share their physical desks, both seeing and editing each other's documents. The following sections describe the shared desk and the "copy and paste" application we implemented on it.

9.3.1. Shared Desk

People often use documents to help them work together and they often need to be able to view and modify a document together. Shared editing of documents has been the focus of a great number of research projects; see [Olson91] for example for a survey. Most work has concentrated on screen-based or virtual documents, but we wanted to support shared editing of paper documents. This has some similarity to VideoDraw [Tang90], but with the addition of digital image alignment and copy and pasting.

Figure 9-2. The Double DigitalDesk. The applications accept input from devices at both desks.

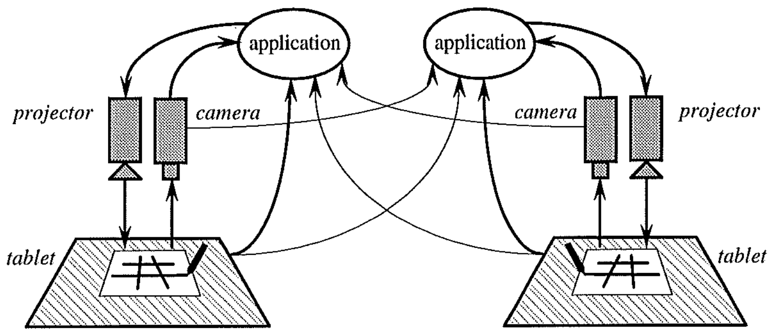
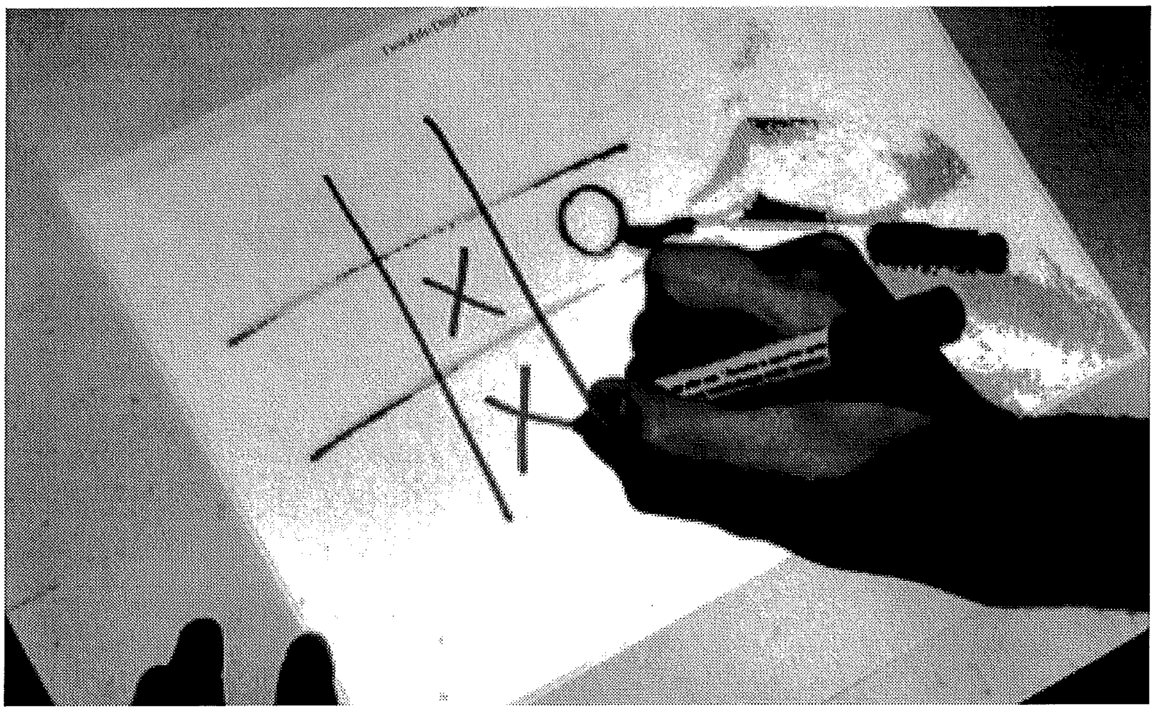


Figure 9-2 shows the basic DDD set-up. Each desk has a computer-controlled video camera and display that point down at the desks. Each computer grabs images from its local desk approximately every three seconds and projects images from the remote desk—*pseudo-video*. The result is that both users see what is on both desks; when a paper document is placed on desk A, its image is projected onto desk B and vice versa. The projections are precisely scaled and positioned to provide What You See Is What I See (WYSIWIS) and users can draw (using a real pen) on either paper documents or on virtual documents. In both cases, the remote user will see the new drawing projected in the corresponding position.

Hand motions are also transmitted, so if a user points to a certain place on a document the other user sees this. Hands block the view of what is underneath them, but this is also true when sharing an ordinary desk and can be dealt with though social protocols. Figure 9-3[4] shows two users drawing in a shared virtual space, giving a sense of how the images of the two drawing surfaces merge. The users are normally connected by telephone or with an audio-video link *via* EuroPARC's RAVE system [Gaver92a].

Figure 9-3. Two users playing Noughts and Crosses in a shared virtual space. The remote user's hand is visible in the projected image.

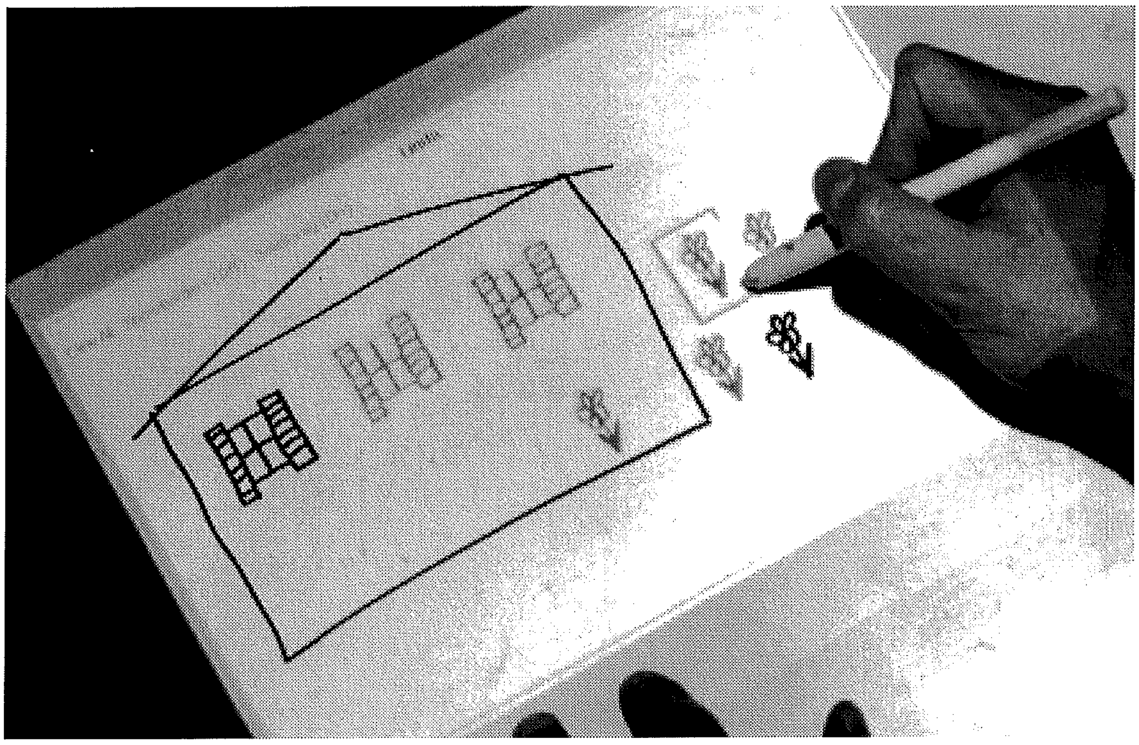


9.3.2. *Copy and paste with real paper*

Although copy and paste is now a standard feature of electronic documents, the same operation is awkward to perform with real paper. The DigitalDesk, however, makes it possible to copy and paste paper documents in the same way that we copy and paste electronic documents. The system, of course, cannot erase real paper, so we cannot implement *cut* and paste. In our prototype, a selection is created by sweeping out an area of the paper with a stylus. This selection is a copy of the area combining the local and remote images and it can be moved around or deleted. Selections are normally not distinguishable—they merge with the rest of the projected image—but a border appears around a selection when any user's pointer is over it, so that it may be found again. The selection can be moved by dragging it with the stylus; the border disappears during dragging to make alignment with other parts of the image easier to see. Figure 9-4 shows a single-user version of copy and paste; the user's selection, a flower, is copied digitally and can be "pasted" onto the front of the house.

[4]I am grateful to Pierre Wellner for generating these illustrations.

Figure 9-4. Single-user copy and paste.



When a user performs an action, such as moving a selection, it is mirrored on the other desk so that both users are effectively sharing the same surface. Selections can be created or moved by either user at any time. If both users select the same area, then two copies are made. If both users attempt to drag the same selection then the first one to grab it gains control, and one user may drag a selection while the other is creating a new one.

9.3.3. Tasks that can benefit from the DDD

One task the DDD may be well suited for is collaborative layout. Two people could, for example, work on designing the cover of a brochure: moving in and out of shared work copying images of standard parts from a catalogue, copying photographs to be included, and drawing (digitally or on paper) any non-standard items to be added. These can be projected onto a standard backing sheet placed on one partner's desk, combining the availability and familiarity of paper with the flexibility of the digital medium. Another task might be the early stages of architectural design, where a customer is discussing requirements for a building with an architect, laying out the facade or a floor plan. So far, we have only briefly tested it on a facade layout task and the game of tic-tac-toe.

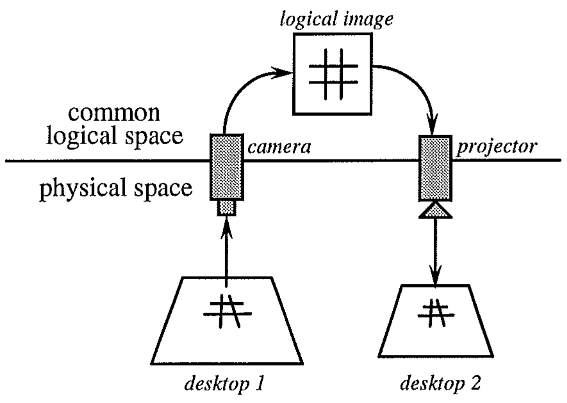
9.4. Implementation Approach

There is a long-standing debate amongst implementers of groupware over the merits of centralised *versus* replicated architectures, discussed in chapter 3. A centralised implementation, although it has many advantages, would be overloaded by the processing for the pseudo-video, so we chose a replicated architecture. We made the input and output components of the DDD—the tablets, the video cameras, and the displays—available as services to client applications. Each tablet, for example, is managed by a tablet server which is a process that accepts events from the physical device, translates them into a common format, and sends them to all interested clients. Each copy of the DDD application connects to the services it needs. Thus, each half of a DDD can work as a stand-alone system, providing local digital copy and paste by connecting to the local tablet, display and camera. The independent sessions become collaborative when each side connects to the other's devices in addition to its own. Two central issues for implementing the DDD are sharing the desk views, and sharing the desk input.

9.4.1. Sharing Desk Views

To give the illusion of a shared desk, some regions of the desk must provide WYSIWIS. One approach would be to use identical DDDs and for both to display exactly the same thing, but this approach is impractical and restrictive. It is very difficult to adjust cameras to point at exactly the same spot, with the same zoom factor on each desk, and to maintain this calibration over time, and the projectors we have are of different resolutions. Users will want to maintain full control over the non-shared portion of their desks (i.e. they should be able to move windows around) and will expect control over the view that the local overhead camera is sending out. A fully digital video capture and projection system allows arbitrary scaling and positioning of the video images.

Figure 9-5. Image alignment between desks.



To align the DDD views, it is necessary to select a global co-ordinate space that all DDDs can share. The systems communicate with reference to the global space and each DDD transforms its view of other DDDs according to its own mapping. Projected pixels (i.e. the image on the desk surface) cannot be used as the global co-ordinate space because different DDDs have different resolutions and sizes of projected image, so we use the camera's field of view. The applications automatically project the remote camera's image scaled and positioned on the desk such that it corresponds exactly to the field of view of the local camera, relying on self-calibration to determine the size and position of the camera's field of view (Figure 9-5).

Aligning the cameras and projected images in this way generates video feedback: a mark drawn under one camera is shown on the remote display which, in turn, is seen by the remote camera and shown on the local display. This is undesirable if it makes images too stable: the image of a user's hand may loop through the system after the user has moved it out of the way. Fortunately, it is easy to adjust the ambient light and projection contrast so that the persistence of moving images is reduced to only one or two cycles. The feedback may also be useful to test the alignment of the system. When properly aligned, projected feedback matches perfectly with the original mark.

Instead of projecting a grey-scale or colour image of the remote desk, the DDD projects thresholded images. A greyscale or dithered image would interfere with the appearance of paper documents on the local desk, and this effect would get worse when combining the images of more than one remote desk. One bit per pixel thresholded images are not only faster to transmit across the network, but they look good when superimposed on paper documents and are easy to combine together. Generating these thresholded images is not, however, trivial because lighting varies across the camera's field of view and over time. An adaptive thresholding algorithm is used that takes into account the local background illumination at each point of the image; the current version takes approximately half a second per frame, enough to demonstrate the principle. Self-calibration and adaptive thresholding are both described in [Wellner93].

9.4.2. Sharing Desk Input

Just as it is necessary to align the video images, it is also necessary to align the digitising tablets so input events for one DDD are mirrored on the other so those events must be aligned with both displays. Our solution is to express device events in the global co-ordinate space described above. We have three co-ordinate spaces mapped onto each other: at the bottom we have the actual device (the tablet). Then there is a mapping from the device co-ordinates to the global co-ordinates used for sending events to client DDDs. Finally, each DDD is calibrated to the global co-ordinate space and so can translate global input events to its local (screen pixel) co-ordinate space. This means that when a user points with a stylus on one DDD, the event is duplicated in the corresponding position on any other DDD which is attached to the same device.

We calibrate a device through its local display. Each local camera represents the global co-ordinate space, so once a camera and projector are aligned to each other we can unmap from projector co-ordinates to global ones. To calibrate the tablet, a series of cross-hairs are projected onto the desk which the user must select with the stylus. This gives a set of point pairs which maps between the tablet and global co-ordinate spaces.

The link between a DDD and a device server is called a *connection*; each connection has a name and a mapping associated with it that relates the device co-ordinate space to the global co-ordinate space. DDDs that share events also share mappings which ensures that when a user points at a specific place on one desk, the corresponding place is pointed to on the other desk. This is discussed further in section 9.6.

9.4.3. Input Distribution

A shared application must distribute its activity between its instances. We chose to distribute basic device events to allow us to experiment with a highly distributed approach and different input device types. This is in contrast with many groupware systems, such as Colab [Stefik87], which distinguish between *syntactic* (i.e. raw input) and *semantic* (i.e. changes to the shared state) events—although it could be argued that the mapping of event co-ordinates between physical and logical spaces represents part of such a conversion. At present, copies of the DDD involved in a conference are co-ordinated only through their input devices, they do not communicate directly with each other. This means that DDDs can be hooked together without the need for an explicit conference component, including (in principle, as we do not have the hardware to do so) many DDDs.

There are two levels involved in distributing input events in this way: managing multiple input streams within a DDD and routing events from devices to DDDs. The next sections discuss how we implemented these two levels.

9.5. Multiple Input Within A DDD.

We wanted to take advantage of existing software and provide a path for future reuse of the DDD implementation. We were already working in *Modula-3* [Nelson91] which provides language support for objects, modules, strong type-checking, lightweight threads and garbage collection, so we chose the Modula-3 user interface toolkit *Trestle* [Manasse91]. This section describes the changes we made to Trestle to support our new requirements.

9.5.1. Trestle

Trestle has a strong model of user interaction which influenced the implementation of the DDD. The basic screen component is an abstract object type, the Virtual Bitmap Terminal (VBT), which represents a share of the workstation's screen, keyboard and mouse. New interaction object types can be defined by subclassing existing VBT types. Like many user interface systems, Trestle maintains a hierarchy of windows in which children of the same parent may overlap; each window is represented by a VBT. A VBT includes a set of methods to handle user input events which a programmer may override to define the behaviour of the VBT. VBTs which are parents of other VBTs, `Splits` and `Filters`, generally pass these events to a child until the event reaches a `Leaf` (the lowest level of child) which may take some action. The top-level VBT in a Trestle implementation provides the connection to the graphics hardware or the host window system; the current version, XClient, interprets between Trestle and the X Window System. One of the original goals of Trestle was to investigate the benefits of multiprocessor hardware to a window system—each VBT may be locked separately—so it also has a sophisticated locking strategy to allow individual VBTs to be locked without risking deadlock.

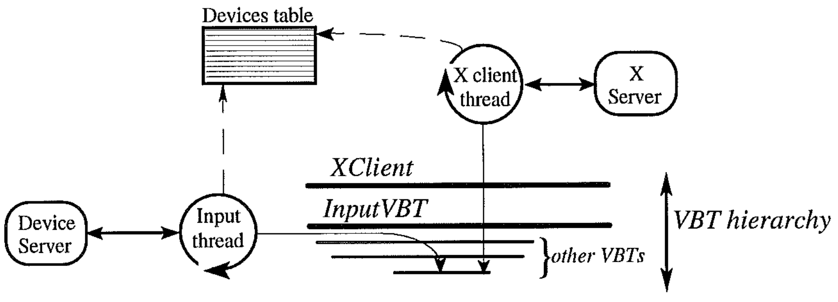
Each VBT also has a `cage` field which represents a set of cursor positions. Cursor motion is only reported to a VBT when the cursor leaves the cage. A VBT can, for example, be notified of the cursor entering its domain by setting its cage to be everywhere outside the VBT—the cursor leaves the cage when it enters the VBT. On the other hand a VBT can track a cursor by resetting the cage to the cursor's current position each time a motion event is reported. When the cage is set for a VBT, it is propagated up the hierarchy so that the cage for each parent VBT is the intersection of its children's cages. This approach avoids unnecessary processing of motion events and reduces the range of events a VBT must handle but, as is discussed later, makes the management of multiple pointers more difficult to handle.

9.5.2. The InputVBT

We wanted to be able to dynamically add and remove input devices to a Trestle application without changing existing code. One option is to synthesise events in the X server, as with earlier versions of the Digital Desk, but this makes it impossible to distinguish between sources of events without changes to the X server and parts of Trestle; chapter 7 discusses the difficulties in extending the X server. Instead, we adopted the solution of inserting external events directly into the application and using Trestle's event-handling mechanism to process them.

The *InputVBT* is a `Filter` which forks a thread to accept input from an external source and pass events to its child as if from the main input device. This allows most VBTs, which cannot distinguish between multiple input sources, to process events as if they had come from the default input devices, thus avoiding alterations to the toolkit. Some VBTs, however, need to distinguish between input sources to allow multiple paths of control, so we provide a `Devices` interface which allows VBTs to determine the origin of the current event. The implementation of `Devices` is based on the observation that input from each device is handled by a different thread and that the general Trestle lock is held during the processing of an individual event so different input threads cannot interrupt each other. Device-handling threads register their association with the device before starting to process events so VBT methods can then use the current thread as a key to look up the input device. A more thorough conversion would implement separate cage hierarchies for each device so that position events from one device would not be reported to a VBT because another device needed finer tracking. Some effort went towards implementing such a scheme, but it was found to be needlessly complex for our purposes, especially as pointer events were not the main computational bottleneck.

Figure 9-6. Adding devices with the InputVBT.



To add an input device, an application inserts an InputVBT at the top of its VBT tree and attaches a `Device` object to it. This can be seen in Figure 9-6: the XClient at the top provides the main window, and input events from the X server are passed through to the relevant descendant VBT. Events from an extra device are distributed by a thread attached to the InputVBT, which is a child of the XClient. Both the XClient and InputVBT threads are registered with the `Devices` table. Any number of devices may be added by inserting extra InputVBTs and, similarly, devices may be disconnected by removing the associated InputVBT.

One tricky point occurs when an application sets a new cage and propagates it up the VBT hierarchy. If the new cage no longer includes the current point, the cursor has effectively left the cage. This requires a new position event to inform the VBT tree but the VBT hierarchy will be locked until the cage-setting procedure returns, so the event cannot be generated from within the call. The solution is to call a procedure, `ForceEscape`, to fork a thread to wait until the VBT hierarchy has been released and then generate the dummy event. This thread, however, will not be registered with the `Devices` interface and so dummy events will always be associated with the default devices. The implementation of `ForceEscape`, however, is embedded in the core Trestle library so we avoided changing it by provided extra cage handling in the DDD code. In particular, cage requests are filtered by each InputVBT, so when the cage is set for its associated device, an InputVBT calls a version of `ForceEscape` that registers the device and new thread. The top-level cage, however, must still be set to the new area, so the method which sets the cage for the XClient was modified to only call `ForceEscape` for events from the default pointer.

9.5.3. Arbitration between devices

The InputVBT approach does not, however, provide a mechanism for balancing input from several input sources—the next event comes from whichever thread acquires the lock. In MMM, each editor has an input queue so it can pre-process outstanding input events, determining which event to accept next. In Trestle, the thread which delivers events to VBTs also performs the processing for the event in the leaf VBT, so each event is handled in isolation. Even were there a separate VBT cage per device, this would only allow crude control over which device to select input from next (by widening the cage during pointer following). Nor would it be easy to filter input by placing all events into a single top-level queue, as the meaning of an event (and, hence, whether it can be dropped) depends on the interpretation placed on it by the destination VBT. There would be other problems with providing each VBT with an individual input queue. In particular, the cage must be reset at the end of the `position` method to receive any further position events, so if the `position` method does not interpret the event but only attaches it to a queue, the choice of cage must be arbitrary. In practice, we found that our tablet, when running at full speed, swamps the application, so we throttled back the rate of event generation in the tablet hardware. This is clearly a pragmatic, rather than a general, solution.

9.6. Distributed Input Devices.

As described in section 9.4, the shared application is based on multiple DDDs having access to common input devices, even when on separate hosts. Furthermore, applications may need to share a *view* of a device rather than access to the base device. In addition, we wanted to be able to use the same infrastructure for both local and remote devices, for testing and other applications. We followed an approach, similar to Network Objects [Birrell93], in which devices are represented by objects which may either drive a local device or be proxies for device objects in a remote server. Modula-3's class system allows us to define a virtual device object type which can then be subclassed for the local and remote implementations. While this technique has been used to provide location-transparent access to more complex devices, such as video recorders in the Component Kit [deMey93], it has not so far been applied to basic user input devices such as mice and pens.

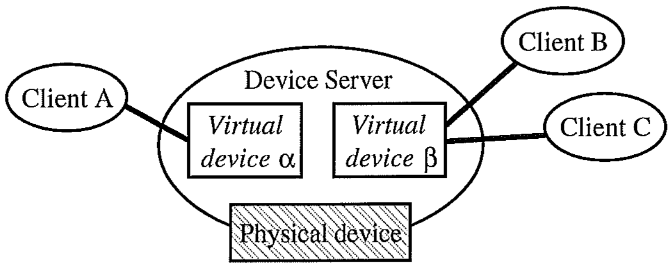
This approach proved particularly useful during the development of the image processing software. The initial implementation of the DDD performed both image processing and display in the same process. This simplified the implementation, as there was no need to write another server or define a protocol for inter-process communication, but there were two main problems. First, the cameras had analogue connections to the frame-grabbers which became impractical after our DigitalDesks were separated for organisational reasons, whereas the computer network covered the whole building. Second, the computational effort devoted to image processing degraded the responsiveness of the user interface, so distributing this part of the system spread the load over more processors and made it easier to isolate bottlenecks.

9.6.1. Virtual Devices

We wanted to hide the fact that a device is shared from its clients, so that applications could be written without having to provide explicit support for sharing. We based our design on a notion of *virtual devices*, analogous to virtual memory, in which a set of logical devices are supported by a common physical device, giving each client the illusion of having its own hardware. Some characteristics of a virtual device are fixed when it is created, such as the image type of a `Grabber.T` (our frame grabber object); if a client needs a device with different characteristics, it should create another virtual device. For example, a client would open a `Grabber` of type `Grabber.ImageType.FastThresholded` to get images quickly for our pseudo-video, but would open a second `Grabber` of type `Grabber.ImageType.GreyScale` to get an image for detailed processing. This avoids synchronisation problems if multiple threads within a client need to use different settings with the same device. Each thread allocates a separate virtual device with different characteristics and the shared implementation of the device determines when to change the physical device settings. There are similar complications when virtual devices are shared between clients (as described below).

Other device characteristics, however, may be changed during the lifetime of a device, as they are more related to the client's *view* of the physical device rather than its hardware attributes. One example is the *warping* (mapping from the physical co-ordinate space on the desk to the DDD's logical co-ordinate space), which may change if the desk or projector move after a calibration. The warpings are held in the virtual grabber, as the frame grabber was designated as the logical co-ordinate space, and other parts of the application query the `Grabber.T` to warp or unwarp pairs of co-ordinates between logical and physical spaces. Of course, the distinction between device-specific and view-specific attributes may not always be clear.

Figure 9-7. Multiple virtual devices and clients in a device server.



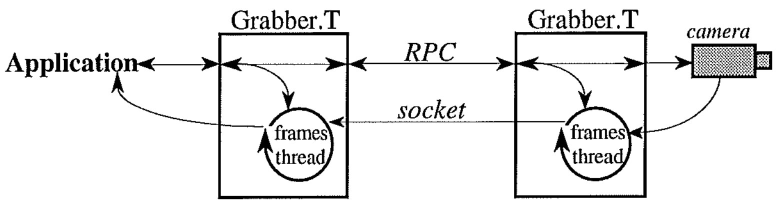
As with shared virtual memory, clients can share a virtual device if they can agree on a common name. When a client opens a virtual device it specifies a name for the device. If a device is networked, the server will look to see if it already has a virtual device with that name and either establishes a second connection to it if such a device exists, or creates a new one. The device servers support an arbitrary number of virtual devices and an arbitrary number of clients of each device. For example, in Figure 9-7, A is the only client of device α, but B and C share the use of device β. This is similar to the Switchboard approach to devices [Tanner86]; a device view can be thought of as logical device implemented over another logical device, except that DDD devices are distributed between hosts and we can use the Modula-3 class system to support a hierarchy of device types.

9.6.2. Network Devices

Given our emphasis on reducing the distinction between local and remote devices, we decided to use Remote Procedure Calls (RPC) [Birrell84] to structure the communication between components, so we took advantage of a Modula-3 implementation from Xerox PARC of SunRPC. This also made interoperability between languages easier as the same interface file could be used to generate C and Modula-3. SunRPC, however, identifies a contact point by a host name and program number, which implies that there is only one logical instance of a program on a given host. SunRPC, unlike ANSA [ANSA89], does not support callbacks from server to client, to notify the client of device events, so some of our device clients exported an RPC interface for the server to call. While it is unlikely (although not inconceivable) that two instances of a device server may exist on a host, it is more likely that there be multiple instances of a client on a given host, which this naming scheme cannot support. Of course, the interface can define a set of query procedures which clients can fork threads to call, each blocking until the server responds, but the inelegance of this solution suggests that we should use a more suitable interface (such as ANSA's).

RPC also proved inappropriate for our high-bandwidth communication such as transmission of our pseudo-video images. The first implementation of our `VideoVBT.T` (a VBT which repeatedly gets images from a `Grabber.T` and displays them in the background) simply looped calling the `Grabber.T.grabRect` method (a synchronous call to grab an area of the frame). While this allowed us to use exactly the same code in local and networked implementations for most of the layers of the application—the networked version is simply an RPC call to the server—it was inefficient in two ways: first, extra copies of the data are made during marshalling and unmarshalling; and, second, RPC is synchronous, so some resources are held locked in the client during the entire processing and transmission of an image. In addition, the RPC model does not map well on to the conceptual model of the application, which is of a device broadcasting data to a set of clients. While group RPC could be used, to take advantage of network-level multicasting for example, this would still be a call from server to client and so subject to the naming limitations described above.

Figure 9-8. Multiple connections in a networked Grabber.T



The solution was to provide a second connection, using sockets, to transmit image data controlled through the `Grabber.T`; a separate thread processes the images as they arrive (Figure 9-8). This is similar to other architectures that support isochronous data, such as IMAC [Nicolaou91] or the TouringMachine [Bellcore93], which provide separate paths for the data stream and the object which controls it. We added a set of methods to the `Grabber.T` to control the extra thread, which repeatedly acquires an image and calls a client-specified procedure, `FrameProc`, to process it. This still allows similar code to be used for different parts of the implementation, but partitions the code which acquires an image from that which uses it. The local version of the image thread gets the image directly from the frame grabber, whereas the networked version waits for the server to send an image through the socket. The `FrameProc` in the server transmits the image to all the clients of that `Grabber.T`, while the `FrameProc` in the client displays the image in the application.

In practice, these techniques, with a few other optimisations, mean that the communication part of our network devices is not the bottleneck in our application—at least in our environment (Sun SparcStations, SunOS, Ethernet). The slowest part of the system is the grabbing, thresholding and warping of the images which require specialised, or very much faster, hardware to approach real-time performance. Other applications implemented over the same infrastructure but without the image processing demands, such as a shared sketching program, were acceptably responsive.

9.7. Conclusions

This chapter describes the Double DigitalDesk and its implementation. The DDD shows how the single-user DigitalDesk can be extended to provide support for multiple users, both at a distance and on the same desk. We envisaged our application forming part of the working environment of a DigitalDesk user, allowing lightweight collaborations at a distance between people using both paper and digital media. The architecture of the DDD demonstrates how facilities in the user interface layer can support the dynamic construction of applications by gathering together distributed input and output devices, providing user collaboration through shared devices.

The implementation of the DDD shows how a user interface architecture, in this case Trestle, had to be extended to support a novel user interface. These extensions were:

- networked virtual devices, which allow applications to be written without regard to the location of the source of an input stream;
- the ability to dynamically attach and detach input devices to a display at arbitrary levels in the window hierarchy; and,
- implementing all input devices as networked services so that clients can acquire an input stream from any device.

The DDD, like MMM, shows that there are models other than the desktop workstation which user interface architectures will have to support and that facilities provided at the level of the user interface system can simplify the construction of distributed multi-user applications.

There are, of course, unresolved issues in the current implementation of the DDD. In particular, we did not develop a general mechanism for arbitrating between input streams to avoid one device swamping the others. It is clear from MMM that the individual VBT must determine which events may be discarded, as this is the level at which an event becomes meaningful to the application, but Trestle handles each event as it arrives rather than queuing it with the VBT. It is also not clear how complex an application this extreme form of distribution can support. Current performance slows the users down sufficiently to avoid many race conditions, but this may get worse as the implementation improves. On the other hand, work such as [Karsenty93] shows that more complex protocols can support parallel activity in replicated applications, and an implementation based on *virtual synchrony* [Birman91] would be interesting to explore.

10. Summary

This part of the dissertation has described three sets of experiments and their implications for the design of user interface architectures. The first chapter describes two experiences with the X Window System: a multi-user application and an attempt to introduce dynamic input device loading to the X server. These made apparent how strongly the assumptions about how users will interact with a display are built into its design. Clients have no influence over the scheduling of an activity in the server, making synchronisation with other media or other servers difficult. Similarly, the optimisation of the internal structure of the server for the model defined in the X protocol makes it difficult to extend for new user interface requirements.

Chapter 8 described the Multi-Device Multi-User Multi-Editor (MMM), a user interface and architecture for multiple users and multiple devices sharing the same screen. It supports fine-grained editing and per-user customisation within an editor. This requires four extra features not normally present in single-user architectures: associations between devices and users are maintained in a system-wide table; events with co-ordinates wait until the destination editor has a well-defined position before being delivered; many data structures, such as modes, selections and insertion points, are replicated for each active editor; and, all editor activity is suspended during a screen refresh to produce a consistent view and smooth motion. MMM shows how input and output devices may be shared in the user's world rather than in a virtual reality and that the implementation costs of providing such flexibility need not be excessive.

Chapter 9 described the addition of multiple input to the DigitalDesk. The Double DigitalDesk (DDD) provides a common virtual drawing space between two DigitalDesks by multicasting its input to both sites, allowing lightweight collaboration using both paper and digital media. If we regard Trestle as a user interface layer, the DDD provides three additional features: networked virtual devices which hide from the application the location of the physical devices they represent; dynamic incorporation of extra input devices into the general event handling mechanisms; and, network devices which allow clients to access and share remote input devices. As with MMM, the DDD shows that user interface systems need to support device models other than the desktop workstation and that such support can simplify the implementation of distributed multi-user applications.

These experiments suggest that user interface architectures should provide support for dynamic associations with multiple users and devices and that such an architecture cannot be easily evolved from existing standards. The next part of this dissertation proposes a model of a user interface architecture which meets some of these demands.

Part IV. The *Gemma* Architecture

So far, this dissertation has shown that current user interface systems do not support new user requirements or take full advantage of innovations in computer systems. The experiments have shown that existing user interface systems are difficult to extend beyond the model of the desktop workstation, but that there is evidence that multiple and distributed devices can be used to provide a more flexible approach. This part proposes a distributed user interface architecture, the Graphical Environment for Multiple users and Multiple devices Architecture (*Gemma*), based on networked devices which provide input and output services to clients. These clients can then assemble low-level devices to provide higher-level services, such as window servers, to application clients.

Chapter 11 discusses the computing environment for which *Gemma* is intended and some of the user interfaces to be supported, outlines the required features of the architecture, and then relates its main features to the work described in Part III. Chapter 12 describes how low-level devices are defined and made available to higher-level clients, such as applications and window servers. Chapter 13 describes an example of a simple window system, sw, built over *Gemma* which provides some of the functionality identified in Chapter 11. Chapter 14 discusses the architecture, describing how *Gemma* may be used to implement some example applications, some unresolved issues in its design and the infrastructure required to support it.

Gemma provides a model for a new generation of interactive systems which are not confined to virtual terminals but use collections of independent devices which may be bound together for the task at hand. It provides mediated shared access to basic devices and higher-level virtual devices. An example window system, sw, shows how these features may be exploited to provide a flexible, collaborative and mobile window system.

11. Introduction

11.1. User Requirements

A first step towards a design is to clarify the activities to be supported. Consider a group of people in a room, working on related tasks, who occasionally want to share devices and applications, without performing complex dialogues to establish a connection, and who want to be able to identify who performed each action; they also want to use multiple devices for simultaneous multi-modal input. The following scenarios provide several interesting user-level requirements:

i) an accountant is having difficulty implementing a calculation on a spreadsheet so he calls over a technical support person. She brings an extra keyboard and the two of them work together for a time at the accountant's desktop. The company has an internal charging system for technical help, so the technical person's presence is logged in her diary as is the formula she develops. Sometimes the client is too far away, in which case the discussion is held over the network using desktop video.

ii) a user has a problem, so she asks a colleague for help. He comes to her desk and they discuss the problem, both making annotations on her desktop display. As they need to look at more information, he brings over a couple of notebook displays which they add to her desktop, but there is still not enough space so they transfer her desktop image to the wall display.

iii) a group holds a design meeting around a wall display. Users write and draw on the display with their pens and the meeting software keeps track of each person's contribution. One person has forgotten his pen and must share his neighbour's, so there must be easy mechanisms for changing the association between person and device. In addition, some people have their own notepads on which they make notes that they sometimes transfer to the display.

iv) in a design class, each student has his or her own drafting display. The teacher wanders around the room correcting and annotating the students' designs with her own pen. Her marks are distinguishable from the students' and can be shown or hidden independently. Sometimes she refers to a document in her own file space; the system can pop it up by identifying her as the origin of the input gesture. Finally, she signs off each student's work on the display in question; the sign-off is automatically copied to a register in her own persistent storage.

These scenarios assume a hardware environment of a room full of output devices: a wall-mounted display, desktop displays, many portable displays ranging in size from a notebook to a badge, and several loudspeakers; there is also a range of input devices: keyboards, pens, mice, position sensors and so on. Many of these devices are mobile.

Aspects of these scenarios can be extrapolated from studies and systems such as:

- Colab [Stefik87] and Tivoli [Pedersen93] which show how large-scale displays can support people meeting in the same room;

- Grove [Ellis91], MMM [Bier91], and CaveDraw [Lu91] which show fine-grained collaboration within an editor;
- Nardi and Miller [Nardi90] who show how people collaborate with "single-user" tools;
- Pettersson [Pettersson89] and Wolf and Rhyne [Wolf92] who show the need to distinguish individual contributions to a shared document;
- Mermaid [Ohmori92] which demonstrates multi-medium collaboration across a wide area;
- Rooms [Card87] which highlights the insatiable need for more display space and demonstrates a portable user environment;
- Tivoli, WeMet [Wolf92] and Commune which show pen-based collaboration; and,
- TeamWorkstation [Ishii92] which shows the integration of different media.

The user-level requirements can be reduced to the following (the numbers in parentheses refer to the relevant scenarios): mobile input devices, ownership of input devices, and shared displays and applications (i, ii, iii, iv); dynamic grouping of displays (ii); mobile displays (ii, iii); and continuous media (i).

11.2. Architectural requirements

The user requirements place several demands on the user interface architecture of which the first is to allow dynamic association of input and output devices. Current user interface architectures view the set of user interface devices as static, but this is too limited as devices become mobile and "group interfaces" become more common. Users could, instead, define relations between sets of devices to combine them together. For example, two people editing together may each define a compound device consisting of their own input devices and the shared screen; the input focus for the keyboard of each compound device is determined by its related mouse. This approach also requires that the user interface software be able to make connections to new devices and handle the loss of existing connections.

Some devices may be shared between users, such as wall-sized displays or touch screens divided into virtual windows [Brown90], or shared between applications—for example, a mouse driving telepointers on several displays or two window systems sharing a screen. Thus, devices must be able to support multiple clients and provide access to special attributes, such as portability or the display of video, without subverting the architecture. Similarly, the user interface software must be able to manage input from multiple sources, providing built-in support for the association of devices with roles—either different users or different functions.

Users must be able to dynamically associate themselves with compound devices. At its simplest, this is equivalent to current login procedures where a user acquires control over a fixed combination of devices (a terminal or workstation) by typing an identifier and password. A more general approach, however, allows users to establish a connection to a group of devices, such as all those within a given room, where the group may change during the connection period. Flexible grouping of devices allows, for example, a user to move between displays while keeping the same set of input devices; this at least would help those with several machines on a desk who have nowhere to put all the keyboards. It also makes easier the impromptu lightweight collaborations which studies such as [Stults89] show to be a feature of teamwork. Thirdly, it allows users to join in each other's interactions without interrupting a session, and facilitates transitions between modes of input—such as from one-handed to two-handed. Finally, it must be possible to locate and connect to all these disparate components.

The architectural requirements can be reduced to: a general mechanism for grouping input and output devices into compound devices; provision for sharing compound and basic devices and for controlling access to particular attributes of a device; and, the ability to identify devices and the sources of events.

11.3. A Distributed User Interface

The following chapters present a design for, *Gemma*, a highly distributed user interface architecture which addresses some of the issues described above. The design assumes the support of some of the facilities discussed in Chapter 4, in particular, support for distributed processing: a lightweight operating system with efficient scheduling, and fast, low-latency connections between hosts. It is also intended to support mobile displays connected by a wireless network which can determine their position in a room.

I use Modula3-like interfaces to specify the components as this is a compact notation which matches the object/interface model on which the architecture is based; these code fragments are intended to show the concepts of the architecture rather than samples of working software. This part presents designs for two abstractions: the *Gemma* model which provides access to networked and mobile low-level input and output devices, and a window system, sw, which dynamically ties low-level devices together.

A first step, however, is to describe how the *Gemma* approach was motivated by the experimental work presented in Part III: the experiences with X, the MMM and the DDD. The attempts to enhance the X server showed that current window systems provide an inappropriate model for a user interface environment not based on workstations; the existing implementations have been optimised over many years to support a particular form of user interaction. Extensions, such as the XIEE, allow developers to enhance the protocol but the core model remains, and some extra facilities, such as multiple cursors, can only be added with ingenuity. *Gemma* is founded on the belief that, to construct user interface environments for multiple mobile devices, the component parts of a distributed user interface system must be made accessible rather than encapsulated in a virtual terminal.

MMM shows that support for multiple devices and multiple users on a display is both useful and possible. The shared user interface encouraged the emphasis on pen-based input and suggested some of the scenarios in Section 11.1, such as the meeting room and design class. sw inherits several features of MMM's architecture:

- like an MMM editor, a window in sw may be embedded in a parent and provides some behaviour for that part of its parent's area. The process associated with a window may handle and distinguish between input from multiple users and devices;
- many window data structures, such as modes, selections, and input focus, are replicated for each active user or input device. sw makes a greater distinction between user and device attributes than MMM; input focus, for example, is associated with a user instance in MMM but a virtual input device in sw; and,
- when an input event arrives, the user it belongs to is identified in a table of user-device associations, sw includes a device registry which allows these associations to be maintained across displays and sessions.

MMM demonstrates how users can share their input and output devices in the physical world, rather than in a virtual world. In addition, sw, unlike MMM, makes provision for its components to be distributed; devices may be remote and the same infrastructure may be used to provide protocols for distributing applications. sw also allows windows to enforce per-user access control to the data they display.

The DDD shows that an interactive environment can be assembled from a set of distributed input and output devices, allowing users to interact with and share computing devices in the real world. The aspects of DDD which have been retained in *Gemma* and sw are:

- basic devices are provided as services, available remotely, rather than bound into a display server. Device services may be multicast to multiple clients. Both DDD and *Gemma* provide a common interface so that clients of a device can ignore implementation issues, such as location, in which they are not interested;
- a client may make its customisation of an input stream, such as the calibration of a pointer, available to other clients. In the DDD, this is retained in the server and other clients ask for a calibrated stream by name, whereas in *Gemma* the calibration is exported as a new virtual device;
- input devices may be dynamically attached to arbitrary levels in the window hierarchy. This has been expanded in sw to make the window the principal point of co-ordination between input and output devices. Unlike Trestle there is not necessarily a separate top-level entity to represent a particular hardware terminal, but such a device may be implemented by attaching input devices to a display window; and,
- a client application assembles those devices it needs for the duration of an interaction. An instance of the DDD can be made multi-handed by incorporating an extra pointer, and a collaborative application can be assembled by two DDD's importing each other's input devices. *Gemma* provides a model for acquiring access to remote devices and sw for tying those devices together.

The DDD is as an antecedent to *Gemma*, it showed how the components of such an architecture would fit together and what infrastructure support is needed. Within DDD, Trestle provided a useful model for the management of embedded windows, particularly its use of filter VBTs to provide particular functions, but it also shows that applications need access to their input queues to arbitrate between input sources.

The next two chapters present low-level *Gemma* and the `sw` window system. After that I describe how this infrastructure may be used to implement MMM, DDD and an example application. I then discuss two examples of issues which have not been resolved within the current design: naming and persistence.

12. Low-level Services

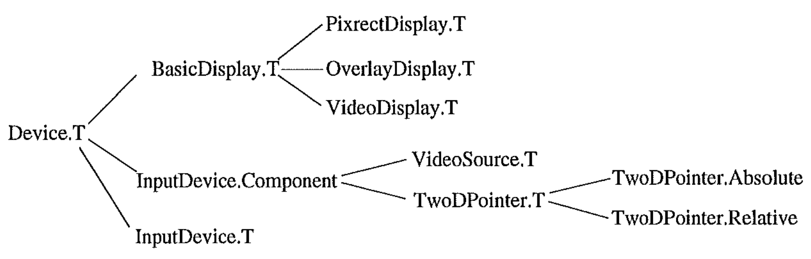
12.1. Introduction

Any interactive system must eventually communicate with its physical devices which, in *Gemma*, may be distributed. The task is to find a means of representing these devices to programmers which is both practical and flexible, especially when devices may be shared or mobile. *Gemma* uses *network objects* [Birrell93] as handles on its devices; a network object has a core object in one address space and is represented in other address spaces by proxies which appear the same to the programmer. The state of a network object may only be obtained through a method, rather than accessing a field, so the location of that state is hidden from the client of the object. Network objects provide a mechanism for distributing data and events across address spaces within the discipline of a strong type system, and so are a convenient approach for specifying the components of a system.

Most current user interface systems combine their input and output mechanisms into a *virtual terminal* (such as Trestle's Virtual Bitmap Terminal [Manasse91]) which simplifies the model but makes it too limited for applications which support simultaneous multiple input or users (or both). This is why Lantz et al [Lantz87a], in their *Workstation Agent* model, advocated the separation of input and output software, similar to that implemented in TheWA [Lantz87], in which separate processes managed the input devices and the display. Hence, basic *Gemma* devices are categorised as input or output with as few assumptions as possible made about the user-level interactions they are to support. These objects may be gathered together by higher-level objects to provide more complex devices, such as a conventional window server or a virtual terminal, for other clients; an example higher-level system is described in the next chapter.

In this chapter, I define some basic types which provide shared access to low-level devices, using displays and input devices as examples, although there may be other device types such as audio. The core `Device` class provides connection management and naming. It has `BasicDisplay` and `InputDevice` subtypes which define those operations which are common to all displays and input devices. These types, in turn, are further specialised to input and output types that, eventually, describe the features of particular physical devices. Figure 12-1 shows the class hierarchy for the device types described in this chapter.

Figure 12-1. A hierarchy of device types.



12.2. A Basic Device

While devices may have different characteristics, there are commonalities in how we communicate with them which map naturally onto a type hierarchy. The basic `Device` type is concerned with locating the device service and establishing a connection to it. Note that the device details are specified during the initialisation of the object rather than when the device is opened. A `Device.T` represents a particular virtual device; if a client needs a connection to another device it should create a new `Device.T` rather than reusing an existing one; this avoids problems such as an interface of one type being used to import a device of another type.

```
INTERFACE Device;
TYPE Changes = OBJECT        (* described below *)
   METHODS
      properties(t: T; properties: PropertyList.T);
      context(t: T; context: TEXT);
   END;
TYPE Details = REFANY;
(* Each device type will subclass this with a record which describes the characteristics of
   the device. Returned from the open method. *)
TYPE Private <: ROOT;
TYPE T = Private OBJECT
   METHODS
      init(device: TEXT; client: TEXT; changes: Changes := NIL): T
         RAISES {Failed.T};
      (* initialise the state of the object. device describes how to find the device to open,
         client describes and identifies the caller which may help the device make policy
         decisions.
         changes determines how to handle changes in the device context or properties *)
      setChangeHandler(changes: Changes) RAISES {Failed.T};
      (* set how client handles changes to the device's status *)
      open(): Details RAISES {Failed.T};
      (* attempt to open a connection to the device. If successful, the method returns a field
         which describes the device characteristics. *)
      close() RAISES {};
      (* explicitly close the device. If the device is already closed, then NoOp.
         This method is always called when the T is garbage collected *)
      isOpen(): BOOLEAN RAISES {Failed.T};          (* is the device currently open? *)
      description(): Description RAISES {Failed.T};
      (* return the current context *)
   END;
PROCEDURE Lookup(READONLY description: Description): TEXT RAISES {Failed.T};
(* described below, the text value returned can be used for the device field in the
   init method *)
TYPE Description = RECORD
      type: TEXT;
      context := "/";
      properties: PropertyList.T := NIL;
   END;
END Device.
```

The `device` parameter of the `init` method describes how to contact the device and a text string is used to allow flexibility in how it is specified; if the caller is creating a new virtual device then it sets this parameter to `NIL`. The `Lookup` procedure provides a mechanism for finding device names from a name server or *trader*, using ANSA terminology [ANSA89]. The parameters to `Lookup` follow ANSA trading conventions and, briefly, consist of a type description, a name space in which to look, and optional properties to constrain the search. While the device type may not change, moving devices may alter their context or some properties, so the client may define a `Changes` object to handle such events. If no `Changes` object is defined, the default response is to mark the `Device` so that the next method call will raise an exception. Access to the devices on a host could be managed by a process such as Nicolaou's `DesktopManager` [Nicolaou91].

A companion interface reveals that `Device.T` is based on a mutual exclusion lock, and provides mechanisms to register it with the trader:

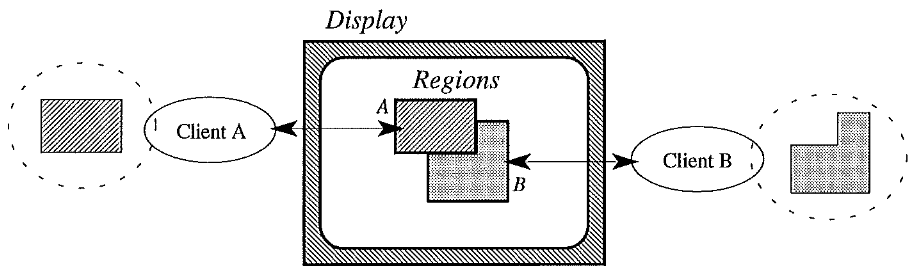
```
INTERFACE DeviceF;
   REVEAL Device.Private = MUTEX OBJECT
      METHODS
         setContext(context: TEXT) RAISES {Failed.T};
         setProperties(property: PropertyList.T) RAISES {Failed.T};
         (* change the current trading values *)
         lookupName(): TEXT RAISES {Failed.T};
         (* returns the trader name for the device *)
      END;
PROCEDURE Export(device: Device.T; READONLY description: Description) RAISES
{Failed.T};
END DeviceF;
```

12.3. Display Servers

12.3.1. Introduction

A display server provides controlled access to arbitrary regions of its screen without imposing a model of a window structure—as Took shows [Took91], not every user interface system need have overlapping windows. A window server can then establish itself as a client of a framebuffer server, requesting regions of the display which represent parts of overlapping windows, as in Figure 12-2. Another window server may also be a client of the same framebuffer, making its own requests for space, so the framebuffer server must arbitrate between conflicting requests and notify other clients when they lose part of their allocation. This, in turn, implies that a window server can no longer assume exclusive control of the frame buffer and must be prepared to handle messages which change its output region.

Figure 12-2. Arbitrary regions give the illusion of overlapping windows.



One benefit from this extra complexity is that a display can support multiple window systems simultaneously. Some advantages of such a facility are outlined by Pascale and Epstein who implemented the Virtual Window System [Pascale92]:

- debugging becomes easier as the different window system implementations can be presented on the same screen and server grabs are limited to the guest window system;

- confidential windows can run on a secure window system while public windows run on another, with copy and paste allowed from less to more secure windows;
- applications written for different window systems can run on the same display. There are many commercial examples of window system integrations, such as X with Microsoft Windows [Giokas92].

12.3.2. A basic display object

A basic display object provides access to a graphic output service; it is used to acquire control over regions of the display. The display service, however, may change the state of a `Region` and the client must be able to respond to such changes, so each client must instantiate a callback object which defines its response to such changes. For example, applications A and B have regions on a display and A moves its region to obscure B's, B's callback will call `regionChanged` so that B may, for example, reshape the image being shown in its region.

```
INTERFACE BasicDisplay;
TYPE Callbacks = OBJECT
   METHODS
      regionChanged(t: T; r: Region; old, new: Region.T);
      (* the server has changed the area of the region *)
      regionReleased(t: T; r: Region.T);   (* the server has released the region *)
      disconnected(t: T);                  (* the server has disconnected the object *)
   END;
TYPE T <: Device.T OBJECT
   METHODS
      requestRegion(area: Region.T;
         callbacks: Callbacks; name: TEXT := NIL): Region RAISES {Failed.T};
      (* request a new region of the BasicDisplay, the callbacks define what the client does when
         notified by the server, the client may also specify a name which identifies the region
         within the server to other clients *)
      shareRegion(name: TEXT; callbacks: Callbacks): Region RAISES {Failed.T};
      (* request a connection to a named Region in the server. This client may or may not be able
         to change the region depending on the permissions set by its creator, but it will
         receive copies of the callbacks *)
      listRegions(): List.T RAISES {Failed.T};
      (* returns a list of all the Regions in the T with names and properties *)
    END;
```

A `BasicDisplay.Region` gives the client rights over an arbitrarily-shaped area of the display, such as the ability to change its shape; these rights depend on the type of region and the policies of the display service. The extent to which the shape is truly arbitrary (as against polygonal) depends on the implementation of the display.

```
RegionPrivate <: ROOT;
Region = RegionPrivate OBJECT
   METHODS
      display(): T;                   (* return the Display for the Region *)
      area(): Region.T RAISES {Failed.T};
      (* return the current area of the BasicDisplay *)
      name(): TEXT RAISES {Failed.T};
      (* return the name of the Region or NIL if there is none *)
      setArea(area: Region.T): Region.T RAISES {Failed.T};
      (* request a different area. Return the area actually granted by the server *)
      release() RAISES {Failed.T};    (* explicitly free the region in the server *)
   END;
END BasicDisplay.
```

12.3.3. More complex display types

Clearly, an interface which provides only the allocation and deallocation of areas of a display is of limited use. Clients must be able to write to the display so a Pixrect [Sun90a] subtype, for example, could be implemented to provide basic drawing operations to the client.

Note that the drawing is done in the `Region` rather than the `Display`, the caller might not be the only client of the display and so cannot assume the right to draw everywhere; a client may, of course, attempt to acquire a `Region` which covers the whole display. Another issue is how the server should respond when a client tries to draw outside its region. The most suitable action for this interface is for the display to clip the drawing operation without informing the client; other interfaces might raise an exception, similar to an address violation in virtual memory systems.

```
INTERFACE PixrectDisplay;
TYPE Region <: RegionPublic;
TYPE RegionPublic = BasicDisplay.Region OBJECT
   METHODS
      rop(destination: Region.T; op: PaintOp.T; source: Point.T) RAISES {Failed.T};
      (* raster operation *)
      polypoint(READONLY points: ARRAY OF Point.T; op: PaintOp.T;
         offset := Point.Origin) RAISES {Failed.T};
      (* draw an array of points *)
      putcolormap(map: Colormap.T) RAISES {Failed.T};
      getcolormap(): Colormap.T RAISES {Failed.T};
      (* routines to change the state of the Region *)
      (* etc. *)
   END;
TYPE T <: BasicDisplay.T;
END PixrectDisplay.
```

Many graphics libraries, such as Pixrect and SRGP [Foley90], set values in the device to be used by subsequent calls to the library. This is straightforward where there is only one client of the library, but a shared device has to multiplex its state between its clients. The techniques for doing so are obvious but an inappropriate implementation may be slow if the client is constantly swapping the device settings. Batching can greatly improve performance [Manasse88], so another interface allows a client to group calls together:

```
INTERFACE BasicDisplayF EXPORTS BasicDisplay;
REVEAL BasicDisplay.RegionPrivate = OBJECT
  METHODS
    startBatch(suggestedSize: CARDINAL := 0) RAISES {Failed.T};
    endBatch() RAISES {Failed.T};
    flush() RAISES {Failed.T};    (* wait until any outstanding requests have been sent *)
  END;
END BasicDisplayF.
```

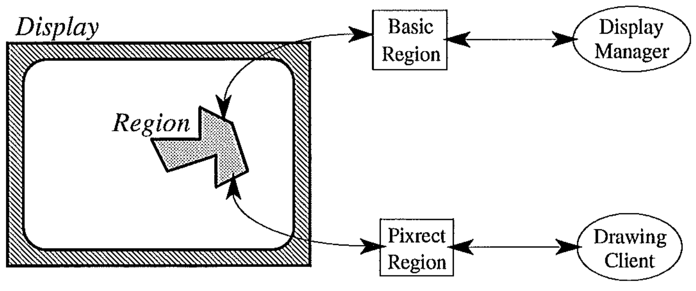
As Schmidtmann et al. [Schmidtmann93] point out in their description of a multi-threaded Xlib, it is increasingly the case that every library should be written with support for multiple threads, regardless of whether the original platform supports them. Similarly, it might be that software designers should also consider the possible distribution of the components of their systems, even if the first implementation is entirely local. This could involve providing hooks in the code where distribution issues arise, or changing the encapsulation boundaries within the software architecture.

12.3.4. Multiple interfaces to a Region

A client may hold more than one connection to a display server at once, so a client can open multiple connections to represent multiple characteristics. This avoids the need for multiple inheritance in the type system when a virtual device with several characteristics is needed, as a client can open several connections to the same named region from multiple interfaces. A client timestamps its requests so communications with multiple interfaces to a `Region` from a single client can be kept in order.

Multiple interfaces also allow access to individual facilities in the server to be controlled for each client. For example, in Figure 12-3, a server supports the `BasicDisplay` and `PixrectDisplay` types and a client requests a named `BasicDisplay.Region` which it can manipulate. Another client asks to share the `Region` using the `PixrectDisplay` interface and receives a connection which is a `PixrectDisplay.Region`. Both `Regions` (the `BasicDisplay` and the `PixrectDisplay`) refer to a common structure in the server, but have access to different aspects of it. This differs from X, for example, where a client, once connected to the server, has access to all its facilities.

Figure 12-3. Multiple interfaces to a Region.



One situation where multiple connections are useful is in the manipulation of overlays and cursors (which are, in effect, a special case of overlays). A telepointing system, for example, might allow remote clients to acquire an overlay region, to make annotations, but not a drawing region to change the common image.

```
INTERFACE OverlayDisplay;
(* The overlay is defined as a (PostScript-like) path which will always be displayed
   on top while it is not empty. To clear an overlay, set the path to NIL.
   The overlay is clipped by the Region, but the Region is transparent and can
   overlap any other Region in the server *)
TYPE Region <: BasicDisplay.Region OBJECT
   METHODS
      setPath(path: Path.T) RAISES {Failed.T};          (* set the overlay path *)
      movePath(offset: Point.T) RAISES {Failed.T};      (* move the path by offset*)
      getPath(): Path.T RAISES {Failed.T};              (* get the overlay path *)
   END;
TYPE T <: BasicDisplay.T OBJECT END;
(* etc. *)
END OverlayDisplay.
```

The interface does not specify how the overlay is implemented, so the server might use a hardware plane for the first overlay and software for others. Another approach would be for the server to allocate a bit plane for overlays and export a private interface for it to a separate overlay manager; the overlay manager would then export the `OverlayDisplay` interface and combine the requests from its clients into a single image. Overlay managers with different policies can be optimised, for example, for either pen-based or mouse-based interfaces. In addition, the interface places no explicit limits on the number of overlays to be supported, so extra cursors may be dynamically added to the higher level window system.

12.3.5. Display servers and graphics systems

This approach could, of course, be implemented over standard hardware (although an efficient version would take some effort), but some other graphics systems suggest interesting possibilities. The approach maps nicely onto the concept of *virtual graphics*, first proposed by Voorhies et al. [Voorhies88]. The authors noted that, whereas CPUs provide hardware support for virtual memory and virtual processors, graphics systems are still oriented to a single client. Current user interfaces, however, support multiple windows, each of which may have different state—this is comparable to the mismatch between trends in hardware and operating system software described by Ousterhout [Ousterhout89]. The authors proposed elevating graphics to a first-class system resource, such as main memory, and providing a virtual interface. Each client then has the illusion of having its own graphics renderer and is protected from other clients' activity.

Virtual graphics also formed part of the *Thistle* display system [Brown91] in which the components of the graphics system, such as rendering and screen update processors, form a symmetric, heterogeneous multiprocessor—they share a consistent view of a single memory system. This supports virtual graphics, as pages of framebuffer images can be swapped out to main memory, and gives client applications access to specialised hardware which would normally be inaccessible in the graphics subsystem, such as the rendering processor. Regions and virtual displays both provide a mechanism for multiple clients to share a common display, switching the state of the display device for each client, for example. Virtual displays, however, hide changes on the physical display from the client, whereas Regions notify the client of such changes.

The Psychology Workstation [Cowan91] was built to provide precise timings for displaying images and user responses with multiple simultaneous tasks. Server-based approaches, such as X, do not allow the client to know exactly when an image reaches the screen as multiple clients can make unpredictable demands on the server. The solution was to provide a separate rendering agent for each client, including the window manager, with direct access to the frame buffer; a client may be running on its own processor, thus avoiding CPU and memory contention with other clients. Access to the display hardware is controlled by a *layer administrator* which keeps track of all the client windows and determines which client 'owns' each pixel. The layer administrator informs renderers of changes in window layering or placement, but the renderers manage the contents of the windows without interference, explicit or implicit, from other clients. Again, the relation to the `Region` and `Display` approach is clear, although the Psychology Workstation is limited by its implementation and does not support distributed applications.

These examples show that there are ways to organise display hardware which are not well supported by current window systems. In particular, systems which allow clients access to components of the graphics pipeline do not match well with the virtual terminal approach.

12.3.6. Summary

This section described a model for allowing clients to share displays. Clients establish a connection to a display and then request control of arbitrary regions, which may be accessed by other clients. Policies for arbitrating between clients are implemented by a separate display manager. Regions are subclassed to provide access to particular features of a display, such as drawing operations or overlays, so access to a display can be controlled more finely than in conventional output servers.

12.4. Input Servers

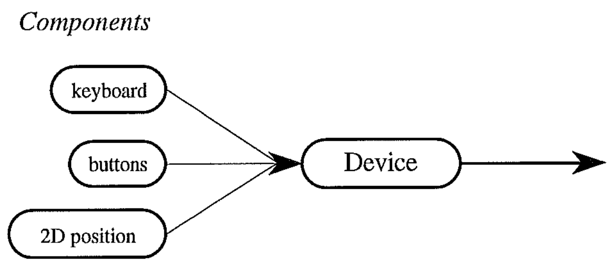
12.4.1. Introduction

This section describes a model for sharing input devices. As with display servers, input servers provide controlled access to devices which may be shared between clients. There are fewer cases for input than for output where a single device is shared, although Brown et al. divide a tablet into windows [Brown90] and the use of multiple pens on a shared desk or Liveboard requires that multiple input sources are abstracted from shared input sensors. An input stream may also be multicast; for example, pen input might be sent to both a display manager and a recognition process to generate feedback and provide 'eager' interpretation of gestures in parallel. More important, perhaps, developments in user interfaces, such as MMM, show the need for flexible and dynamic routing of input events. DDD, Dialogo [Lauwers90] and Mermaid [Ohmori92], for example, implement multi-user applications by distributing low-level input events, rather than application-level events.

12.4.2. Devices and components

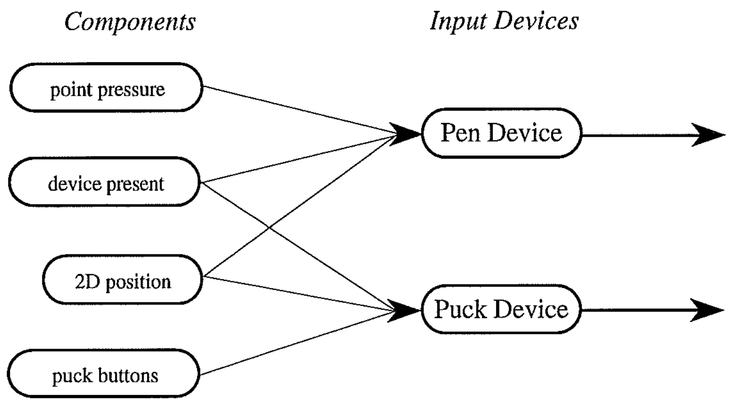
Unlike `Displays` and `Regions`, there are difficulties in using multiple objects to represent aspects of a single input device. The events in an input sequence must arrive in a consistent order and multiple objects with multiple threads of control would require co-ordination in the client. The solution is to allow users to add components to a virtual device, rather than abstract them from it. Figure 12-4, for example, shows a conventional input device assembled from keyboard, buttons and 2D position components. The virtual device provides a synchronisation point for the physical devices it manages; it timestamps events as they pass through it.

Figure 12-4. Assembling an input device from components



Multiple virtual devices may be constructed over the same physical devices, but a client which uses more than one virtual device must perform its own synchronisation between them; this is similar to the input model for the Workstation Agent. For example, a tablet may be able to distinguish between a pen and a puck, and detect whether either is present, so two virtual devices could be extracted from the same physical device. As in Figure 12-5, both the pen and puck devices contain *position* and *present* components, but the pen also contains a *point pressure* component and the puck contains a *puck buttons* component. The pen and the puck can be treated as different devices, although both drive the same tablet, and can be handled by different dialogue threads.

Figure 12-5. Sharing components between two input devices



12.4.3. The basic InputDevice

The `InputDevice.Component` provides access to all or part of a physical input device or an existing virtual input device; the basic interface provides details of the device and simple flow control. The `getState` method returns the current state of the component and so can be used for *sample* mode input (as described in [Duce90]) in which the client periodically polls the device. If the component is disabled, then no events will be delivered until it is re-enabled, although `getState` will still return its current state.

```
INTERFACE InputDevice;
TYPE Event = OBJECT
      t: TimeStamp.T;
      device: T;
      component: Component;
   END;
TYPE Component <: Device.T OBJECT
   METHODS
      getState(): Event RAISES {Failed.T}; (* return the current state of the component *)
      enable(on := TRUE; flush := FALSE) RAISES {Failed.T};
      (* tell the component to start or stop sending events.
         If flush, then clear any input queues *)
      isEnabled(): BOOLEAN RAISES {Failed.T}; (* returns whether device is on or not *)
      reset() RAISES {Failed.T};    (* clear the device and reset it. Could be noop *)
   END;
```

The `InputDevice.T` allows the client to assemble a compound device. The `enable` method enables or disables the components of the device and also manages any internal threads which feed input from the components to the device queue. The `getState` method gets the state of all the components and returns a compound state, if a suitable type is defined, or a list of the individual states. The `nextEvent` method returns the event at the head of the device's input queue, if one is available, or blocks the calling thread until one arrives.

```
TYPE T <: Public;
TYPE Public = Device.T OBJECT
   METHODS
      addComponent(component: Component) RAISES {Failed.T};
      removeComponent(component: Component) RAISES {Failed.T};
      (* add or remove components from the device. addComponent fails if (amongst other things)
         component is already present in the device *)
      listComponents(): ComponentList.T RAISES {Failed.T};
      (* return a list of all the current components *)
      getState(): Event RAISES {Failed.T}; (* return the current state of the device *)
      nextEvent(): Event RAISES {Failed.T};
      (* return the next event from the device. If there are no events to be processed,
         it blocks the thread until one arrives *)
      enable(on := TRUE; flush := FALSE) RAISES {Failed.T};
      isEnabled(): BOOLEAN RAISES {Failed.T};
      reset() RAISES {Failed.T};
   END;
END InputDevice.
```

Where performance is an issue, some clients may need more detail, which the following interface reveals. The `eventFromRaw` method translates data as it arrives from a component device into typed data structures, the `InputDevice.T` uses this method to parse the input stream and insert the result in its input queue. This interface allows clients to implement their own processing of the input stream.

```
INTERFACE InputDeviceF EXPORTS EventInputDevice;
REVEAL InputDevice.Component = Device.T OBJECT
   METHODS
      eventFromRaw(READONLY buffer: ARRAY OF WORD): Event RAISES {Failed.T};
   END;
```

Some clients may find repeated calls of `nextEvent` too inefficient and require access to the input queue—as Clark and Tennenhouse [Clark90] point out for networks, performance can suffer from inappropriate enforcement of protocol layers. One possible optimisation is that a client starts a thread which periodically examines the input queue to remove unnecessary events. The following revelation makes the queue available to clients and provides a method for raising errors:

```
REVEAL InputDevice.T = InputDevice.Public OBJECT
      queue: InputQueue.T;               (* incoming events, not yet processed *)
      components: ComponentList.T;       (* a list of the components of the device *)
   METHODS
      markFailed(failure: Failed.Failure);
      (* mark the device so that failure will be raised during the next method call *)
   END;
END InputDeviceF.
```

12.4.4. More complex input types

As with the display types, we need to subclass these basic classes to do something useful; a two-dimensional pointer, for example:

```
INTERFACE TwoDPointer;
   TYPE Event = InputDevice.GenericEvent OBJECT
      point: Point.T;
   END;
   TYPE T <: InputDevice.Component;
```

Pointers may give absolute or relative co-ordinates, so the `TwoDPointer.T` needs two subclasses which provide different controls for each type. Absolute pointing devices, such as pens, may need calibration, so in the `TwoDPointer.Absolute`, a `Warping.T` defines a mapping from one set of points to another and setting `warping` to `NIL` clears any current warping. In addition, the `warping` is a value which might be changed by other clients which share this connection, so an extra `Changes` method must be defined.

```
TYPE AbsoluteChanges = Device.Changes OBJECT
      METHODS
         warping(absolute: Absolute; READONLY newWarping: Warping.T);
      END;
   TYPE Absolute <: T OBJECT
      METHODS
         setCalibration(READONLY warping: Warping.T := NIL) RAISES {Failed.T};
         getCalibration(): Warping.T RAISES {Failed.T};
      END;
```

On the other hand, there are pointing devices, such as mice, which give relative motion. In this case the relevant control is to set the acceleration parameters.

```
   TYPE Relative <: T OBJECT
      METHODS
         setAcceleration(READONLY acceleration: Acceleration.T) RAISES {Failed.T};
         getAcceleration(): Acceleration.T RAISES {Failed.T};
      END;
   (* etc. *)
END TwoDPointer.
```

12.4.5. Other input models

There are some interactive systems based on input devices providing services to clients. The Adagio workstation [Tanner86], for example, used a multi-tasking operating system (Harmony) to support parallel input from several devices; a *Switchboard* task routed events from device tasks to interested clients. The Adagio implemented generic devices and provided a flexible mechanism for linking applications with input and output devices, but was not distributed and was specialised for its role as a robotics workstation. The VUE architecture [Appino92] is based on distributed device servers which filter raw device events and pass them to higher level interpreters. VUE is designed for virtual reality and so the emphasis is on combining devices together into a common shared virtual world. The *Gemma* architecture, however, is aimed towards embedding computation in the users' environment and so the emphasis is on extracting the features of a device which interest a particular client.

There are also models which construct logical input devices from smaller components. Duce et al [Duce90], working within GKS, propose a hierarchy in which composite devices can be used to build higher level logical devices. This hierarchy can be supported within *Gemma* as is shown by the Tablet example (Figure 12-4) in which two components are combined into a single service, although the GKS model also includes feedback. The X Input Extension [Patrick89] specifies complex devices in terms of their constituent parts, but the devices must be compiled into the X server and so cannot be changed dynamically. Furthermore, clients cannot acquire access to individual component parts, as against the whole device. This is not so significant when everything is in a single process and input can be filtered at the window level, but *Gemma* devices may be distributed and unnecessary events consume bandwidth.

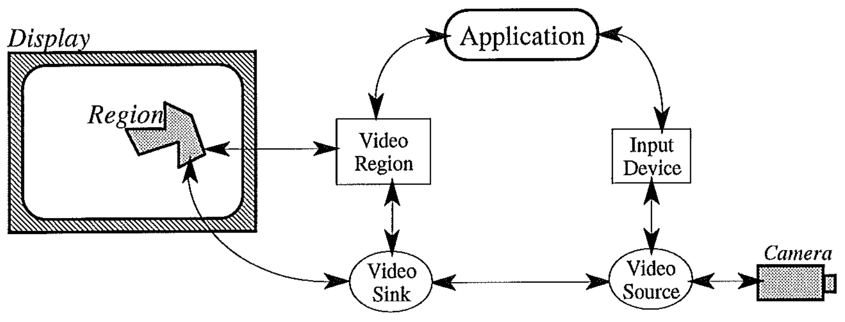
12.4.6. Summary

This section described a model for collecting together distributed input devices. A compound input device is constructed from simpler components for which it provides a serialisation point. Compound devices may share components to reflect different uses of the same device and components, and exported `InputDevice.Ts`, .may be shared between clients.

12.5. Continuous media

Gemma devices can also manage continuous media such as audio and video. The `VideoDisplay` interface allows video to be displayed on a screen. Unlike other input devices, continuous media distinguish between in-band and out-of-band data (in-band data consist of the video frames or audio signal, out-of-band data are control and synchronisation signals) so the video stream itself is controlled by a `Video.Sink` object which manages quality of service specifications, for example. This division between device state and network state is common in continuous media systems, examples include the Touring Machine [Bellcore93] and the Etherphone [Vin91]. The connections between components for displaying video in a `Region` are shown in Figure 12-6.

Figure 12-6. The components of a video stream.



The `VideoDisplay.Region` establishes a link between a video sink and a display and provides access through its event handlers to the out-of-band data which is relevant to the user interface layer.

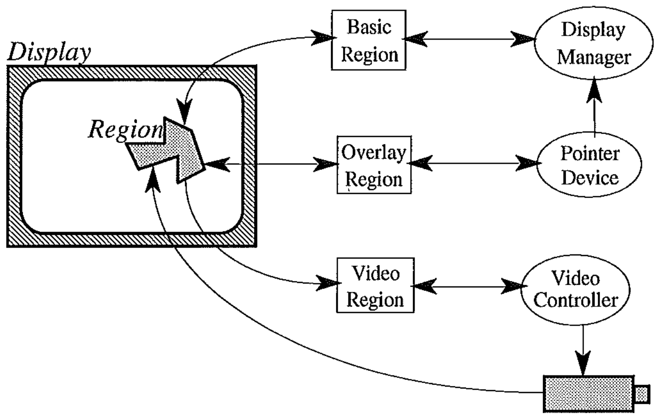
```
INTERFACE VideoDisplay;

TYPE Region <: BasicDisplay.Region OBJECT
   METHODS
      setSink(sink: Video.Sink; syncHandler: SyncHandler := NIL) RAISES {Failed.T};
      (* set the video sink to use. A SyncHandler will only by called if syncHandler <> NIL.
         reset the sink to change the handler  *)
      sink(): Video.Sink RAISES {Failed.T};
      (* return the video sink associated with the Region *)
   END;
```

There are two types of event handler: the `Callbacks` handles events arising from control of the stream and the `SyncHandler` handles events arising from the contents of the stream. The `Video.Sink` object may, of course, have handlers of its own to manage communications-level activity whereas these handlers should specify user-interface level behaviours. For example, a `VideoDisplay.SyncHandler` would be used to drive an application's frame counter, whereas a `Video.SyncHandler` would be used to smooth jitter.

```
TYPE Callbacks = BasicDisplay.Callbacks OBJECT
   METHODS
      started(t: T);                       (* video stream has been started *)
      stopped(t: T);                       (* video stream has been stopped *)
      failed(t: T; failure: Failed.Failure);      (* video stream has failed *)
   END;

TYPE SyncHandler = OBJECT
   METHODS
      frame(region: Region; frameNumber: Video.FrameNumber; time: TimeStamp.T);
   END;

TYPE T <: BasicDisplay.T OBJECT END;

(* etc. *)

END VideoDisplay.
```

For example, in Figure 12-7, the display manager controls the shape of the region, an overlay region allows a pointer device to drive a cursor, and the video region tells the video controller which parts of the screen to send images to; this provides the user with a pointer-driven cursor over a moving video image.

Figure 12-7. A Region with a video interface.



The `VideoSource` interface is a subclass of an `InputDevice.Component` which allows control over the source of a video stream; communications-level issues are managed by the `Video.Source` object. As with the `VideoDisplay`, the `started`, `stopped` and `failed` events are handled by callbacks because they are part of the administration of the stream, whereas the frame number is part of the contents of the stream.

```
INTERFACE VideoSource;
TYPE Changes = InputDevice.Changes OBJECT
   METHODS
      started(t: T);                      (* video stream has been started *)
      stopped(t: T);                      (* video stream has been stopped *)
      failed(t: T; failure: Failed.Failure);      (* video stream has failed *)
   END;
TYPE Event = InputDevice.GenericEvent OBJECT
      frameNumber: Video.FrameNumber;     (* sequence number of last frame sent *)
   END;
TYPE T <: InputDevice.Component OBJECT
   METHODS
      start() RAISES {Failed.T};
      stop() RAISES {Failed.T};
      (* these methods start and stop the video stream, the inherited enable method
         controls the delivery of frame numbers to the client *)
      setSource(source: Video.Source) RAISES {Failed.T};
      source(): Video.Source RAISES {Failed.T};
   END;
END VideoSource.
```

This application is similar to the desktop video example in Chapter 8 of [Nicolaou91] except that it hides more of the networking detail and is more integrated with the user interface layer. As pointed out above, it allows more specific control over access to facilities in the server, unlike the Pandora extension to X [King92], for example, in which a client, once connected to the server, has access to everything including (as the author points out) other peoples' video cameras. A similar approach can be applied to audio services, as shown by Angebranndt et al [Angebranndt91], who construct logical audio devices (LOUDS) from virtual device objects, which can be described with a class name and a set of attributes, to provide server-based audio. For example, an answering machine is made up of a recorder, a player and a telephone. These virtual devices are then mapped onto the physical devices the server controls. The AudioServer, however, only allows devices in the same server to be put together, whereas *Gemma* allows arbitrary devices to be combined. Constructing applications from connections to multiple servers has been shown by Arons, [Arons92], who combines audio and graphics servers.

The Desk Area Network (see Chapter 4) provides an interesting platform for multiple media applications. It allows distributed access to end-devices within a host, unlike existing workstations where interactions with devices must pass through the host operating systems. Remote clients, such as a video source or sink, can have a data stream directly routed through to the device, but they still need to co-operate with other clients of the device. Multiple virtual interfaces to components provide flexible access to the hardware, or software implementations of hardware functions, and flexible control of that access.

12.6. Summary

This chapter describes how low-level input and output devices are specified in *Gemma* using a class hierarchy and network objects. Output devices may support multiple virtual devices (`Displays` and `Regions`) each of which exposes one or more facilities of the base device. Virtual input devices may be built up from one or more base components and provide a synchronisation point for those input streams. Time-sensitive media are controlled by virtual devices which provide a handle for control of the medium—the data themselves are managed by the communications objects.

Gemma supports some of the architectural requirements described in Chapter 11: the basic device type makes devices accessible to arbitrary clients and notifies those clients when the status of a device, such as its location, changes; clients may share a device by importing the same device object; the output model provides controlled access to facets of a display device; the input model allows input devices to be assembled into compound devices, clients of the compound device may then unpack input events to obtain as much detail as they need; and, the model supports continuous media.

Other requirements, such as the association of input and output devices and compound output devices, are supported in the window system described in the next chapter.

13. A Simplistic Window System

13.1. Introduction

Low-level devices are too primitive for building all but the simplest applications, so we need a level which will bind them into more complex services. In particular, logical devices at this level provide a synchronisation point for their components: in space, by defining a mapping between input and output device co-ordinates, and in time, by timestamping events. These are the objects usually described as servers in window systems; they provide higher-level services from the low-level devices they import. This level defines how the input and output devices are to be combined, and presents a logical set of devices to the application level above it. For example, a windowing object manages polygonal regions of a screen which the users see as overlapping windows, and the relative motion events of a mouse are mapped to absolute display co-ordinates.

This chapter describes a simplistic window server, sw, that provides basic support for a hierarchy of windows and event dispatching in a highly distributed user interface. This design is intended to demonstrate how a window service might be constructed using *Gemma*, rather than describing a definitive system. sw is unusual in that it is intended to support an environment in which there are many input and output devices scattered around a room, some of which may be arbitrarily combined. Two user interaction styles from those described in Chapter 11 show the sort of issues that need to be addressed:

- a person uses an identifiable pen to draw on a liveboard for the first time. The window system must provide hooks to allow applications to accept and validate this new source of input.
- a person uses a portable display to enlarge another display, then carries it away bearing the new data from this interaction. The window system must be able to manage multiple displays which might be used together.

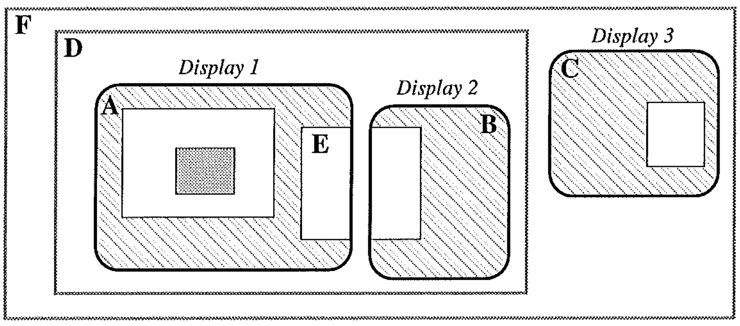
sw uses two techniques to support these requirements: a common window abstraction used at all levels of the hierarchy which can also bind physical displays together, and the ability to group input devices and attach them to a window at any level of the hierarchy. sw provides a model within which multiple distributed devices can be combined into a higher-level user interface, using the window as a co-ordination point for linking input and output to provide timestamping and alignment between devices. After describing these techniques, the chapter discusses input handling in more detail and the relationship between sw and *Gemma*.

13.2. The SWwindow

The basic component in `SW` is the *window*[*] which is an arbitrary area of 2D space that may contain child windows. An unusual point is that this general definition does not require a top-level window to be associated with a physical display—it is not always the case that the top-level window is the "closest" to the output device. A top-level logical window may handle drawing requests by either refusing to accept the request and raising an exception or by passing it down to the relevant child.

For example, a logical window may contain a set of displays around a room with an arbitrary policy for defining the spatial relationships between the children (Figure 13-1). A logical window may allow paint operations to span two children, so two physical displays could be tiled together to make a larger virtual display; the parent forwards its paint requests to the children who overlap the image being drawn.

Figure 13-1. A logical hierarchy of windows. Windows A, B and C represent the top level of displays 1,2, and 3 respectively. A and B are children of a logical window D which is used to bind displays 1 and 2 together, so that window E can cross between them. C and D are children of a logical window F which represents a common domain for access control and handling unexpected events.



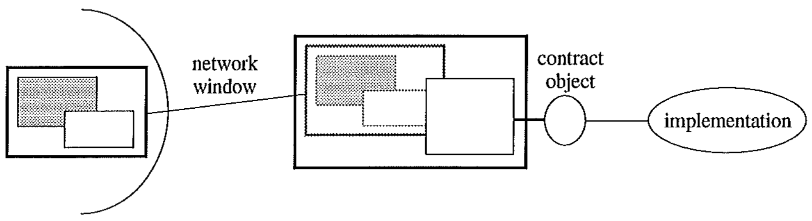
The use of a single abstraction means that applications need only be concerned with manipulating a single top-level window. An application may be moved between displays by reparenting its top-level window, and the role that a display plays with respect to neighbouring devices may also be changed by reparenting the window associated with that display.

The basic window type is subclassed from a *Gemma* `Device.T` which provides import/export facilities. Distribution is achieved by inserting a *network window* between parent and child that relays events and requests between the two. This is similar to Trestle's cross-address-space filter, except that the child window does not have to be attached to a parent window—if exported, it can maintain its existing state and be attached to another parent, local or remote. A process may then export its significant windows, such as the bottom-level window of a display or the top-level window of an application, which other applications may then import and use as an interaction device. In addition, a

[*] It may no longer be possible to find a term for this component which does not have connotations from existing systems. Other systems use "sheet," "screen," "canvas," "frame" and so on. I stick with *window* for simplicity.

window subclass, such as one which manages the layout of its children, may encapsulate its specialised behaviour in an object like a Silica *contract* [Rao90]; a network version of this object may then be driven from another process (Figure 13-2). This is similar to the X window manager where some of the functionality of the window system is implemented by a remote process using a specialised protocol.

Figure 13-2. Two mechanisms for application distribution. Network windows to connect parent and child in different address spaces and a contract object to encapsulate, and distribute, the behaviour of the contents of a window.



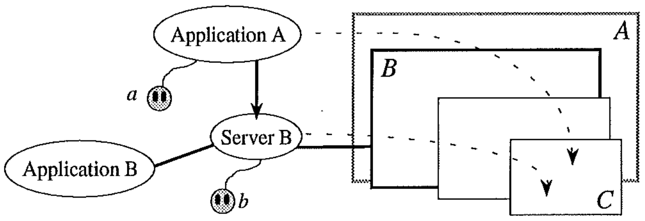
SW only supports a single parent per window but it is possible to conceive of other models where, for example, the parent is a group or there may be a different parent per client for different functions of a window; these, however, raise too many issues to be treated within the scope of this dissertation.

13.3. Input Handling

Much input handling in SW is simply concerned with routing events that pass through a window to the appropriate child, as with any conventional window system. A client may also, however, attach an input device to a window which makes that window the root of the tree for interactions with that device. For example, selections made with a device are retained at the level of the window to which the device is attached. A thread is then attached to the window to dispatch events from its devices to its descendants. This is similar to the InputVBT in the DDD in which most of the interactions with a particular device are handled below the VBT to which the device is attached. As with MMM, each window maintains an input queue with an associated thread to process its events and arbitrate between devices; parents may ask children to accept events to add to their input queue (see section 13.5).

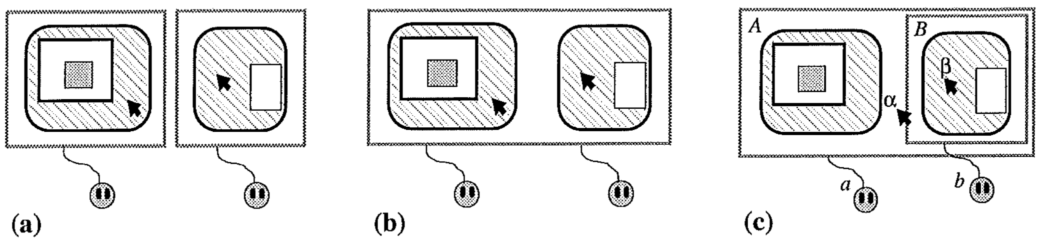
In Figure 13-3, for example, application *B* sees a window service which includes window *B* and mouse *b*, application *A* creates an internal window hierarchy of which window *B* is a member; server *A* sees only events from mouse *a*, while server *B* sees events from both mice. The approach allows input to be managed at multiple levels in a window hierarchy; in Figure 13-4 (c), mouse *b* may only move within window *B*, whereas mouse *a* can cover both windows. In the example of the drafting class, all the drafting boards are children of a common logical window; input from the students' pens is handled at the level of the individual board, whereas the teacher's pen is registered with the top-level window so she does not have to establish a separate connection with each board.

Figure 13-3. Event dispatching in different window levels. Server *B* uses mouse *b* and window *B*, which is the top-level window of a display device, and provides a window service to application *B*. Application *A* creates a virtual window *A*, which includes window *B*, and accepts input from mouse *a*; input events and output requests which fall into the domain of window *B* are passed on to server *B*.



This approach also allows clients to determine whether event handling for two neighbouring displays should be kept separate or combined in a larger virtual display (Figure 13-4, (a) and (b)). In the latter case, a client can make both displays descendants of the same logical window and associate the input devices with that ancestor.

Figure 13-4. Cursors and windows. In (a) the pointers are restricted to their respective displays; (b) the displays are part of the same logical window and the pointers have access to both. In (c) mouse *a* is attached to window *A* and drives cursor α, mouse *b* is attached to window *B* and drives cursor β; cursor β is restricted to window *B*, but α may enter window *A* or *B*.



13.4. The SWinputDevice

Users need a mechanism for grouping arbitrary sets of devices to provide an event order across the devices and an input focus for non-positional events such as key presses. In addition, other clients may wish to receive a copy of the event stream from an input device once it has been calibrated with a display. The approach taken here is to create a further virtual device which may be exported for other clients. When attached to a window, events from the component devices are distributed through the window hierarchy as if from a single device—although a client can unpack the original source of the event. There may be an arbitrary number of pointing components, of which one may be nominated as the *focus pointer* which is then used set the input focus window; the default input focus window is the one to which the device is attached. As with the `SWwindow`, the application may chose to export the new device.

```
INTERFACE SWinputDevice;
TYPE T <: Public;
TYPE Public = InputDevice.T OBJECT
   METHODS
      setFocusPointer(pointer: InputDevice.Component) RAISES {Failed.T};
      focusPointer(): InputDevice.Component RAISES {Failed.T};
      setCursor(pointer: InputDevice.Component; cursor := SWcursor.None)
        RAISES {Failed.T};
      cursor(pointer: InputDevice.Component) SWcursor.T RAISES {Failed.T};
      (* set the cursor shape for a pointing component which is part of this device.
        used when the device is attached to a window *)
      calibratePointer(component: InputDevice.Component;
         warping := Warping.None) RAISES {Failed.T};
      (* sets transformation for positional events from input co-ordinates to
        display co-ordinates *)
   END;
END SWinputDevice;
```

This arbitrary combination of input sources allows some novel interaction techniques. For example, a person at a digital drafting board could switch between two or three different pens with one hand, to represent different drawing functions, with a button device in the other hand to provide modifier keys common across all the pens. Each pen can be represented by a different `SWInputDevice.T`, all of which include a copy of the button device.

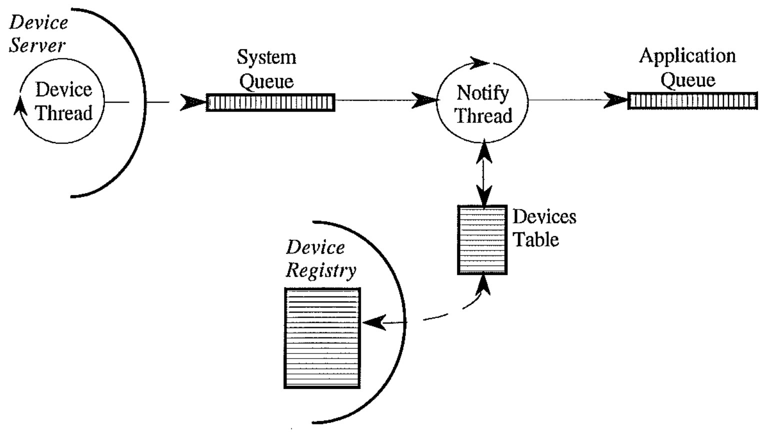
13.5. Users and Input Devices

13.5.1. Making associations

MMM shows that multi-user and multi-device interfaces need to be able to make associations between devices and users and roles. MMM runs within a single address space and so can maintain internal device and user tables, but many of the scenarios described in Chapter 11 need to maintain device associations between displays. This assumes that some input devices have an identity which they carry with them, rather like an IP address on an Ethernet, which is a property of the physical device rather than the virtual devices implemented over it. To make these associations available across hosts, `SW` holds them in a *device registry* which is accessible over the network; the need for rapid response means that device registries may need to be cached locally, but this matches the intended use. Clearly, at most one user can own an identifiable device, although there may be multiple clients receiving events from it. `SW` also provides a *user registry* which holds details about users, including their application preferences.

Within `SW`, each event dispatcher keeps a list of devices it recognises, with their respective owners. When a dispatcher encounters an event from a device it does not recognise, it queries its device registry to see if the device is registered with anyone. Figure 13-5 revisits MMM, Figure 8-9 (page 56) to show how processes and data structures which were internal in MMM are now distributed. If the registry returns an unrecognised owner, `SW` queries the user registry to acquire the user's details. In addition, the new device may also be part of a user's compound device, such as a pen associated with a portable button pad, in which case the dispatcher must import the compound device. The dispatcher then attempts to pass the event to a descendant window for interpretation by a client application.

Figure 13-5. Distributed event handling in SW. The Notify Process is attached to a window and feeds events to its children. The Device Server and Device Registry are remote services.



Assembling all this information may take some time for an interactive application, so it may best be delegated to another thread. To avoid excessive switching, a user who is moving frequently between displays can make them children of a common logical window and attach the input device at that level; the logical window will store the user details and dispatch events to the appropriate child.

13.5.2. Access control

The device registry is also used to control user access in a shared interface. The event mechanism attempts to dispatch positional events from leaf to root (unless the device has been grabbed). Each window has an `acceptOrRejectEvent` method which either accepts the event and returns `TRUE` or returns `FALSE`; where a window contains children, `acceptOrRejectEvent` should first try to pass the event to any child which is appropriate before considering it for its own window. Thus, the interpretation of an event and access control is localised to the application in charge of each window. Client applications can retain a list of users who may make changes to an application's appearance or its data and check the owner of the source device with the access list before accepting an event. An implementation of `acceptOrRejectEvent` for a leaf application would be:

```
PROC AcceptOrRejectEvent(event: Event): BOOLEAN =
   device ← event.device();
   ownerRec ← LookupOwner(device);
   IF ownerRec = None THEN
      RETURN FALSE; (* we're not interested in anonymous devices *)
   FI;
   deviceRec ← LookupLocalDeviceRec(device);
   IF deviceRec = None THEN
      deviceRec ← NewLocalDeviceRec(device);
   FI;
   IF ISTYPE(event, Event.AcquireFocus) THEN
      (* only owner may alter the contents of this window *)
      IF ownerRec.writePermission() THEN
         AcquireInputFocus(device, event);
         RETURN TRUE;
      ELSE
         RETURN FALSE;
      FI;
   FI;
   (* add event to local application queue *)
   AddEventToQueue(event, deviceRec, ownerRec);
   RETURN TRUE;
END;
```

Whereas the implementation for a simple parent which allows anyone to change the positions of its children would be:

```
PROC AcceptOrRejectEvent(event: Event): BOOLEAN =
   device ← event.device();
   (* try to pass event on to child *)
   child ← FindDestinationChild(event, device);
   IF NOT child = None THEN
      IF child.acceptOrRejectEvent(event) THEN
         RETURN TRUE;
      END;
   END;
   (* event not for any child, add to local queue *)
   deviceRec ← LookupLocalDeviceRec(device);
   IF deviceRec = None THEN
      deviceRec ← NewLocalDeviceRec(device);
   FI;
   AddEventToQueue(event, deviceRec, None); -- not interested in owner of device
   RETURN TRUE;
END;
```

If stricter control is needed, the application can query the device registry, to validate its cached values, before making significant changes. For example, a mail tool may use locally-stored owner identification for altering the appearance of the tool and for constructing a message, but query the registry before sending a message.

Unaccepted device events are returned to the device window which can provide a default handler. Pen strokes, for example, could be passed to a signature recogniser to allow people to "sign in." Otherwise, events might be passed further up the window hierarchy, if any, to be caught by a higher-level handler. This can be exploited by users who wish to switch between a number of displays by making all the displays children of the same logical parent. The users may then connect to the parent window and move freely between the displays; the parent will perform all the ownership validations for its children.

13.6. Summary

This chapter describes the example Simplistic Window system and how it provides a mechanism for grouping sets of devices. It supports the architectural requirements defined in Chapter 11 of co-ordinating input and output devices and provides a mechanism for constructing and making available compound interaction devices. Users can construct their own interaction environments by dynamically linking together input and output devices. `SW` would, for example, be particularly suitable as an infrastructure to support the Pad model of user interaction, described in section 2.6.

The *window* provides a common abstraction for defining the scope of interactions. It can be extended above the hardware ("meta-windows") to link displays together, and it provides a co-ordination point between input and output devices. It also allows applications to move between displays by reparenting their top-level window. In addition, input devices may be gathered into a `SWinputDevice.T` to impose an order on events from multiple source and to route non-positional events.

`SW` supports the two interaction styles described in the introduction. The input mechanism can provide default handling for unfamiliar devices, such as looking up the owner, and the device and user registries allow users to maintain the ownership of devices as they move between displays.

14. Discussion

14.1. Introduction

The design of *Gemma* and SW grew out the experiences and frustrations of writing several multi-user applications. It defines a model for constructing user interface software for the ubiquitous approach to computing outlined by Weiser [Weiser91], of which some examples are described in Chapter 11. This chapter first shows how *Gemma* may be used to construct such applications, using MMM, DDD and one of the scenarios from Section 11.1 as examples. It then discusses two examples of issues which are unresolved in the current design but will have to be addressed in any implementation of *Gemma*, and the state of the computing infrastructure needed to support such an architecture.

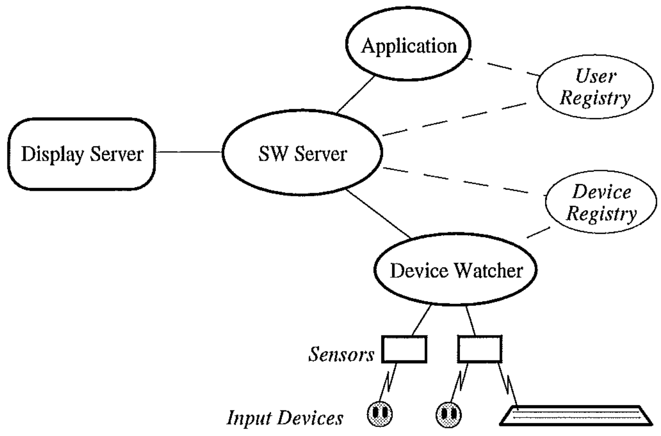
14.2. Applying SW and *Gemma*

How, then, might *Gemma* be used to actually build an application? This section shows how the systems which motivated *Gemma*, namely MMM and DDD, would benefit from it. MMM becomes more open and various components of the interface, such as user management, become eligible for distribution. DDD, on the other hand, acquires a clearer division between its logical and physical co-ordinate spaces so that applications can operate entirely in the logical space and be expected to handle arbitrary combinations of hardware. Finally, I describe how the design meeting example from Section 11.1 may be implemented. This shows, again, that a *Gemma* application can be written independently of a specific combination of hardware and that its behaviour can be changed to match external circumstances by connecting to a different set of virtual devices.

14.2.1. Implementing MMM

Consider, as an example, an environment to support MMM-like activity (Figure 14-6). Each person has a desk-based display and uses cordless input devices, such as mice and keyboards, which communicate via low-power infra-red (to avoid the signals being picked up at other desks). The local SW Server implements a top-level window which it attaches to the display; one of its sub-windows is driven by the Application. There is also a *Device Watcher* which manages the input device sensors and implements virtual devices for the physical devices it detects. This is similar to Lantz's Workstation Agent [Lantz87] except that we also provide support for distributed components and for multiple users sharing an interface.

Figure 14-6. Using SW and *Gemma* to support MMM-like activity. Remote connections are shown by a dashed line and infrared by a zigzag.



As a user moves an input device into the range of the sensors, its presence is noted by the Device Watcher. Like Active Badges, the input devices periodically broadcast a signature to announce their location—this also allows the Device Watcher to detect whether a device is not in use at the moment or has been moved away. The Device Watcher then contacts the Device Registry to get the details of the device and to claim ownership. If the device is still registered at another site, the Device Watcher at the previous site can be notified to clean up any local data structures. The Device Registry also records whether a device is owned by a particular user. If not, then the device may simply be presented on the display, as a cursor for a pointing device or an icon for a keyboard, to allow a user to acquire control of it. Otherwise, possibilities include:

- if this is the first device for this user on this `SW` Server, this action may count as a form of login. The `SW` Server may then query the User Registry to acquire details of the user and create a Home Area to represent him or her. The `SW` Server may start a dialogue to ask the user to further authenticate himself or herself.
- if a pointer is registered as the focus pointer of a virtual `SWInputDevice`, the Device Watcher at the previous site may be notified to send the current state of the `SWInputDevice` and to route its non-positional events to this `SW` Server.
- for other devices, the `SW` Server may create a representation of the device to be added to the user's state on the screen. The representation may depend on the role the user has registered for the device—a trackball, for example, may be used with the weak hand for scrolling.

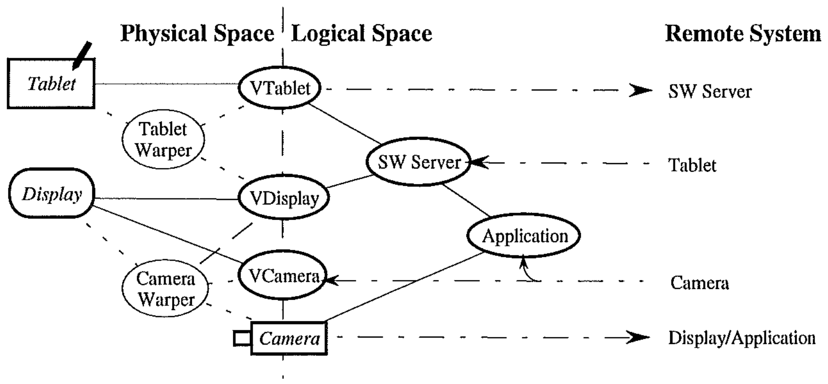
The Device Watcher feeds events from its devices to the `SW` Server which places them on its system input queue for dispatching to its associated applications. When a new device arrives, the `SW` Server creates a new entry in its devices table and attaches the device to its top-level `SW` window so that further events are associated with the Server's co-ordinate space.

This approach opens up the MMM architecture to make it considerably more flexible. For example, the focus window for non-positional events is associated with a virtual `SWInputDevice.T`, rather than the `UserInstance` record as in MMM, so users can maintained combinations of devices as they move around. Similarly, users can attach arbitrary devices to a particular display, maintaining ownership, state and preferences as they move between displays. This approach allows new device types to be connected, without interrupting the session, by starting a new Device Watcher and directing its output to the SW Server. Furthermore, the display can be shared with other systems; a video application, for example, could acquire a region of the display and draw directly to it without affecting the performance of the SW server. The addition of distribution also allows components to be implemented separately. For example, the representation of users within the SW Server could be managed by another process, a *user manager*, which determines access policies, maintains user/device associations and provides a user interface, such as the Home Areas. Different user managers could then be run to support different policies.

14.2.2. Implementing DDD

The DDD, on the other hand, is already distributed but benefits from the infrastructure which *Gemma* provides. Figure 14-7 shows the component parts of one end of a *Gemma* DDD. The DDD logical space is defined with respect to the camera's frame buffer, so mappings must be defined between this space and the input and output devices. The DDD includes a *Camera Warper* which provides a dialogue to calibrate the display with the camera and makes that calibration, or *warping*, available to interested clients. The most important local client is the virtual display, *VDisplay*, which translates display requests from the DDD's logical space to the physical display's co-ordinates; this translation is managed by the Camera Warper. Frames from the remote camera are translated to the local display by a virtual camera, *VCamera*, which paints directly into an SW Region on the display; this avoids unnecessary processing in the SW Server. The tablet has a virtual device, *VTablet*, to translate its input to the logical co-ordinate space; this translation is managed by a *Tablet Warper* which also provides a calibration dialogue.

Figure 14-7. The components for one half of a DDD. The "V" devices translate between the physical device co-ordinates and the common logical space in which the Application and SW Server operate. The logical space is defined with respect to the camera's frame buffer, so it has no local virtual device.



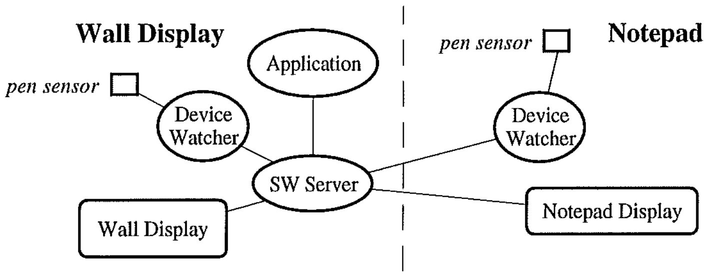
This approach allows the application and SW Server to run entirely in the DDD logical space; all the warping is managed by virtual devices so new hardware, and new *types* of hardware, can be introduced simply by connecting to different services.

14.2.3. The design meeting

The implementation of the design meeting described in Section 11.1 breaks down into two parts. Firstly, the use of the shared wall display is simply a case of the MMM approach in which a Device Watcher monitors the pens applied to the display. As with mice and keyboards above, the wall display Device Watcher detects different pens from their infra-red signatures and establishes a virtual input device for each one. The second part implements the communication with the notepad displays. Much of the time, notepad owners will be working independently of the main display, but occasionally they will want to transfer work between the notepad and the display and to use the notepad as an input device for the wall display.

One approach is to include both the wall display and the notepad in a common SW window—the notepad area is treated as if attached to an edge of the wall display area. Graphical objects may then be dragged from one display to the other, even though the two displays are physically separate; once an object has been moved to the main display it can then be picked up with another device. Figure 14-8 shows how input events from the notepad may be diverted to the common SW Server process which, in turn, drives the notepad display. All the pens are absolute pointing devices, so their input events can be mapped so that their range is limited to the area of the display with which they are associated. As with the DDD, the application is removed from the details of the hardware and sees only an oddly-shaped top-level window with input coming from multiple users.

Figure 14-8. The connections for combining a wall display and a notepad into an SW window. Notepad input events are forwarded to the SW server which, in turn, dispatches output requests to the two displays.



Another approach is to use simple cut and paste by associating the selection with the user, rather than the window server, and storing it with the User Registry; the selection may then be retrieved from the User Registry if the user pastes to another display. Thus, a user can make a selection on his or her notepad, to be stored with the User Registry, and paste to the wall display with any pen with which they are associated. This is related to a technique used in some meeting room systems, such as the Capture Lab [Mantei88], which allow users to maintain their copy buffer when switching between control of their personal machine and of a common display. A further step is to allow a notepad owner to switch between using it as a local device and as a pointer for the wall display. When used as a pointer, the notepad display replicates a part of the main display which can be panned around using a local dialogue area; the notepad owner can interact with the replicated area as if drawing on the wall display.

Figure 14-9. The connections for using a notepad as a pointer on a wall display. The Scroll Dialogue process allows the user to control which part of the wall display the notebook replicates.

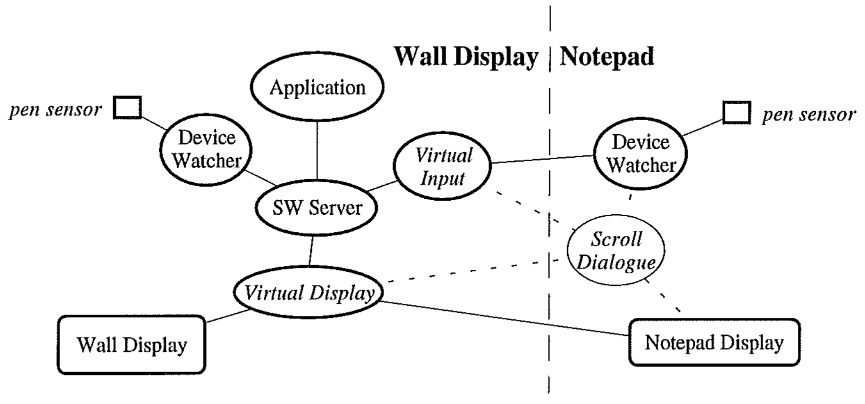


Figure 14-9 shows how this may be achieved. The SW Server draws to a *Virtual Display* device which replicates parts of the display image to its connected displays. The input from the notepad pen is sent to a *Virtual Input* process which maps its co-ordinates to that part of the virtual display which is visible on the notepad. The *Scroll Dialogue* process manages a dialogue box on the notepad which provides an interface to allow the user to move around the virtual display; as the user scrolls, the scroll dialogue notifies the virtual input and output devices of the change of view. If the user switches to a local application, the Scroll Dialogue notifies the virtual devices to suspend their connections with the notepad. Once again, the application sees only a display with input coming from a variety of sources.

14.3. Unresolved Issues

There are, of course, many unresolved issues in the design of *Gemma* which are beyond the scope of this dissertation, such as the sort of user interface that would allow users to manage numbers of virtual devices. Naming and persistence, however, provide interesting examples for further discussion:

14.3.1. Naming

The interface to a `Device.T` uses a naming scheme like ANSA [ANSA89], in which a name consists of a type description, a hierarchical context and a list of properties, to allow clients to find devices. The type description may be determined by the device class, but the context depends on the meaning a user associates with the device. For example, where *location* is important the context for a display might be:

```
/uk/cambridge/darwin-college/reading-room/by-window
```

with properties including

```
being-used-by: steve-freeman@computer-lab.cambridge.academic.uk
top-window-application: diary
display-type: pad-v17.2
```

Where the *role* the device is playing is important, the context and properties might be:

```
/uk/academic/cambridge/computer-lab/steve-freeman/diary
location: by-window.reading-room.darwin-college.cambridge
display-type: pad-v17.2
```

The physical device itself might also have a context:

```
/uk/academic/cambridge/computing-service/pad-v17.2
location: by-window.reading-room.darwin-college.cambridge
serial-no: 358483
allocated-to: steve-freeman@computer-lab.cambridge.academic.uk
```

Which name space is appropriate depends on the client's use of the device, so an application which concatenates several displays into a logical window will use the location context, a pager application will use the role context, and system administration will use the device context. The complexities of maintaining such interrelated name spaces and of finding one device from among a large number are obvious. Some subtle issues also arise if, for example, the process of looking up a service reveals information about the person using a device. The architectural consequences are being considered by researchers involved with Active Badges, such as Spreitzer and Theimer [Spreitzer93], and, as Harper points out [Harper92], the social organisation of the institution which administers the devices is also significant.

14.3.2. Persistence

Like MMM, sw embeds applications in windows, so a natural model for storing application state is a persistent object store, the question being which components to store. A user may wish to keep a "home" window tree which can be attached to the user's current device and retained when not being used; the user will expect the tree to include his or her interaction preferences within each window. If other users also work in this tree and set up preferences of their own then it is not clear how, or whether, the storage system should drop one user's state but not the other's. Maintaining per-user state for each application instance could lead to the storage of large volumes of data which may never be used again. One possible avenue is to store user preference data with the user; it is then up to the user to remove unnecessary records or acquire more space. This has the added benefit that more information about the user stays within her or his domain, but complicates the storage and restoration of application state. The approach might be combined with a convention for specifying default preferences for categories of window, as with X Resources; window-specific user preferences would then be required only for exceptional cases. Another avenue, possibly complementary, is to run a very slow garbage collection over the application objects to remove user preference data which has not been accessed for some time (weeks or months). These data may be viewed as expendable on the grounds that no user need remember the previous insertion point in small file, for example, after several weeks.

14.4. System Requirements

The *Gemma* places some severe demands on its computing infrastructure. It requires mobile output devices which can be accurately located, input devices which can be uniquely identified, low-latency communications, and large amounts of rapidly accessible storage. The system software must also provide mechanisms to dynamically bind this hardware together without imposing too great a computational cost; these include storage services for naming, user and device details and application data, efficient higher-level communication protocols, and real-time scheduling in the operating system.

Some of these facilities are available or becoming so, as described in Chapter 4; these include, in particular, ATM networks, micro-kernel operating systems and mobile computing. Other examples of infrastructure are Chameleon [Fitzmaurice93], which includes a prototype hand-held display that uses its position and attitude in 3D space as an input device, and Liveboard [Elrod92] which supports multiple pens on wall-sized display. Many of the required facilities, however, are still experimental or too expensive and so are unstable platforms on which to build higher-level systems. For example, there is at present no technology that could be used across a range of displays that would identify an individual stylus.

The existing environment which is closest to these requirements is that at Xerox PARC, described in [Weiser93]; it includes Liveboards, portable Tabs and Pads, and wireless networks. Their user interface software (apart from that for the Tabs), however, is based on an X toolkit which allows an X window to migrate between servers and so bears some resemblance to an orphan `SWWindow.T`; such an approach differs from `SW` by restricting a display to a single pointer and keyboard.

14.5. Summary

This chapter discusses some implications of *Gemma*, showing first how it facilitates the construction of applications such as MMM and DDD and how it may be used to support one of the scenarios described in Section 11.1. *Gemma* provides a flexible environment with a clean separation between the application and hardware environments, and even allows an application's behaviour to be modified by changing the virtual devices it is connected to. The next section discusses two examples of issues which must be resolved in order to implement *Gemma*: a suitable naming scheme, or set of naming schemes, for *Gemma* entities; and, how persistent storage for applications and user preferences is to be managed. The final section outlines some of the infrastructure needed to support the implementation of such a system and notes the experimental status of many of necessary technologies.

15. Conclusions

15.1. Introduction

This dissertation proposes a novel architecture, *Gemma*, for distributed user interfaces. The phrase "distributed user interface" can be understood in three ways, all of which are supported by *Gemma*:

- *distributed* user interface: this is the conventional meaning in which the user interface for a single display is implemented across several machines, examples include the X Window System and NeWS. *Gemma* can be used to gather a set of input and output devices together to implement a window server.
- *distributed user* interface: groupware applications and architectures tie together multiple users who may physically separated. *Gemma* allows applications to share devices at the most appropriate level of abstraction.
- distributed *user interface*: this is the sense implied by the Ubiquitous computing approach in which the user interface is not limited to a machine on a single desk, but is made up of many devices. Each device may represent a particular aspect of the collective interface.

Gemma allows applications to bind devices together for the task in hand, with as few assumptions as possible about the type of interaction to be supported.

15.2. An Architecture for Distributed User Interfaces

Gemma defines a model for interactive systems which exploits new developments in computing systems to support new user requirements. It is based on a rigorous distribution of basic user interface components which existing monoloithic systems have kept inaccessible to client applications. *Gemma's* open architecture allows people to use input and output devices as the need arises without having to restart applications, while its use of multiple interfaces supports fine-grained access control for any *Gemma* service. *Gemma* devices may be shared in the users' familiar physical world, so people may exploit their existing everyday skills rather than having to learn to survive in the computer's virtual world.

The sw system shows how basic *Gemma* devices may be combined to support the user interaction scenarios described in Section 11.1. It provides a clear separation between an application and the devices, or even the types of devices, that application is running on. Thus, a core behaviour can be defined in an application object, but its manifestation can be changed by connecting the application to different virtual devices. Device ownership and user state are managed by networked services and so can be maintained across applications, devices and sessions. The provision of such services at the sw level means that sw applications automatically support the sharing and mobility which users require.

The design of *Gemma* and sw was motivated and informed by three sets of experiments which showed, first, how existing window systems are fundamentally unsuited to support the next generation of user requirements. Second, the MMM showed that it is possible to break the assumption of the single-user at a display—that the personal computer is not the only model for user interfaces. Third, the DDD showed that it is feasible, and sometime necessary, to distribute the internal components of a user interface, rather than the user interface as a whole. The implementation of these unusual applications made clear the limitiations of existing user interface architectures and provided prototypes for aspects of the design of *Gemma*.

Finally, existing user interface architectures are udner pressure from two directions. One the one hand, people are attempting to build applications which stretch the limits of available architectures—motivated by both a more sophisticated understanding of users' needs and practices and by the assumption that more powerful computers can take on a large part of the effort of human-computer communication. One the other hand, computing systems now consist of collections of hardware bound together by fast networks and flexible operating systems. The general purpose operating system which implements all the necessary services, is being replaced by a micro-kernel which binds service-providers together. *Gemma* exploits these developments in computing systems to provide an infrastructure suitable for the next generation of user interface.

15.3. Further Work

Gemma is based on a model of user interaction in which many people use a large number of communicating user interaction devices. Much of the infrastructure which would make *Gemma* most useful, such as ATM networks and identifiable pens, is not yet widely available, but there are some smaller projects which could prove, or disprove aspects of the design, for example:

- a workstation-based implementation which included an X server would show whether *Gemma* could be made efficient enough to support conventional forms of user interaction. Building further window systems over the same platform would highlight problems of interoperability and arbitration between device clients.
- *Gemma* may be particularly suitable for use with low-power devices, such as mobile displays. For example, using multiple levels of virtual devices allows output processing to be distributed to optimise the use of computational power and network communication.
- the issue of maintaining per-user state could be investigated in a conventional object-oriented user interface system with persistence; users' interaction preferences and state could be stored for each interactive object they work with. The issues concern which data are to be stored by whom and for how long.

The greatest benefits of the *Gemma* approach, however, will accrue when most of the people in an organisation can operate within the same system, sharing devices and applications and taking advantage of its flexibility; the issue is similar to that of critical mass for CSCW applications [Markus90]. Thus, the *Gemma* design ultimately requires a large scale implementation in everyday use to be fully validated.

Bibliography

[Abdul-Wahab90] Abdul-Wahab, H.M., M. Feit. XTV: a framework for sharing X Window clients in remote synchronous collaboration. In *Proc. IEEE Conf. on Communication Software*, Chapel Hill, NC, 1990, pp. 159–167.

[Adobe85] Adobe Systems, Inc, *PostScript Language Reference Manual*, Addison-Wesley, Reading, MA, 1985.

[Ahuja90] Ahuja, S.R., J.R. Ensor, S.E.Lucco, A Comparison of Application Sharing Mechanisms in Real-time Desktop Conferencing Systems. In *Proc. ACM Conf. on Office Information Systems*, Cambridge, MA, April 1990, pp. 238–248.

[Altenhofen91] M. Altenhofen, B. Neidecker-Lutz and P. Tallet, P., Upgrading a window system for tutoring functions. In *Proceedings of the ARGOSI Workshop on Distributed Window Systems, Abingdon, UK,* EuroGraphics Technical Report (December, 1991).

[Angebranndt88] Angebranndt, S., R. Drewry, P. Karlton, T. Newman, B. Scheifler, K. Packard. *Definition of the Porting Layer for the X v11 Sample Server*, X Consortium, (March 1988).

[Angebranndt91] Angebranndt, S., R. L. Hyde, D. H. Luong, N.Siravara, C. Schmandt. Integrating Audio and Telephony in a Distributed Workstation Environment, in *Proceedings of USENIX* (Summer 1991), pp 419–435.

[Appino92] Appino, Perry A., J. Bryan Lewis, Lawrence Koved, Daniel T. Ling, David A. Rebenhorst and Christopher F. Codella, An Architecture for Virtual Worlds, *Presence* 1(1) (Winter 1992) pp.1–17.

[Apple91] Apple Computer, Inc., *Inside Macintosh, Volume VI*, Addison-Wesley, Reading, MA, 1991.

[ANSA89] ANSA, ANSA Reference Manual, APM Ltd, Cambridge, UK., 1989.

[Armand91] Armand, François, *Give a Process To Your Drivers*, Technical Report CS/TR-91-97, Chorus Systèmes (1991).

[Arons92] Arons, B. Tools for Building Asynchronous Servers to Support Speeach and Audio Applications. In *Proc. ACM Symp. on User Interface Software and Technology*, Monterey, CA (November, 1992), pp. 71–78.

[Bannon91] Bannon, L.J., K. Schmidt, CSCW: Four characters in search of a context. In Bowers, J.M., S.D. Benford (eds), *Computer Supported Cooperative Work*, Elsevier, Amsterdam, 1991, pp. 3–16.

[Beaudoin92] Beaudoin-Lafon, M., A. Karsenty, Transparency and Awareness in a Real-Time Groupware System, In *Proc. ACM Symp. on User Interface Software and Technology*, Monterey, CA, November 1992, pp. 171–180.

[Beck93] Beck, E., V. Bellotti. Informed opportunism as strategy: supporting collaboration in distributed collaborative writing. To appear in *Proc. European Conference on Computer Supported Cooperative Work*, Milano, September 1993.

[Bellcore93] Bellcore Information Networking Research Laboratory, The Touring Machine System, *Communications of the ACM*, 36(1), January 1993, pp. 68–77.

[Bentley92] Bentley, R., T. Rodden, P. Sawyer, I. Sommerville. An architecture for tailoring cooperative multi-user displays. in *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, September 1992, pp. 123–129.

[Bier91] Eric A. Bier and Steve Freeman, MMM: A User Interface Architecture for Shared Editors on a Single Screen, in *Proc. ACM Symp. on User Interface Software and Technology*, Hilton Head, NC (November, 1991) pp.79-86.

[Bier91a] Eric A. Bier, Embedded Buttons: Documents as User Interfaces, in *Proc. ACM Symp. on User Interface Software and Technology*, Hilton Head, NC (November, 1991) pp.45-54.

[Birman91] Birman, K., Cooper, R., Gleeson, B. *Programming with process groups: group and multicast semantics*. Technical Report, Dept. of Computer Science, University of Cornell, (January 1991).

[Birrell84] Birrell, A.D., B.J.Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, 2(1), February 1984, pp. 39–59.

[Birrell93] Birell, A., G. Nelson, S. Owicki, T. Wobber. Network objects. Submitted to *Symposium on Operating System Principles* (1993).

[Bly90] Bly, S.A., S.L. Minneman, Commune: a shared drawing surface. In *Proc. ACM Conference on Office Information Systems*, Cambridge, MA, 1990, pp. 184–192.

[Bly93] Bly, S.A., S.R. Harrison, S. Irwin, Media Spaces: bringing people together in a video, audio and computing environment. In *Communications of the ACM*, 36(1), January 1993, pp. 28–45.

[Bobrow90] Bobrow, D.G., M. Stefik, G. Foster, F. Halasz, S. Lanning, D. Tatar. *The Colab Project Final Report*, Xerox PARC Technical Report SSL-90-45, (1990).

[Brown90] Brown, Ed, W.A.S. Buxton and K. Murtagh. Windows on Tablets as a Means of Achieving Virtual Input Devices. In Diaper, D., et al (eds) *Human-Computer Interaction—INTERACT '90*, Elsevier, Amsterdam, 1990, pp.675–681.

[Brown91] Brown, David J. *Abstraction of image and pixel—The Thistle display system*, Technical Report 229, University of Cambridge Computer Laboratory, August 1991.

[Buxton86] Buxton, W., Brad A. Myers. A study in two-handed input. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Boston, MA, (April, 1986), pp. 321–326.

[Campbell92] Campbell, Roy H., Nayeem Islam and Peter Madany, Choices, Frameworks and Refinement, *Computing Systems*, 5(3), Summer 1992, pp.217–1255.

[Card87] Card, S.K., A. Henderson Jr., A Multiple, Virtual-Workspace Interface to Support User Task Switching. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, 1987, pp. 53–59.

[Carr91] Carr, Robert, Dan Shafer, *The Power of PenPoint*, Addison-Wesley, Reading, MA, 1991.

[Cheriton84] Cheriton, David R., The V Kernel: A software base for distributed systems, *IEEE Software* 1(2) (April 1984) pp.19-42.

[Clark90] Clark, D.D., D.L. Tennenhouse. Architectural Considerations for a New Generation of Protocols, in ACM *Computer Communications Review*, 20(4), September, 1990, pp. 200–208.

[Clark92] Clark, Sean, Steve Scrivener, *The ROCOCO sketchpad distributed shared drawing surface.*, LUTCHI report No. 92/C/LUTCHI/0150, LUTCHI Research Centre, Loughborough University of Technology, 1992.

[Codella92] Codella, C., Reza Jalili, Lawrence Koved, J. Bryan Lewis, Daniel T. Ling, James P. Lipscomb, David A. Rabenhorst, Chu P. Wang, Alan Norton, Paula Sweeney and Greg Turk, Interactive simulation in a multi-person virtual world, in *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, (May, 1992), pp.329–334.

[Cowan91] Cowan, W.B., Christopher Wein, Marceli Wein and Kellog S. Booth, Hardware Support for Multitasking Graphics,, pp.199–206 in *Graphics Interface* (1991).

[Cox92] Cox, D.C., Wireless Network Access for Personal Communications, *IEEE Communications Magazine*, December 1992, pp. 96–102.

[Crowley90] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., Tomlinson, R. MMConf: an infrastructure for building shared multimedia applications. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp. 329–342.

[deMey93] de Mey, V., S. Gibbs, A Multimedia Component Kit. In *Proc. ACM Conference on Multimedia*, Anaheim, CA, August 1993, pp. 291–300

[Dixon92] Dixon, M.J., *System Support for Multi-Service Traffic*, Technical Report 245, Computer Laboratory, University of Cambridge, January 1992.

[Dourish92] Dourish, P., S.A. Bly. Portholes: supporting awareness in a dsitributed work group. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Monterey, CA, April 1992, pp. 541–547.

[Dourish] Dourish, P. “Dissecting Shdr: Anatomy of a collborative writing tool” in Dourish., P. (ed). *Implementation Perspectives on CSCW Design*, Springer-Verlag, in preparation.

[Droms90] Droms, Ralph, Wayne R. Dyksen, Performance Measurements of the X Window System Communication Protocol, *Software—Practice and Experience* 20(S2) (October 1990) pp.S2/119–S2/1136.

[Duce90] Duce, D.A., R. van Liere, P.J.W. ten Hagen, An Approach to Hierarchical Input Devices, in *Computer Graphics Forum*, 9(1) (January 1990) pp. 15–26.

[Economist93] Retail Technology: Remote Control, *The Economist*, 29 May 1993, pp. 110–111.

[Ellis91] Ellis, C.A., S.J. Gibbs, G.L.Rein. Groupware: some issues and experiences. in *Communications of the ACM*, 34(1), January 1991, pp. 38–58.

[Elrod92] Elrod, S., R. Bruce, D. Goldberg, F. Halasz, W. Janssen, D. Lee, K. McCall, E. Pedersen, K. Pier, J. Tang, B. Welch, Liveboard: A large interactive display supporting group meetings, presentations and remote collaboration. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI),* Monterey CA, 1992, pp. 599–608.

[Engelbart88] Engelbart, D., The Augmented Knowledge Workshop,, pp.187–232 in *A History of Personal Workstations*, ed. Adele Goldberg, ACM, New York, NY (1988).

[Fahlén93] Fahlén, L.E., O. Ståhl, C.G. Brown, C. Carlsson. A space-based model for user interaction in shared synthetic environments. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Amsterdam, April 1993, pp. 43–48.

[Feiner93] Feiner, S., B. MacIntyre, D. Seligmann, Knowledge-Based Augmented Reality. In *Communications of the ACM*, 36(7), July 1993, pp. 52–63.

[Fitzmaurice93] Fitzmaurice, G., Situated Information Spaces and Spatially Aware Palmtop Computers, *Communications of the ACM*, 36(7), July 1993, pp. 38–49.

[Foley90] Foley, J.D., Andries van DAm, Steven K. Feiner, John F. Hughes, *Computer graphics: principles and practice*, Addison-Wesley, Reading, MA, 1990.

[Gajewska90] Gajewska, Hania, Mark S. Manasse and Joel MCormack, Why X Is Not Our Ideal Window System, *Software—Practice and Experience* 20(S2) (October 1990) pp.S2/137–S2/171.

[Gaver92] Gaver, W.W. The affordances of media spaces for collaboration. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, September, 1992, pp 17–24.

[Gaver92a] Gaver, W.W., T. Moran, A. MacLean, L. Lovstrand, P. Dourish, K.A. Carter, W. Buxton. Realizing a Video Environment: EuroPARC's RAVE System. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Monterey, CA, April 1992, pp.27–36.

[Gettys90] Gettys, James, Philip L. Karlton and Scott McGregor, The X Window System, Version 11, *Software—Practice and Experience* 20(S2) (October 1990) pp.S2/35–S2/67.

[Gibbs88] Gibbs, S.J. LIZA: An Extensible Groupware Toolkit. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, April 1989, pp. 29–35.

[Giokas92] Giokas, D.G, Andrew T. Leskowitz. eXcursion for Windows: Integrating Two Windowing Systems. *Digital Technical Journal*, 4(1) (Winter 1992), pp. 56–67.

[Gosling89] Gosling, James, David S. H. Rosenthal and Michelle J. Arden, *The NeWS Book: an introduction to the Network Extensible Window System*, Springer-Verlag, New York (1989).

[Greenberg91] Greenberg, S. Personalizable Groupware: Accomodating Individual Roles and Group Differences. In *Proc. European Conference on Computer Supported Cooperative Work*, Amsterdam 1991, pp. 17–31.

[Greenberg92] Greenberg, S., M. Roseman, D. Webster, R. Bohnet. Human and technical factors of distributed group drawing tools, *Interacting With Computers*, 4(3), 1992, pp. 364–394.

[Gust88] Gust, P., Shared X: X in a distributed group working environment. In *Proc. 2nd Annual X Conference*, Boston, MA, 1988.

[Harper92] Harper, R., Looking at Ourselves: an Examination of the Social Organisation of Two Research Laboratories. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, November 1992, pp. 330–337.

[Hayter91] Hayter, M., The Desk-Area Network, *ACM Operating Systems Review*, 25, October 1991, pp. 14–21.

[Hill92] Hill, R.D.The Abstraction–Link–View paradigm: using constraints to connect user interfaces to applications. In *Proc. Conference on Human Factors in Comp. Sys. (CHI)*, Monterey, CA, 1992, pp. 335-342.

[Hill93] Hill, R.D., Brink, T., Patterson, J.F., Rohall, S.L., Wilner, W.T., The Rendezvous language and architecture. *Communications of the ACM*, 36(1) (January 1991), pp. 62-67.

[Hodges89] Hodges, M, R. Sasnett, M. Ackerman, A Construction Set for Multimedia Applications, IEEE Software, 16(4), January 1989, pp. 37–43.

[Ioannidis93] Ioannidis, J., G.Q.Maguire Jr., The Design and Implementation of a Mobile Internetworking Architecture, *USENIX*, San Diego, CA, January 1993, pp. 491–502.

[Ishii90] Ishii, H. TeamWorkStation: Towards a Seamless Shared Workspace. *Proc. ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, September 1990, pp. 13–26.

[Ishii92] Ishii, H., M. Kobayashi, J. Grudin, Integration of Inter-Presonal Space and Shared Workspace: CleadBoard Design and Experiments. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, September 1992, pp. 33–42.

[Jeffay92] Jeffay, K., J.K. Lin, J. Menges, F.D. Smith, J.B. Smith. Architecture of the Arifact-Based Collaboration system Matrix. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, September 1991, pp. 195–202.

[Jones90] Jones, W., P.Williams, G. Robertson, V. Joloboff, M. Conner, In Search of the Ideal Operating System for User Interfacing. In *Proc. ACM Symp. on User Interface Software and Technology*, Snowbird, UT, October 1990, pp. 31–35.

[Karsenty93] Karsenty, A., M. Beaudoin-Lafon, An Algorithm for Distributed Groupware Applications. In *Proc. International Conference on Distributed Computing Systems*, Pittsburg, PA. May 1993.

[Karshmer91] Karshmer, A., J. Nehmer (eds), *Operating Systems of the 90s and Beyond*, Springer-Verlag, Berlin (1991).

[Kay88] Kay, A., Adele Goldberg, Personal Dynamic Media. In *A History of Personal Workstations*, ed. Adele Goldberg, ACM, New York, NY, 1988, pp.254–263.

[King92] King, T. Pandora: An Experiment in Distributed Multimedia. *EuroGraphics*, 11(3), 1992, pp. 23–34.

[Kleinrock92] Kleinrock, L., The Latency/Bandwidth Tradeoff in Gigabit Networks, *IEEE Communications Magazine*, April 1992, pp.36–40.

[Kraut88] Kraut, R., C. Egido, J. Galegher. Patterns of contact and communcation in scientific research collaboration. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Portland, OR, 1988, pp. 1–12.

[Lampson88] Lampson, B.W., Personal Distributed Computing: The Alto and Ethernet Software,, pp.293–335 in *A History of Personal Workstations*, ed. Adele Goldberg, ACM, New York, NY (1988).

[Lantz84] Lantz, Keith A., Willliam I.Nowicki, Structured Graphics for Distributed Systems, *ACM Transactions on Graphics*, 3(1) (January 1984) pp.23-51.

[Lantz87] Lantz, Keith A. Multi-process Structuring of User Interface Software, *ACM Computer Graphics*, 21(2), April 1987, pp.124–130.

[Lantz87a] Lantz, Keith A., Peter P. Tanner, Carl Binding, Kuan-Tsae Huang, and Andrew Dwelly. Reference Models, Window Systems, and Concurrency, *ACM Computer Graphics*, 21(2) April 1987, pp.87–97.

[Lauwers90] Lauwers, J.C. *Collaboration Transparency in Desktop Teleconferencing Environments*, Technical Report CSL-TR-90-435, Computer Systems Laboratory, Stanford University (July 1990).

[Lauwers90a] Lauwers, J.C., K.A.Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Seattle, WA, (April 1990), pp. 303–311.

[Leslie83] Leslie, I.M., *Extending the Local Area Network*, Technical Report 43, Computer Laboratory, University of Cambridge, February 1983.

[Leslie93] Leslie, I.M., D.R.McAuley, D.L. Tennenhouse, ATM Everywhere?. In *IEEE Network*, 7(2), March 1993, pp. 40–46.

[Liedtke91] Liedtke, J, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, G. Szalay, Two Years of Exerience with a µ-Kernel-Based OS, *Operating Systems Review*, January 1991.

[Liedtke92] Liedtke, J., Fast Thread Management and Communication Without Continuations. In *Proc. 1st USENIX Workshop on Microkernel and Other Kernel Architectures*, Seattle, WA, 1992.

[Linton89] Linton, M. A., J. M. Vlissides and P. R. Calder. Composing user interfaces with InterViews, *IEEE Computer* 22(2) (February 1989) pp.65–84.

[Lövstrand91] Lövstrand, L, Being Selectively Aware with the Khronika System. In *Proc. European Conference on Computer Supported Cooperative Work*, Amsterdam, September 1991, pp. 265–277.

[Lu91] Lu, I.M., M.M. Mantei, Idea management in a shared drawing tool. In *Proc. European Conference on Computer Supported Cooperative Work*, Amsterdam, 1991, pp. 97-112.

[Luff92] Luff, P., C. Heath, D. Greatbatch, Tasks-In-Interaction: Paper and Screen Based Documentation in Collborative Activity. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, September 1992, pp. 163–170.

[MacAuley90] MacAuley, Derek R, *Protocol Design for High Speed Networks*, Technical Report 186, University of Cambridge Computer Laboratory, Cambridge, UK (January, 1990).

[Mackay90] Mackay, W.E., Patterns of sharing customizable software. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, September 1990, pp. 209-221.

[Manasse88] Manasse, M., G. Nelson, *A Performance Analysis of a Multiprocessor Wndow System*, Unpublished MS, 1988.

[Manasse91] Manasse, M., G. Nelson, *Trestle Reference Manual.* Research Report 68, Digital Systems Research Center, 1991.

[Mantei88] Mantei, M. Capturing the Capture Lab concepts: a case study in the design of computer supported meeting environments, *Proc. ACM Conf. on Computer Supported Cooperative Work*, Portland, OR, USA. September 1988, pp. 257-270.

[Mapp92] Mapp, G. E., *An Object-Oriented Approach to Virtual Memory Management*, Technical Report 242, Cambridge University Computer Laboratory (January 1992).

[Markus90] Markus, M.L., T. Connoly, Why CSCW applications fail: problems in the adoption of interdependent work tools. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, 1990, pp 371-380.

[McCormack88] McCormack, J., Paul Asente, An Overview of the X Toolkit. In *Proc. ACM Symp. on User Interface Software and Technology*, October 1988, pp.46–55.

[Moran81] Moran, T. An Applied Psychology of the User. In *ACM Computing Surveys*, 13(1), March 1981, pp. 1–12.

[Nardi90] Nardi, B.A., J.R. Miller, An ethnographic study of distributed problem solving in spreadsheet development, in *Proc. ACM Conf. on Computer Supported Cooperative Work*, Los Angeles, CA, September 1990, pp 197-208.

[Navarro92] Navarro, L., T. Rodden, W. Prinz. Open CSCW and ODP. In *Proc. IEEE Workshop on Future Trends in Distributed Systems*, Taipai, Taiwan, IEEE Press, 1992.

[Needham91] Needham, R.M., What Next? Some Speculations, pp.220–222 in *Operating Systems of the 90s and Beyond*, ed. A. Karshmer and J. Nehmer, Springer-Verlag, Berlin (1991).

[Nelson91] Nelson, Greg, *Systems Programming with Modula-3*, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Newman92] Newman, W., P. Wellner, A Desk Supporting Computer-based Interaction with Paper Documents. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Monterey, CA, May 1992, pp. 587–592.

[Nicolaou91] Nicolaou, C. A Distributed Architecture for Multimedia Communication Systems, Technical Report 220, University of Cambridge Computer Laboratory, May 1991

[Nielsen93] Nielsen, J., Non-Command User Interfaces. In *Communications of the ACM*, 36(4), April 1993, pp. 83–99.

[Nigay93] Nigay, L., J. Coutaz, A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Amsterdam, April 1993, pp. 172–178.

[Ohmori92] Ohmori, T., K. Maeno, S. Sakata, H. Fukuoka and K. Watabe, Distribute Cooperative Control for Sharing Applications Based on Multipartyand Multimedia Desktop Conferencing System: Mermaid. In *Proceedings of the IEEE 12th International Conference on Distributed Computing Systems, Yokahama*, 1992, pp.538–546.

[Olson91] Olson, J.S., Olson, G.M., Mack, L., Wellner, P. Concurrent editing: the group's interface. In *Proc.Human-Computer Interaction (Interact)*, Cambridge, UK, September 1991, pp. 835-840.

[Ousterhout89] Ousterhout, J. *Why aren't operating systems getting faster as fast as hardware?*, Technical Note TN-11, Digital WRL October, 1989.

[Pagani93] Pagani, D., W.E. Mackay, Bringing media spaces into the real world. To appear in *Proc. European Conference on Computer Supported Cooperative Work*, Milano, September 1993.

[Pascale92] Pascale, R., Jeremy Epstein, Virtual Window Systems: A New Approach to Supporting Concurrent Heterogeneous Windowing Systems, in *Usenix* (1992).

[Patterson91] Patterson, J.F. Comparing the programming demands of single-user and multi-user applications, In *Proc. ACM Symp. on User Interface Software and Technology*, Hilton Head, NC, November 1991, pp. 87–94.

[Patrick89] Patrick, M., G. Sachs, *X11 Input Extension Protocol Specification*, X Consortium, Cambridge, MA (1989).

[Pedersen93] Pedersen, E.R., McCall, K., Moran, T.P., Halasz, F.G. Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings, in *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Amsterdam, 1993, pp. 391–405.

[Perlin93] Perlin, K., D. Fox, Pad: An alternative approach to the computer interface. In *Proc. ACM Computer Graphics (SIGGRAPH)*, August 1993, Anaheim, CA, pp. 57–64.

[Pettersson89] Pettersson, E., Automatic Information Processing in Document Reading: A Study of Information Handling in Two Intesive Care Units. In *Proc. European Conference on Computer Supported Cooperative Work*, Gatwick, UK, September 1989, pp. 63–72.

[Pike83] Pike, Rob, Graphics in overlapping bitmap layers, *ACM Transactions on Graphics* 2(2) (July, 1983) pp.135-160.

[Rao90] Rao, R. *Implementational Reflection in Silica*, Technical Report SSL-90-63, Xerox PARC, 1990.

[Rao92] Rao, R., S.K. Card, H.D. Jellinek, J.D. Mackinlay, G.G. Robertson, The Information Grid: a framework for Information Retrieval and Retrieval-Centered Applications. In *Proc. ACM Symp. on User Interface Software and Technology*, Monterey, CA, November 1992, pp. 23–32.

[Rashid93] Rashid, R., Seminar at Computer Laboratory, University of Cambridge, 1993.

[Robertson89] Robertson, G. G., Stuart K. Card and Jock D. Mackinlay, The Cognitivie Coprocessor Architecture for Interactive User Interfaces, pp.10–18 in *Proc. ACM Symp. on User Interface Software and Technology*(November 1989).

[Roseman92] Roseman, M., S. Greenberg, GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications, *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, 1992, pp. 43–50.

[Rosner92] Rosner, P., M. Slater, A. Davidson, A Generalised Event Mechanism for Interactive Systems. In Gray, P, R. Took (eds), *Building Interactive Systems*, Springer-Verlag, London, 1992.

[Sarin88] Sarin, S., I. Greif. Computer-based real-time conferencing systems. In Greif, I. (ed), *Computer Supported Cooperative Work*, Morgan Kaufman, 1988, pp. 397–422.

[Scheifler86] Scheifler, Robert W., Jim Gettys, The X Window System, *ACM Transactions on Graphics* 5(2) (1986) pp.79-109.

[Schmidtmann93] Schmidtmann, C., Michael Tao, Steven Watt, *Design and Implementation of a Multi-Threaded Xlib*. In Proc. USENIX, San Diego, CA, January 1993, pp. 193–204.

[Sheng92] Sheng, S. A. Chandrakasan, R.W. Brodersen, A Portable Multimedia Terminal, *IEEE Communications Magazine*, December 1992, pp. 64–70.

[Smith87] Smith, R.B., Experiences with the Alternative Reality Kit. An Example of the Tension Between Literalism and Magic, in *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Toronto, April, 1987, pp. 61–67.

[Spreitzer93] Spreitzer, M., M. Theimer, Scalable, Secure, Mobile Computing with Location Information, in *Communications of the ACM*, 36(7), July 1993, p. 27.

[Sreenan93] Sreenan, C.J., *Synchronisation Services for Digital Continuous Media*, Technical Report 292, Computer Laboratory, University of Cambridge, March 1993.

[Stefik87] Stefik, M., G. Foster, D.G. Bobrow, K. Kahn, S. Lanning, L. Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1), january 1987, pp. 32–47.

[Stults86] Stults, R. *Media Space*, Xerox PARC Technical report, 1986.

[Stults89] Stults, R., *Experimental Uses of Video to Support Design Activities*, Xerox PARC Technical Report, SSL-89-19, 1989.

[Sun90] Sun Microsystems. *NeWS 2.1 Programmer's Guide*, Sun Microsystems, Mountain View, CA (1990).

[Sun90a] Sun Microsystems. *4.1 Pixrect Reference Manual*, Sun Microsystems, Mountain View, CA (1990).

[Swinehart86] Swinehart, Daniel C. , Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment, *ACM Transactions on Programming Languages and Systems*, 8(4), 1986, pp. 419-490. Also available as Xerox PARC Technical Report CSL-86-1

[Swinehart92] Daniel C. Swinehart, Personal Communication.

[Tang90] Tang, J.C., S.L. Minneman, VideoDraw: a Video Interface for Collborative Drawing. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Seattle, WA, April 1990, pp. 313–320.

[Tani92] Tani, M., K. Yamaashi, K. Tanikoshi, M. Futukawa, S. Tanifuji, Object-Oriented Video: Interaction with Real-World Objects Through Live Video. In *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, Monterey, CA, 1992, pp. 593–598.

[Tanner86] Tanner, Peter P., Stephen A. MacKay, Darlene A. Stewart and Marceli Wein, A Multitasking Switchboard Approach to User Interface Management, *ACM Computer Graphics* 20(4) (August, 1986) pp.241–248.

[Tatar91] Tatar, D.G., Foster, G., Bobrow, D.G. Design for conversation: lessons from the Cognoter. In *Int. J. of Man-Machine Studies*, Vol **34**, N **2**, February 1991, pp. 185-210.

[Teitelman84] Teitelman, Warren, *The Cedar Programming Environment: A Midterm Report and Examination*, Report CSL-83-11, Xerox PARC, Palo Alto, CA (June 1984).

[Tennenhouse90] Tennenhouse, D.L., The *ViewStation* Research Program on Distributed Video Systems. In *Proc. Workshop on Network and Operating Systems Support for Digital Audio and Video*, Technical Report TR-90-062, International Computer Science Institute, Berkeley, CA, November 1990.

[Tennenhouse92] Tennenhouse, D.L., *Telemedia, Networks and Systems Group, Annual Report*, Internal Report, Laboratory for Computer Science, MIT, August 1992.

[Thacker79] Thacker, S. P., E. M. McCreight, B. W. Lampson, R. F. Sproull and D. R. Boggs, *Alto: A Personal Computer.*, Xerox Palo Alto Research Center, Palo Alto, CA (1979).

[Thekkath93] Thekkath, C.A., H.M.Levy, Low-Latency Communication on High-Speed Networks, *ACM Transactions on Computer Systems*, 11(2), May 1993, pp 179–203.

[Took90] Took, R., Surface Interaction: A Paradigm and Model for Separating Application and Interface, in *Proc. ACM Conf. on Human Factors in Computing Systems (CHI)*, April, 1990, pp.35–42.

[Took91] Took, R., The Active Medium: A Conceptual and Practical Architecture for Direct Manipulation,, pp.249–264 in *People and Computers VI*, ed. D. Diaper and N. Hammond, Cambridge University Press, Cambridge (1991).

[Unison90] UNISON, *The Final Report of the UNISON Project*, Loughborough University of Technology, UK, 1990.

[Voorhies88] Voorhies, Douglas, David Kirk and Olin Lanthrop, Virtual Graphics, *Computer Graphics* 22(4), August 1988, pp.247–253.

[Vin91] Vin, H.M., P.T. Zellweger, D.C. Swinehart, P.V. Rangan, Multimedia Conferencing in the Etherphone Environment, *IEEE Computer*, 24(10), October 1991, pp. 69–79.

[Want92] Want, R., A. Hopper, V. Falcão, J. Gibbons, The Active Badge Location System, ACM Transactions on Information Systems, 10(1), January 1992, pp. 91–102.

[Weiser91] Weiser, M., The computer for the 21st century, *Scientific American*, 265(3), September 1991, pp 66–75.

[Weiser93] Weiser, M. Some Computer Science Issues in Ubiquitous Computing. In *Communications of the ACM*, 36(7), July 1993, pp. 75–84.

[Wellner93] Wellner, P, Interacting with Paper on the DigitalDesk, *Communications of the ACM*, 37(7), July 1993, pp. 86–95.

[Widener90] Widener, Glenn, The X11 Inter-Client Communication Conventions Manual, *Software—Practice and Experience* 20(S2) (October 1990) pp.S2/109–S2/118.

[Wittie91] Wittie, L.D., Computer Networks and Distributed Systems, in *IEEE Computer*, 24(9), September 1991, 67–76.

[Wolf92] Wolf, C.G., J.R.Rhyne, Communication and Information Retrieval with a Pen-based Meeting Support Tool. In *Proc. ACM Conf. on Computer Supported Cooperative Work*, Toronto, October 1992, pp. 322–329.