Surface Interaction: Separating Direct Manipulation Interfaces from their Applications

Roger Kenton Took

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

July 1990

Table of Contents

Acknowledgements **1**

Declaration **2**

Abstract **3**

1. Introduction **4**

1.1. The Needs of the User 4
1.1.1. Performance 5
1.2. The Needs of the Interface Designer 6
1.2.1. Power 7
1.2.2. Freedom 7
1.3. Separation 8
1.3.1. Abstraction 8
1.3.2. Binding 10
1.4. Existing User Interface Systems 12
1.5. Premises and Issues 13
1.5.1. Presentation 14
1.5.2. Terms: Object and Application 16
1.5.3. Formal Design 16
1.6. Overview of the Thesis 17
1.6.1. The Thesis 17
1.6.2. Structure of the Thesis 19

2. Architectures for Separation **20**

2.1. Separation 20
2.1.1. Motivation for Separation 21
2.1.2. Flow of Control 25
2.2. Input Frameworks 27
2.2.1. Input Types and Modes 27
2.2.2. Input Routing 28
2.3. Interaction and Semantics 35
2.3.1. Feedback 35

2.3.2. Directness 37
2.3.3. Semantic Perspectives 38
2.4. Linguistic Architectures 38
2.4.1. Dialogue Abstraction 39
2.4.2. Transition Networks 39
2.4.3. Grammars 41
2.4.4. Problems of Dialogue Abstraction 43
2.5. Agent Architectures 48
2.6. Refinements of the Agent Architecture 51
2.6.1. User Interface Toolkits 51
2.6.2. Device Abstraction 52
2.6.3. Homogeneity 52
2.6.4. Logical Devices 53
2.6.5. Object-Orientation 55
2.6.6. The Model-View Paradigm 59
2.6.7. Problems of Model-View Separation 66
2.6.8. Separation Problems in Agent Architectures 67
2.7. Conclusions 68

3. A Formal Perspective on Dialogue Separation 70

3.1. Interaction 70
3.1.1. State 71
3.1.2. Functionality 72
3.1.3. Object 74
3.1.4. Range 74
3.1.5. Behaviour 75
3.1.6. Dialogue 76
3.2. Relation between Functionality and Behaviour 78
3.3. Taking account of the user 79
3.3.1. Implementing Trace Constraints Separately 82
3.4. Limitations on Separation 83
3.4.1. Classes of Dialogue Separation 84
3.5. Input and Output 85
3.5.1. Communication 90
3.6. Conclusions 93

4. Surface Interaction 96

4.1. Abstract Models of Interaction 97
4.2. The Medium 99
4.2.1. Model 100
4.2.2. Presentation 100
4.2.3. Abstracting the Medium 101
4.2.4. Separating the Medium 103
4.2.5. Directness in the Medium 105
4.2.6. Consistency 106
4.2.7. The Medium: Summary 107
4.3. Surface Interaction 107
4.3.1. Premise 107
4.3.2. The Surface 108
4.3.3. Refinements 109
4.4. The UMA Architecture 111

4.4.1. The Medium 113
4.4.2. The User Agent 114
4.4.3. The Application 117
4.4.4. The Surface 118
4.4.5. An Observation of Surface Interaction 119
4.5. A Simple Surface 121
4.5.1. The Concrete Medium 122
4.5.2. The Concrete Application 123
4.5.3. The Concrete User Agent 124
4.5.4. The Communication Structure 127
4.6. Implementation Issues 128
4.6.1. Performance 128
4.6.2. Timing 129
4.6.3. Binding User Agent and Medium 131
4.6.4. Multiple Applications 131
4.6.5. Fairness 132
4.6.6. Object Structures 132
4.6.7. Picking 133
4.6.8. Stylistic Binding 133
4.6.9. Buffering 135
4.6.10. Channels 136
4.6.11. Synchronisation 137
4.6.12. Echoing 138
4.6.13. Pruning State 139
4.6.14. Error Handling 140
4.6.15. Logical Events 141
4.7. Conclusions 141

5. Surface Models 143

5.0.1. Procedural and Declarative Models 143
5.0.2. Marks and Media 144
5.1. Window Management 145
5.1.1. The Model 147
5.1.2. Features: Icons and Menus 152
5.1.3. The Window Interface 153
5.1.4. Window System Architecture 154
5.2. Graphics 158
5.2.1. Imaging 158
5.2.2. Modelling 164
5.2.3. Procedural Modelling 165
5.2.4. Declarative Modelling 166
5.2.5. Structure 171
5.2.6. Viewing 177
5.3. Text 182
5.3.1. Content 182
5.3.2. Logical Structure 183
5.3.3. Properties 184
5.3.4. Editing 184
5.4. Documents 186
5.4.1. Formatting 187
5.4.2. Layout 188

5.4.3. Integrating Format and Layout 190
5.5. Conclusions 194

6. A Formal Model for the Surface Medium 196

6.1. Introduction 196
6.2. The Presenter Model 199
6.2.1. The Specification 199
6.3. Objects, Structures, and Properties 200
6.3.1. Fundamental Objects: REGIONs 200
6.3.2. Fundamental Representation 200
6.3.3. Fundamental Structure: Ordered Tree 201
6.3.4. Basic Relations 204
6.3.5. Fundamental Properties 204
6.3.6. Content 206
6.3.7. Geometric Properties 208
6.3.8. Visualisation 211
6.3.9. Behaviour 213
6.3.10. The Core Model 214
6.4. Surface Presentation 214
6.4.1. Projecting the Tree 214
6.4.2. Imaging 216
6.4.3. Geometric Transformations 216
6.4.4. Clipping 219
6.4.5. Combining Transformation and Clipping 221
6.4.6. Propagation of Attributes 222
6.4.7. Visualisation Attributes 225
6.4.8. Presentation 225
6.5. Manipulating the Model 228
6.5.1. Initialisation 228
6.5.2. Operations on the Medium 228
6.5.3. Picking and Selecting 241
6.5.4. The User Agent 245
6.6. Conclusions 249

7. Presenter 250

7.1. Brief Outline 251
7.2. Differences 252
7.2.1. Clipping 252
7.2.2. Application Confirmation of Input 254
7.3. Additions 254
7.3.1. Editing 254
7.3.2. Linking 257
7.3.3. Persistence 260
7.3.4. Hardcopy 261
7.4. Refinements 261
7.4.1. Presentation 261
7.4.2. Selection 262
7.4.3. Highlighting 264
7.4.4. Grouping and Scaling 264
7.5. Deficiencies 265
7.5.1. Text and Graphics 265

7.5.2. Input Masks 265
7.5.3. The User Agent 266
7.5.4. Client-Server Working 266
7.5.5. SunView Dependence 266
7.5.6. Memory Limitations 267
7.5.7. Manipulation Efficiency 268
7.5.8. Constraints 269
7.5.9. Higher Constructs 269
7.6. Issues 270
7.6.1. Empty Leaves 270
7.6.2. Access to Text 271
7.6.3. Rectangularity 271
7.6.4. Dimensionality 272
7.7. Conclusions 273

8. Future Work: An Alternative Model for the Surface 274

8.1. An Informal Description of the Model 276
8.1.1. Framing 279
8.1.2. Embedding 280
8.1.3. Multiple Inheritance 281
8.1.4. Constraints 284
8.1.5. Tables 285
8.2. Conclusions 287

9. Conclusions 289

9.1. The Thesis of Surface Interaction 289
9.1.1. Surface Models 291
9.2. Contributions of the Thesis 291
9.3. Limits of Surface Interaction 293
9.4. Postlude 294

Appendix I: Presenter Applications 296

Appendix II: Generic Functions 302

Appendix III: Glossary 303

References 310

Acknowledgements

My sincerest thanks must go to the many people who have supported and helped this work. In particular, I owe a debt of gratitude to Anthony Hall, whose enthusiasm provided the initial impetus. My thanks are also due to Professor Michael Harrison for his patience and optimism; to my supervisor, Dr. Ian Benest, for keeping me on a long lead; to Sylvia Holmes, for her many suggestions which were incorporated in *Presenter*; and to Greg Abowd, for many fruitful discussions, comments, and corrections on the more mathematical side of this work (although of course any errors that remain are mine).

My final and warmest thanks must go to Kate and Douglas, my wife and son, who have put up with my absences and abstractions over this Thesis for as long as they have both known me. They have given me nothing but love and encouragement. I hope I can make it up to them.

Declaration

Minor parts of this Thesis, in a very modified form, have already appeared in [Took90a] and [Took90b].

Presenter was originally specified and written within the Aspect project - the code therefore belongs to System Designers PLC. The formal specification of *Presenter* which appears in [ASPECT87] is very different from that given here, which is an idealisation based on several iterations of implementation and use. The notion of Surface Interaction, and the *UMA* architecture, were developed later as a result of research on this Thesis. Indeed *Presenter* does not conform to the *UMA* architecture since it has no separable user agent.

```
#define he      he or she
#define him     him or her
#define his     his or hers
```

Abstract

To promote both quality and economy in the production of applications and their interactive interfaces, it is desirable to delay their mutual binding. The later the binding, the more *separable* the interface from its application. An ideally separated interface can factor tasks from a range of applications, can provide a level of independence from hardware I/O devices, and can be responsive to end-user requirements.

Current interface systems base their separation on two different abstractions. In *linguistic* architectures, for example User Interface Management Systems in the Seeheim model, the *dialogue* or *syntax* of interaction is abstracted in a separate notation. In *agent* architectures like Toolkits, interactive *devices*, at various levels of complexity, are abstracted into a class or call hierarchy.

This Thesis identifies an essential feature of the popular notion of *direct manipulation*: directness requires that the *same* object be used both for output and input. In practice this compromises the separation of both dialogue and devices. In addition, dialogue cannot usefully be abstracted from its application functionality, while device abstraction reduces the designer's expressive control by binding presentation style to application semantics.

This Thesis proposes an alternative separation, based on the abstraction of the *medium* of interaction, together with a dedicated user agent which allows direct manipulation of the medium. This interactive medium is called the *surface*. The Thesis proposes two new *models* for the surface, the first of which has been implemented as *Presenter*, the second of which is an ideal design permitting *document* quality interfaces.

The major contribution of the Thesis is a precise specification of an architecture (*UMA*), whereby a separated surface can preserve directness without binding in application semantics, and at the same time an application can express its semantics on the surface without needing to manage all the details of interaction. Thus *UMA* partitions interaction into *Surface Interaction*, and *deep* interaction. Surface Interaction factors a large portion of the task of maintaining a highly manipulable interface, and brings the roles of user and application designer closer.

Chapter 1

Introduction

The medium is the message - Marshall McLuhan

The context of this Thesis is both the making and using of interactive computer applications. The purpose of the Thesis is to advance a new architecture for the structuring of applications and their user interfaces which reduces the cost of making direct manipulation interfaces while providing for ease of use. This architecture is based on the separation of a *generic presentation model* from applications. The essence of the Thesis is that by giving this presentation model (the *surface*) its own state, operations, and agent, it can be manipulated either by applications, or directly by the end user. The surface can thus abstract common manipulative tasks from applications, and also act as a *medium* of communication [Draper86] between applications and end users.

We explore the context of this architecture by examining first the needs of the user and the needs of the interface designer.

1.1. The Needs of the User

The user is concerned primarily with the *quality* of the interface. This may involve stylistic issues like 'look and feel', or the interface's ability to prevent or undo errors. At base, however, the quality of the interface must be judged on how well it enables the user to perceive the current state of an application, and how well it allows him to manipulate that state in order to attain some user-defined goal.

The interface is thus necessarily, in the general case, a two-way medium between applications and users. However, an important criterion is the degree to

which the interface allows the *same* object to be used for both input and output. We can call this criterion *directness*.

A glass teletype, for example, has low directness. Interaction is textual, and references to previous output are *symbolic* rather than direct. The user might ask for a directory listing, and then, if he wishes to delete a file in the list, must retype the file name as a parameter to the appropriate command. In the worst case, when output scrolls off the screen, the user must maintain the relevant state in his memory, and make references from that.

A graphical mouse-driven interface, by contrast, has potentially high directness, since graphical objects persist and can be referenced geometrically by the mouse. It is conventionally agreed that 'direct manipulation' [Shneiderman83, Shneiderman82, Hutchins86] enables higher quality interfaces. However, graphical displays and pointing devices are a prerequisite, but not a guarantee of directness.

Directness itself does not ensure consistency between the state of the interface and the state of the underlying application. The interface operations need to form some 'complementary algebra' [Harrison90] to the operations possible on the application state. It is precisely mismatches between operations in the interface, and their denotation in the semantics of the application, which leads to poor interface usability. It is an open question whether these algebras (i.e. the interface operations and their semantic counterparts) can be specified independently. This Thesis attempts to define precisely the bounds on interface independence, and the consequent communication requirements between application and interface.

1.1.1. Performance

Human users may also have constraints on their performance which are not taken into account in the functionality provided by the application. They may be naive, colour-blind or otherwise disabled. They may have a limited short term memory, and go away for cups of tea. The hope is often voiced that these 'human factors' issues of aesthetics, ergonomics or cognitive psychology can be represented in the interface. The problem of determining and formulating these human constraints is the concern of research initiatives like user modelling [Young89, Kass88]. Accommodating such constraints has been a traditional goal for User Interface Management Systems (UIMS) [Bennett87].

As well as disabilities, human users may also have *skills* which equally may not be exploited by the application. They may be able to assimilate graphical information rapidly. They may have good hand-eye coordination, and be used to handling and manipulating physical objects. An interface system should present application functionality in a way which maximises the use of these common human skills. Bailey et al [Bailey88], for example, quote a user productivity gain of 77% through re-engineering of the user interface alone.

1.2. The Needs of the Interface Designer

In 1972, Meads [Meads72] criticised graphics software for being either too complex for the occasional user or too inflexible for the sophisticated programmer. Recently, Myers [Myers88b p.17] was still able to list ten problems with existing user interface design tools, including the difficulty of coming to grips with new interaction languages or libraries consisting of hundreds of tools, and limitations in functionality.

The interface designer may well consider a successful interface to be 'saleable, fabricable, and cost-effective' [CohenB86]. That is, he may be concerned primarily with the *economy* of the interface system. From the point of view of cost, the high proportion of user interface code typically found in interactive graphical applications (over 50% [Szekely88b]; 80% [Myers88b]) makes it a target for rationalisation. Such a cost would not be tolerated, for example, in interfaces to hardware peripherals like disks.

Economy and quality in interface systems may conflict. The high cost of writing interactive graphical software often results in interfaces restricted to 'cheap' static panels of 'clickable' objects (buttons, menus, icons) which simply invoke application functions. There may thus be as many levels of *indirection* between actions and effects in a mouse-driven interface as there are in a command-line interface: there is a loss of 'engagement' [Hutchins86] between the user and the objects he may be directly interested in manipulating.

In this context, we consider the interface designer (who may of course also be the application writer) to have two basic needs: power and freedom.

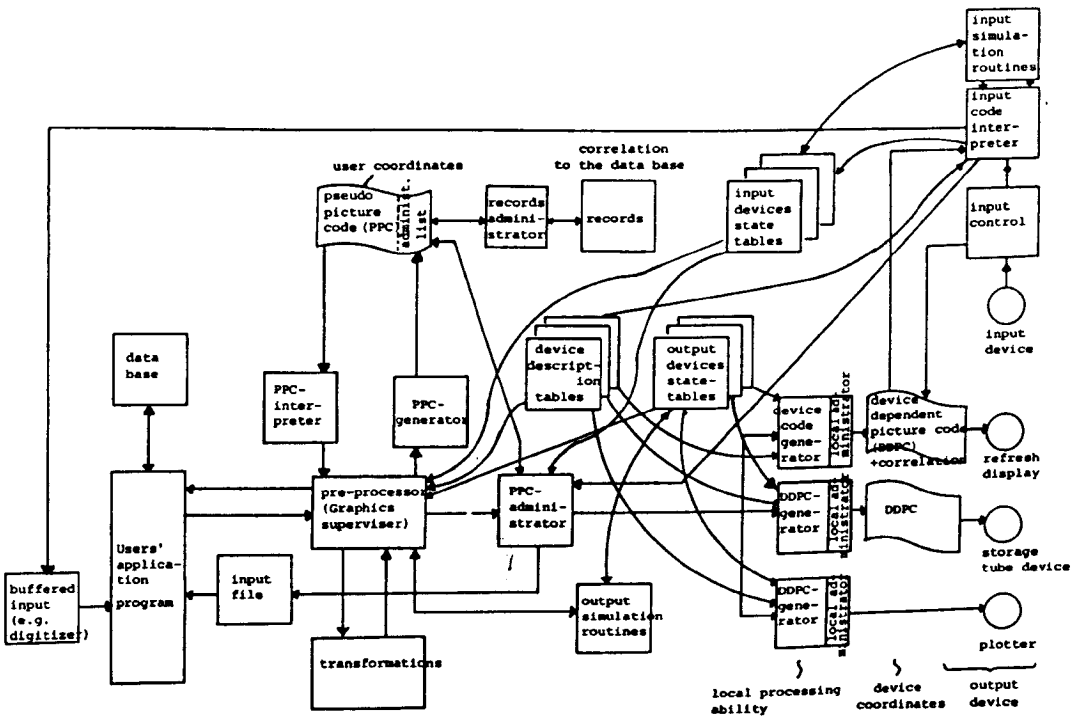
1.2.1. Power

Interface systems vary in their constructive power. An *under-powered* system only provides low level primitives, such as RasterOps [Newman79] or BitBLT [Goldberg83 p.333, Ingalls81]. Higher types of object must be constructed and maintained by the interface designer. At the extreme, an interface system which only provided a

setpixel (position, colour)

operation on its medium could display any image, but at the expense of much iterative coding on the part of the designer.

On the other hand, we may consider an interface system *over-powered* if the complexity of the functionality it provides is as much a barrier to effective use as a lack of functionality. The following diagram appears to represent a system that is over-powered in this sense [Encarnacao79 p.89]:



Concept for a device independent graphics system

1.2.2. Freedom

As well as considering the *ease* with which the interface designer can construct the interface he wishes, we can also consider the *possibility* of realising his

design. That is, the medium may limit the scope of interface design, either by constraints in the presentation types (Cedar windows [Beach85], for example, cannot be overlapped), or by withholding control over its generation (at the extreme, an interface may be generated automatically from an abstract interaction specification [Olsen83b, Scott88], or even, in theory, from a description of the task model [Green87, Singh89].)

Such design constraints can be called *style* [Newman88]. Viewed positively, style can impose a pleasing consistency. As Kasik [Kasik89 p.60] says, "attractive interfaces matter". However, it is an open question whether consistency is worth the loss of design freedom. Applications and user groups may wish to customise their user interface to some house style [Marcus84], and HCI researchers may wish to explore alternative styles. Since an interface medium which gives design freedom may be constrained to produce a consistent style, but not vice versa, it is clear that imposing consistency in the interface system is a less fruitful approach. It can also be argued that the ergonomics and aesthetics of interface design have not yet been so thoroughly researched that they can be standardised in a fixed 'look and feel'. [Took90b] explores these issues in more detail.

We go on to examine a basic mechanism for catering for the needs of both the user and the interface designer: separation.

1.3. Separation

The provision of interface services for applications is conventionally justified in terms of application/interface *separation*. This is a notoriously vague term (see Chapter 2). We define it more precisely here in terms of abstraction and binding.

1.3.1. Abstraction

The conflict of economy and quality in user interface construction can best be addressed (as in most fields) by *abstracting* common features and implementing these separately. At a very idealistic level we can view the interface itself as an abstract, which is *applied* to application functionality to produce a usable system:

interface (application) = system

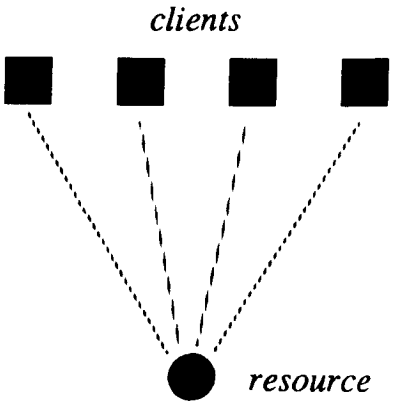
This formulation suggests that it should be possible to apply the same interface to a number of different applications, or apply a number of different interfaces to the same application. We might even be able to make the interface *generic* over a range of styles (or users) [Wiecha89]:

interface [style] (application) = system

A major concern of this Thesis is to examine the limitations of performing this abstraction. Abstraction has two main benefits: factoring and independence.

Factoring

If a number of applications duplicate the same operations, it makes sense to abstract these into a single separate resource. We can thus factor the work done, and reduce any unnecessary effort. As a rough illustration:

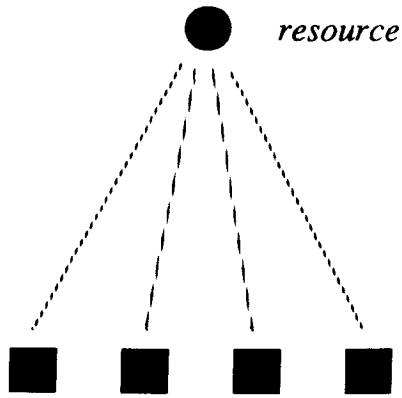


The cost of factoring is the difficulty of designing the data types and operations in the factored resource. On the one hand there is no point in factoring functionality that is rarely used. On the other hand it is equally pointless to factor functionality that is so frequently used that the bandwidth of the communication medium becomes a bottleneck. In general, as Stroustrup notes [Stroustrup88], finding 'commonality' in a set of objects and designing appropriate operations is far from trivial.

Independence

Not everything has been standardised. There exist many different types of input and output devices, communication protocols, text and graphics libraries, languages, and so on. If an application is not to be completely rewritten for each different software or hardware environment, then there must exist a representation at some level between the hardware and the application code which is common over

a range of environments. This representation is then a resource which provides a level of *independence*. This hides the diversity of the underlying implementations, and anything written to the representation is easily ported between these (so long as appropriate back ends already exist). Standard graphics languages such as GKS [ISO85, Enderle84] or PHIGS [ISO87b, Brown85] are predicated upon such independence. As a rough illustration:



The user interface is an ideal site for a level of independence. That is, the interface can provide a common representation over a range of input and output devices and software. This commonality might be realised on a set of normalised devices, or at higher levels, for example on logical devices, interactive techniques, or even dialogue.

At the highest level, it is even possible to conceive of the user as requiring independence from applications (applications are simply *implementations* of the user's tasks). In this view, the interface should allow the user to impose his own concerns on the representations of the applications. For example, he should be able to cut and paste representations from one application to another.

1.3.2. Binding

Once identifiers declared in one component are bound to values (constants, variables, operations, functions, procedures) defined and implemented in another, then *communication* can take place. Whereas independence allows a conceptual distinction between abstractions and their implementations, binding allows a *temporal* distinction between communication mechanisms. Early or static binding permits communication via a shared environment. Late or dynamic binding permits

communication via a distributable protocol. We can thus use binding in a *relative* sense, to compare the separations achievable between two components. If we *delay* the binding of the interface abstraction to the application, we gain in separation.

We can distinguish four common classes of separation, in order of increasing lateness of binding. In each case we can say what tasks can be *factored* over the binding:

- Application and interface are *designed* separately, but coded as one process. In this case only the design can be factored.

- Application-specific and interface-specific code are held in separate classes or libraries and bound together at compile time. This in addition factors the programming cost of the interface. Toolkits typically have this class of separation.

- The formalisms (languages or primitives) for interfaces and applications are distinct. The interface language is interpreted at run time, but cannot be changed. If the communication between application and interface is by messages rather than subroutine calls, then the interface and application may run on separate devices. The interface, for example, may run locally on the workstation for optimal performance. This in addition factors the running cost of the interface. UIMS ideally have this class of separation.

- The interface to an application can be changed while it is running, without the application being aware. This is the principle behind Coutaz' Dialogue Socket [Coutaz86]. This (in theory) factors the user's control over the *style* of interaction, and makes the application and the interface mutually independent.

As the last point illustrates, incorporating user concerns dynamically into the interface ideally requires maximum separation. In interactive systems, user input is necessarily bound late to application functionality. Otherwise the application would not be *responsive* and would need to be run in batch mode. However, in order to provide dynamically *adaptive* interfaces [Kantorowitz89, Alty84, Benyon84], it is necessary to delay the binding of (the stylistic component of) the interface to the application at least until run time. On the other hand, this reduces the designer's expressive control [Bos83 p.89] over the interface.

Representations which are maximally late bound may in addition be *persistent*, in that their lifetimes are not tied to the lifetime of the objects which create or use

them. Interface objects which are persistent may be created prior to the applications which use them, and may be saved and recreated between application sessions, and even passed as messages between different applications.

Separation thus necessarily involves abstraction (if one component were not an abstraction of some functionality in another, there would be no need, or basis, for communication). In addition, abstractions may be more or less separated from their use, depending on their binding time.

1.4. Existing User Interface Systems

Existing interface services can be categorised in terms of abstraction and binding:

- Graphics languages abstract the production of output. Typically the primitive operations are bound early to the client application, but their implementation may be delayed so that equivalent images can be produced on a range of workstations.

- Input frameworks abstract the routing and first level parsing of user input. Applications are usually written on a particular framework, and so are bound early to this. The implementation of input frameworks, for example in the X Intrinsics layer [MIT88], may be late bound and therefore compatible with a range of workstations.

- UIMSs abstract the interactive *dialogue* from applications, that is, the sequencing of input and output events. Ideally the dialogue is bound late to the application functionality, so that the application can be isolated from interface issues. The UIMS itself can be seen as the *implementation* of the dialogue, and ideally this is bound at run time (the UIMS *interprets* the dialogue) so that the dialogue can be presented in a variety of styles on a variety of machines, and possibly to a variety of users. This ideal, as we show, is very limited in practice.

- Agents in general abstract components of application functionality into *devices*. They are necessarily bound early to the application task, since they encapsulate it.

- logical device agents abstract sub-dialogues from applications, typically to provide input functions such as option choice, strings, or location. These may be bound early to the application, but their implementation in terms of particular styles or hard devices may be bound late.

- Toolkit agents typically provide logical devices, except that these are often bound early to a particular stylistic 'look and feel'.

1.5. Premises and Issues

We should only want to abstract some component of human-computer interaction into a separated interface if we can thereby serve a wide range of potential users and applications. Thus in this Thesis we are concerned not with the performance of particular interfaces to particular applications, but with providing generic interface support. Such a system should be free both of user and application bias but capable of incorporating both.

The major premise for the Thesis is that, in order to promote both *quality* and *economy*, the interface system should be maximally *separate*, that is, maximally abstracted and late bound. This is in contrast to library and toolkit paradigms, which concentrate on just abstraction of functionality. We make the assumption that quality is best promoted through economy, since it is thus cheaper to iterate interface design.

Whereas user interface services have typically concentrated on separating either the *form* of interaction (i.e. dialogue) or the *devices* of interaction, this Thesis concentrates instead on separating the *medium* (i.e. *content*) of interaction.

The medium itself could be any domain which can be directly addressed by the user. It could, for example, be a domain of sounds or speech, or text, or limited graphics such as windows, or a more general graphical domain. This Thesis, however, is concerned with visual, as opposed to audio, tactile, or other media. This includes text and graphics. Interactive visual devices are considered at a certain level of abstraction, but in implementation a bitmapped screen for output and a mouse and keyboard for input is assumed.

The Thesis is not concerned with judging visual interfaces against human factors criteria. We simply aim to provide the basic constructs whereby user interfaces

can be built by the designer and modified by the user. The implicit standpoint throughout the work is that usability is best promoted by flexibility at the designer level rather than by a fixed stylistic 'look and feel'.

A core problem is to provide constructs for the medium which have objectivity but *not* style. It is considered that the best way to design or discover these is to examine the basic features of the medium itself, rather than any *use* to which it might be put. Text, for example, is considered independently of its use as a medium for applications like mail systems, databases, or document processors.

1.5.1. Presentation

The visual domain subsumes what is commonly called *presentation*, that is, the display of screen objects. This is obviously an essential part of any visual user interface. Many designs for user interface systems, however, simply assume the existence of a presentation layer [Alexander87 p.22, Olsen86 p.322]. Green [Green86] and Hudson [Hudson87 p.120] point out that the main emphasis of UIMS research has been on the dialogue rather than the presentation component. Olsen claims that presentation has been 'sorely neglected' [Olsen87a p.135], and that

> *all ... UIMS that we are aware of ... [do] not account for the presentation of application data. This most important aspect of a user-interface implementation has not been adequately addressed by current research.* [Olsen86 p.322]

He himself addresses these problems in the GRINS UIMS [Olsen85b].

However, there is no clear agreement as to what the presentation level should comprise. Green [Green85b p.13] sees it as a fairly static layer concerned with output types (he extends this to sound and the control of mechanical movement) and input types (again extended to include video, voice, and gestural input). Olsen [Olsen85a p.126] and Dance et al [Dance87 p.99], on the other hand, impute more 'input/output linkage' [Olsen85b] to the presentation level, in the form of 'logical devices' or 'interaction techniques'. That is, a certain amount of the echoing between input and output devices is allowed to migrate from the dialogue level to the presentation level. Still another interpretation is given by Szekely [Szekely87, Szekely88b] and Moreland [Moreland87] who view presentation as output only. In Szekely's view, presentation is a display mapping from underlying application

objects, whilst input is a separate mapping from physical devices to application operations.

The issues are thus whether input is to be included in the presentation domain, and if it is, the amount of control autonomy that is to be given to the presentation level in order to 'link' input and output prior to application involvement.

Presentation Constructs

Constructs for presentation have traditionally been addressed by standard graphics packages, like GKS [ISO85] and PHIGS [ISO87b]. Rosenthal [Rosenthal83 p.38] calls this 'mainstream' graphics. However, presentation on bitmapped workstations has essentially pursued a separate development path. To a large extent this is due to the unsuitability of the traditional vector-oriented paradigm to the mechanics and capabilities of raster displays, and the poor input facilities of the standard graphics packages. It may also have something to do with the task domains: standard graphics has typically been used in an industrial environment where a model of a complex object like an automotive part or an oil refinery is constructed in virtual space, and then is viewed from a variety of angles. This may be called *scenic* modelling. The parts of the object have no denotation other than their visual qualities (they stand for no other information or functionality). Operations (pan, zoom, rotate) are global to the space, rather than being targeted on particular parts of the object.

Bitmapped workstations, on the other hand, have their main application in office environments [Newman83, Newman87] as a medium for what may be called *schematic* modelling. In a schematic model the information content of the interface is paramount, while accuracy of geometry or rendition are secondary issues. Such systems include software engineering environments [Benest85, Took86b], database systems, spreadsheets, and document processing applications. In these applications displayed entities *denote* information or functionality, rather than *represent* real world objects. These reasons may account for the lack, in bitmapped environments, of a global graphical model for all objects of visual interaction. In these environments, only low-level operations like RasterOp, and various windowing protocols, such as X [Scheifler86], have reached even the status of *de facto* standards.

A similar situation is developing in the domain of textual presentation. Emerging international document standards like ODA [ISO87a] and SGML [ISO86b]

address issues of device independence and document structuring and transmission, but take little account of the suitability of their constructs for *interactive* document preparation. Independently of, and in contrast with, this standardisation effort, the workstation community is experimenting with hypertext [Conklin87], and active [Zellweger88, Allen 81 p.74], interactive [Arnon88], multimedia [Crowley87, Angell87] and hypermedia [Meyrowitz86] documents.

The models proposed in this Thesis attempt an integration of document and application concerns by tightly coupling textual and graphical presentation. At the same time they address issues currently underdeveloped in bitmapped environments: structuring in text and modelling in graphics. Finally, they attempt to provide a broader covering of the domain of presentation than that possible with the customary opaque, rectangular windows. Considerations include transparency, hierarchical structuring, tabular layout, and persistent graphical links.

1.5.2. Terms: Object and Application

A central concept in this Thesis is the notion of an *Object*. By *Object* we mean an abstract data type [Guttag78] which encapsulates not only operations but also *state*. The identity of an Object persists, but its state may be modified by its operations. We do *not* assume class inheritance or any other feature of object-oriented programming in this definition, although clearly these may create Objects. Since the word *object* is common and useful, when we mean it specifically in the sense above, we use the capitalised form.

Throughout the Thesis we use the term *application* to refer simply to some domain-specific functionality, and do not imply thereby any *particular* computational model like procedural, declarative or object-oriented, unless otherwise stated. However, we do assume that an application may have *control*, that is, we do not necessarily think of an application as simply a collection of semantic functions to be called by the interface.

1.5.3. Formal Design

A final premise of the Thesis is that a well-powered interface system can best be achieved via a formal design. The hope is that such a design elucidates the *a priori* features of the objects of interest themselves, uninfluenced by implementation strategies. If unavoidable conflicts occur (the screen resolution might not be quite up

to displaying mathematical points, for example!) then a formal design at least forms a basis upon which sensible trade-off decisions can be made. The model is expressed here in the formal notation Z [Spivey89], developed at the Programming Research Group in Oxford, and its communication architecture in CSP [Hoare85].

1.6. Overview of the Thesis

1.6.1. The Thesis

The Thesis is that by separating the *medium* of human-computer communication we can provide for both economy and quality in user interface design. The abstraction is supported by a *model* (i.e. state and operations) by which the semantics of the medium is defined. The late binding of the medium and applications is supported by an *architecture* (*UMA*) which defines how the user, the medium, and the application communicate.

The medium's model

- has encapsulated state and operations on the state. Applications can invoke the operations and address objects in the state, which have persistent identity.

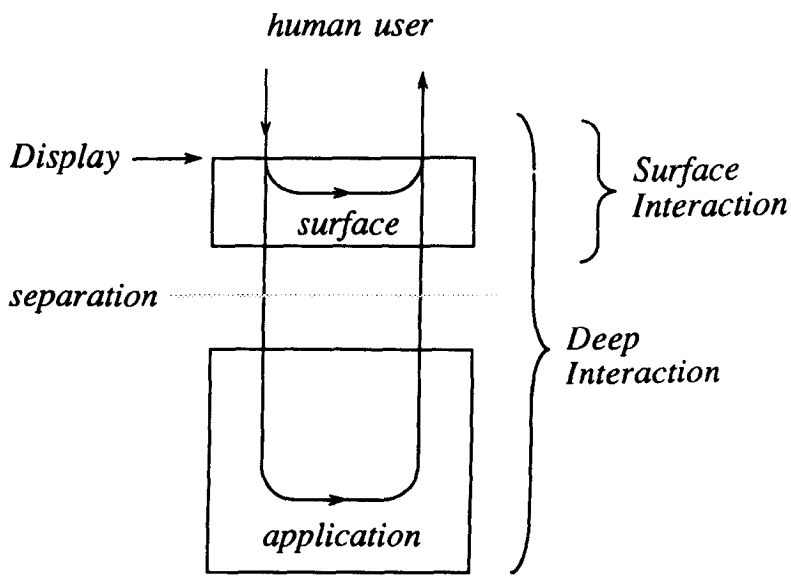
- has a presentation function by which the objects in the state can be presented on a display. The presentation function can be inverted to allow users to pick surface objects by addressing the display directly with a pointing device.

The *UMA* architecture incorporates a *user agent*, dedicated to the medium, which translates all user input either to operations on the medium, or to messages to the application. Together, the medium and the user agent form an *interactive medium*, here called the *surface*. The term surface is used deliberately to suggest a more specialised domain than interface in general.

The major benefit of a surface in the *UMA* architecture is that, because it is abstract and late-bound, its *operations* may also be invoked independently of the application. Thus application objects can be manipulated both by the application, and directly by the user. This allows the surface to *factor* manipulations which are irrelevant to application semantics, but which may be significant for the user.

In practice this economises on the cost of creating application interfaces, as well as allowing the user greater power over the appearance of the surface. It also allows surface objects to be interactively constructed prior to being bound to application semantics. Thus the roles of user and interface designer are closer. This application-independent manipulation is here called *Surface Interaction.* Surface Interaction allows surface objects to have *behaviour* without *functionality.* This is the core of the Thesis.

Very schematically, we think in this Thesis in terms of the following separation:



Thus the surface has *some* depth, that is, it has its own semantics. However, this is separated from application semantics. At the boundary between the surface and the human user there is some *display*, which we usually think of as a screen (we also assume appropriate input devices). Surface Interaction takes place simply between the user and the surface, while deep interaction takes place between the user and the application.

1.6.2. Structure of the Thesis

The Thesis is in two broad parts, the first of which (Chapters 2-4) presents Surface Interaction and its *UMA* architecture, and the second of which (Chapters 5-8) presents two alternative models for the surface.

Chapter 2 categorises existing architectures which provide separation of application and interface, in particular into what it calls *linguistic* architectures and *agent* architectures. Linguistic architectures abstract the *syntax* of interaction, while agent architectures fragment application functionality into *devices*.

Chapter 3 examines critically and formally the premises for dialogue abstraction, and accounts for the lack of success of systems which employ this as the basis for separation.

Chapter 4 is the main formulation of the Thesis. It establishes the existence of the surface and the possibility of Surface Interaction, and defines formally, in CSP, a minimal architecture (*UMA*) by which Surface Interaction can occur. It also examines implementation issues arising from the architecture.

Chapter 5 describes existing models for the surface and its medium, in particular window and graphics systems.

Chapter 6 gives a formal model, in Z, of the surface which has been implemented as *Presenter*.

Chapter 7 gives an account of the implementation of *Presenter*, and how this differs from the formal model. Lessons are drawn from its difficulties and deficiencies.

Chapter 8 describes informally an alternative, more ambitious architecture which forms the basis of future work.

Chapter 9 concludes.

Chapter 2

Architectures for Separation

2.1. Separation

The extreme positions on the separation of interface from application are represented by Coutaz and Sibert respectively. Coutaz 'accepts the principle of separation' [Coutaz85 p.21]. Separation brings the following benefits (see [Szekely87 p.235]):

- User interface and applications can evolve independently. It may be possible to program, analyse or prototype each in isolation from the other, and using different formalisms.

- One interface can be made common to a range of applications, and thus interface consistency can be enforced, and code and development effort shared. Generic commands, for example to invoke abort, undo, or help operations, and status or error reporting, can be provided [Lieberman85 p.182]. As well as such run-time support, a common interface could offer support for design, analysis, or evaluation of interfaces [Dance87 p.97]. Myers [Myers88b p.4] gives a more detailed list of such facilities.

- A range of interfaces can be applied to the same application, so that user preference or designer experimentation can be catered for. In this way various levels of independence can be built into the interface, from device independence to style and dialogue independence.

Barth, for example, claims to 'maintain a strong separation' [Barth86 p.147] between interface and application in GROW.

At the other extreme is Sibert [Sibert86 p.261]:

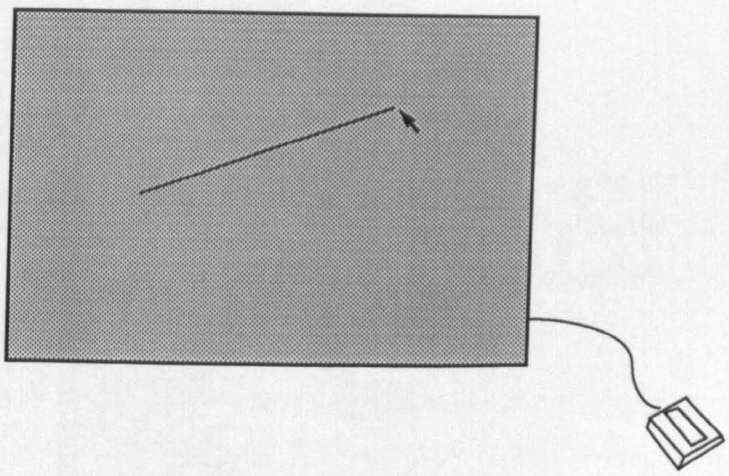
> *We are convinced that it is not possible to build systems which handle semantic errors and feedback intelligently if we maintain a strict separation between the lexical/syntactic domain in the UIMS on the one hand, and the semantic domain of the application on the other.*

See also [Sibert85 p.183]. This is supported by Green, who argues that if the application can directly influence the user interface, then 'the notion of a separate user interface module breaks down' [Green86 p.257]. Recent experience [Manheimer89 p.131] underlines this.

Clearly there is no consensus on the possibility of separation. Even papers that deal centrally with the topic [Hartson89] come to no firm conclusion. This chapter examines different architectures for separation and their success, in particular what we here call *linguistic* architectures and *agent* architectures. Some of the material in this Chapter has also appeared in [Abowd89].

2.1.1. Motivation for Separation

Architectures providing separation originate in response to the problems of coding an interactive system as a single process in a standard procedural language. We illustrate these problems, and their various proposed solutions, with an example due to Newman [Newman68]:



What is required is a draughting system which minimally allows the user to draw arbitrary lines on the screen. The functionality is as follows: upon the first

mouse click (or push on the light pen [Benest79 p.99] in Newman's paper) the system goes into line drawing mode (in Newman's paper it is only at this point that cursor echoing begins). Once in line drawing mode the user can move the mouse cursor around the screen until he decides on the starting point for the line, which he signals with another mouse click. He can then continue to move the mouse, but now a rubber line starting at the first point stretches to the cursor. Upon a third mouse click this line is fixed in place, and the system goes back to its uncommitted state.

A procedural coding of this system is as follows (we assume an input event of type *button* | *point*, where *point* is a pair of coordinates, and also primitives to *draw* a line between two points and to *clear* the screen):

```
var start: point;
repeat
    repeat
        read (event);
    until event = button;
    repeat
        read (event);
        if event ≠ button then start := event;
    until event = button;
    repeat
        read (event);
        if event ≠ button then
        begin
            clearscreen;
            drawline (start, event);
        end;
    until event = button;
until false;
```

This approach is recommended by Jones [JonesDW88] as 'the best way' to code a finite state machine, which indeed the draughting system is. Notice, however, a number of deficiencies of this code from the point of view of abstracting its interactive behaviour:

- The states of the system are only *implicit* in the organisation of the program. A more complex FSM might result in a more deeply nested program structure in which the states would be even less obvious.

- This is a concise representation only for FSMs in which the control flow is *well-structured.* If arbitrary jumps are permitted in the FSM, for example to

reach abort states, then much redundant code might result. This is because the aborting code would have to be replicated in each block of the program in which an abort could occur.

• Input consumption (*read (event)*) is scattered through the program. A change to the event types might require many modifications to the code.

• *Read*s occur in places where only particular events are expected. If there were two concurrent FSMs, such that their input could be arbitrarily interleaved (for example, if there were a second mouse button which drew circles) it would be impossible to code both FSMs together using this technique without much redundant code. This is because, in general, the state space of two concurrent FSMs is the *product* of the number of states in each, since for *each* state in one machine there may occur an event moving the other machine into *any* of its states.

• A simple modification to the FSM, for example the addition of a transition, may require a radical restructuring of the code.

An alternative approach ('the wrong way', according to Jones) is to make the states of the FSM explicit in the program:

```
state := initial;
while true do
begin
    read (event);
    case state of
        initial:    if event = button then state := startline;
        startline:  if event = button then state := endline;
                        else start := event;
        endline:    if event = button then state := initial;
                        else begin
                            clearscreen;
                            drawline (start, event);
                        end;
    end case;
end while;
```

Jones' main criticism of this style of control structure is that it is in effect a series of *goto* statements. Indeed the assignments to *state* could simply be replaced by *goto*s to the appropriate section of code. However, systems may well be driven by relatively unstructured FSMs which it is difficult to code any other way. For

example, although a system may have nested states, it may also, as noted above, provide unstructured jumps to common facilities like help systems and abortion. [Bohm66] demonstrates formally that while any program can be expressed without conditional jumps, unstructured programs will require either explicit state variables or repetitive coding.

This second style of control structure in fact resolves many of the problems of the first:

- Its states are explicit.
- Its complexity is independent of the *structuredness* of the FSM.
- Input consumption takes place at *one* location.
- So long as there is some method of despatching input to the appropriate case statement, a concurrent FSM could be added such that the total number of states would be simply the *sum* of the states of the two FSMs.
- Modifications to the FSM are easily incorporated in the code.

Nevertheless, this coding is not ideal. There are two remaining problems:

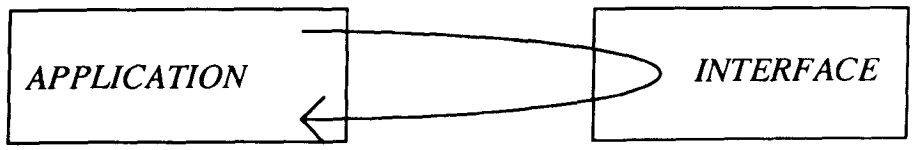
- All acceptable sequences of input are *explicitly* coded. If it is not important in what *order* some inputs occur, so long as they all do occur, then nevertheless each possible sequence would have to be coded. Thus, paradoxically, giving the user more freedom to choose his style of interaction involves extra work for the programmer.
- The programmer must construct the input despatching *framework*, in this example the *while* or *repeat* loops and the *read* primitive. This is clearly a generic structure which could be provided as a service.

These problems form the fundamental motivation for all user interface management systems and services. They illustrate how the primary concern with abstracting interactive *dialogue* from application functionality has arisen.

We first examine various *mechanisms* for separation.

2.1.2. Flow of Control

Separation of interface and application can be characterised simplistically by their flow of control. Historically, the proposals for user interface systems have differed clearly in this respect. The programming language model above, in which the user interface is coded as part of the application program, can be illustrated:



This has been called the 'internal control' model [Thomas83 p.17] or 'embedded control' [Kamran83 p.59], or the 'prompting' model [Young88 p.371]. In this model the interface modules are typically bound in at compile time from a library, and it is difficult to separate interface services at run time. For example it is not possible to separate dialogue, simply because control resides in the application. In this model also the application must have knowledge of, and thus be dependent upon, the interface, since it calls on its functions. Lantz claims that most applications have internal control [Lantz87a p.40].

The 1982 Seattle workshop which laid the foundations for UIMS development proposed an alternative model in which control resides in the interface rather than the application. This is called 'external control' [Thomas83 p.17], or the 'despatching' model [Young88 p.371]. This can be illustrated:

This configuration makes dialogue separation possible, since the dialogue interpreter can reside in the interface. Many early UIMS and window systems have external control. This is true of Sun and Tajo [Teitelman86 p.40], AIH [Kamran83 p.59], Tiger and Oasis [Kasik89 p.56], SODDI [Gangopadhyay82], and MINICORN [Strubbe83 p.1041]. Dialogue separation in the linguistic architecture requires at least an external control model, since it is the dialogue control module in the interface which determines the invocation of application functionality.

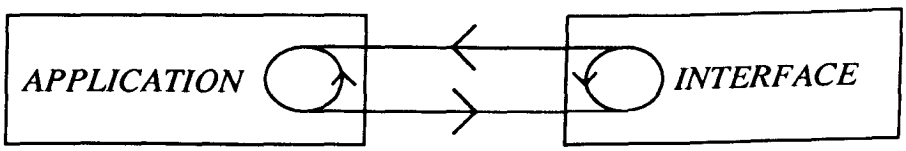
In the external control model, application functionality is thus typically fragmented into 'action' routines [Swick88 p.224] or 'callback' routines [Rao87 p.120]. User input must be multiplexed and despatched to these semantic routines with reference both to the type of the input event, and to the object upon which the event occurred. Depending upon the order in which this selection is performed, routines may conceptually be attached

- to the particular interface agent (as is the case with widgets in the X Toolkit [Roberts88 p.272] and in GROW [Barth86 p.155], PAC [Coutaz87 p.434], the Box [Coutaz84b p.4], Descartes' 'interactive extensions' [Shaw83 p.105], and Minicorn [Strubbe83 p.1039]).

- to particular events (as happens in Cardelli's Toolkit [Cardelli87 p.22]).

A common characteristic of both internal and external control models is that there is a *single* thread of control. This makes it difficult to cater for asynchronous events which occur at the level in which control does *not* reside. In the internal control model, for example, spontaneous user input cannot be accepted. On the other hand, under external control, internal events (such as signals to the application from other processes) are difficult to handle. Continual operations, also, are difficult to achieve at the level in which control does not reside. It is therefore difficult to animate views or monitor state using the external control model.

These problems are noted in the 1984 Seeheim workshop on UIMSs [Pfaff85], and by the time of the 1986 Seattle workshop a third model emerges in which the interface and application components are truly concurrent [Hill87b, Tanner87, Lantz87b]:



In this model the components can each retain control and thus monitor input or generate output asynchronously. Communication is by messages or events, and not by handing over control. If required, either the internal or external control models can be simulated by the concurrent model, simply by using blocking sends or reads. In addition, the components need not be monolithic: the granularity of communicating

components can be increased (in theory) arbitrarily, as for example in actor systems [Agha86]. The concurrent model also allows interface and application to run on separate processors. The concurrent model therefore seems by far preferable. The *UMA* architecture in Chapter 4 allows concurrency between application and interface.

We go on to examine specific classes of architecture that provide some level of separation between applications and generic user interface tasks. These architectures are input frameworks, linguistic architectures, and agent architectures.

2.2. Input Frameworks

As we saw in the programming illustration above, reading and acting upon user input is likely to be a common task in interactive applications. Input *frameworks* hide implementation details such as device polling loops, and allow the application to deal with more abstract events.

2.2.1. Input Types and Modes

At the most abstract level, user input simply delivers values of some type. In mainstream graphics this is known as the 'measure' of the input device. Measure is some function of the state of the device, for example producing a character or a location. In graphical user interfaces the mouse is so pervasive that a useful first composition of input is into a value, a time, and a location (a *what*, a *when*, and a *where*). These are construed as happening simultaneously.

Orthogonally, physical input *devices* may be divided into two classes: discrete and continuous. Discrete devices are generally two-state (like buttons) which generate events upon *transitions* between these states. Continuous devices, on the other hand, (like a mouse or a potentiometer) must be *sampled* upon some trigger in order to generate a measure.

The conventional interpretation is that discrete devices generate events (for example, a keypress), whereas continuous devices are sampled (for example, to get the position of a dragged mouse). In the general case, however, all devices have state, and the measure of that state may be triggered arbitrarily. Thus discrete devices can be made to deliver continuous input (such as the time interval between

press and release of a button), and conversely continuous devices can be made to deliver discrete input (such as an event generated when the mouse starts moving).

In practice, GKS for example allows these permutations, but CORE (GKS' failed standardisation competitor, but still in use [Kasik89 p.56]) binds the input classes with particular modes: valuator is sample-only input, pick event-only [Rosenthal80 p.364]. Similarly, in VGTS [Lantz84 p.33] the mouse can be used in sample or event mode, but pick operates only in request mode and the keyboard only in event mode. Further, the construction of a logical input token may involve the reading of a number of physical input devices simultaneously. In mainstream graphics, there may for example be both measure and trigger processes and associated devices [Rosenthal82 p.34]. Mainstream graphics also adds a *request* input mode, which in effect implements an infinite wait for an event of a particular type. These mismatches in interpretation and synchronisation between logical and physical devices are major problems in the management of input.

User interface frameworks essentially perform two functions on raw user input:

- they *route* input to the appropriate processes. An appropriate process might be one which is expecting input, or one to which the user has directed input.

- they *interpret* input with respect to some context. A keypress event, for example, may be interpreted as a character input in the context of a table mapping keys to characters.

We examine these two functions in detail.

2.2.2. Input Routing

Routing is the passing of raw input tokens from the physical devices to processes which are interested in them, or at which the user has directed the input. Without an input framework this could only be achieved by requiring all processes needing input to poll all input devices to ascertain if their state had changed since the last poll. Application polling makes the synchronisation of different devices difficult, and may result in an application consuming an event not directed at it.

These disadvantages to device polling have meant that frameworks typically provide *event* rather than *sample* input. In cases where the event is not immediately generated by the user (by activating a discrete device like a button), a trigger pro-

cess is usually stationed in the server to emit events on some criterion, for example when the mouse starts moving or has moved a certain distance. An input framework which provides events essentially hides input device polling from clients.

Events provide a mechanism for *asynchronous* input processing. In this way, in theory, input can be handled immediately, independently of the state of the underlying computation. This is guaranteed if events are signalled to the process by a hardware interrupt. In practice, however, while immediate handling of input might be useful in dealing with catastrophic occurrences such as 'abort', in many cases the *sequencing* of user input carries significance. A fast typist, for example, would not be happy to find that many characters failed to be registered because they were constantly being either interrupted by the next character typed, or, depending on the prioritising scheme, locked out while the previous character completed its processing. If the interrupt routines were stacked, the typist might even find the characters coming out in reverse order! For this reason a general interrupt mechanism is not normally used at a high level, although CSI proposes one possibly using UNIX* signals [Williams87 p.6].

A more effective solution is the provision of an input queue. Events are therefore not lost if they are not consumed before the next event. A main issue here is the queue mapping between devices and processes. Typically there is a *single* queue per process (see [Lantz87b p.90, Lantz87a p.41, Lantz84 p.33]) upon which all its input events are interleaved. However, CSI clients can set up multiple queues [Williams87 p.27], and Pike makes a proposal for a window system that has separate mouse and keyboard channels [Pike89].

The major routing problems are event synchronisation and event despatch.

Event Synchronisation

A single queue abstracts the task of event synchronisation from the application, since events from different devices which occur together appear together on the queue. The cost of queuing in comparison with interrupt-driven input processing is a loss of immediacy. The input event must wait on the queue until its process is ready to deal with it. Similarly, the typical process action is to WAIT (see Foley [Foley84a p.57]) until there are events ready on the queue.

*UNIX is a registered trademark of AT&T.

However, the problems of event synchronisation must still be dealt with in the framework. These are particularly severe in a networked environment, where network latency (round-trip response) may be unpredictable. A typical problem is 'mousing-ahead': a user requests a pop-up menu, for example, by pressing a mouse button, and then drags the mouse and releases the button on the menu item he wants. However, an expert user may know where the menu item is going to be, and is capable of releasing the button before the screen manager has had time to draw the menu (or, in NeWS, the menu process has had time to express an interest in input [NeWS87b p.50]). The danger is that the button release event will be wrongly despatched to the application under the menu, rather than to the menu process. A simple but effective mechanism to handle this is for the menu process to freeze input processing until it is sure the menu has been drawn. Both X ('synchronous mode' [MIT88(1) p.124]) and NeWS ('blockinputqueue' [NeWS87a p.21]) provide a mechanism to block input in this way. It is important to note that input events are not lost by blocking, just delayed in their despatch.

There is a converse problem, however. As Myers notes [Myers86a p.65] a *novice* user may be confused by network latency into thinking that the system simply has not responded (for example, to a mouse button push) and repeat the action. Contrary to what he expects, his input is queued, and he finds he has made multiple invocations of, say, some menu command. A simplistic solution is to flush the buffer (resulting effectively in a single-event record rather than a queue) or to allow events some limited lifetime [Tanner86 p.247]. Essentially, however, there is a conflict here between the needs of the novice and the expert user. Whereas the expert user needs to be guaranteed that his input is despatched to what he *predicts* is its target (a soon-to-pop-up menu, for example), the novice user needs to be guaranteed that events are despatched to what he *sees* is their target (for example, a plain background against which he has no means of knowing that a menu is about to pop up). This is not easily resolved, and seems to be a matter of case by case tactics rather than an overall strategy.

A more general event synchronisation problem is the ability to synchronise multiple devices in the interpretation of logical input events [Hill87b, Tanner87, Buxton86]. Tanner [Tanner86 p.246] distinguishes between 'simultaneous input' (multiple devices, multiple tasks) and 'user's choice input' (multiple devices, single task). These categories, however, assume only one device per task at any one time. Clearly, as in other physical input systems like cars, several devices may share one

task simultaneously. For example, a configuration may enable the user to draw a line using the mouse in one hand, whilst at the same time controlling the width of the line using a touch tablet with his other hand [Hill86 p.195]. The capability of handling input from multiple devices is also highly relevant to Supervisory and Control systems, in which input may arrive not only from a human user, but also from sensors in the system being controlled [Alty87 p.1008]. Tanner [Tanner86 p.247], conversely, discusses the sharing of a single device by multiple processes. Salmon and Slater [Salmon87 p.263] give an example of this: a mouse button press might generate three events - a CHOICE that segment rotation is required, a PICK identifying the segment to be rotated, and a LOCATOR to specify the centre of rotation. Although a basic configuration like a mouse and a keyboard in itself represents a multiple device configuration, in the general case there is thus a need to connect arbitrary devices dynamically. There is also an increased need for synchronisation the more closely devices are associated with a common task.

A number of different schemes have been adopted to handle such device synchronisation. The mainstream graphics proposals of CORE and GKS addressed multiple devices in their full generality. In both these proposals all device events are placed on the input queue marked with the trigger process which occasioned them. In CORE, simultaneously occurring events (i.e. those with the same trigger) were packaged together as a single compound event [Rosenthal82 p.37], whereas in GKS the application is allowed to 'INQUIRE MORE SIMULTANEOUS EVENTS'. Salmon and Slater [Salmon87 p.262] point out, however, that most implementations of GKS do not implement the full level *c* input (sample and event modes). In contrast, recent workstation-based systems have, perhaps unwisely, exploited their limited range of input devices by including the total device state, along with a timestamp, in each event. CSI [Williams87 p.6] and NeWS [NeWS87a p.110] do this, for example. X and VGTS [Lantz84] have less general schemes. X divides events into a number of different classes (e.g. 'KeyPress', 'MotionNotify'), only some of which (but including the standard device events) have timestamps. At the same time, the X server maintains a number of global time values for recent events, for example last keyboard or last pointer grab. All X mouse and keyboard events include the mouse location, but not all appear to give the total key state. VGTS distinguishes keyboard, mouse, and pick events [Lantz84 p.32]. It is thus not clear how easy it is to perform arbitrary device synchronisation in X or VGTS, for example to register key chord events. NeWS's event mechanism is cleaner, at the risk of including more redundant information in each event.

Event Despatching

Input routing also requires events to be despatched to the processes at which the user has directed them. In window systems screen objects are usually used to multiplex input. That is, events occurring in a window are sent to the process owning the window. However, it is not necessarily evident *which* window is receiving input.

There are primarily two options: spatial and state-based (Schiefler and Gettys [Scheifler86 p.101] call these *real estate* and *listener*). Using a spatial criterion, keyboard input, for example, is directed to the queue associated with the window currently containing the mouse cursor. In the state-based approach, on the other hand, a particular window is designated *current*, (or *active* or in *focus*) and all keyboard input is directed to it until its state changes. At the user interface this obviously requires some echoing of the state of 'currentness' - often this is by highlighting the window borders or title bar. Tanner's Switchboard [Tanner86 p.245], for example, displays a keyboard icon in the current window. 'Currentness' also requires a method by which the user can assign this state - for example by clicking the mouse in the window he wishes to make current.

However, in a hierarchical window environment several windows may be layered underneath the cursor. This is particularly a problem in object-oriented or object-based (iconic) systems where the screen object/process granularity is high, such that on the normal spatial criterion there may be several candidate processes for a particular mouse event. Where the management system cannot decide on behalf of processes (perhaps on the basis of some expressed 'interest' [Lantz87b p.90]) if they are to receive the event or not, this contention can be resolved in three ways:

- A process (say the one owning the window 'in front') can be allowed to look at the input first and then pass it on to the window 'behind' if it is not interested (as in NeWS). In a hierarchical system this is usually a parent window. Rosenthal [Rosenthal83 p.44] calls this 'passing the buck' up the hierarchy. This only works if there is a strict geometric nesting of child windows in their parent.

- An input event can be distributed to all candidate processes at once (as in X, TheWA [Lantz87b p.90], or the Local Event Broadcast Method of the Sassafras UIMS [Hill86 p.187]), on the expectation that only the interested process will take action.

• An input event can go first to the root of the visual hierarchy, and then be passed down to ever smaller objects until one accepts the event. Rosenthal [Rosenthal83 p.44] calls this the 'I'll handle it' method. The Andrew Toolkit uses this, and calls it the 'parental authority' concept. The justification is that it is often 'wider' objects which need to arbitrate the behaviour of component objects.

A process may also need to divert input to itself which would not normally be directed to it. There are three main reasons for allowing this preferential access:

• A process may wish to 'lie in wait' for a particular input event (a function button or mouse button press, for example).

• Processes with urgent business may wish to force the user to pay attention (for example to a 'modal' dialogue box) by blocking input to any other process.

• A process may wish to implement a manipulation *mode* in which all mouse input is directed to it while (and irrespective of where) the mouse is dragged.

Popping a menu is an example of a manipulation mode. A pop-up menu may need to appear when a mouse button is pushed, and disappear when (and wherever) it is released. The release event must therefore be despatched to the menu process, even if the mouse is currently over some other object. This input access contention between processes can be solved by imposing a priority.

There are a variety of prioritising schemes. In X and CSI, any process is allowed to 'grab' all mouse or keyboard input. X also calls this an 'active' grab, and in addition allows a 'passive' grab, where only a specific set of keys is grabbed. X can thus also allow a process to lie in wait for a particular input event. X11 in fact imposes a mouse button grab automatically during mouse drag events (i.e. between press and release of a mouse button) [MIT88]. This is needed in order to implement a manipulation mode. Since the grabbing process preempts all others until it explicitly drops the grab, these servers have only a two-level priority for input despatching. NeWS, on the other hand, has a more complex prioritising scheme. Each canvas has a prioritised list of processes interested in input events occurring within it, while in addition there exists a 'global interest list' containing processes interested in the whole screen. A process can only express one interest at a time. In order to grab input, therefore, a process can either put itself on the global interest list, or draw an

'overlay canvas' in front of all other windows. In the second case, it may be preempted by another process which overdraws it with yet another overlay canvas.

Input to a process may come not only from the user, but also from another process. This may be via an operating system signal, or by a change in an active value [Szekely88a p.37] or other message in an object-oriented system. In general, again, the event is the most useful mode. In systems with a high process granularity, like Smalltalk and its MVC paradigm [Burbeck87], the Sassafras UIMS [Hill86], Beach's 'anthropomorphic' paint program on the Thoth operating system [Beach82], Tanner's Switchboard system on Thoth's successor Harmony [Tanner86], or Lantz et al's proposals for a Workstation Agent [Lantz87b p.91], an important function of this input will be for process synchronisation. All these systems use a simple FIFO queuing mechanism to resolve contention between parallel events. That is, input events to a process converge on a single queue, and are thus automatically sorted by time of arrival.

As Borning points out [Borning86 p.365], however, this is a coarse mechanism, since processes generating the events have little control over when they may be received. He gives as an example the problem of *interleaving* the update of two different views of a model. In MVC this would not be possible, since there is no way that the sending process (the model) can synchronise the behaviour of two separate view processes. It is also useful to be able to generate what Lantz calls 'out-of-band' events [Lantz87b p.91] in order, for example, to abort some previously entered sequence of inputs. If this is not to be handled by process interrupts, there must be some mechanism whereby such an event can jump to the head of the queue.

Both Borning's Animus system [Borning86], and NeWS, allow processes generating events to specify their timestamp, rather than leaving it to the system. The input queues are then ordered by this timestamp (and will thus not necessarily be FIFO). All events are guaranteed not to be despatched before their marked timestamp. Borning regards this mechanism as a temporal *constraint*. Processes have thus finer control over the despatch of their events, and there may be arbitrary interleaving and (pseudo) concurrency among events (many events can be given the same timestamp). Processes can even send themselves future-dated events, thus emulating system-generated timer events. NeWS in addition can emulate both synchronous (directed) and asynchronous (broadcast) message sending, simply by specifying (or not) the receiving canvas.

2.3. Interaction and Semantics

Interaction essentially consists of alternations of input and output between the user and the computer. As an introduction to architectures which subsume both input and output, we consider in general the ways in which input and output can be linked.

While many writers see input and output as separate languages (see [Rosenthal80 p.361, Foley74 p.465, Green86 p.251]), it is clear there must be some link [Olsen85b] between the two in order for *transactions* to take place between the computer and the user. In general, users impute *semantics* to precisely the transformations that occur *between* input and output.

2.3.1. Feedback

Feedback is the most fundamental linkage between input and output. All interaction can be seen in terms of feedback. Feedback, however, can occur at a number of levels. It can thus be used to 'short-circuit' [Lantz84 p.29] input and output before application involvement. We can categorise these levels linguistically into lexical, syntactic, and semantic feedback.

Lexical Feedback

Lexical feedback consists of low-level *echoing* of input events for the purpose of informing the user that input has registered. It has a high granularity, for example individual keystrokes or mouse movements. Ideally it should also be highly responsive.

Lexical feedback is typically *incremental*. That is, it occurs in a context which does not change. The echoing of characters to the screen, for example, is not simply a function from keypresses (K) to screens (S):

$$K \rightarrow S$$

(except in the most simple calculators), but a function from *sequences* of keypresses to screens:

$$seq\ K \rightarrow S$$

This is because we expect keypresses to be echoed by characters that are *appended* to the text, or, in the case of *delete* characters, that remove the previous

character [Shaw83 p.102]. Thus in order to generate this feedback either the history of characters entered, or (equivalently) the screen state, must be retained.

Syntactic Feedback

Syntactic feedback reinforces the current state of the interactive dialogue. That is, it enables the user to predict the effect of his next action. A caret in text, for example, indicates (or should indicate) where the next keyed character will be inserted. Similarly, with the mouse button pressed over it, a menu item may be highlighted as a *potential* selection if the button were to be released.

Syntactic feedback may also be used to report syntactic errors, or to give the user an opportunity to amend, withdraw or cancel inputs at an intermediate stage of the dialogue. A spelling error may be corrected in a command line before pushing return, for example, or an icon moved away from the trashcan before releasing the mouse button, or the 'quit' button pushed in a dialogue box.

Semantic Feedback

The most problematic form of feedback in this consideration of architectures for separation is *semantic feedback*, that is, feedback from the deepest level of functionality. Here we may be satisfied with a *single* response, for example a document emerging from the printer. This is the *result* of the computation. However, in order to achieve this result we may need to go through a number of interactions in order to query or update computer objects.

Therefore there is a need to present to the user not only the application functions available to him, but also an appropriate visualisation of the state of his data, and how this affects the availability and progress of the functions. In this way, the user's actions can have immediate, incremental consequences [Shneiderman83] with no 'gulf of evaluation' [Hutchins86] which the user must bridge himself by reference to a complex conceptual model.

For example, functions might be presented as menus, while data might be visualised by text, or any of a wide domain of graphical representations. Unavailability of functions (because the data is in an inappropriate state, or there are type mismatches) could be signalled by shadowing the menu items (as on the Mac), or by more elaborate mechanisms, for example having buttons 'fall through' inappropriate objects (as in ARK [Smith87 p.62]). Progress of functions can either be represented

by continual incremental updates of the data representation (for example, the 'slow' global replace operation in the text editor spy [Jones-Ng86]), or by 'percent-done' indicators.

The consequence of these two requirements is that the operations available to the user (for example through the mouse) should provide not only lexical and syntactic feedback, but should also be incrementally sensitive to underlying application state. Myers [Myers87a p.132] gives examples of such semantic feedback in direct manipulation, from rubber lines 'snapping' to gravity points on an application-determined grid, to chess pieces which are directly manipulable only along their legal moves, or even only along their *possible* moves in the current state of the game. Young recognises that 'immediate semantic feedback' [Young88 p.368] must not be sacrificed in an interactive system. Hudson feels that providing semantic feedback at the lexical level is one of the major challenges UIMSs must face in order to support direct manipulation [Hudson87 p.122, Hudson90]. Lantz [Lantz87a p.41] and Myers [Myers88b p.2] echo this. Dance et al [Dance87 p.97] feel that separation has not worked in practice because of the problem of semantic feedback.

2.3.2. Directness

In addition to giving semantic feedback, interactive graphical systems should ideally also give the user the impression of manipulating his data objects *directly*, rather than manipulating syntactic agents (commands, icons, or menus) which do the job on his behalf. That is, the *same* object should be used for both input and output.

Mallgren notes [Mallgren83 p.27] that whereas in batch systems input may depend only on previous input, in interactive systems input may also depend on previous output. Even at the level of a framework, as we have seen, input may be interpreted in the context of some output configuration of the screen, for example to despatch input depending on which window the mouse cursor is currently in.

Directness thus has two fundamental requirements:

- The display medium must have *state* which persists over a number of cycles of input and output.

- The user must be able to *dereference* this state by addressing the displayed objects using a pointer like a mouse. This is conventionally called 'picking'. Picking is further examined in Chapters 4 and 5.

Without directness the user is reduced to making *symbolic* references to application state, such as typing command names on a glass teletype.

2.3.3. Semantic Perspectives

As a preparation for the more formal parts of this Thesis, we note that there are two fundamental perspectives we can take of the semantics of an application.

- Extensional: we can define the *behaviour* of an application in terms of the sequences of operations which it accepts.

- Intensional: we can define the *result* of an application in terms of changes to its state produced by its operations.

Chapter 3 will show formally that these two definition methods are equivalent. There are however subtle differences of emphasis between the two. The first stresses *input*, and suggests that semantics is best expressed in terms of syntactic constraints on sequences of input. The second stresses *output*, and suggests that semantics is best expressed by defining the result of the operations.

We go on to consider two classes of architecture which match these two perspectives: linguistic architectures which exploit *dialogue* abstraction, and agent architectures which exploit *device* abstraction. In particular we examine how both architectures cope with directness.

2.4. Linguistic Architectures

Linguistic architectures see interaction as layered, comprising at least lexical, syntactic, and semantic levels. Separation based on such linguistic divisions is closely associated with Foley [Foley80b, Foley80a, Foley84a p.220]. Moran's CLG [Moran81] and a protocol proposed by Nielsen [Nielsen86] have a similar linguistic structure. In addition to these layers there may be a 'conceptual' (Foley) or 'task' (Moran) layer at a more abstract level, while at a more concrete level there may be some consideration of what Buxton calls 'pragmatics' [Buxton83] - issues of device ergonomics that underlie the lexical layer. At the core of all these models, however, is the lexical/syntactic/semantic layering.

User Interface Management Systems

The UIMS model, particularly in its Seeheim formulation [Olsen85a, Green85b] of presentation, dialogue, and application linkage components, is the clearest example of a linguistic architecture. The most concise characterisation of a UIMS is that it *implements* a user interface [Shaw83 p.101, Cockton88b p.510], that is, a UIMS provides a formalism for user interface syntax that can be interpreted *separately* from application semantics.

Even as early as 1972, George's Meta system [George72] proposes a linguistically layered, prototypical UIMS which has a separable control language. However, it is at the 1982 Seattle workshop and the 1984 Seeheim workshop that the concept of a UIMS receives its fullest definition. While the term UIMS is sometimes used in a wider sense, we here use it in the precise sense of an interface system which separates and gives central importance to the syntactic, or dialogue, component [Pfaff85]. Typically, the linguistic architecture in its UIMS form is *monolithic* [Lantz87a p.39] - there is a *single* dialogue component.

2.4.1. Dialogue Abstraction

Chapter 3 will demonstrate formally the fundamental issues of dialogue *separation*, that is, the binding time of dialogue and functionality. In this section we examine the prior problem of dialogue *abstraction*.

Dialogue consists of sequences of both input and output events, and thus represents the observable *interaction* between user and computer. Two main formalisms are used to express allowable dialogue sequences: transition networks and grammars.

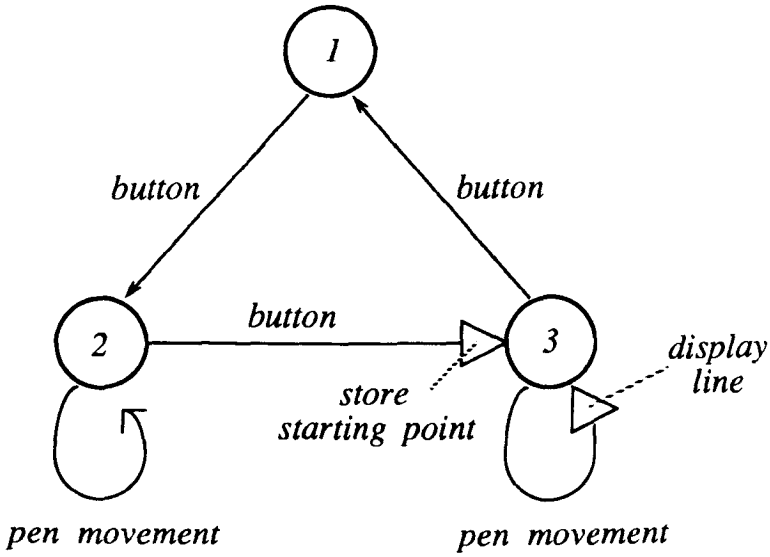
2.4.2. Transition Networks

Transition networks express allowable sequences of events by associating events with transitions between system states. In its simplest form a transition network is a finite state automaton and expresses a regular grammar, although in practice the formalism is often extended to give greater power. Transition networks are usually coded as a table of tuples of the general form:

$$State \times Input \rightarrow Newstate \times Output$$

Dialogue separation based on transition networks was first used by Newman in his early Reaction Handler [Newman68], examined by Foley and Wallace [Foley74], and taken up by Boullier et al in Metavisu [Boullier72 p.248], and by Wasserman in USE [Wasserman85].

Newman (op. cit. p.47) gives the following example of a transition network representation for the draughting task introduced above:



From an initial state, the user first presses the button to initiate the operation. In the second state the user can move the light pen around the screen until he decides on a starting point for the line. He then pushes the button again and changes to a state in which pen movements are continuously echoed by a rubber line whose endpoint is the pen position. On the final button push the line is fixed and the system returns to the initial state. Even this simple example illustrates the *moded* [Tesler81] nature of transition networks: in any state only a fixed number of transitions out are specified. The system response to user input that does not match these allowed transitions is undefined.

The transition network notation has been extended in three main ways.

- Large networks may be modularised by allowing labels on arcs to refer to separate networks: the labelled arc may be traversed only if there is a path through the associated subsidiary network. The label can thus be viewed as a non-terminal symbol in a grammar. If recursive labelling is allowed, then the network has the power of a context-free grammar [Jacob86b p.213].

• Transitions may be made to depend not simply on the current input token and state, but also on a global data structure. Transitions may enquire and update this structure. Woods [Woods70] calls these networks Augmented Transition Networks (ATN's). In general, an ATN has the power of a Turing machine (since any computable function can be applied to the data structure by a transition), and this has been exploited to enable the dialogue to encapsulate all application computation, as for example in Kamran's 'Abstract Interaction Handler' [Kamran83]. A more restricted form of ATN, the pushdown automaton, in which the data structure is limited to a stack, can implement recursion and therefore parse context-free grammars. Olsen in SYNGRAPH [Olsen84] and GRINS [Olsen85b] uses a form of these called 'interactive pushdown automata'.

• Local, independent transition networks may be embedded in a wider environment scheduled non-deterministically by input events. Jacob is most closely associated with this extension [Jacob86c, Jacob86a], but Coutaz in her PAC model [Coutaz87 p.434], Images [Simoes87], and Myers' Garnet [Myers89] have similar schemes.

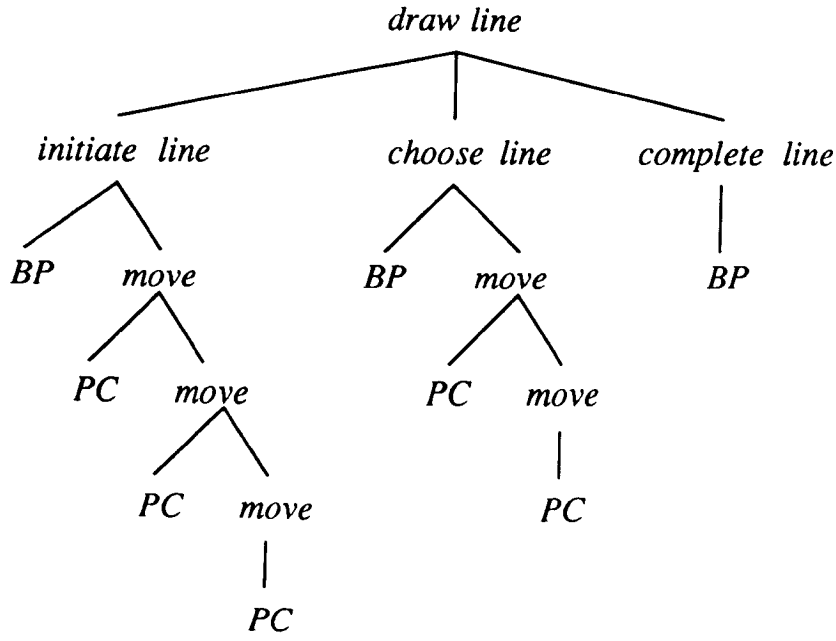
2.4.3. Grammars

Dialogue parsing on the basis of a grammar allows a *task* abstraction. That is, each of the symbols in the grammar can be associated with a task. Terminal symbols specify basic input tasks such as keystrokes, while non-terminal symbols express higher, logical tasks. The grammar determines the sequences of basic input symbols necessary in order to achieve a task.

The grammar is conventionally specified in a variant of BNF productions (as, for example, in SYNGRAPH [Olsen83a], or Reisner's ROBART languages [Reisner81]). Here is Newman's line drawing task expressed in Reisner's version of BNF (| is alternation, + is sequential concatenation. Terminal symbols are in upper case):

draw line ::= initiate line + choose line + complete line
initiate line ::= BUTTON PRESS + move cursor
choose line ::= BUTTON PRESS + move cursor
complete line ::= BUTTON PRESS
move cursor ::= POSITION CURSOR | POSITION CURSOR + move cursor

This grammar thus expresses a hierarchical breakdown of the task, from the top-level 'draw line' to the terminal lexemes like 'BUTTON PRESS'. A parse tree for a particular sequence of user actions which results in the drawing of a line could therefore be (using BP and PC for BUTTON PRESS and POSITION CURSOR respectively):



Thus this parse represents the sequence of actions:

<*BP, PC, PC, PC, BP, PC, PC, BP*>

This shows the sequence of actions: a button press to indicate the intention to draw a line, a drag of the cursor to the start point of the line, indicated by a second button press, a drag of the cursor (followed by a rubber line) to the end point of the line, terminated by a third button press.

Since this grammar is equivalent to the state transition machine given above, it is *regular* - that is, its productions expand from one end only (in this case the right). A higher grammar, however, for example a context-free grammar in which the productions expand from the centre outwards, can easily model nested sequences of actions.

Note, however, that this grammar generates only sequences of *input* actions. In order to specify output, output echoing can be incorporated in the productions. For example, POSITION CURSOR could be expanded to consist of an input of the

mouse position, followed by an output of the cursor at that position. However, the semantics of output rapidly reaches a complexity which a context-free grammar is not capable of expressing, especially if picking is required. In spite of the loaded symbol names in this example, there is no semantics here - this could just as well result in a circle being drawn as a line.

In order to express more general semantics, the grammar must essentially be 'attributed'. In SYNGRAPH, for example, Pascal procedures are inserted into the productions of the grammar to perform the semantic operations [Olsen83a p.50] like drawing a line between the start point and the current position of the cursor. Liere and Hagen [Liere87] also note the need for an attributed, and therefore at least context-sensitive, grammar in order to incorporate the semantics of the task.

2.4.4. Problems of Dialogue Abstraction

Semantic Feedback

Under the impact of the direct manipulation style, a number of fundamental problems with dialogue formalisms have come to light. Transition networks suffer from a quadratic growth in the number of possible transitions as the number of states increases. This is a severe problem in graphical interfaces, where significant state distinctions may depend on incremental graphical changes such as moving an icon to a new location. This is compounded by the fact that the number of screen objects may vary dynamically (see Sibert et al [Sibert85 p.186]). In a typical direct manipulation interface, therefore, the overall state space may be enormous. It is generally agreed that a higher than regular grammar is required to abstract and modularise the dialogue in such interfaces.

The basic state transition approach is also incapable of handling call/return sequences (i.e. nested states): as outlined above, a labelling mechanism or push-down state is at least required for this. As Newman points out [Newman68 p.48], this deficiency means that common semantic functions cannot be abstracted (for example as a sub-network) and invoked by any of a number of actions. Such a capability is needed to define globally-accessible user actions such as abort, help and undo. Kasik also notes the difficulty of doing this [Kasik89 p.57].

Various extensions have been proposed to handle these actions. Olsen's SYNGRAPH has the notion of distinguished 'escape' and 'reenter' states for each

nonterminal, which he calls 'pervasive' states [Olsen84 p.182]. For example, a task is aborted via the escape state, while help might be invoked at any time via an escape and then a reenter state. Help is, of course, in addition context-sensitive. Equivalently, but more generally, Cockton [Cockton88b] proposes 'Generative Transition Networks' by which transitions can be defined over sets of states, rather than state-by-state as in the standard notation. Thus an abort or help transition can easily be defined for *all* states.

Abort, undo, and general syntactic error recovery present special problems related to the parsing algorithm used. A top-down parsing algorithm commits the dialogue to a task as soon as the first possible input symbol for that task is received. The only solution for a subsequent abort, undo, or illegal input may be to cancel the parse. This may be difficult if output from the task such as prompting has already taken place. On the other hand, a bottom-up parsing algorithm may be easier to backtrack, but provides poor intermediate feedback, since the task may not be invoked until the whole input sequence is complete.

A bottom-up algorithm may be acceptable in a textual interface, where no action might be expected until the entire command string is typed and despatched (by pushing 'return'). A direct manipulation interface, on the other hand, requires a top-down algorithm, both because users expect incremental feedback of their actions [Shneiderman83], and because a graphical screen retains no unambiguous trace of user actions (such as a command line does), over which a parser could backtrack. Green [Green86 p.252] Bos [Bos80 p.167] and Kamran [Kamran85 p.46] all examine this problem. It is also interesting to note that as early as the Seillac II conference, Alan Kay was able to report [Guedj80 p.22] that experience with the well-used (textual) learning system PLATO had shown that error handling and back-tracking took up most of the interaction, and that finite-state grammars were unable to cope with this dialogue.

Finally, as Reisner points out [Reisner81 p.237], not all syntactically correct dialogues allowed by a grammar-driven parser may be legal in terms of the underlying task. That is, there may be *semantic* (contextual) errors not trapped by the dialogue parser. There are two approaches that may be taken in this case. In one, errors may simply be allowed to pass through to the application task, which may then need to instigate a special dialogue with the user in order to correct them. This strategy is adopted by GWUIMS [Sibert86 p.262]. An alternative approach is to allow the dialogue knowledge of or communication with the application task. For

example, parameter types may be declared in advance against which the dialogue can check input, or enquiry operations may be allowed on the task state, as in MIKE [Olsen86].

This is a particular instance of the general problem of incremental semantic feedback. Directness, in the definition of this Thesis, also requires that the output of semantic feedback be reusable as dialogue input. This is difficult if the dialogue is separated from the application semantics.

Multi-Threading

The ability to interrupt a task, get help or other information, and then return to the task at the point where it was interrupted is simply a particular case of the general need to run multiple tasks concurrently. This need is especially high in systems with interactive graphical, and particularly window-managed, interfaces. The problem from the point of view of a monolithic dialogue parsing system like a standard UIMS is that input destined for the various tasks arrives arbitrarily interleaved: the user may type a few characters in one window, move an icon against the background, then draw a line in another window. To handle this in a *single* parse a grammar must be evolved whose states (or symbols) is the Cartesian product of the states of each of the tasks.

Some systems handle this interleaving complexity by disallowing it. For example, SYNGRAPH, like GKS REQUEST input, is highly moded: physical and virtual input devices are dynamically 'acquired', 'enabled', and prioritised so that inputs are delivered only in the expected contexts. A single thread of control is therefore forced on the user. The need to cater for arbitrarily *multi-threaded* dialogues, and the inadequacy of formal grammars for this, was recognised early by Alan Shaw [Shaw80 p.378] and Anson [Anson80 p.123] (their comments even predate the flourishing of the UIMS model). Mary Shaw [Shaw83 p.107] uses the phrase 'data-driven' to convey similarly the notion of the user's freedom to update any visible data, as opposed to a 'control-driven' model where the order of updates is determined by the program. More recently, there has been a revival of interest in the handling of multi-threaded dialogues. The fundamental perception is of the user as a realtime system - asynchronous and unpredictable [Tanner86 p.248] - and that therefore interaction should be treated as a problem in parallel computation [Mallgren83 p.185].

Determinism

Syntactic dialogue parsing suffers two further fundamental objections. Firstly, whereas some use of formal grammars in parsing is for descriptive and analytical purposes [Reisner81, Payne84, Moran81], current use in UIMS is *prescriptive*. That is, the grammar *determines* the acceptable input and output sequences. We can distinguish between this problem and the problem of multi-threading: in the latter a grammar restricts the number of alternative concurrent dialogues, in the former a grammar restricts the number of alternative sequences in the same dialogue. Kamran [Kamran85 p.47], for example, admits that the Interaction Language of his AIH permits only a rigid sequencing of actions, and that more flexibility is required. This problem is more severe the higher the grammar, since as grammars become more context-sensitive the tasks they model become more *moded.*

Chapter 3 argues that there are two cases where dialogue determination is necessary or useful:

- when there is a necessary sequencing in the operations provided by the functionality, for example non-commutative operations like pushing and then popping an empty stack, or logging in and then opening a file.

- when one of the participants in the dialogue cannot be expected to be responsible for its actions, for example a novice user who does not know that exiting from an editor does not automatically save his edited file, or a nuclear reactor that does not 'know' that raising its damping rods and voiding its coolant would result in a melt-down. In these cases it is useful to impose temporal or logical constraints on the possible traces of the functionality for the good of the user. However, it is not clear that a grammar is the best formalism for doing this.

Chapter 3 also shows that there are cases where dialogue determination is unnecessary, for example in the ordering of parameters to an operation. A just balancing of these factors should result in what Thimbleby [Thimbleby80] calls a *well-determined* dialogue.

Practical Experience

Parsing human-computer dialogue according to a grammar, therefore, has a number of theoretical drawbacks. In practice, also, experience of using grammars has

not been positive. Two complaints are voiced. Firstly, *specifying* the dialogue in a separate language or formalism from the application functionality is often difficult [Myers88b p.17, Myers87a p.130, Olsen86 p.320]. SYNGRAPH was not widely used for this reason [Olsen87a p.135]. The only real solution is to generate the dialogue automatically. Green [Green87 p.114] proposes this, but there are few prototypes [Myers88b p.15]. Secondly, *parsing* user input according to the grammar often presents problems. Hekmatpour and Woodman complain of this [Hekmatpour87 p.7].

In recent papers, Olsen, Hudson, and Hill have strongly criticised the syntactic approach to dialogue. Olsen [Olsen87a p.135] thinks that ease of use is often more critical to the success of a UIMS than syntactic capability. Having used syntactic dialogue parsing in the SYNGRAPH and GRINS, Olsen's recent system, MIKE [Olsen86 p.320], abandons the syntactic component. Coutaz similarly abandons the single dialogue component in her PAC model [Coutaz87]. Hudson [Hudson87 p.121] views syntactic input as reducing 'engagement' in a direct manipulation system, since the user is communicating with the system rather than with the objects of interest, and concludes that syntax should be minimised. Hill [Hill87b p.118] regards the parser-based approach as "clumsy and awkward", and argues for a user interface specification language with programming power. This requires that the user interface system *run* the interface specification, as in the Blit [Pike84, Pike85], NeWS [NeWS87a], and CLAM [Call87]. Downloading user interface programs in this way, however, factors execution but not programming costs.

We conclude that syntactic dialogue specification fulfills neither the requirements for separation, nor the original goals of UIMSs. Nevertheless, there may be some more restricted domain in which syntactic specification is useful. Chapter 3 isolates this domain precisely.

In 1982 the perceived benefits of a UIMS were device exploitation for human factors optimisation, cost savings, reliability, interface consistency, rapid specialist prototyping, adaptability, extensibility, portability, and ease of debugging [Thomas83 p.7]. Unfortunately, by the time of the 1986 Seattle workshop it was possible to say (in the chairman's introduction) that: "it was not clear that the UIMS concept or structure was still valid after four years" [Olsen87b p.71]. There appear to be two main reasons for this lack of success: the impact of the direct manipulation style (in particular the problems of semantic feedback and multi-threaded dia-

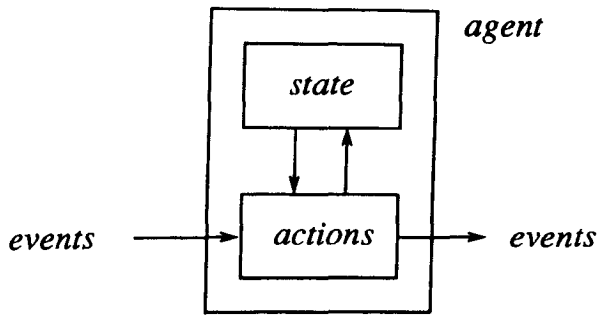
logues), and the difficulty of using the UIMS's formalisms. Most commercial UIMS have in fact been textual, at least in input [Kasik89 p.56, Prime90].

2.5. Agent Architectures

If a separate dialogue parser, as in a UIMS, is not used, then it is up to the application to interpret the sequence of input events. A more recent approach to the problem of coding large complex dialogues *within* the application is to fragment the application into specialised *agents* each of which manages a relatively simple dialogue. The agents communicate and cooperate to achieve the application task. Dialogue control is thus distributed [Coutaz89a p.11] among the agents, and as a result is minimally determined. That is, dialogue with a number of different agents can be interleaved by the user arbitrarily.

Agents are a very general architectural paradigm, and do not in themselves represent a solution to the problems of separation. Thus there are a number of different models that have been supported on top. Indeed, the syntactic UIMS itself can be seen as a monolithic agent. However, the tendency is to regard agents as medium or small scale objects which coexist in teams.

In general, agents encapsulate any functionality, including possibly input and output handling. They do this by maintaining their own state, and so are formally equivalent to Objects. We adopt Sugaya's diagram [Sugaya84] as a canonical model:



Hurley and Sibert give a slightly more detailed model (CREASE) [Hurley89].

Agents are scheduled by events. Events can be construed as input tokens, or as messages from other agents. We simply assume the agent 'fires' when events

are available which match its input rule. The temporal behaviour of an agent can thus be modelled by a process in CSP [Hoare85]. The input rule may require the tokens or messages making up the event to be single or multiple, ordered or not, depending on the agent.

A basic formalism for events and agents is a set of <input, action> pairs [Shaw83 p.107] (Chapter 3 gives a more precise formulation of this in terms of a *behaviour* function which maps input to state transitions). Agents can thus be viewed as event-handlers. Green [Green86] shows that this is greater in expressive power than either state-transition networks or context-free grammars (in fact, it has Turing power). This is because the event handlers are allowed programming constructs over state. There is thus little or no notion of a syntax over the events themselves other than what might be imposed by individual event handlers - events are simply despatched as they arrive to matching handlers. This is what gives the model its flexibility and frees it from the restrictions of syntactic parsing.

In this form, an agent can be expressed as a *production system* [Hopgood80]. In contrast to a formal grammar, symbols on the left hand side of productions used in agents are typically input tokens rather than task abstractions. Hopgood and Duce, for example, give a production system for Newman's line drawing task (although they suppose three buttons rather than one) (op. cit. p. 250):

B1		→	*<enable tracking device>*
X		→	*<display cursor>*
B2		→	*<store start point>* *S*
S	*X*	→	*<display rubber band line>* *S*
B3		→	*<store end point>*

In this system, productions (held in Long Term Memory (*LTM*)) are invoked on each time interval if the events to the left of the arrow are present in Short Term Memory (*STM*). The events are not ordered. *X* is the position of the cursor, which is generated on each time interval. *B1*, *B2*, and *B3* are button events (presumably generated by different buttons). The *S* event after the action specification is generated by the rule and written back to *STM*. Events are consumed on each time interval, but may match more than one rule. This formalism is thus more expressive than either the transition network or BNF grammar given above. For example, if *S* and *X* are in *STM* then both the second and the fourth rules are satisfied. The formalism is also less moded, in that it does not determine the order of some sets of events. For

example, the end point could be given before the start point (by pressing *B3* before *B2*). A refinement of the production system does impose an ordering on the production rules in *LTM*, which reduces the ambiguity but increases the modedness.

The power of the notation is mainly exhibited in the ease with which systems can be combined. For example, if a similar production system were defined for another set of buttons (say, *B4, B5, B6*), then the two sets of productions could simply be combined to implement a system which allowed the user arbitrarily to interleave the drawing of two lines. To give the same power using a state-transition network would require many extra states for all the permutations of interleaving.

Green in the University of Alberta UIMS [Green85a], Cardelli and Pike (with Squeak) [Cardelli85], Tanner [Tanner87], Hill [Hill87a], Olsen [Olsen90], and Lantz [Lantz87b, Lantz87c] have all produced systems or formalisms for handling interaction using events and agents, usually in the form of a production system. These, however, may be bound early to their functionality. Squeak, for example, is precompiled into C.

There are also some hybrids in which agents use state machines or grammars to maintain their individual input syntax independently of other agents. As noted above, Jacob [Jacob86c] expresses task syntax using state-transition networks, but his top-level input is event-driven. When an individual task is suspended (because the current input does not match any of its possible transitions) it maintains its state until control returns. The tasks thus behave as coroutines in which there is a single thread of control. Similarly Garnet [Myers89] has 'interactors' each of which runs a predefined state machine. On the other hand, Scott and Yap [Scott88] express task syntax as a context-free grammar, but allow parallel invocations of tasks.

A useful benefit of an agent architecture, exploited by Hill, is that it easily handles concurrent *multi-device* input. This is because events from a number of input devices can be interleaved and synchronised by monitoring agents. Green's more general agent model also allows event handlers to *generate* events, which brings it close to the communication model of NeWS processes and the object-oriented paradigm.

2.6. Refinements of the Agent Architecture

We consider ways in which the basic agent model has been exploited to produce architectures for separated user interfaces. We first introduce Toolkits, which are the most specific instantiation of an agent architecture, and then examine the abstractions upon which Toolkits have been based.

2.6.1. User Interface Toolkits

It is useful to distinguish user interface Toolkits from UIMSs, as do Lantz [Lantz87a p.39] and Myers [Myers88 p.1]. Toolkits aim to provide a pre-packaged set of useful tools (agents) to the interface designer. Toolkits are generally designed to take advantage of some underlying window system, and may extend or hide the facilities provided there. The NeWS Lite Toolkit [NeWS87a p.43], and the X [Swick88, Rao87] and Andrew [Palay88] Toolkits, for example, are designed to run on their respective window systems (although the Andrew Toolkit has now been ported to X). Recent window systems, in fact, are not intended to be immediately used by the interface designer, but to be a 'platform' or substrate for Toolkits [Williams87 p.2]. Rosenthal gives a good example of how difficult it may in fact be to write directly to the window manager without using a Toolkit [Rosenthal87].

A Toolkit may provide its own input framework (such as the X Intrinsics layer [Rao87 p.121]). The Andrew Toolkit framework [Palay88 p.13] and the InterViews framework [Linton87 p.261], for example, completely hide the underlying X Intrinsics framework.

Most Toolkits profess to be object-oriented, and so also allow some degree of customisation and composition of tools, although the ease with which this can be accomplished varies. Some Toolkits are concerned mainly with the mechanisms for creating and maintaining tools, but others concentrate on providing a set of predefined tools which may have their appearance bound in to their functionality [Swick88 p.227]. The X Toolkit, for example, distinguishes this as a set of 'widgets'. The X and Andrew Toolkits, Interviews [Linton87], MacApp [Schmucker86], Cardelli's Toolkit [Cardelli87], and Coral [Szekely88a], for example, all provide basic button, scrollbar, and menu tools. However, these and other basic tools will vary, across the different Toolkits, in their functionality, structure, composability, and conceptual integrity (see the comments in [Roberts88 p.279]).

2.6.2. Device Abstraction

Underlying Toolkits, and a major use of agent architectures, is *device* abstraction. In contrast to dialogue abstraction, the fundamental formalism for device abstraction is not syntax, but *type*. Physical devices, under some measure, deliver values of a type. However, it is possible to abstract from physical devices to *logical* devices, whose type may be more complex, for example even documents or databases. The measure in these cases may be a complex function dependent on a complex state, rather than a simple transducing function. Similarly, the triggering of logical devices may be the result of complex event synchronisations. Nevertheless, any logical device can be expressed using the basic agent model above.

An important refinement of the agent architecture in supporting such logical devices is that agents be *recursively composable*. That is, complex agents can be built out of simpler ones, to any level, such that at the top level the application itself is an agent (see [Bos83 p.91]). In implementation this simply requires that the output of one agent can form the input of another. This is minimally satisfied if the I/O protocols are the *same*, as for example with UNIX filters.

There is thus a correspondence between the grammar formalism, in which symbols expressed tasks, and devices, which represent the *result* of tasks. Just as the productions of a grammar express orderings of lower symbols, device values may consist of some structuring of values from lower devices. The correspondence is closer if the structure is given some temporal interpretation. Hagen's 'dialogue cells' merge the grammar and device formalisms in just this way [Hagen85].

2.6.3. Homogeneity

Systems based on an agent architecture can be *homogeneous* [Dance87 p.98], in which there is fundamentally only one sort of agent which may vary in its functional content and type. Other agent-based systems can be *heterogeneous*, in which there may be a variety of agents specialised for particular tasks.

Both object-oriented and actor [Agha85, Agha86] systems are basically homogeneous agent architectures, in that they do not in themselves determine the semantics of the objects. However, systems which attempt to separate user interface concerns by providing a set of predefined interface objects, like Toolkits, are heterogeneous. In heterogeneous agent-based systems there is often a linguistic

alignment, with at least a logical partition of agents into lexical, syntactic, and semantic classes.

For example, GWUIMS [Sibert86 p.259], Smalltalk's MVC [Burbeck87], Voodoo [Scofield85 p.56], EZWin [Lieberman85], Nephew [Szekely88b], Garnet [Myers89 p. 320], and the Sassafras UIMS [Hill86 p.187] are all heterogeneous, distinguishing different types of object which can be classified linguistically. GWUIMS, for example, has 'graphic', 'technique', and 'representation objects' (lexical/syntactic), 'interaction objects' (syntactic/semantic), and 'application objects' (semantic); correspondingly, MVC has view, control, and model objects; Voodoo has images, editors, and objects; EZWin has presentation, EZWin objects, and commands; Nephew has presenters, commands/recognisers, and models; Garnet has object-oriented 'views', interactors and Lisp code, and the Sassafras UIMS has modules which are specialised for I/O, dialogue control and application routines. In Coutaz' PAC system [Coutaz87 p.431] the linguistic levels are made explicit *within* each agent.

2.6.4. Logical Devices

We make a distinction (as does Rosenthal [Rosenthal82 p.34], but *not* Tanner [Tanner86 p.247]) between *virtual* and *logical* devices. A virtual device does not define any output, but simply composes input into some logical token. Examples might be the 'double click' event which Roberts et al [Roberts88] complain that X does not provide, or Squeak's E and L events (entering or leaving a rectangle) [Cardelli85 p.200]. A logical device, on the other hand, we take to include the production of output. This may be for prompts, echoing, and other feedback. In GKS, for example, devices in REQUEST mode can provide prompting output [Enderle84 p.275], although this is highly implementation dependent. The logical device is thus the more complete device abstraction.

Logical devices underlie the input classes of mainstream graphics: string, valuator, locator, pick, stroke and choice. Formulated first by Foley and Wallace [Foley74] and Wallace [Wallace76], and critically examined by Rosenthal [Rosenthal82, Rosenthal83 p.19] and Mallgren [Mallgren83 p.28], they form the basis for the input models of CORE, GKS, and PHIGS. They have in common the characteristic that they abstract a single value from a possibly complex set of user actions, and generate echoing and possibly prompting. However, in the standards

they are not composable or extensible (other than through the programming language in which they are embedded), but form a single layer of input primitives. For this reason, the abstractions they express should ideally be complete and orthogonal over the domain of interactive input. The differences between CORE, GKS, and PHIGS (and between these and other logical device layers - see [Kamran83 p.60] and [Kasik82 p.103]) suggest that this is not yet so. According to Rosenthal, the input devices mainstream graphics appear "either inadequate or inelegant when applied to interactive as opposed to passive graphics applications" [Rosenthal80 p.361]. This is echoed by Myers [Myers87a p.132], who claims that the input model of the graphics standards is inappropriate for direct manipulation interfaces. More recently, Duce et al [Duce90] have proposed a generalisation of the standard model which allows device composition.

A forerunner of the logical device is Newman's 'Reaction Handler' [Newman68]. In this system, reactions can be programmed to issue an immediate 'response' to user action, for example the prompt string "point at line to delete" when the user hits the 'delete' button. In addition to this, the reactions contain a procedure which typically handles feedback, such as redrawing a line. However, Newman's reactions are scheduled as transitions in a network. In addition, they are not composable: the group of reactions in the network only conceptually comprises the behaviour associated with a logical device, for example to maintain a rubber line. A very similar system is Metavisu [Boullier72] which again has 'reaction' statements scheduled in a network.

Classical logical device models which do allow composition are Anson's Device Model [Anson80, Anson82], and Bos' Input Tool [Bos80, Bos78] (or his later variant IOT [Bos83]). Other systems and architectures which can be thought of as being within the logical device model are Hopgood's adaptive productions [Hopgood80], Hill's event-response paradigm (with its extension, 'outgoing events') [Hill87a], Sugaya's Logical Device Modules [Sugaya84], Hagen's Dialogue Cells [Hagen85], and 'interaction techniques' [Foley84b, Dance87 p.99, Kamran83 p.58] (although Kamran's are expressed as part of an interpreted 'Interaction Language').

2.6.5. Object-Orientation

Object-orientation [Stroustrup88] extends the basic notion of a logical device which delivers a value of some type to include abstractions which declare instances of these (for example, classes in Smalltalk [Goldberg83]). In combination with inheritance, this allows abstract objects to be partially specified, and instances to be created which inherit some operations and variables but supply others more specific to their task. In terms of type, the classes represent supertypes, and the instances subtypes. Their relationship is such that a value of a subtype can be used wherever a value of the supertype can, since all the operations of the supertype should be inherited in the subtype.

There are a growing number of interface systems that are built on top of objects [Szekely88a, Coutaz87, Borning86, Linton87, Call87, Szekely87, Schmucker86, Crampton87, Barth86, Simoes87, Palay88, Sibert86, Lieberman85]. However, it is worth repeating Olsen's caveat [Olsen87a p.134] (echoed by Cockton [Cockton88a p.18]) that object-orientation, like agents, is a very general model, and in itself sheds no light on the particular problems of interface separation.

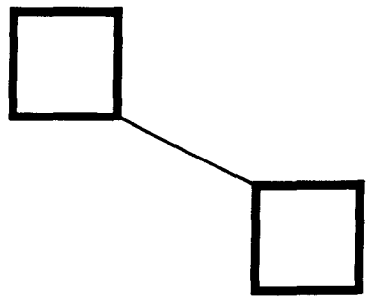
Object Customisation

The argument for the suitability of object-orientation for the construction of user interfaces rests on its inherent strategy of design by modification rather than creation [Nanard87 p.83, Scofield85 p.156]. In user interfaces, it is argued, there can be a basic set of interactive classes (menus, scroll bars etc.) which need simply to be customised for a particular application.

However, with the trend towards visual programming [Cook88, Ingalls88, Myers86b, Reiser88, Myers88a, Chang86] and the *interactive* design of user interfaces [Myers87b, Cardelli87] there is a need to visualise such classes. A representative or default instance must therefore be created on the screen. The standard object-oriented model, however, does not support (in Stroustrup's sense [Stroustrup87 p.162]) instance *specialisation*, that is, the ability to use an instance (with default values) as a *prototype* [Borning86 p.359] for a class of objects.

[Took90b] also argues that finding the optimum set of basic classes is a critical design problem in an object-oriented system, involving finding 'commonality' [Stroustrup87 p.165] in objects over the whole domain. Where this fails, the result is often an arbitrary collection of loosely distinguished classes. Toolkits often fall

into this category. On the other hand, where a true taxonomy has been achieved, as in GROW [Barth86], object instances may not be much use on their own. GROW, for example, can provide class paths such as *Vector:Open:RightAngles* or *Vector:Closed:Box*. Coral [Szekely88a p.39] has a simpler taxonomy: *Graphical-Object:Line-GO* or *Graphical-Object:Rectangle-GO*, for example. In order to construct useful interactive objects, it is necessary to build these primitive instances into *compound* objects [Swick88 p.227]. Consider the compound visual object:



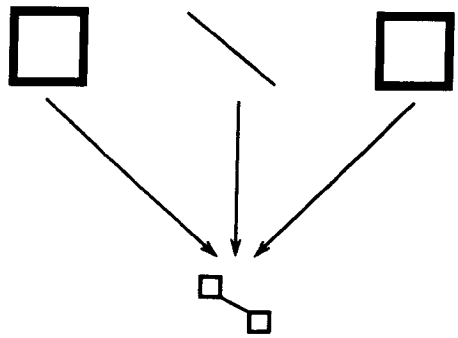
There are two strategies whereby such an object might be constructed from simpler objects such as squares and lines. We call these strategies object combination and object composition.

Object Combination

In single inheritance systems (where an instance can inherit from only one class), objects which have a number of distinct properties are awkward to implement. For example, a rectangle filled with text may have some properties relating to polygonality, and others relating to text handling. However, not all text need be displayed in a polygon, nor all polygons filled with text. These classes are therefore orthogonal. In a single inheritance system this can lead to code duplication, since either a text instance must be extended with a rectangular border, or a polygon instance extended with filled text, and either of these extensions might need to be performed on instances of different classes.

Within the object-oriented paradigm, therefore, construction of objects is more natural through *multiple* inheritance [Linton87 p.262] from a *number* of classes. The

compound object above, for example, could be constructed (with the *square* class inherited twice with different labels):



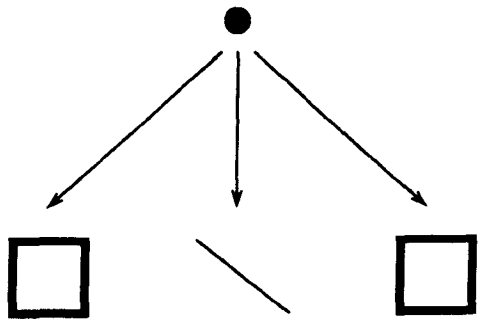
Object-oriented systems capable of multiple inheritance are becoming important in user interface implementations for this reason [Barth86, Szekely88a, Borning p375 81, Borning86].

Object combination through multiple inheritance has a number of problems, however. At a very fundamental level there are the possibilities of name clashes (if two inherited classes use the same name in different contexts) and aliasing (if two inherited classes themselves each inherit from the *same* class). More specifically, the *combined* object is a single object, rather than a structure. Thus, dynamic restructuring, for example to exchange a circle for a square, is impossible. Also, any inherited modularity breaks down. In the example, the implementations of the square and the line would each be visible from the other. Most importantly from the perspective of this Thesis, user interface issues like presentation are bound in to the semantics of the objects.

Object Composition

True multiple inheritance, however, is rarely supported (again in Stroustrup's sense). InterViews [Linton87 p.262], and the Andrew [Palay88 p.14] and X [Swick88 p.227] Toolkits, are all based on taxonomic (single inheritance) systems, for example.

An alternative mechanism for building compound objects on single inheritance systems is object *composition*. This is essentially the creation of a *data structure* of component objects. Thus the compound object above could be created:



The structure is maintained by a super-object which holds pointers to, or slots for, the component objects. Objects can thus be dynamically created or deleted. The most common structure is a part-whole hierarchy, or tree structure, which can model a compound object which does not have cyclic dependencies. Nearly all visualisable objects fall into this category.

Object composition is supported in GROW [Barth86 p.151] and ThingLab [Borning81]. Similarly Coral [Szekely88a] provides 'aggregates', and the X Toolkit provides 'forms' [Swick88 p.225].

Object Dependencies

Composite objects in user interfaces typically have geometric or textual dependencies between their states. For example, the appearance of a sub-object should bear a constant geometric relationship to its parent object. A scroll bar, for example, would be expected to remain at the side of a window when the window is moved. Thus primitive objects in Coral [Szekely88a] have a geometric specification (endpoints in class *Line-GO*, for example) which allows them to be positioned with respect to any composite object of which they are a part. Similarly, objects in InterViews [Linton87 p.256] can be composed geometrically using Knuth's more flexible 'box' and 'glue' model (see Section 5.4.2). In the same way, textual dependencies might govern the placement of sub-text, for example paragraphs or sections, within a wider document.

Geometric or textual dependency is a simple case of a *constraint* since the relationship between dependent interface objects is often constant over their lifetime.

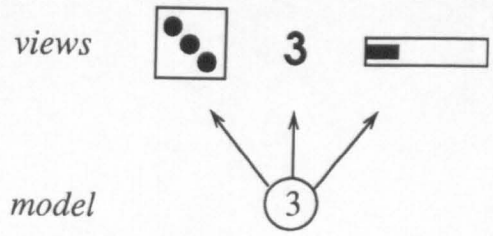
(Note, however, that not all constraint-based systems are object-oriented, for example Sketchpad [Sutherland63]). In general, however, as Borning notes [Borning81 p.356] there is a *tension* between the object-oriented (and agent) policy of encapsulation, and the need to apply state constraints between otherwise unrelated objects. In the linked box example above, the endpoints of the link are constrained to lie on corners of the boxes. While this constraint may be constant with respect to components like the boxes and the linking line, it will not necessarily be constant with respect to the *parent* space, since the boxes may move independently.

One strategy for resolving this tension, as represented by ThingLab and Animus, is to exploit part-whole composition to capture the dependencies. That is, inter-dependent objects can be incorporated as parts in a wider object, which itself holds the constraints on them (and the methods for solving them). The sub-objects are not accessed directly, but via a pathname starting with the enclosing object. This object can therefore monitor all access to its component objects, and apply the constraints. The strategy is analogous to the Andrew Toolkit's 'parental authority' (see Section 2.2.2). A restriction of this is the technique of 'merging' [Sutherland63 p.337] [Borning81 p.380], by which objects or parts of objects are constrained to be the same (for example, the endpoints of two lines), and thereafter referred to by the same pathname.

2.6.6. The Model-View Paradigm

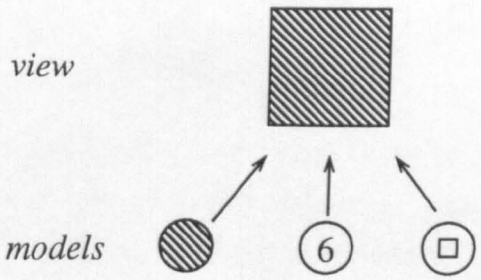
A disadvantage of this approach is that logically important objects (like the boxes and their content) may be bound in, and subordinate, to presentation objects (like the composite object representing the linked pair). A further refinement of the agent architecture seeks to separate logical objects from presentation objects. The most common terminology is *model* objects and *view* objects, respectively. In the model-view paradigm some agents represent the *model*, or application state, while others generate *views* on the model. The view agents may be parameterisable by style or format definitions, so that different visualisations can be applied to the same

state, simply by applying different views to the same model. The number three, for example, may be viewed as the face of a dice, a numeral, or a value on a sliding bar:



These views may in addition be presented concurrently, for example if the model support views distributed over a number of workstations.

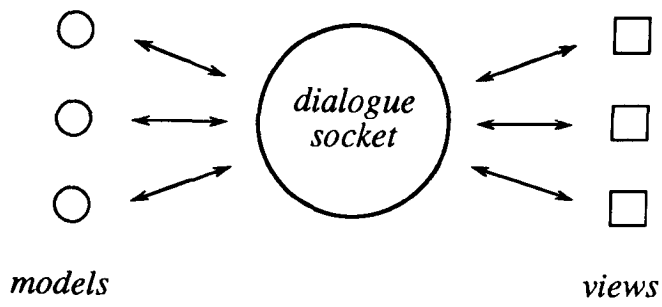
It is also conceivable that a single view have aspects which are controlled by a number of models. For example, the colour, size, and shape of a single view may depend on distinct models:



The values in these models may be determined by separate physical devices. Sassafras [Hill86], for example, allows the size and colour of a drawing stylus to be controlled concurrently by different devices.

Szekely's 'models' and 'presenters' [Szekely88b], Young's 'artists', which contain 'models' and 'views' [Young88], Linton's 'subjects' and 'views' [Linton87 p.256], the Andrew Toolkit's 'data objects' and 'views' [Palay88 p.11], Scofield's objects and images [Scofield85], and Ciccarelli's presentation and application databases [Ciccarelli85] typify the model-view paradigm. In theory, if the agents communicate by messages, then the model and the view can be late bound and therefore separable.

Coutaz exploits the two possible mappings between models and views above by proposing an intermediate component by which the models and the views can be mutually independent:



As originally proposed [Coutaz84a, Coutaz85, Coutaz86], the Dialogue Socket was a syntax-based interpreter in the manner of a UIMS. A more recent formulation [Coutaz90, Coutaz89a] plays down the syntactic component.

There are clearly state dependencies between model and view agents. It is usually the case that changes to the model cause changes to the view. There are two basic ways in which this can be implemented:

- The model can *maintain* the view. When the state of the model changes, it updates the view. This is called variously 'presentation' [Ciccarelli85], 'announcement' [Szekely87 p. 239], or a 'procedural API' [Coutaz90].

- The view can *monitor* the model. When the view is aware that the state of the model has changed, the view updates itself accordingly. This is variously called 'recognition' [Szekely87 p. 239, Ciccarelli85], or a 'declarative API' [Coutaz90].

Maintenance is the traditional method by which applications update their interfaces, and is an example of the internal control model (see Section 2.1.2). The disadvantage is that the interface objects are thereby bound in to the application, and different views cannot be applied dynamically. Monitoring, on the other hand, is an example of the external control model. It has the theoretical disadvantage that it is costly to expect the view explicitly to poll the model in the expectation of a change of state.

Access-Orientation

A mechanism by which model monitoring is provided as a service to views is *access-orientation* [Stefik86]. Here, variables can be 'annotated' with procedures or properties. They thus become *active values* [Stefik86, Myers87b p.55], in that when they are accessed or updated the associated procedures are called or properties re-evaluated automatically.

Myers refines this slightly so that active values keep lists of dependent objects rather than themselves encapsulating methods or procedures. Smalltalk's Model-View-Controller architecture [Burbeck87] similarly uses an explicit 'changed' message to inform the view(s) that the model has been updated. The MVC model must keep note of all dependent view objects (in a global dictionary) to which the 'changed' message must be sent. These views can then recalculate any applicable constraints. Active values are used in this way in Incense [Myers83], in Coral [Szekely88a p.37], in GWUIMS [Sibert86 p.262] (where they are called 'active containers'), in Nephew [Szekely88b p.50] (where they are called 'changes communication concepts'), and in Descartes [Shaw83 p.106] (where they are not named).

Active values have wide applicability in model-view architectures. They can be used to monitor program state for debugging purposes ('program visualisation' [Myers86b, Myers88a]), or to provide process monitoring interfaces, as in Stefik's gauges and control panels [Stefik86 p.14]. However, at a lower level they can also provide input transducing in a way closely analogous to logical devices. Here the *model* is the hardware-generated value of the input device, while the view that is activated is a measure of this. Myers notes this use [Myers87b p.55]. Turner [Turner84] similarly makes an early proposal for 'graphics variables' for languages like GKS and PHIGS, which extends the notion of logical device to something very close to active values.

If models and views can be composed hierarchically, such that a view at one level becomes a model for a higher view, then hierarchical control schemes, analogous to 'passing the buck' (see Section 2.2.2), can be constructed. These can be used to build agents which simply wait for their parameters to accumulate, as in MIKE [Olsen86], EZWin [Lieberman85], and Szekely's 'input gathering' mode [Szekely88b p.56], and in forms-based systems such as Cousin [Hayes84], and the Karlsruhe system [Bass85].

As in general constraint-based architectures, it is thus theoretically possible for model-view dependencies based on active values to be cyclic. In practice, however, this is rarely allowed. Stefik, for example, explicitly disallows the procedures annotated to active values from having interfering side effects [Stefik86 p.11]. Chiron [Young88 p.372] allows only hierarchical dependencies, in which complex agents receive events only from their component agents. The dependencies in GROW are similarly hierarchical and uni-directional [Barth86 p.152]. Coral's dependencies are uni-directional, but in contrast allow cycles [Szekely88a p.37]. However, constraint satisfaction here is not allowed to iterate around these cycles (op. cit. p. 44).

Using models and views hierarchically in this way, however, results in a construct very similar to a logical device in which the model forms the input processor, and the view the output processor. This makes separating models from views difficult. This is unfortunate, since the model/view distinction is a clear candidate for separation. For example, views can have part-whole dependencies which may be orthogonal to the logical dependencies of their associated models (c.f. the linked boxes above). This Thesis is predicated on the notion that views may even have *behaviour* which is orthogonal to their model.

A number of systems make a distinction between model and view structures, for example Smalltalk, Chiron, Incense (where views are 'artists'), Coral, Animus, Voodoo [Scofield85 p.115], Nephew [Szekely88b p.47], MIKE [Olsen86 p.327], and the Andrew Toolkit [Palay88 p.13]. At the top level, the view structure may simply be mapped directly to a window hierarchy, as in Xtk [Rao87 p.118].

An alternative mechanism to provide access-orientation is *daemons*. Daemons are attached to model operations, rather than to model values. When the operation is invoked, associated procedures are triggered to update the relevant view. Young's 'artists' [Young88 p.369] use a daemon-like mechanism to update the view as a side effect of each operation.

Directness in Model-View Architectures

The pure model-view architecture is uni-directional. Changes to the model are reflected in the view, but there is no inverse mapping (the model cannot be affected by the view). Such a system cannot be *direct*, in the sense defined in Chapter 1. Incense, for example, has this restriction [Myers83 p.123]. This is fine for program visualisation [Myers88a], process monitoring [Stefik86], animation [Borning86,

BrownMH88], and database presentation [Mackinlay86, Herot80], but not for interactive direct manipulation systems, where the user expects to control the model directly *through* the view (he may even believe the view *is* the model!). Directness would enable the user, for example, to change the model number in the illustration above by clicking on the dots on the dice, or dragging the bar. Lack of directness is a major limitation of access-orientation.

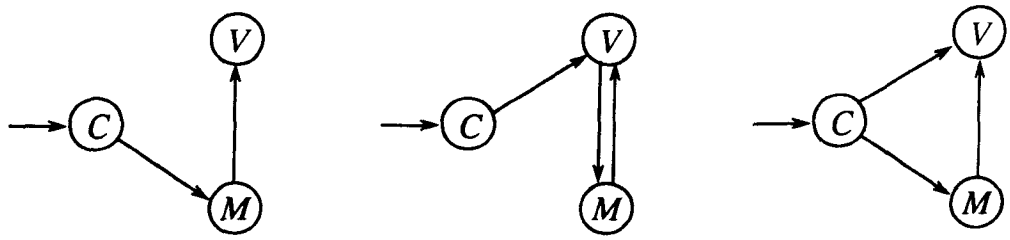
Directness therefore requires minimally that the model-view dependencies be *bi-directional*. Borning and Duisberg [Borning86 p.370] suggest a construct called a 'filter' which is a bi-directional constraint between a model and a view. Wills [Wills87b pp.10,14] has a notion of 'transformer' along similar lines. A transformer is not only a 'projection' function between model and view, but also a mapping between operations on the view, and operations on the model. That is, manipulating the view invokes the corresponding operations on the model. More formally, Harrison [Harrison90] expresses the viewing function as having a 'complementary algebra': in the same way as Will's transformers, operations on the view, via some parsing function, have corresponding operations on the model. These examples, however, are as yet paper models.

Mapping operations on the view to operations on the model assumes that, just as the model can have state, so can the view. In a multi-agent environment changes to view state are likely to be incremental. This suggests *editing* as a general paradigm for model-view interaction. Scofield's is the classic thesis in this respect [Scofield85]. In terms of text, the editing paradigm allows a model-view transformation (fonts, formatting, pagination etc.) as well as a view-model transformation (insert, delete etc.). Olsen claims that 85% of interaction is editing or browsing through some underlying data model [Olsen87a p.136]. Clearly other domains, such as graphics or databases, can, in a general sense, be edited [Wills87a p.34, Fraser80]. It simply requires that the operations of an abstract type be conceived as 'editing' operations. Thus one 'edits' a stack, for example, through the operations *pop* and *push*.

The problem of bi-directional views has also been addressed in the database domain [Claybrook85, Wiederhold86]. Views with state can similarly be used as a *database* for the application [Olsen87a p.135, Green87 p. 115]. Some representative examples are EZWin, in which EZWin objects hold application state [Lieberman85 p.182], Ciccarelli's presentation database [Ciccarelli85], Hudson's shared objects [Hudson87 p.123], GWUIMS 'A_objects' [Sibert86 p.262], frames in Zog and KMS

[Akscyn88], Cousin [Hayes83], and Higgens and DOMAIN/Dialogue [Dance87 p.99]. The danger here is a loss of data freedom: applications are constrained to a common object representation. Young criticises this approach, and gives the further example of structure editors like Mentor which impose this restriction [Young88p.368]. Alternatively, there is a risk of unnecessary duplication of data between model and view [Lantz84 p.29]. *Presenter* (Chapter 7) has been utilised for database presentation [Brown90].

Directness requires not only bi-directional model-view dependencies, but also some mechanism whereby raw user input can be mapped to operations on the view or on the model (for example, operations to position dots on the face of dice, or operations to increment or decrement a number). Where this mapping is performed is critical to the separation of interface and application agents. If input is routed raw by a framework, then the semantic interpretation of input is performed in either the model or the view. Some systems, however, distinguish a separate component, often called a *controller*, as in Smalltalk's MVC paradigm [Burbeck87]. To the extent which the controller performs this mapping, it is bound to the model or to the view. Three routing variants can be distinguished:



In the first, input is passed by the controller to the model. The model then updates the view (in which case the view is simply a projection and requires no state) or it sends a 'changed' message to the view to ask it to update itself, as in Smalltalk's MVC. In either case, in order to provide directness the model must be aware of, and thoroughly determine, the view, so that it can interpret the input in the view context (for example, as a click on the face of a dice). The view is thus bound early to the model, and so is difficult to separate. UIMSs, in which the dialogue component can be identified with the controller, typically use this variant.

In the second variant, it is the view which interprets user input. If the view updates itself on this basis then, in order to provide directness it must also be aware of model semantics, and thus be bound early to this. Toolkit objects (views) typical-

ly use this control variant. If the view does not update itself, but relies on the model to do this, then this variant degrades to the first.

In the last variant, the controller passes input both to the view and to the model. If in addition the controller *interprets* input as operations on the view and the model, then clearly the controller must know both about the view and about the model, and these two are bound early here. However, separation is possible if the view and the model interpret raw input independently. The *UMA* architecture presented in Chapter 4 is a refinement of this third variant which provides both for directness and separation as a consequence of *Surface Interaction.*

2.6.7. Problems of Model-View Separation

In any non-trivial application the model state may be large and structured. It is likely to be optimised for its logical content, rather than for any concrete view. For example, a network may be represented by storage structures interconnected using pointers. Possible screen views may vary enormously in appearance and layout - they may be tabular as well as graphical. Therefore there is not likely to be a clear mapping between components of the model state and the view. As Szekely [Szekely87 p.240] points out, this is a major failing of access-orientation as a mechanism for prompting view updates: in a complex model data structure there may be no single variable which can be isolated as containing the 'value' which is to be viewed. The view might be a complex function of the model, and updating the model and updating the view may each require a number of incremental changes which bear little relation to each other.

Secondly, as Young [Young88 p.370] points out, the view may contain elements not derivable from the model and of relevance only to the view. The graphical representation of a network, for example, may be laid out in various ways independently of its connectivity. The user may even wish to manipulate the view after the lifetime of the model which generated it, for example to include this in a document. This is not possible in Andrew, for example, since its views are not persistent [Palay88 p.11]. One might also wish to have generic views which could be applied to a variety of models (scroll bars, for example). These facilities would be impossible if the view were simply a projection of the model.

It is thus necessary to give views their *own* model, separate from the 'semantic' model, for example the face of a dice as a separate entity from the number

three. But even that view could have a separate model, for example a high level graphical description, which itself could have several views, and so on. That is, there is a need for a nesting of models. The PAC architecture [Coutaz87] is in theory recursive in this way. This problem has also been recognised in the Andrew Toolkit [Palay88 p.12] where the trick is to use 'auxiliary' data objects.

Similarly, but separately, it is possible to envisage a nesting of views. That is, stylistic and other parameters of a view could be controlled through a separate view, and so on. Interviews, for example, has a concept of 'metaviews' [Linton87 p.256], which are views not of the semantic model, but of the view model. Stylistic details of the view can thus be manipulated through the metaview. Szekeley makes the same point with what he calls *presentations* [Szekely87 p.238]. As an illustration, the Framer system [Fischer89 p.48] allows the user interactively to change the visual substructure of a window through a special view representation of this.

2.6.8. Separation Problems in Agent Architectures

Agent architectures in general face two major problems in providing a basis for the separation of interface and application: media sharing and semantic seepage.

Media Sharing

In multi-tasking graphical environments, input media like the mouse and keyboard and output media like the screen are resources necessarily shared between agents. Agents cannot therefore encapsulate all aspects of either their input or output. For example, view agents cannot simply project themselves onto the screen without consideration for the presentations of other objects, which may have visual priority.

Agents therefore have to comply with the synchronisation and prioritisation requirements of a resource manager like an input framework or a window manager. In poorly modularised systems, this can lead to low-level screen 'damage/repair' [Gosling86] being bound into agents in the form of a mandatory 'repaint' procedure which the agent must supply.

Semantic Seepage

Semantic seepage refers to the early binding of application semantics and interface presentation. That is, there is a tendency for application functions to move into the interface, and thus make it more domain specific.

This can occur in logical devices, in which presentation in the form of prompts, echoing or feedback is written into the device. It can also occur in specialised interface objects, such as Toolkit components like menus or scroll bars, where inherited presentation features are extended with application semantics. In the limit, it is possible to import all of the application semantics into the interface in this way.

Even in model-view systems which allow the composition of views into visual structures, there may be complex dependencies between the application (model) objects and the structure of the interface (view) objects, leading to semantic seepage. In the example of the linked boxes, the logical connectedness of the boxes must also be bound in to the interface object which maintains the link in the view.

Thus while it is possible to *abstract* interface concerns in agent (particularly model-view) architectures, for example into classes of tools or other presentation objects, delaying *binding*, and thus full interface *separation*, is more difficult.

2.7. Conclusions

This Chapter has examined a number of architectures whose motivation has been to provide a service to application writers by separating interface and application concerns.

Input frameworks simply provide an input routing service whereby applications receive input directed at them, without having to poll devices. However, the framework may be more or less bound in to some 'desktop' model (for example, a window manager, or more specialised environment), and thus by determining output necessarily involve some semantic seepage.

The Chapter then examined two major classes of full architectures (i.e. those that support both input and output, and therefore semantics). These two classes differ fundamentally in their granularity, and in the perspective they allow of application semantics.

Linguistic architectures, particularly in UIMS, are monolithic, and attempt to separate dialogue. This separation is compromised by the need for incremental semantic feedback and directness, and the need to cope with multi-threaded dialogues.

The difficulty of even abstracting multiple dialogues has led to the flourishing of *agent* architectures, which by contrast are manifold, and attempt to separate the *devices* of interaction. However, the independence of agents is compromised by the need to share input and output resources. In addition, particular refinements of the agent architecture, such as logical devices and the model-view paradigm, may suffer semantic seepage into their interface agents. It may therefore be difficult to supply a *generic* interface system within an agent architecture.

We can draw the following overall conclusions from this chapter:

- In order for separation to be concrete enough to run application and interface on different machines, there needs to be an initial clear boundary between the two. That is, the architecture should allow an *integrated* interface component.

- In order to provide directness, either the interface must be aware of application semantics, or the application must be aware of the interface presentation. In particular, the notion of a dialogue socket by which application and interface can be mutually independent is not compatible with directness.

There appears to be a fundamental conflict between the need for directness and the need for separation, since directness suggests that interface presentation is bound early to application semantics, whereas separation seeks to delay that binding. However, this Thesis exploits the fact that some aspects of the interface can be determined purely by the user, while other aspects are necessarily determined by the application. Separation can be achieved between these two aspects of interaction (later called *surface* and *deep*), while directness can be supported by each independently.

Chapter 3

A Formal Perspective on Dialogue Separation

The purpose of this chapter is to show formally that while dialogue management can be *abstracted*, it cannot usefully be *separated* from the functionality it drives, as these terms are defined in Chapter 1. This strongly limits the degree to which

- dialogue management can be used to incorporate user-level concerns in the interface, independently of the application.
- the *execution* of dialogue management can be distributed from its functionality.

These are the two major motivations for linguistic architectures like UIMS. This Chapter is thus a complement to the previous Chapter, in that it seeks to make precise the limitations on dialogue separation recognised in practice. The overall motive is to support the Thesis that Surface Interaction provides a better architecture for user interface *separation* than the linguistic architecture.

3.1. Interaction

We consider three major components of interactive computation:

- **State** - the *type* of values which the computation can take on. The state is likely to be a complex structure itself containing different types. A distinguished subset of the set of states forms the *start* states for the computation.

• **Functionality** - the operations possible on the state, which result in changes to the state (state transitions).

These two components constitute the *semantics* of the computation, since the functionality defines the *set* of states, or values, derivable from the start states.

• **Dialogue** - the set of possible *sequences* of operations that can take place in a computation. Dialogue is an abstraction of *control*. For example, common control abstractions provided in programming languages are sequence, iteration, and selection. When used as communication (i.e. when one process invokes an operation in another process), the operations can be called *events*. Events subsume both input and output, since one event can be the output of one process and the input of another.

We can easily model the human user as a process which communicates with the computer. Dialogue can be thought of as the *syntax* of this interaction.

As we shall see, dialogue is *determined* by the semantics. These points can be illustrated by a simple example which will also serve as a tutorial for the formal notations of Z and CSP used in the Thesis.

3.1.1. State

Imagine a nuclear reactor, which (as far as I understand it) consists essentially of a core of radioactive material, some damping rods to control the chain reaction, and some coolant to take away the heat generated. At the formal level we can abstract away from all engineering detail such as specific materials or dimensions, and also from any components of the system that remain unchanged throughout its operation, such as the radioactive core itself. The critical information we wish to know or control in the nuclear reactor, therefore, is simply the position of the damping rods and the level of the coolant.

We can define such a system in the schema language of Z:

REACTOR

$rods: up \mid down$
$coolant: in \mid out$

The schema *REACTOR* defines a *state space* consisting of the variables *rods*, which can take the values *up* or *down*, and *coolant*, which can take the values *in* or *out*. In Z, the state space can be defined explicitly as the set

$$\{REACTOR \bullet \theta REACTOR\}$$

that is, all the possible bindings between values and component variables which conform to the constraints in the schema *REACTOR* (the prefix θ indicates a *binding* of the schema - a particular set of values for its variables). In this case there are no constraints (these would be expressed in a separate *predicate* within the schema), and so *REACTOR* represents the set of *all* possible combinations of values for *rods* and *coolant*.

3.1.2. Functionality

The *functionality* of the reactor consists of the operations we can perform to modify this state. A *state-based* specification like Z allows us to define operations by specifying how the state changes as a result of the operation. That is, we specify pre- and post-conditions on the state. Typically, the post-condition expresses a relationship between values of the variables prior to the operation and their values after the operation, possibly as a function of input parameters.

For example, the *REACTOR* clearly needs an operation to set the rod position:

SET_RODS

$\Delta REACTOR$
rodpos?: up | down

---

rods' = rodpos?
coolant' = coolant

This schema illustrates a number of features of the Z schema language. The top box is the *signature* which declares terms and their types, while the lower box is a *predicate* consisting of a conjunction of clauses (the logical operator $\wedge$ is understood between lines). One schema can be *included* in another simply by referring to its name in the signature. All of the declarations and predicates of the referenced schema are then understood to be included in the referencing schema.

In the case of *SET_RODS*, the schema reference *REACTOR* is prefixed with Δ, which has the effect of including, as well as the original signature and predicate, a mirror version in which all the variables are decorated with prime ('). Thus *SET_RODS* includes the variables *rods*, *coolant*, *rods'*, and *coolant'* (*REACTOR* has no predicate). By convention, the unprimed variables refer to state before an operation, while the primed variables refer to state after an operation. Also by convention, variables suffixed with *?* or *!* are input or output variables respectively.

Δ*REACTOR* thus indicates that an operation is being defined on the *REACTOR* state, to which an input parameter *rodpos?* is provided. The schema establishes that after the operation *SET_RODS*, the variable *rods* has been set to the value of the input *rodpos?*, while the value of *coolant* remains the same as it was before.

In exactly the same way we can define an operation to set the coolant level:

SET_COOLANT

Δ *REACTOR*
coolantlevel?: in | out

---

coolant' = coolantlevel?
rods' = rods

To complete the specification, we need to define an initial state for the *REACTOR*. We do this by providing an operation (prefixed conventionally by *Init*) which ignores the prior state of the variables, and sets their primed state to appropriate values:

Init_REACTOR

Δ *REACTOR*

---

rods' = down
coolant' = in

The operations *SET_RODS*, *SET_COOLANT* and *Init_REACTOR* in effect define sets of possible state transitions in the system. The type of each operation therefore is

$$REACTOR \rightarrow REACTOR$$

That is, each defines a function between reactor states. For example, if *REACTOR* is in a state defined by the tuple

$$(rods \Downarrow down, coolant \Downarrow out)$$

(using $\Downarrow$ to indicate a binding of a value with a variable name), then the operation *SET_COOLANT* with parameter *in* will convert this state to

$$(rods \Downarrow down, coolant \Downarrow in)$$

In the general case with an arbitrary state S, as Sufrin and He point out [Sufrin90 p.154], it is necessary to weaken the state transitions to a *relation* $S \leftrightarrow S$, since the predicate of an operation may not fully determine the resultant state.

3.1.3. Object

For convenience, we can combine the description of state and functionality into a single tuple of type *OBJECT*:

$$OBJECT == S \times O \times Init$$

where S is a state, O a set of operations defined as state transitions (relations) on S, and *Init* is an initialisation operation on S. In mathematical terminology, an *OBJECT* defines an *algebra*, if we regard the *Init* operation as producing constants of the type.

Thus we can convert the *REACTOR* system defined above into an *OBJECT*:

$$Reactor = (REACTOR, \{SET\_RODS, SET\_COOLANT\}, Init\_REACTOR)$$

3.1.4. Range

The states of *REACTOR* which it is possible to reach using the operations *SET_RODS*, *SET_COOLANT*, and starting with the state resulting from *Init_REACTOR*, are in fact all the states in *REACTOR*. In general, this is not the case. For example, using an operation *(+2)* on natural numbers, and starting with *0*, it is only possible to reach all the even numbers. The set of reachable states is therefore a useful measure. We call this the *range* of a system.

Range can be defined generically over any object:

$$\forall O: OBJECT \mid O = (state, ops, Init) \bullet$$
$$range\ O = (\bigcup ops)^* \ (\!|\ ran\ Init\ |\!)$$

That is, the *range* of an *OBJECT* is the image of the initialised states through the closure of the union of all the operations. More simply, range is the set of all states that can be reached from any initial state, using any combination or iteration of operations. Fairly clearly,

$$\forall O: OBJECT \mid O = (state, ops, Init) \bullet range\ O \subseteq state$$

That is, the range of an *OBJECT* can only ever be a subset of, or equal to, the state on which it is defined. This is a simple consequence of the state-based definitions of the operations.

3.1.5. Behaviour

Following Sufrin and He (op. cit.), it is possible to think of the system, or object, as a *process*, and its operations as an alphabet of *events* (*E*) in which the process can engage. It is therefore possible to derive a single *behaviour* function (outside the Z specification) which maps such events to the state transitions they produce:

$$\beta: E \rightarrow (S \leftrightarrow S)$$

This is the formal equivalent of the event-action pair notation for agents outlined in the previous Chapter.

For example, if the reactor consisted just of the *Init_REACTOR* operation, then

$$\beta = \{\textbf{\textit{Init\_REACTOR}} \mapsto$$
$$\{Init\_REACTOR \bullet \theta REACTOR \mapsto \theta REACTOR'\}\}$$

That is, the behaviour function would consist of the single mapping which takes the event ***Init_REACTOR*** (bold type is used to distinguish the event name from the operation name) to the relation defined as the set of pairs of bindings of *REACTOR* and *REACTOR'* which satisfy the predicate of *Init_REACTOR*.

However, we also need to take into account parameters to operations. Given a particular operation, its state transition relation is in fact a *function* of its parameters (since the state transitions will differ depending on the parameters). For simplicity we can think of the parameters (which may be arbitrary in number) as a single tuple P of the actual parameters. In the general case, therefore, we need to upgrade the behaviour function:

$$\beta_1 : E_1 \rightarrow P \rightarrow (S \leftrightarrow S)$$

For example, if the reactor consisted of just the *SET_RODS* operation, then

$$\beta_1 = \{SET\_RODS \mapsto \lambda\, rodpos?: up \mid down \bullet \{SET\_RODS \bullet \theta REACTOR \mapsto \theta REACTOR'\}\}$$

It is convenient, however, to expand the set of events to incorporate all the possible parameters, so that we can revert to the simpler behaviour function β. In these terms the alphabet of the reactor, for example, can be defined as the union:

$$E ::= \textbf{\textit{SET\_RODS}} \langle\langle up \mid down \rangle\rangle$$
$$\mid \textbf{\textit{SET\_COOLANT}} \langle\langle in \mid out \rangle\rangle$$
$$\mid \textbf{\textit{Init\_REACTOR}}$$

(so that one event might be ***SET_COOLANT in***).

3.1.6. Dialogue

In the general case, since the state resulting from an event may or may not satisfy the pre-conditions for a potential subsequent event, it is possible to determine which *sequences* of events the process can theoretically engage in. These are known as *traces* of the process. In user interface terms, the traces record the *dialogue* with the user. With an arbitrary behaviour function β, the set of traces (*traceset*) can be defined:

$$traceset = \{t: seq\, E \mid \fatsemi / (t \fatsemi \beta) \neq \{\}\}$$

That is, traceset consists of any sequence of events the composition ($\fatsemi$) of whose state transitions also defines a state transition. In yet other words, a sequence of events is a trace if the state transitions each event defines form a con-

nected sequence of states such that the whole can be thought of as a single state transition between the initial state of the first event and the final state of the last event.

A necessary feature of the *traceset* is that it is *prefix-closed.* That is, if a process can engage in a sequence of events t, then it clearly can also engage in any initial subsequence of t (since it already has). These must therefore also be possible traces.

Since there are no preconditions for the ***Init*** event, this can occur at any point in a trace, and has the effect of returning the process to the initial state. In fact in a Z specification there is an implicit constraint on the *traceset* such that the first event in all traces must be an ***Init***:

$$Ztraceset = \{t: traceset \mid head\ t = \mathbf{Init}\}$$

(Equivalently, the traces could be constrained by designating certain states as *start* states. This technique is used in defining state *machines*, for example.)

Ztraceset therefore must at least contain the trace <***Init***>, because the ***Init*** event is always possible. Since *Ztraceset* is thus non-empty as well as prefix-closed, there must be a CSP-defined process which corresponds to it. For example we can define the *Reactor* process (R) by:

$$R = \mathbf{Init\_REACTOR} \rightarrow \mu X \,.\, (x: E \rightarrow X)$$

That is, R is the process which engages first in the initialising event, and then in any sequence of events from its alphabet E.

Since there must be a process which corresponds to the observable traces of an *OBJECT*, and since the traceset can be derived deterministically from the definition of an *OBJECT* as we have seen, there must also be a function which can convert an *OBJECT* into a process:

$$observe: OBJECT \rightarrow Process$$

(where *Process* is the set of CSP-defined processes). We can define *observe* simply:

$$\forall\ O: OBJECT \bullet traces\ (observe\ O) = Ztraceset$$

(*traces* is the CSP operator. We assume *Ztraceset* is quantified over all *OBJECT*s). Also, if β is the behaviour function of the *OBJECT O*, then

$$dom\ \beta = \alpha\ (observe\ O)$$

That is, the alphabet of the process observable from *O* corresponds to the operations for which state changes are defined in the behaviour function. Thus we can convert the *Reactor OBJECT* into the process *R*:

$$R = observe\ Reactor$$

This equivalence between *OBJECT*s and *Processes* holds whether the processes are deterministic or not. In fact the possibility of *failure* in a non-deterministic process [Hoare85 p.129] corresponds to *looseness* [Spivey89 p.135] in the specification of a state transition (which is the reason for specifying the state transitions as a relation rather than a function in β). At this level, also, we abstract away from any distinction between input and output.

3.2. Relation between Functionality and Behaviour

What is the relationship between a state-based description of functionality, and an event-based description of behaviour? It is theoretically possible to recover the semantics of a trace (i.e. the particular state transition it effects) from the trace alone. But in order to do this we would need to pack more information into the name of the event by 'uncurrying' the behaviour function:

$$\beta_2 : (E \times S \times S) \rightarrow (S \times S)$$

In effect we enumerate *all* state transitions, and expand the name of the event with the unique transition to which it maps. A typical event in such an alphabet might be:

SET_RODS down when rods is up and coolant is in
resulting in coolant in and rods down.

It is always possible to recover state effects from such a description of behaviour since clearly the state transitions are made explicit in the event name (or some simple interpretation of it). In the abstract, therefore, traces can describe any functionality. In practice, of course, this would result in an impossibly large number of events.

On the other hand, not all sets of traces can be described purely by a state-based definition of functionality. The most obvious counter-example is *Ztraceset*, which can in general only be constructed by an explicit constraint on *traceset*. It might be imagined that the same effect could be achieved by including a flag in the state which is set by *Init* and is a precondition of every operation. But then we are assuming either that *Init is* the first operation, or that there is some prior initialisation of the flag.

In practice, therefore, both a state-based description of functionality, and an event-based description of behaviour, seem necessary. The need for this 'hybrid' approach has been recognised in a number of places [Sufrin90, Josephs88, Lamport89, Abowd90]. Without a description of functionality we would be reduced to implementing behaviour as a state *machine* in which each state is explicitly enumerated. In this case, the number of states, or the number of events/transitions, rapidly becomes unmanageable.

For example, since the parameters for the operations in *REACTOR* are essentially Boolean, it is easy to enumerate its alphabet of events and its states. But if we had allowed the level of the rods or coolant to be specified by a number, then in theory the states and hence the alphabet would be infinite. Conversely, without a description of behaviour there may be some orderings of the operations which we could not exclude. In the general case, one constraint may best be expressed over the traces, another over the state. We give an example of this below.

3.3. Taking account of the user

Excluding certain traces from those possible under the functionality is useful when we need to incorporate external concerns. For example, the *Reactor* allows any sequence of the operations *SET_RODS* and *SET_COOLANT* and their parameters, and this would presumably also be true of the hardware. However, it will be in the interests of the user of the reactor to exclude those traces which generate a state in which *rods* is *up* and *coolant* is *out*. In this state the reactor will suffer a meltdown or worse.

In order to prevent a meltdown we need to place the following constraint P on the traces:

$$P\ (t: seq\ E) = \forall\ u: seq\ E \mid \neg <SET\_RODS\ down> \textbf{ in } u \bullet$$
$$\neg <SET\_RODS\ up> \frown u \frown <SET\_COOLANT\ out> \textbf{ in } t$$
$$\wedge\ \forall\ v: seq\ E \mid \neg <SET\_COOLANT\ in> \textbf{ in } v \bullet$$
$$\neg <SET\_COOLANT\ out> \frown v \frown <SET\_RODS\ up> \textbf{ in } t$$

(***in*** is the contiguous subsequence operator). A safe reactor will then have the following traces:

$$Safetraces = \{t: Ztraceset \mid P(t)\}$$

The constraining predicate P is clumsy. What we are in fact avoiding is a particular *state* of the reactor. It may be easier to incorporate the constraint in the state-based description, and in this case it is possible to do this:

SAFE_REACTOR

$$REACTOR$$
$$\neg\ (rods = up \wedge coolant = out)$$

That is, the particular state in which the rods are up *and* the coolant is out is *not* allowed in *SAFE_REACTOR*:

$$SAFE\_REACTOR = REACTOR - \{(rods \Downarrow up, coolant \Downarrow out)\}$$

However, we ought also to ensure that no operation can *put* the safe reactor into this state. The original *Init* operation clearly does not, but we need to restrict the other operations:

SAFE_SET_RODS

$$SET\_RODS$$
$$SAFE\_REACTOR'$$

That is, *SAFE_SET_RODS* is the same as *SET_RODS*, except that the *SAFE_REACTOR* constraint must apply after the operation (i.e. on the primed variables). Similarly:

SAFE_SET_COOLANT

SET_COOLANT
SAFE_REACTOR'

We do not prove it here, but such a functionality, along with the initialising constraint, generates the traces *Safetraces*.

This is a clear example in which the restrictions placed on the system are in the interests of *all* users. There is therefore an argument here for incorporating these constraints *within* the functionality in this way. However, there are many cases where different users may require differing access to the operations of the functionality. For example, the access of ordinary users to an operating or database system is likely to differ from that of the system manager. There may also be cases in which the need for constraints arises because of some *interference* between a number of processes, for instance when two processes access the same resource. In all these cases we need ideally to make the user constraints on the traces *independent* of the functionality.

For example, we might wish a trainee user of the reactor never to push the same button twice in succession (assuming there is a button for each event). A suitable constraint might therefore be:

$$Q\ (t{:}\ seq\ E) = \forall e{:}\ E \bullet \neg \langle e, e \rangle\ \textbf{in}\ t$$

and his traces would be:

$$Traineetraces = \{t{:}\ Safetraces \mid Q\ (t)\}$$

This constraint is not dependent on the functionality, in the sense that (so long as the alphabet E is generic) it could be applied to any set of operations.

Representing user concerns by placing constraints on the traces of a functionality is a fundamental concept in User Interface Management Systems. The practicalities and limitations of this approach were examined in Chapter 2. Research has also been carried at York [Dix85, Dix86, Dix87a, Dix87b, Dix88b, Harrison90]

and Oxford [Sufrin90] into devising and formulating user-oriented properties of interactive systems as predicates over traces.

3.3.1. Implementing Trace Constraints Separately

User constraints *expressed* within the functionality, that is, as pre- or postconditions of the state transitions caused by the operations, would naturally be *implemented* as part of the functionality. This would be the case, for example, with an implementation of *SAFE_REACTOR* above. Constraints expressed over traces, on the other hand, are possible to implement separately.

We can model such constraints with a separate user interface process (U), which communicates with the functionality. For example, if we wished to prevent a trainee from pushing the same button twice, then we could implement U such that

$$U \textbf{ sat } (\forall e: \alpha U \bullet \neg \langle e, e \rangle \textbf{ in } traces\ (U))$$

Let us assume (by *observation*) that there is a process S whose traces are exactly those of the operations on *SAFE_REACTOR*, and that U has the same alphabet as S. If we run U in parallel with S:

$$U \parallel S$$

then we have a process whose traces, by the definition of the CSP $\parallel$ operator, consisted of only those traces which were possible under both U and S. That is,

$$traces\ (U \parallel S) = Traineetraces$$

This prevents the trainee from pushing the same button twice, which is exactly what was required. By creating a special interface process in this way we can implement such user-oriented interface constraints *separately* from the application.

In general, for any application process A and interface process U, the only restriction we need to apply is

$$\alpha U \cap \alpha A \neq \{\}$$

That is, that their alphabets should have some events in common. If this were not the case, then they would not be able to communicate. On the other hand, we need to allow the possibility that the interface process engage in some events with the end user which the application does not see (such as building up a command from

keystrokes or mouse clicks), and conversely that the application engage in some events (such as reading from disk), which the end-user does not see.

3.4. Limitations on Separation

In the abstract, then, this argument supports the case that it is possible to implement a UIMS which represents user concerns as constraints on dialogue traces, separately from application functionality. However, there are two major limitations to this separation.

Firstly, the interface process must know some or all of the application events. This clearly limits the genericity of the interface process. There are two strategies.

- The interface process may be deliberately specialised to some application domain, for example database or process control.

- The interface events may be quantified over sets of application events. The constraint in *U* above, for example, that no event should happen twice in succession, applies to any application event. It is still an active issue, however, whether effective 'user engineering principles' [Thimbleby84, Thimbleby85] can be expressed independently of application semantics in this way.

Secondly, the interface process must know some or all of the application state. This is a much more severe limitation. For example, consider if we wished to implement the *SAFE_REACTOR* process (*S*) by running the *Reactor* process (*R*) in parallel with an interface process (*U*) which maintained the safety constraints:

$$S = R \parallel U$$

We would need a process *U* such that

$$U \textit{ sat } P\ (tr)$$

That is, the traces of *U* should satisfy the safety predicate *P* defined above.

However, in order to decide *P*, the least information that *U* must maintain is the state of the rods and the state of the coolant. That is, *U* must know that a *SET_RODS up* event cannot be accepted if the coolant is out, and that a *SET_COOLANT out* event cannot be accepted if the rods are up. This information is

in fact the whole of the *Reactor* state. The interface process U thus needs to duplicate the application state in order to *predict* these errors.

These limitations suggest that it is in fact impossible to separate (late bind), or even abstract, the interface from the application process, if the interface is to incorporate any user-level constraints on the application behaviour other than those which are completely generic (like Q in *Traineetraces*).

One might imagine that it would be possible to express meta-constraints in the interface that make no specific reference to application state but yet impose useful properties, for example a *Safe* constraint that could be applied to any class of application. Upon examination, however, some communication protocol must be agreed between application and interface such that either the application itself specifies what is safe (which misses the point of *separate* user constraints), or the interface is specialised to that class of application by referencing its functionality.

3.4.1. Classes of Dialogue Separation

This necessary binding of interface control to application functionality can be classified by examining the *level* of dialogue constraints required in the interface. As has been noted, the constraint Q in *Traineetraces* is generic over any functionality, whereas a *Safe* constraint necessarily depends upon some semantic interpretation of what is safe in the application domain.

In general we can view the traces that the interface process should accept as a formal language. We therefore have an automatic classification of interface systems in terms of the Chomsky hierarchy of languages, and can be precise about the semantic requirements of each. Both *Traineetraces* and *Safetraces* are regular languages (Chomsky type 3), since they can be recognised by a finite state machine. State-transition systems are a common implementation mechanism for parsing user input in a UIMS.

Any user interaction that has a nested structure (for example, opening a window and then invoking an application within the window) is part of at least a context-free language (Chomsky type 2), and requires at least a state machine with a pushdown stack to implement its grammar. The stack keeps track of the level of nesting.

If the interface is required to manage error-handling, aborting or undoing, then it must recognise at least a context-sensitive (Chomsky type 1) or even a Turing-power (Chomsky type 0) language. This is because at the very least it needs to jump about in the stack - that is, it needs a random access memory. Thus, in this case we need to *program* the interface as well as the application, and it is debatable to what extent we have abstracted the dialogue, rather than simply fragmented the functionality.

It is therefore inescapable that the more powerful the dialogue control we wish to build into the user interface, the more application semantics it has to know about. This leads to three alternative pathological situations in UIMS based on the separation of dialogue and functionality:

- Application state, as in the example of the *SAFE_REACTOR* process above, is duplicated in the interface. In the limit, all application state might need to be duplicated.

- The interface repeatedly *enquires* about application state. In the limit, the interface can take no action without first checking the application state, and the communication link between them is heavily used.

- The application and interface are fragmented into numbers of objects (as in the agent architecture (Chapter 2)) each of which handles some application functionality and some interface control and presentation. In the limit this results in a 'homogeneous object space' [Dance87 p.98] in which it is impossible to separate application objects from interface objects.

Collectively we can call these situations *semantic seepage* from the application into the interface, because they all arise from the need to know about the application. They are pathological to the extent that they compromise the separation (that is, the late binding) of the interface and application components.

3.5. Input and Output

We can make the domain of application-independent dialogue constraints clearer by refining the earlier notion of simple communication *events* into separate input and output events.

Up to now we have considered the *result* of an operation to be simply a change in the state of the system. In the case of the *REACTOR* this clearly is the most important effect. Even here, however, we may wish to specify some component of the state, or some function of this, as an explicit *output*. For example, we may wish an indicator light on a panel to flash, showing that the rods are up.

Z allows us to specify output using, by convention, a '!' suffix. Thus we can refine the *SET_RODS* operation:

SET_RODS1

$$
\begin{array}{|l}
\hline
SET\_RODS \\
rodindicator! : off \mid on \\
\hline
rods' = up \Rightarrow rodindicator! = on \\
rods' = down \Rightarrow rodindicator! = off \\
\hline
\end{array}
$$

We assume that there is a similar indicator for the coolant, and that the indicators are initialised correctly.

We can abstract away from the distinction between input and output because a Z operation is conceived to be atomic, and therefore at one level we can take output as part of the invoking event, just as we have done with input. A typical event therefore might be

SET_RODS up and rodindicator on

At this level, the behaviour function β does not change with the introduction of output parameters.

In practice, however, the operation may take some time, and we may wish to direct output to a specific receiving process along some channel. It is also the case that, in considering interface design, we might well want to differentiate different classes of input events. For example, we might want to treat the specification of an *operation* (for example, by clicking an icon) as a separate event from the specification of its *operands*.

It is fairly easy to decompose the invocation of an operation into separate input and output events. The alphabet of the process is in this case extended to be the

union of the operation names, and input or output parameters and their values. For example, at this level the alphabet of *REACTOR* could be:

$$
\begin{aligned}
E ::= & \ \textbf{\textit{SET\_RODS}} \\
& \mid \textbf{\textit{SET\_COOLANT}} \\
& \mid \textbf{\textit{Init\_REACTOR}} \\
& \mid \textbf{\textit{rodpos?}} \ \langle\langle \textbf{\textit{up}} \mid \textbf{\textit{down}} \rangle\rangle \\
& \mid \textbf{\textit{coolantlevel?}} \ \langle\langle \textbf{\textit{in}} \mid \textbf{\textit{out}} \rangle\rangle \\
& \mid \textbf{\textit{rodindicator!}} \ \langle\langle \textbf{\textit{off}} \mid \textbf{\textit{on}} \rangle\rangle \\
& \mid \textbf{\textit{coolantindicator!}} \ \langle\langle \textbf{\textit{off}} \mid \textbf{\textit{on}} \rangle\rangle
\end{aligned}
$$

(We use emboldened parameter names as constructors).

This maps easily to CSP, by treating the parameter names as channels (i.e. we put *?* or *!* between the constructor and the value). An invocation of an operation now consists of the *set* of events containing an operation name, and input and output parameter values. For example,

$$\{\textbf{\textit{SET\_RODS}}, \textbf{\textit{rodpos?down}}, \textbf{\textit{rodindicator!off}}\}$$

Let us call such sets of lower level events *invocations*. Correspondingly, the behaviour function β has simply to be modified so that *invocations*, instead of just events, map to state transitions:

$$\beta_i : \ \mathbb{P}\, E \rightarrow (S \leftrightarrow S)$$

We generate the traces of the behaviour at this lower granularity in two stages. Firstly, we modify the original trace generation so that it produces sequences of invocations, rather than sequences of events:

$$\textit{invocation\_traceset} = \{t : \textit{seq}\ \mathbb{P}\, E \mid \ 8 \,/\, (t \ 8 \ \beta_i) \neq \{\}\}$$

Secondly, in order to collapse the sequences of invocations into sequences of events, but still preserve a deterministic mapping between sequences of events and sequences of operations on the state, we need only place two constraints on the sequencing of events from the invocations:

- the input events from any one invocation occur *before* any output events from the same invocation.

• an operation *fires* only when all its input events have occurred, and the next events are its output.

In order to express these constraints we need predicates to determine if an event is input or output. We define these simply by saying that *input(e)* and *output(e)* are true if *e* is an input or an output event respectively. We can then expand any invocation *i* into a set of possible (injective) sequences of events:

$$expand\ (i) = \{t: iseq\ E \mid ran\ t = i \wedge \forall\ e1, e2: i \mid input(e1) \wedge output(e2) \bullet t^{\sim} e1 < t^{\sim} e2\}$$

For example,

$$expand\ (\{\textbf{SET\_RODS}, \textbf{rodpos?down}, \textbf{rodindicator!off}\}) = \{\langle \textbf{SET\_RODS}, \textbf{rodpos?down}, \textbf{rodindicator!off} \rangle, \langle \textbf{rodpos?down}, \textbf{SET\_RODS}, \textbf{rodindicator!off} \rangle\}$$

That is, in order to invoke the *SET_RODS* operation to set the rods down, we can either specify the operation name ***SET_RODS*** followed by the parameter value ***rodpos?down***, or vice versa, but in either case the output event ***rodindicator!off*** must occur last.

The possible traces of input and output events from any behaviour (i.e. any *sequence* of invocations *it*) can thus be defined:

$$iotraceset = \{it: seq\ \mathbb{P}\ E; i: 1..\#it; t: seq\ seq\ E \mid \#t = \#it \wedge t\ i \in expand\ (it\ i) \bullet \frown / t\}$$

For example, an *iotrace* of a ***SET_RODS*** operation, followed by a ***SET_COOLANT*** operation on *Reactor*, might be:

$$\langle \textbf{SET\_RODS}, \textbf{rodpos?down}, \textbf{rodindicator!off}, \textbf{SET\_COOLANT}, \textbf{coolantlevel?in}, \textbf{coolantindicator!on} \rangle$$

But it could also be:

$$\langle \textbf{rodpos?down}, \textbf{SET\_RODS}, \textbf{rodindicator!off}, \textbf{coolantlevel?in}, \textbf{SET\_COOLANT}, \textbf{coolantindicator!on} \rangle$$

In the second the parameter is accepted before the operation name. This is a common ordering in iconic environments because it gives the user freedom to amend the

parameters before invoking the operation. We could even unambiguously allow these modes to be mixed:

<SET_RODS, rodpos?down, rodindicator!off,
coolantlevel?in, SET_COOLANT, coolantindicator!on>

It is clear that any one trace from *invocation_traceset*, i.e. a sequence of invocations, allows a number of possible sequences of input and output events in *iotraceset*. This is because we are not strict about the ordering of input events or output events to a particular operation. This is just as it should be, since we thereby allow, within the semantic constraints, a range of dialogue syntaxes. It is appropriate that any *further* constraints on the dialogue be imposed by an *interface* process, for example one that ensures interface consistency by imposing a syntax in which operator precedes operand (e.g. typing a command followed by its arguments) or vice versa (e.g. selecting some text and then clicking on the delete icon).

However, the predicate in *iotraceset*, as defined, is too strong given the original constraints. That is, it excludes some semantically possible traces of input and output events. In particular, it excludes traces in which input events to one operation start before the completion of a previous operation. One such trace might be:

<SET_RODS, SET_COOLANT, coolantlevel?in,
rodpos?down, rodindicator!off, coolantindicator!on>

This trace is still unambiguous in terms of the invocation of operations and their parameters. That is, it is a valid invocation of the semantic behaviour, since the operations still fire in the same sequence, even though some of their inputs may have occurred early and may be interleaved. These extra traces match an interactive syntax in which operations may be specified by filling slots in forms or dialogue boxes, so that a number of operations may be partially specified at the same time.

The mathematics to express this is cumbersome (and so is not given), since it is necessary to identify to which invocation an input event belongs. Otherwise, if operands had the same types, there would be no way of knowing if an input event completed an operation invocation that had already started, or was destined for a subsequent operation. Thus each event in the traces must be tagged with the operation to which it belongs. In a graphical interface this tagging is performed by location - we know that an input is destined for one operation rather than another because, for example, it is typed by the user into a particular named dialogue box.

By definition, the constraints on the interleaving of events within input and output sets, and between different invocations, have been designed to preserve the mapping between sequences of events in *iotraceset* and the possible sequences of operations on the underlying functionality. That is, we can assume there exists an interpretation function I which is total but not necessarily bijective:

$$I: iotraceset \rightarrow traceset$$

That is, when we refine events into input and output, there may be a *number* of acceptable dialogue sequences for the *same* sequence of application operations. Which particular dialogue sequences are allowed, then, is entirely independent of the application, and can rightly be made an *interface* design issue.

3.5.1. Communication

It remains to demonstrate how two processes defined in this way might actually communicate along the channels. For example, we could define a display panel consisting of two lights:

$$DISPLAY \triangleq [rodslight, coolantlight: on \mid off]$$

(this is a *horizontal* schema definition in Z). Operations on the panel switch the appropriate lights on or off (we assume correct initialisation of the lights):

RODS_LIGHT

$\Delta DISPLAY$
switch?: on | off

---

rodslight' = switch?
coolantlight' = coolantlight

COOLANT_LIGHT

$\Delta DISPLAY$
switch?: on | off

---

coolantlight' = switch?
rodslight' = rodslight

We can construct a process D from the *DISPLAY* panel by observation (we assume an initialisation operation):

$$D = \\ observe\ (DISPLAY, \{RODS\_LIGHT, COOLANT\_LIGHT\}, Init)$$

D thus is defined over events which switch the appropriate lights on or off. We could also output the light state to some other processes which represented lights, and so on, but clearly at some point we must be satisfied simply with effecting a change of state in a process. We would hope for the correct communication to occur if we ran the two processes in parallel:

$$R \parallel D$$

Unfortunately, however, the alphabets of the processes are disjoint. For example, the output event

rodsindicator!.on

from R does not coincide with the input event

switch?.on

to D, as we might hope. If we use one common channel called *switch*, then it would not be clear which of the operations ***RODS_LIGHT*** or ***COOLANT_LIGHT*** were to be invoked when ***on*** or ***off*** appears on the channel. On the other hand, if we assign different channels, for example *rodsindicator* and *coolantindicator*, to each operation in *Reactor*, there would remain the problem that the behaviour (β_i) of D is defined over events consisting of operation name and parameter value, not just of the parameter value.

This illustrates the essential binding that must occur between communicating processes. That is, either the sending process R must specify the operations to be invoked in the receiving process D by outputting composite events consisting of an operation name and its parameters, or each operation in the receiving process must have a distinct channel, and the binding between the channel and the operation name must be made in or before the receiving process.

We can represent such dynamic or delayed binding by another process:

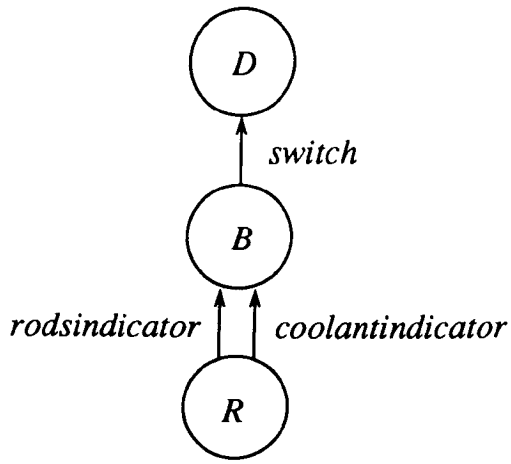
$B =$

rodsindicator?on $\rightarrow$ ***RODS_LIGHT*** $\rightarrow$ ***switch!on*** $\rightarrow B \mid$
rodsindicator?off $\rightarrow$ ***RODS_LIGHT*** $\rightarrow$ ***switch!off*** $\rightarrow B \mid$
coolantindicator?on $\rightarrow$ ***COOLANT_LIGHT*** $\rightarrow$ ***switch!on*** $\rightarrow B \mid$
coolantindicator?off $\rightarrow$ ***COOLANT_LIGHT*** $\rightarrow$ ***switch!off*** $\rightarrow B$

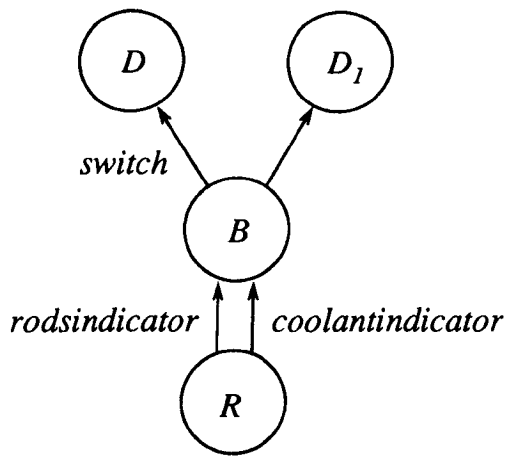
B simply takes the output from R and converts it into input suitable for D. The communication we want - that is, the indicators lighting correctly on the display panel - takes place if the three processes are run in parallel:

$$B \parallel R \parallel D$$

This communication can be illustrated:



Delaying binding in this way has the advantage that, without changing R, we could implement an alternative display simply by changing B:



Making R call directly on D by incorporating the operation names in the output does away with B, but binds R to D. Note, however, two important points:

- It is not possible to plug in *arbitrary* applications R or displays D, in the manner suggested by Coutaz Dialogue Socket [Coutaz85]. This is simply because the alphabets of new Rs or Ds may be disjoint from B, and thus they would not be able to communicate. In the general case, the alphabet of B will have to be extended for each new application or display.

- This architecture is uni-directional. Thus it does not cope with *directness*, whereby some external change to D can be communicated back to R. As was noted in Section 2.7, directness requires either that D is bound early to R, or R is bound early to D.

These are fundamental design issues in providing separated interfaces.

3.6. Conclusions

There are two major motivations for separating dialogue from functionality:

- if dialogue can be *abstracted*, then it should be possible to express this in a notation particularly suited to user interaction.

- if dialogue can in addition be bound late to its functionality, then it should be possible to run this as a separate process, and possibly to incorporate some user-level constraints at run time.

Separate dialogue management is the essential architecture of the classical UIMS. As we have seen, there may be a wide range of possible interleavings of discrete input and output events which are unambiguous in interpretation (via I) into application operations. The justification for dialogue management is that, for cognitive or other reasons, not all of these possible ways of invoking functionality may be unambiguous or at least easy to use for the user.

The design goals for dialogue management in UIMS are thus often to *restrict* the number of ways in which a user may invoke some underlying functionality, by imposing an interactive style such as forms, dialogue boxes, or icon-based syntaxes. Clearly, if the underlying functionality is inadequate or poorly designed, we cannot expect dialogue to be able to *extend* its possible traces in any useful way.

By contrast, the motivation for work on *abstract models of interaction* at York and Oxford is to *guarantee* certain properties of the traces of the functionality visible through the interface. For example, the principle of *reachability* [Dix88a] ensures that any state of the application can be reached from any other state. These abstract principles thus act as checks on *over*-constrained dialogues, but also are principles for the design of functionality. They are therefore bound very early, and we cannot expect them to be *executable.*

Many of the principles of these abstract models require access to application state, at least to effect an equality operation. The principle of reachability, for example, is expressed [Dix88a p.50] by using the equality of the *effects* of two programs on system state.

However, in separating (i.e. late binding) dialogue and functionality we cannot even assume that it is possible to determine whether two application states are the same. In contrast to the PIE model [Dix88a p.37], therefore, we have defined the interpretation function I not from programs (P) to effects (E), but simply from dialogue sequences to sequences of operations.

This chapter has attempted to clarify the scope of constraints on such a separated dialogue. If the dialogue management process is to be generic and effectively separable (fundamental premises in a UIMS), then *semantic seepage*, that is, the binding of the interface to application semantics, must be avoided. This limits user-oriented dialogue management to two classes of constraint:

- constraints which impose a command-invocation syntax, for example, operator-operand or the reverse.

- constraints which are expressed over sequences of commands, but which require no access to application state, that is, which do not require any *interpretation* of the effects of the commands. A constraint like this, for example, might simply limit the number of command invocations per session.

Even the second type of constraint might be undesirable in practice. Imagine the scenario where the human operator of a *Reactor*, following union pressure, was assigned only a certain number of operation invocations per shift, and that this was enforced by a separate dialogue manager. Now the *Reactor* springs a leak and the coolant level drops. It would be a pity if the *Reactor* suffered a meltdown simply

because the operator had run out of his allotted operations and could not raise the rods out of danger.

It therefore can be argued that no constraints imposable by the dialogue manager should limit the possible traces of the functionality, because without knowing the application domain completely it is impossible to predict the effect of such a limitation. On the other hand, as we argue above, if there are useful constraints which do depend on the application state, then the proper place to implement these is in the application itself, not in a separate dialogue manager.

The domain of separable dialogue constraints (or equivalently, principles or predicates on dialogue) is thus effectively limited to syntactic variations on the *style* of command invocation, that is, to the first type of constraint above. This can be expressed formally. If D is an acceptable dialogue constraint under this principle, then when D is used to constrain the *iotraceset* (the input and output events of the dialogue), then it is still possible to generate any sequence of application operations (*traceset*) by applying the interpretation function I to this reduced set of *iotraces*:

$$D \in \textit{separable\_dialogue\_constraints} \Rightarrow$$
$$I\ (\!|\{\textit{iot: iotraceset} \mid D\ (\textit{iot})\}|\!) = \textit{traceset}$$

This allows the implementation of separated UIMSs in which some aspects of the dialogue (i.e. those controlled by the *separable_dialogue_constraints*) can be modified at run time. However, it ignores *how* the dialogue is to be abstracted and expressed (Chapter 2 examined various notations for this). It also ignores important issues of pragmatics and aesthetics in the *use* of the resultant interfaces (for example whether it is useful to switch arbitrarily from a menu/mouse interface to a command line interface). Consideration of these issues can only bind the interface more tightly to the application functionality.

It is possible to conclude therefore that, both in practice and in principle, dialogue management is not suited to implementation in a *separate* interface. Chapter 4 presents an alternative class of models which it *is* possible to separate, and defines an architecture by which this separation can be implemented.

Chapter 4

Surface Interaction

This chapter contains the central formulation of the Thesis of Surface Interaction. Its context is what has gone before: that both *dialogue* within the linguistic architecture, the basis of UIMSs, and *devices* within the agent architecture, the basis of Toolkits, are difficult to separate from application semantics, and therefore provide little opportunity for factoring and user independence.

If there is a creative core to this Thesis then it is the decision to abstract the *medium* of interaction, rather than the *form* (dialogue) or the *devices* of interaction, as a basis for separation.

The medium of interaction, at its most abstract, is the set of values in which the user is interested, and which the application can generate. This might be, for example, the result of some calculation, or a formatted document. At any more concrete level, the medium is some *presentation* of these values which the user can directly perceive, and which represents, under some preferably determined mapping, the application values. Thus in practice the medium may consist of symbols or graphics on a screen or a piece of paper.

This Chapter argues that, so long as the medium's model is sufficiently generic, the implementation of its manipulation and presentation can be factored out from a range of applications. In addition, we expect objects in the medium to interact (or interfere) visually, for example by overlapping, since the display is a resource shared by a number of application processes. The medium and its presentation must therefore be integrated and common to these applications in order to adjudicate these conflicts. It must therefore also have a separate thread of control. This is in contrast

to both to binding the medium in at compile time as a library of primitive tools, and to object-oriented tools which encapsulate their own presentation.

However, the Chapter argues further that there may be some changes to the layout or appearance of medium objects which are irrelevant to the application semantics. For example, the application effect of a dialogue box is usually independent of its position, and so moving the dialogue box to a new position has no relevance to the application. If such changes can be made directly by the user, and independently of the application, then even more can be factored out from applications, to the benefit of both the user and the application writer. This requires, minimally, a separate *user agent* which communicates with the medium and the application on behalf of the user. The Thesis calls the composition of the user agent and the medium an *active* medium, or *surface*. The user interactions which the surface abstracts from applications the Thesis calls *Surface Interaction*.

Prior to specifying particular models (i.e. semantics) for the medium (which we do in Chapters 6 and 8), we need to show in general how a separated surface can factor application concerns as well as provide some measure of independence for the user from particular application domains. To do this we need to establish an architecture whereby the user and the application can communicate via the medium. The architecture must express the communication structure, and preserve the semantics, of the participating components.

This Chapter has six sections. The first section gives the abstract model of interaction which underlies the Thesis. The second section defines the medium. The third section gives the premises and principles for Surface Interaction. The fourth section gives a minimal architectural framework for Surface Interaction, using three processes: the user agent, the medium, and the application (*UMA*). The fifth section gives a concrete example of Surface Interaction, using this architecture and notation from Chapter 3. Finally the sixth section examines implementation issues that necessarily arise in refinement of this architecture.

4.1. Abstract Models of Interaction

The *PIE* model [Dix88a] models an interactive process as an interpretation function (*I*) between sequences of input (*P - programs*) and their effects (*E*) on the computer state. The type of the *PIE* interpretation function can be expressed:

$$I: P \rightarrow E$$

I is thus closely equivalent to the behaviour function β (Chapter 3), since, for any program p,

$$I(p) \in \; ? \; / \; (p \; ? \; \beta) \; (S_0)$$

That is, the effect of a *PIE* interpretation of a program ($I(p)$) is one of the possible states reached if the same program is allowed to produce a composed sequence of state changes according to β, applied to some starting state (assuming that the *PIE* programs consist of events in the domain of β). If the range of β were tightened to a function, then the equivalence would be exact.

As long as we take a global view of interaction, the *PIE* model is sufficient, since necessarily all that we can ever say about interaction is captured by sequences of input and their output effects. However, *PIE* is a very bland model of interaction. While it makes it easy to express properties over input in terms of effects, such as predictability, observability and reachability, it does not reflect some unavoidable constraints that emerge with refinement.

This Thesis therefore uses a slightly richer (i.e. more refined) fundamental model for an interactive process. This takes account of two important features highlighted in the Chapters so far:

- An interactive process may run in an environment where there are a number of *separate* processes, each of which shares the interactive medium.

- An interactive process must allow *directness*. That is, it must allow input to be interpreted in the context of previous output to the medium.

The consequence of these is that we cannot define application interpretation to take place simply over sequences of input (*Input*), as in *PIE*, since for *directness* the matching sequences of states of the medium (*Medium*) must also be taken into account in deciding what the input means. A mouse click, for example, can only be interpreted (as, say, the selection of an icon) in the context of the current state of the display. However, the medium is not fully determined by the input sequences of any *one* application, since we allow multiple applications to effect changes to the medium. If we modelled *each* application as a *PIE*, then the individual *PIE*s, using

input consisting only of mouse clicks and positions, could not determine the interpretation of input without external reference to the state of the medium.

We therefore need to refine the *PIE* model of interaction to make explicit this querying of the medium state. Similarly, we cannot model output from a single application simply as a sequence of medium states, since the medium is a shared resource. Output is therefore modelled in terms of commands (*Command*) to the medium. The commands invoke medium operations, which have the effect of changing the medium state. Our general interactive application (*GA*) is thus of type:

$$GA: seq\ (Input \times Medium) \rightarrow seq\ Command$$

This is sufficient to model an application since any internal state changes which it may undergo are only observable through the effects of its commands on the medium.

The behaviour of the medium itself (*M*) must therefore be modelled *separately* by a function converting sequences of *Command*s (interleaved from possibly a number of applications) into medium changes:

$$M: seq\ Command \rightarrow seq\ Medium$$

However, although we assume here that the *Medium* states which are paired with the *Input* in *GA* are actually those generated by *M*, this formulation says nothing about how this synchronisation and communication is achieved. It is the purpose of this Chapter to make this connection more precise.

4.2. The Medium

We wish to think as generally as possible about the connections between an underlying application state, and a presentation of this in a form the human user can assimilate. For this purpose we define the display *Display* as the set of possible states directly communicable to the user. Thus representations in terms of sound or other media are not excluded, although we think of the display primarily in visual terms.

4.2.1. Model

Clearly there may be a number of nested representations between the application state and its display (the application itself might consist of views, for example of a database). Each of these levels may introduce some non-determinism in the mapping from state to display. That is, given a particular underlying application state, there may be a large number of possible displays that can be generated.

It is useful, however, to isolate a particular intermediate representation whose mapping to the display we can assume *is* determined. In graphics terminology, we can call this representation a *model* for the display. The model may, for example, be expressed in a graphics language like GKS or PostScript, or the graphics package of a window manager or a Toolkit.

4.2.2. Presentation

We define *Model* to be the set of all possible model states. We can therefore assert the existence of a function:

$$present: Model \rightarrow Display$$

present takes a particular model description, and projects this onto the display. If there are any further device dependencies such as pixel resolution which might lead to non-determinism in presentation, then we can hide these in *Display* - that is, if need be we can think of *Display* as some *normalised* display, rather than a physical display.

An important property of *present* is that it should be *total*. That is, we expect all models to be *displayable*. However, we cannot expect *present* to be a bijection (although it may be). For example, assuming a definition (in Postscript) of *square* as follows:

```
/square
    {newpath
        200 200 moveto
        0 72 rlineto
        72 0 rlineto
        0 -72 rlineto
    closepath
    fill
    } def
```

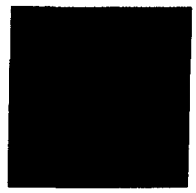
Here are two distinct models:

square

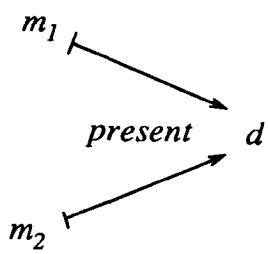
and:

square
square

both of which produce the same display:



This of course is because in the second model the same image is simply drawn twice in the same spot. However, this is sufficient to show that there is not necessarily a bijection between the model and the display. These mappings can be illustrated:



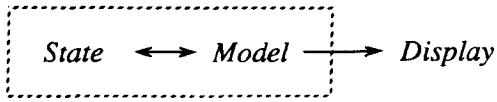
4.2.3. Abstracting the Medium

Terminals and bitmapped workstations commonly provide only a set of low-level text or graphics primitives with which to modify the display. On a terminal there may be operations to insert or delete characters or lines, while on a workstation there may be RasterOps [Newman79] to move arbitrary areas of the screen to and from memory, and possibly some graphics primitives like lines or circles (see Chapter 5).

These primitives, however, have no *identity*, and do not *persist*. For example, the operation to draw a line returns no handle to that line, and there is conventionally no corresponding delete or move operation which can thereby be applied to the

same line. Similarly, there are no facilities provided to allow the user to *pick* lines, simply because the lines do not exist other than as marks on the screen.

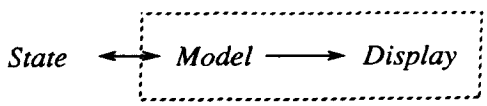
In the implementation of applications on these devices, therefore, it is usually necessary for the application to maintain its own display model by reference to the underlying application state, and to manage all the display updates itself from this model:



application *interface*

In order to provide a direct manipulation style of interaction, for example, the application has to monitor directly all the input devices, and effect the display updates in terms of low-level primitives. If the application wishes to implement a model that allows overlapping and arbitrary geometric movement on the screen, then the *present* function between the model and the display may be very complex. In addition it is usually necessary, for reasons of efficiency, to implement *present* incrementally, so that only those areas of the screen that have changed are actually updated. This compounds the complexity and accounts for a large part of the difficulty of writing highly manipulable user interfaces [Myers88b p.2], particularly in multi-tasking environments.

Since the *present* mapping from *Model* to *Display* is determined, it is clearly possible and advantageous to abstract the model, and to represent this in the interface rather than in the application. This is particularly useful if the model is generic over a range of applications. We think of a *medium* as such an interface:



application *medium*

In this configuration, the medium incorporates both the model, and the presentation of the model onto the display. The medium thus has its own state, and can act as a presentation database [Garrett82] (or *proxy* [Scofield85 p.66]) for the application. The model can be thought to consist of possibly dynamic numbers of

components (for example windows or icons). We refer to these components (without being specific as to their appearance) as medium *objects*.

This definition of a medium corresponds to graphics systems like GKS [ISO85] and PHIGS [ISO87b], in that these provide abstracted models whose presentation implementation is hidden. Perhaps surprisingly, there are fewer current text media, if any. The standard schemes of ODA [ISO87a] and SGML [ISO86b] (see Chapter 5) are intended more for document *transmission* than efficient document presentation.

Note, however, that in the above architecture there is a *relation* between the application and the medium, whereas in the previous architecture there was a *function* between the application and the interface. Thus for any one application state, there may exist a *set* of possible medium states. Thus conceptually we free the medium from complete dependence on the application.

In order to exploit this potential independence we need some other mechanism whereby the medium can be manipulated *separately* from the application. In this way we can allow a number of applications, or the user, to access the medium concurrently.

4.2.4. Separating the Medium

In order to allow the application to effect changes to the medium's model, it would be possible to give it direct access to this. This would be analogous to an older display processor like the DEC VT-11 [Eckhouse79], in which the model is a display program. The central and display processors share the memory in which the display program is held. The central processor updates the display program, which at the same time is repeatedly executed by the display processor in order to present the screen image.

This clearly entails problems of synchronising access to the display program. For example, the display processor must not display a line after the update of one endpoint, but before the update of the second. This synchronisation problem is compounded if a number of concurrent application processes are each trying to update the display program. A number of window systems also give access to their representation structures in this way, and thus require mutual exclusion around such critical updates.

In order to protect the medium we need to do more than simply *abstract* the model into some independent representation like a display program. For true data abstraction we must modularise the medium by providing operations which generate values of the model, without revealing details of their representation.

If we wish in addition to *separate* the medium, we need to delay the binding between the application and the model. This can be accomplished by making the medium an *Object*, that is, by encapsulating model states within the medium. A general type for the medium can therefore be given:

$$MEDIUM = (Model \cup Display \cup present, Command[Model], Init[Model])$$

That is, a medium is an *OBJECT* whose state consists of a model, a display, and a presentation mapping between them, and whose operations *Command* and initialising operation *Init* are defined on the model, but not on the display. The point of this definition is that since the display can be generated deterministically from the model by *present*, then it is only necessary to define the effects of the operations on the model.

As the original Z derivation of an *OBJECT* suggests (see Chapter 3), there is no access either to the model or to the display other than through the operations $Command \cup \{Init\}$. This excludes window managers and other display systems which allow direct access either to their model representation, or to their display screen, from being media in this definition.

Such a separation of the medium has a number of benefits:

- The application handles *identities* of medium objects, rather than their internal representations. These identities *persist* over changes to their properties. Their lifetime is not even tied to the lifetime of the application.

- Simply by knowing the identity of a medium object, a number of agents can access and modify its properties concurrently. Thus the medium, as its name suggests, can form a channel of communication *between* these agents.

- So long as the operations are atomic, the medium can effectively manage the presentation synchronisation, and can handle interleaved updates from a number of agents.

There is yet a further requirement we wish to make of the medium.

4.2.5. Directness in the Medium

If the medium is to act as a channel of communication, then all parties need access to its objects in order to be able to *send* messages by creating or modifying objects, or to *receive* messages by determining the properties of objects. From the point of view of applications, the sending and receiving of messages takes place via symbolic references to object identities in the programming interface to the medium.

The human user, however, receives messages via the *present* function. That is, his only access to the state of the medium, and thus to the state of the application, is via the display. In order to *send* messages back, the user must be able to *address* the display using a pointing device. This is a basic precondition for directness.

Our main requirement is that there should exist a *pick* function which enables the user to address objects (*Ob*) of the model through locations on the display. These locations will form a component of input, for example input generated using a mouse:

$$pick: (Display \times Input) \nrightarrow Ob$$

Thus given a particular input on a particular display, *pick* will return the object which is the target of that input. *pick* may be partial, since not all locations may display objects. It is not likely to be injective, since a large object may be displayed at a number of locations simultaneously. It is also not likely to be surjective, since some objects may not be displayed at all because they are obscured or clipped.

pick cannot be defined in isolation, since clearly it requires reference to the state of the medium's model. We therefore expect *pick* to be one of the *Commands* that the medium offers:

$$pick \in Command$$

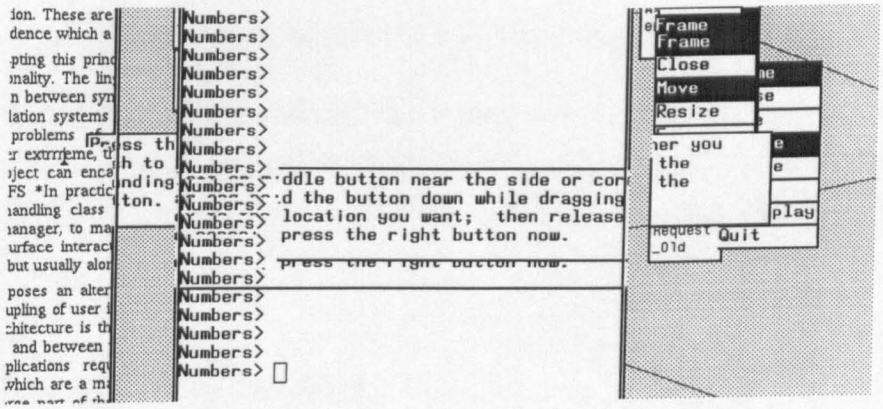
In practice, the *Display* argument to *pick* can be supplied internally by the medium, so that externally the *pick* operation need only be parameterised by the *Input*. That is, we need only ask the medium what object is at the location specified by *Input* in its current *Display*.

The important feature of this definition is that the *pick* mapping is *determined*, that is, that the display is not *ambiguous* to the user with respect to the objects in the model. We do not address here the issues of constructing a *surjective* mapping, through composition of *pick* with viewing functions like scrolling or popping, thereby allowing the user to select *any* component of the model. Formulating abstract models of interfaces which include such viewing operations is an active research area [Harrison90] that is outside the scope of this Thesis.

We also do not address here the relation between the medium state and the application state, as important as this is to the overall interface. This is because at this level all we are concerned with is simply to give the application freedom to construct any medium state by issuing *Command*s to the medium.

4.2.6. Consistency

We need finally to be assured that the displays caused by the medium operations are consistent. In the general case, it is certainly possible to generate corrupt screens. The following screen, for example, was generated simply by mouse operations within SunView:



Given an appropriately low-level operation, it should be possible to generate *all* displays, even those which are simply random configurations of pixels. However, we expect that the displays generated by *present* from any of the medium models are in fact a subset of the possible displays (*range* is defined in Chapter 3):

$$\{m: range\ MEDIUM \bullet present\ m\} \subseteq Display$$

We mean by *consistency* that no matter how a particular state of the medium model has been generated, its presentation is the same. This follows simply from the fact that *present* is defined on model states, rather than on sequences of model operations. However, we restate this here to make it clear that in implementation the presentation of the model should be independent of how the model was created.

This also explicitly excludes giving any application direct access to the display, as often happens in window managers. If there is direct access to the display, then the presentation cannot be predicted from the medium model. A major benefit of this restriction, however, is that any application using the medium does not have to be involved in the complexities of *presentation*, but simply with the objects of the *model*.

4.2.7. The Medium: Summary

We can summarise our requirements of a medium as follows. A medium consists of:

- an encapsulated *model* or *state*
- a set of operations or commands on the model, one of which is a *pick*.
- a total function *present* between the model and the display

Clearly we also imply that the model have some *displayable* content. However we wish to say nothing specific about this here (Chapters 6 and 8 give detailed models for the medium). Note that we also say nothing at this point about how input is gathered.

4.3. Surface Interaction

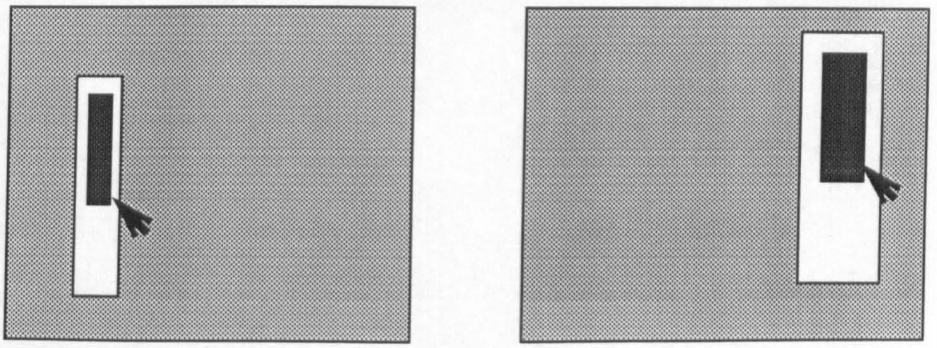
4.3.1. Premise

The fundamental *premise* of Surface Interaction is that there may exist some sequences *si* of input and changed medium states which have *no* application interpretation (using the application interpretation function *GA* from Section 4.1):

$$\exists\, p, si: seq\ (Input \times Medium) \bullet GA\ (p) = GA\ (p \frown si)$$

The effects of such sequences *si* can take place on the display without involving the application. Such interactions we call *Surface Interaction.*

All that is needed to validate this is one example:



We take the case of a standard scroll bar box in which users can change the position and height of the scroll bar. The application monitoring this object makes some interpretation of the height and position of the scroll bar with respect to its box, and uses this to control, for example, the view of a document. This interpretation is entirely independent of the position or size of scroll bar box itself on the screen. Thus input and medium changes pertaining to modifying the size or position of the scroll bar box have no interpretation within the application semantics. The scroll bar box can thus be moved about the screen through Surface Interaction only.

4.3.2. The Surface

There are two fundamental *principles* which are necessary to allow Surface Interaction:

- There must exist a medium which provides the operations to *effect* Surface Interaction. The medium *abstracts* the presentation of its model from applications.

- There must exist a separate process which reads user input and sends commands directly to the medium in order to *invoke* Surface Interaction independently of (any number of) applications. We call this process the *user agent.*

The *surface* we consider to be the composition of the user agent and the medium. The surface is common to a range of applications, and there is typically one

surface per workstation or display. The surface is *separable*, since it is dynamically bound to its applications. The surface model (i.e. the medium model in the surface) can thereby be manipulated as much by the user, via the user agent, as by the application. Directness can thus be supported both through Surface Interaction, and through semantic feedback from the application.

4.3.3. Refinements

The formulation of the application semantics *GA* given in Section 4.1 requires the application to dereference input in the context of the paired medium state. In order to do this it would need to make use of the *pick* function which is one of the *Commands* of the medium. *pick* is clearly generic, since the medium is part of the surface. It should therefore be possible to factor *picking* also into the surface. The application semantics can thus be refined to accept sequences of input paired not with whole medium states, but simply with picked objects:

$$A: seq\ (Input \times Ob) \rightarrow seq\ Command$$

Ob is the set of surface objects (i.e. components of medium state). Any particular picked object (*o*) is determined within the surface from the current input (*i*) and state of the medium's model (*m*) using the *pick* function:

$$pick\ (present\ (m),\ i) = o$$

The invocation of the *picking* operation can thus be taken over by the user agent, which then passes the result to the application. In fact, we can make the user agent a general dispatcher which sees all user input and passes this, as *(input, picked object)* pairs, to the appropriate application of type *A*.

Two further refinements are evident. If some sequences of Surface Interaction have no meaning to the application, then there is no point in informing the application of these. There must therefore be some mechanism to filter input sequences before they arrive at the application. The surface is the ideal site in which to do this, and the user agent the ideal mechanism.

However, we do not wish to be prescriptive about this filtering, since in this way we would reduce the power of the application to impose its semantics on the surface. The solution adopted in this Thesis is to allow the application to determine in advance, on a per-object basis, what filtering will be performed.

Secondly, if the surface is to act as a medium of communication between the user and the application, then the application must have some means of obtaining information from the surface about its medium state. We thus assume that some of the medium *Commands* are enquiries, and some of the input which the application is prepared to accept consists of replies from the surface to these enquiries.

Formally specifying these refinements, however, requires much more semantics (and therefore design decisions) than is appropriate at this level of abstraction. However, a major omission remains the precise connection between the medium objects generated on the surface, and the objects against which input is interpreted by the application. Clearly we want these to be the same, but this formulation in no way defines this.

The *UMA* architecture presented in the next section is a further refinement of the fundamental principles for Surface Interaction which accounts for how the surface *communicates* with the application. The *UMA* architecture also shows precisely how the surface can accept both application commands and direct input from the user by allowing both user/user agent and application/medium communication. In this way the user can impose his own semantics directly on an application's surface objects. For example, he may manipulate a graphical representation of a database schema into a configuration that is meaningful for him (see the example in Section 7.3.2).

The principles of Surface Interaction also presuppose no particular semantics for the surface. That is, there may be any number of surface models and sets of operations on these. Chapters 6-8 for example give a variety of models for the surface. Even a UIMS within the linguistic architecture fulfills the conditions so far, in that it may provide operations for display changes (for example, packaging of command strings, or dialogues to correct input parameter type errors) which occur independently of the application. To this extent such a UIMS has, or has need of, a surface.

The critical difference between a standard UIMS and a surface, however, as we have shown in Chapter 3, is that a UIMS requires knowledge of the operations or state of its client application. We wish a surface, on the other hand, to be ignorant of everything about applications except their existence and identity.

4.4. The *UMA* Architecture

The restriction that the surface should have no knowledge of application semantics (i.e. their state or operations) is supported by showing how application semantics can nevertheless be expressed on the surface. The problem of attaching semantics to surface or presentation level objects is well recognised [Lantz87b p.95].

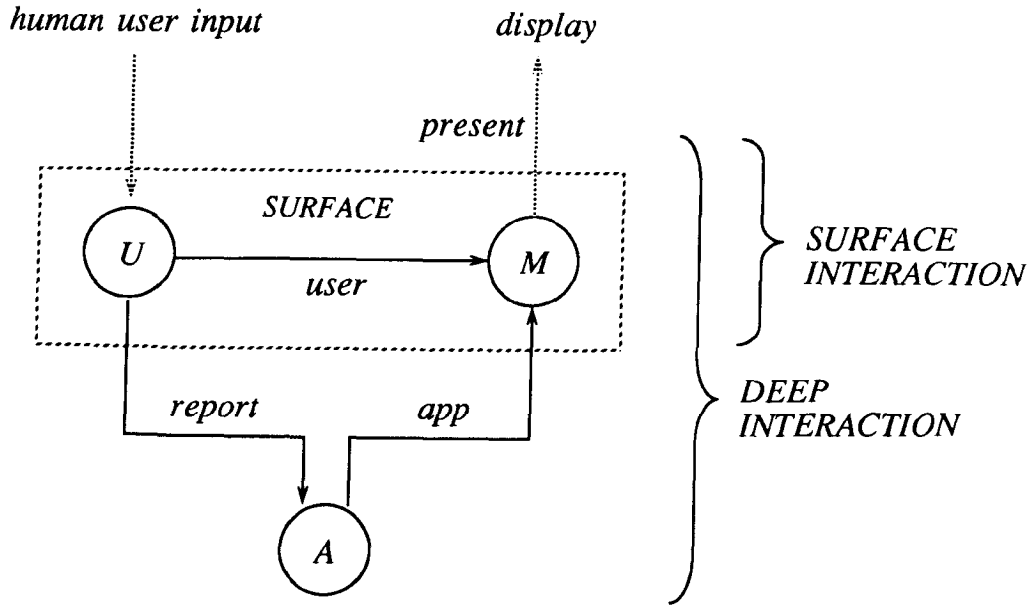
We do not wish to decide *a priori* which user actions will result in Surface Interaction, or which will be reported to the application. We therefore assume that *all* user actions result in Surface Interaction (that is, changes in the state of the surface medium), but that we report these actions to the application *before* their surface effects. In addition, we give the application the power to permit, cancel or subvert these input reports, and to interleave surface operations of its own. The application's involvement we call by analogy, *deep* interaction.

The application can always elect *not* to be informed of certain events on certain objects. In these cases Surface Interaction can take place truly independently of applications. In general, Surface Interaction can be used to manage manipulations that either have no meaning to the application (like moving a dialogue box), or that have meaning but which the application has delegated (like changing the size of the scroll bar in a scroll bar box).

We show how the surface can be affected both directly by the user, and concurrently by applications, by establishing a communication infrastructure between the surface, the user, and applications. By *observing* these communications in abstraction from particular semantics for any of the processes, we can demonstrate how user and application events can be interleaved on the surface, and how synchronisation problems can be resolved.

We use the process notation of CSP [Hoare85] to observe these communication events. Minimally, three processes are involved, which we will call *U* (*user*

agent), *M* (*Medium*), and *A* (*Application*). The communication infrastructure between these processes can be represented diagrammatically:



The solid arrows here are channels, and show the directions from which the communication is initiated. For reasons given later, even though each request is immediately followed by a reply in the opposite direction, we do not need to specify explicitly a channel for replies.

The diagram thus shows that the medium *M* is passive, responding only to commands from the user agent or from the application. It is therefore not dependent on either of these, in the sense that it needs no access to any of their operations.

The actual semantics of the system, that is, what values it generates in response to these actions, will be determined by the particular models of the processes and the operations they allow. Our purpose here is to show that this communication architecture can accommodate any semantics, either for the surface or for the application. This in fact is implied by our use of CSP, which ignores the semantics of its processes, and by the fact that we place no upper restrictions on the alphabets of the processes involved. This *UMA* architecture is a complement to and an essential justification of the principle of Surface Interaction, in that it provides a concrete and practical mechanism for its instantiation.

We now examine the communication requirements on the three processes in the *UMA* architecture in order to provide for the separation of surface and deep interaction. We look first at the medium, then at the user agent, and finally at the application.

4.4.1. The Medium

As we have seen, we think of the medium as a set of commands or operations on an encapsulated state, from which there is a presentation projection to some output display. For the purpose of this description of communication, we simply assume that the presentation projection is hidden within the medium process.

The only assumption we make about the semantics of the medium is that it consist of a number of objects *Ob*, about which we do not want to be specific, except that they are likely to have textual or graphical qualities. The medium will accept a set of commands (called *COM* in order to distinguish these events from the *Commands* of the static description), each of which will be parameterised possibly by an object and some other values. A typical *COM*mand might be:

MOVEX object distance

That is, move object *object* distance *distance* horizontally.

In the general case, we expect each command to return a reply, which minimally may simply be confirmation of receipt or completion, but which more usefully may return information on the state of the medium. We can define, without being more specific, a set of replies: *REPLY*.

In process terms, we wish the medium simply to be ready to accept any sequence of commands, irrespective of the origins of the request. An observation of the medium process *M* is therefore simply defined:

$$M = user?c{:}\ COM \rightarrow r{:}\ REPLY \rightarrow M$$
$$\mid app?c{:}\ COM \rightarrow r{:}\ REPLY \rightarrow M$$

That is, its traces consist of any sequence of commands, each immediately followed by an appropriate reply (the precise value of which is determined by the medium semantics, which here we do not consider). The commands may originate either from the user (along the channel *user*) or from the application (along the channel *app*), and may be interleaved arbitrarily. Having accepted a *COM*mand, however,

the medium cannot be interrupted or preempted, but must return a *REPLY* before being able to accept another *COM*mand.

4.4.2. The User Agent

While we want to allow both the human user and the application to *COM*mand the medium, it is clear that their respective means of communicating with the medium are quite different. Given that we think of the medium as an *OBJECT* consisting of state and operations, its most appropriate implementation is as a software process with a separate thread of control. Applications may therefore communicate with the medium using some internal mechanism like messages.

The human user, on the other hand, can only communicate using physical devices such as a mouse and keyboard. Minimally, therefore, it is necessary to translate device input into medium commands of type *COM*. For this reason a *user agent*, which we can model as the separate process *U*, is necessary to perform this translation. In principle this interpretation should be as direct as possible, to minimise any domain or style bindings at this level. However, some device binding, for example a dependence on a certain number of mouse buttons, may be difficult to avoid. The *Presenter* system, which is described in Chapter 7, for example has a surface model which maps the three mouse buttons of the Sun workstation to the operations of selecting, moving and sizing medium objects.

The user agent, in order to interpret user actions with respect to the medium objects, must first send a *pick* request to the medium to determine which object, if any, is under the current cursor position. While here we are clearly thinking in terms of a mouse and a screen, this approach does not exclude at this level of description any other output medium which contains discrete identifiable objects and a means of indicating or addressing them.

The user agent, in order secondly to allow the application to impose its own *deep* semantics on the interaction, must allow the application to cancel or subvert user input. To do this, the user agent next *reports* the user's input *and* the picked object to the application, and waits for confirmation to continue. This confirmation is granted by the application's returning an input value and object, which may or may not be the same as those which were reported. We thus assume that *input/object* pairs *(i, o)* are part of the alphabet of the application, as is suggested by the static

specification *A* above. The user agent then interprets this returned pair, and sends an appropriate command to the medium.

Clearly, if the *input/object* pair *(i', o')* returned by the application is the same as that generated by the user, then the user's input, as interpreted by the user agent, is allowed to go ahead. However, the application may substitute another input and/or object, and thus change the user agent's interpretation of the user's input. The application may also substitute a null object, in which case the user agent gives no *COM*mand to the medium, and the user's input is effectively cancelled.

The effect of this is to give the application the opportunity to modify user input before it is interpreted by the user agent as a *COM*mand to the medium. This is important since there may well be cases where for semantic reasons user input should not be allowed to have its usual effect. For example, when a user attempts to move one object, such as the background of a diagram, the application may instead require the diagram itself to move. Similarly, an application may have stylistic or ergonomic reasons for switching the user agent's default interpretation of the effect of the mouse buttons.

There is one refinement of this user agent scheme which is critical to the success of Surface Interaction as a principle of interface separation: in some cases user input is *not* reported to the application, and its surface effects (as invoked by the user agent on the medium) therefore occur autonomously. In this way Surface Interaction, seen in effect as the composition of the processes *U* and *M*, allows surface objects to have *behaviour* independently of the application which may have created them. The application can determine in advance, by setting suitable attributes on the surface objects, which events, on which objects, are to be reported. This per-object event mask is interpreted by the user agent. A typical case might be not to report *drag* events to the application, so that the user agent, as the user moves the mouse, makes repeated move commands to the medium without reporting these to the application. The effect is that an object moves autonomously on the display under direct user control.

From a semantic point of view therefore, we can define a general type for *U*:

$$U: seq\ Input \rightarrow seq\ (\textbf{deep}\ \langle\langle Input \times Ob \rangle\rangle \mid \textbf{surface}\ \langle\langle Command \rangle\rangle)$$

Thus the user agent *U* takes a sequence of user input, and generates a sequence of either **deep** *(i, o)* reports to the application, or **surface** Commands to the medium.

From the process point of view, the user agent process U, which incorporates these features, can be defined by observing its communications. In order to do this, we assume a set I of user-generated input events. To be most general, we can think of each of these events as consisting of a value, a location, and a time (a *what*, a *where*, and a *when*). In conventional terms, the value may be the keystate, and the location the cursor position in screen coordinates. In addition, we need to specify one command subset *pick* $<<I>>$ from the set of medium commands *COM*. We assume that a *pick* command along the *user* channel, parameterised by the user's input, returns a target object o (which may be null) as a *REPLY* from the medium:

$$
\begin{aligned}
U = i{:}\, I \rightarrow\ & user!pick\,(i) \rightarrow o{:}\, REPLY \rightarrow \\
& (user!c{:}\, COM \rightarrow r{:}\, REPLY \rightarrow U \ \sqcap \\
& report!(i, o) \rightarrow (i', o') \rightarrow user!c{:}\, COM \rightarrow r{:}\, REPLY \rightarrow U)
\end{aligned}
$$

Thus the user agent U is the process which always first accepts a user input event i, then asks the medium to *pick* the object o which is the target of the input (based on the location component of i). Having received this *REPLY* from the medium, there is then a choice of two possible courses of action, determined internally by the attributes of the object o returned from the medium. The courses of action are:

- the picked object o's attributes allow the default Surface Interaction to occur without informing the application, so the appropriate *COM*mand is sent immediately to the medium.

- the picked object o's attributes determine that deep interaction should occur on this event. It is therefore reported to the application. Once the input/object pair (i', o') is returned from the application, an appropriate *COM*mand is sent to the medium.

For brevity, this description of the user agent's behaviour omits two obvious refinements:

- if a standard *drag* interpretation is made of mouse buttons, then it is not necessary repeatedly to send a *pick* request to the medium during the drag operation - the user expects the same object to be moved on the screen throughout the drag. In effect the user agent gets the pick information for each move action from its own simple memory of which object was picked when the mouse button was depressed. Indeed, in some situations, for example when moving an object like a scroll bar which can only move in one direction, it is

possible for the mouse cursor to slip off the object. In this case a repeated pick request would lose the object, yet normally the required interaction is for the user to continue to move the originally selected object within a manipulation *mode* (see Section 2.2.2).

• if the returned input/object pair (*i'*, *o'*) is null (i.e. the application has cancelled the input), then no *COM*mand need be given to the medium.

4.4.3. The Application

To show how Surface Interaction allows application semantics to be imposed on the surface, we define the behaviour of a generic application without reference to any particular application domain. Since also we want the architecture to be general, we must support not only the possibility of Surface Interaction, but also its absence. That is, we must cater for applications which need to make an interpretation of *all* user input, as well as those which can afford to let the user agent handle some part of the interaction.

The application is thus able to determine its involvement in interaction by the settings it gives to the event masks of its objects. In the extreme, all objects could report all events. With this understanding, we go on to describe the communications of the application.

The application receives a reported input event and *pick*ed object pair *(i, o)* from the user agent, and is then able to send an unlimited number of commands to the medium (for example to pop up menus, or make other surface changes), before confirming, altering, or cancelling the report as the pair *(i', o')*. Of course, in the meantime the application is also able to perform its own domain-specific computation, but this is hidden here.

In a standard callback or action routine situation where main control is external to the application, which is structured as a set of event handlers to be called by the input level (like Sun's Notifier [Sun86]), this behaviour would be all that is required. However, we wish the application to have full concurrent operation, possibly to monitor other processes, and to be able to *initiate* medium operations spontaneously, for example to manage animation or to report to the user important events in the environment. This may be interleaved with receiving reported user events *(i, o)*. This capability is an essential requirement in the construction of process monitoring and

control applications. The general behaviour of the application process A is thus defined:

$$
\begin{aligned}
A = \; & report?\ (i, o) \rightarrow \\
& \quad \mu X \,.\, (app!c: COM \rightarrow r: REPLY \rightarrow X \sqcap (i', o') \rightarrow A) \\
& \mid app!c: COM \rightarrow r: REPLY \rightarrow A
\end{aligned}
$$

That is, the application A is the process which cycles over two possible subsequences:

- it accepts an input report (i, o) on the *report* channel, sends a number of commands to the medium along the *app* channel, and then, at some point determined internally, returns confirmation to the user agent in the form of an input/object pair (i', o').

- it spontaneously sends a command to the medium.

If a user input report is available, then we wish A to deal with this as a matter of priority. However, we cannot express this requirement in this notation.

The application has thus four ways of imposing its semantics on the surface medium:

- It can set the attributes of medium objects which determine which events will be reported to the application (this is not expressed in this notation).

- It can modify these reported events and thus alter the actions of the user agent.

- It can cancel the actions of the user agent (by returning a null (i', o')), and instead itself send *COM*mands directly to the medium in response to the user's input.

- It can spontaneously send *COM*mands the medium in response to some internal events.

4.4.4. The Surface

The surface consists simply of the user agent and the medium running in parallel. Their internal communications along the *user* channel can be hidden from both the user and the application:

$$SURFACE = (U \parallel M) \setminus \{user.v \mid v \in \alpha user\}$$

4.4.5. An Observation of Surface Interaction

Surface Interaction occurs when the user agent, the medium, and the application are run in parallel:

$$U \parallel M \parallel A$$

Essentially, changes to the medium may originate either from the user, or from the application. The medium is ignorant either of the user agent or of the application, and simply accepts *COM*mands and returns *REPLY*s repeatedly. Application commands to the medium are dealt with directly. User input, however, is read by the user agent, which first reports it (and the *pick*ed object) to the application which returns it modified or unchanged. The user agent then interprets this modified input as a command to the medium. The benefits of Surface Interaction as a principle of application and interface separation arise from the fact that in some cases the user agent may act independently, and send *COM*mands to the medium as a result of user input without reporting to the application.

Control ultimately lies with the application, as it should. It can determine in advance, by setting attributes on the surface objects, which subsequences of the interaction it can leave to the user agent, that is, to Surface Interaction. An application maintaining a dialogue box, for example, may decide that the location of the box on the screen is immaterial to its semantics, and allow the user to move this around through Surface Interaction, without being informed of these events.

An important feature of this behavioural structure is that the application can impose its semantics dynamically either by modifying user input, or by interpolating its own commands to the medium. That is, we allow the application access both to the low-level input events in I sent to it by the user agent, and to the set of medium commands *COM*. In practice, most applications may be content with being informed of only a small subset of the user's input (for example, mouse button clicks), and concentrating their output semantics in *COM*mands to the medium. Others, in particular applications which control the look and feel of the surface environment, may wish to take a hand in user input at a finer granularity (drag events, button releases), and even to supplant the user agent's default interpretation of user input.

There is no danger that any of these communications should go awry, for example that the application should accept a *REPLY* from the medium that was actually in response to a *COM*mand from the user agent. This is because all processes here observe a strict alternation of input and output events [Hoare85 p.198]. For example, once the application *A* has had a command accepted by the medium *M*, then the next event can only be a *REPLY* by *M*, and only *A* will be ready to receive this. Similarly, the application cannot accept two reports from the user agent in a row, without first returning confirmation to the first.

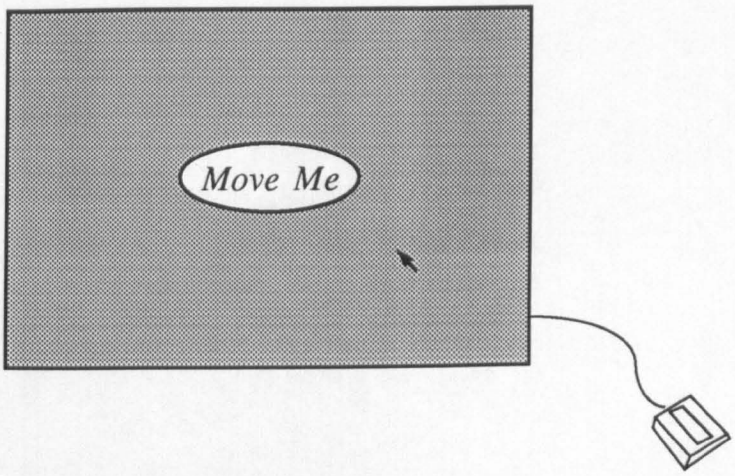
The medium is prevented by the same mechanism from handling more than one *COM*mand at once, without *REPLY*ing first to the first. Thus multiple access to the medium model is automatically synchronised, and the surface actions of the application and the user agent cannot interfere with one another. This also allows the medium to be implemented as a single thread of control, which is a convenient mechanism for implementing this synchronisation [Scheifler86 p.84, Gentleman81 p.445]. Without this, there would have to be some complex locking of screen updates from competing applications [Lantz87b p.94]. This synchronisation is made more powerful by the fact that the medium, because it is accessed directly by the user, is designed to provide user-level objects and operations rather than bitmaps (see Chapters 6 and 8). Whereas in a conventional window manager a complex screen operation might have to be accomplished by a sequence of bitmap operations, in the surface this can be atomic.

Thus the medium can never be blocked, since it does not initiate any communications, and its *REPLY*s are always awaited. The application can only be blocked waiting for a *REPLY* from the medium. The only real restriction on the traces of Surface Interaction is that the user agent may be blocked from accepting user input either while waiting for a *REPLY* from the medium, or while waiting for confirmation from the application of an input report.

We have given the bare bones of a communication scheme, in the form of the *UMA* architecture, detailed enough to show how Surface Interaction works in practice, and general enough to be able to incorporate any application or surface semantics.

4.5. A Simple Surface

The effectiveness of the *UMA* architecture in promoting the separation of surface and deep interaction can be illustrated by a simple example which could easily be scaled up into a useful system. We take a medium consisting of a single icon:



The only input device is a one-button mouse. The overall behaviour that we want is for the user to be able to move the icon around the screen by clicking on it and then dragging. However, as he *releases* the mouse button after a drag, an action is taken which depends on the position of the icon. The precise action is not important - it might be to change the angle of a video camera, or to set the selling price of shares.

The principle of Surface Interaction exploits the clear conceptual distinction here between the operations to pick and move the icon on the display (Surface Interaction), and the 'semantic' operation to initiate some domain action (deep interaction). In this way, we can separate these two concerns. On the other hand, in a conventional implementation within a window manager, the application would manage both the domain action *and* the movement of the icon around the screen.

This example will show how the communication requirements for Surface Interaction established above are satisfied in practice with particular semantics for the *U*, *M*, and *A* processes.

4.5.1. The Concrete Medium

A state-based description of the surface semantics can be given in Z. The basic medium state is simply the icon position, defined by a pair of coordinates:

ICON

$$iconpos: \mathbb{N} \times \mathbb{N}$$

In order to dereference the mouse position, there is a *pick* operation:

PICK

$$\Xi ICON$$
$$pos?: \mathbb{N} \times \mathbb{N}$$
$$hit!: true \mid false$$

$$hit! \equiv (pos? = iconpos)$$

The Ξ prefix indicates that while *PICK* is an operation (i.e. it has pre- and post-states), it does not *change* the state of *ICON*. That is,

$$iconpos' = iconpos$$

We also assume for simplicity here that the granularity of *iconpos* and *pos?* is sufficiently coarse to allow a hit simply when they are equal.

In order to move the icon, however, we need an operation that does change the *ICON* state:

MOVE

$$\Delta ICON$$
$$to?: \mathbb{N} \times \mathbb{N}$$

$$iconpos' = to?$$

MOVE does not have any explicit output, because we assume that the medium contains a presentation mapping that displays the icon at the appropriate position.

The *ICON* state and operations can be thought of as an *OBJECT* (see Chapter 3):

$$\textit{ICON} = (\textit{ICON}, \{\textit{PICK}, \textit{MOVE}\}, \textit{Init})$$

We assume an initialising operation *Init* that puts the icon in some starting state, for example in the centre of the screen.

The ***ICON*** *OBJECT* can be *observe*d to produce a process, and the alphabet of this process is obvious from the specification of the operations:

$$\alpha(\textit{observe ICON}) = \textit{PICK} << \mathbb{N} \times \mathbb{N} \times (\textit{true} \mid \textit{false}) >> \mid \textit{MOVE} << \mathbb{N} \times \mathbb{N} >>$$

In order to fit this process into the communication architecture for Surface Interaction presented above, we have to make a number of simple refinements:

- We put the output component of ***PICK*** into a separate event ***hit***, using the notion of an *invocation* from Chapter 3.
- We assume ***MOVE*** returns an event ***ok*** $\in$ *REPLY* upon completion.
- We assign the operations to the channels *user*, *app*, and *report*.

The event ***ok***, and the values *true* and *false* which ***hit*** can take on, are the medium's *REPLY*s. Thus we can define the traces of a concrete medium process *CM* whose functionality is that of ***ICON***, and which conforms with the architectural requirements of Surface Interaction:

$$\begin{aligned} \textit{CM} = {} & \textit{user?}(\textbf{PICK}\ p) \rightarrow \textbf{hit}\textit{: (true} \mid \textit{false)} \rightarrow \textit{CM} \mid \\ & \textit{user?}(\textbf{MOVE}\ p) \rightarrow \textbf{ok} \rightarrow \textit{CM} \end{aligned}$$

Thus *CM* accepts either a ***PICK*** or a ***MOVE*** from the user agent, and returns an appropriate *REPLY*. *CM* conforms with the canonical medium process *M* in the sense that its traces are a subset of the traces of *M*. The major simplification is that *CM* does not accept any *COM*mands from the application.

4.5.2. The Concrete Application

In exactly the same way we can define a concrete application with the required functionality, and fit this into the architectural framework.

We do not want to be explicit about the application state or operations, other than to assume there is some domain action ***DO*** which takes a pair of coordinates as a parameter. The traces of this concrete application *CA* are therefore:

$$CA = report?\ p \rightarrow \mathbf{DO}\ p \rightarrow\ p\ \rightarrow\ CA$$

Thus *CA* cycles around a fixed sequence of events which starts with a report from the user agent containing a location p (in the canonical form, the user agent reports both an input event and a picked object, but in this case there is only one object, and the only component of relevance is its location). The application then performs its domain action ***DO*** using p as a parameter, and then returns p to the user agent as confirmation of receipt (in this example there in fact would be nothing to stop the application returning p immediately, and then performing the ***DO***).

Again, the traces of *CA* are a subset of the traces of *A* in the architectural model, and so *CA* conforms with this. The major simplification is that *CA* does not send any *COM*mands to the medium, and does not act spontaneously.

4.5.3. The Concrete User Agent

We want to be explicit about the semantics of the user agent, just as we have been about the medium. We also want to show how its communications conform with those of the architectural model *U*. However, the communication traces of the user agent are more complex than those of the medium or the application, both because they involve three channels rather than two, and because they are dependent on *values* passed from the other processes. A strict CSP definition of the user agent cannot capture all that we want. We therefore use an alternative method of specifying the traces of the user agent that incorporates its semantics.

The concrete user agent *CU* must perform a number of simple, but necessary, functions:

- It must determine the target of the user's selection by sending a ***PICK*** request to the medium.
- It must implement some interpretation of raw input as medium operations, for example to ***MOVE*** the *ICON* when drag events occur.
- It must report some subset of user input to the application.

First of all we define a simple tuple type to carry raw input events from the user:

RAW_INPUT

location: $\mathbb{N} \times \mathbb{N}$
mode: down | *up* | *drag*

Thus the input events that come in from the mouse have two components: a mouse *location* in screen coordinates, and a *mode* which indicates whether the mouse button has been pressed (*down*), released (*up*), or that the event has been generated because the mouse has been moved some set distance (*drag*).

We define the communications of *CU* by specifying a function *trace* which converts the sequence of *RAW_INPUT*s that come from the mouse into sets of sequences of communication events from the alphabet of *CU*:

trace: moving | *notmoving* $\rightarrow$ *seq RAW_INPUT* $\rightarrow$ *seq* α*CU*

trace m <> = <>
trace notmoving (i:si) = trace notmoving si
if *i.mode = down*
    trace m (i:si) = P $\frown$ *trace moving si* ***if hit*** *= true*
        = P $\frown$ *trace notmoving si* ***otherwise***
if *i.mode = drag*
    trace moving (i:si) = M $\frown$ *trace moving si*
if *i.mode = up*
    trace moving (i:si) = R $\frown$ *trace notmoving si*

Where
    *P = <user?**PICK** i.location, **hit**>*
    *M = <user?**MOVE** i.location, **ok**>*
    *R = <report?i.location, **p**>*

Here we use a recursive definition in which we assume *m* is a variable with values *moving* or *notmoving*, and *(i:si)* is a sequence of *RAW_INPUT*s with *i* as its head. Even this simple functionality illustrates how the user agent needs to make an

interpretation over *sequences* of input events, rather than over single input events. This is because the interpretation of a *drag* event is dependent on whether or not the mouse button has been previously pressed and not yet released. Only in these cases can we make a ***MOVE*** request to the medium.

This could have been expressed simply as constraints over sequences. However, *trace* makes more explicit that in implementation the user agent would have to keep some state flag (*m*) to indicate whether or not it was in ***MOVE*** mode. If there were more objects than just *ICON*, then the user agent would also have to retain a memory of which object was current during a drag.

In addition, the traces generated are influenced by the value returned from the ***PICK*** request. If there is a ***hit***, then *CU* can go into *moving* mode, whatever the mode before (thus we do not depend on a strict alternation of button *down* and *up*). If ***hit*** is *false*, then *CU* goes into *notmoving* mode. Finally, upon button release (*up*), and so long as *CU* is in *moving* mode, then a report can be sent to the application, and the *CU* returns to *notmoving* mode.

Because the value of ***hit*** is determined externally, and not within this definition, *trace* is strictly a relation rather that a function. That is, we specify alternative traces following the ***hit*** event. Thus an application of *trace* to an input sequence actually generates a *set* of sequences. We use the notation loosely at this point, but the intention is clear.

Given that we place no restrictions on the sequence of *RAW_INPUT*s, we can define *CU* from *trace*:

$$traces\ (CU) = \bigcup \{i: seq\ RAW\_INPUT \bullet trace\ notmoving\ i\}$$

That is, the traces of *CU* are the union of all the sets of traces generated from particular input sequences, starting with the *CU* in *notmoving* mode.

Using the abbreviations *P*, *M*, and *R* from the definition of *trace*, the traces of *CU* are essentially the language

$$(P^{+}M^{*}R)^{*}$$

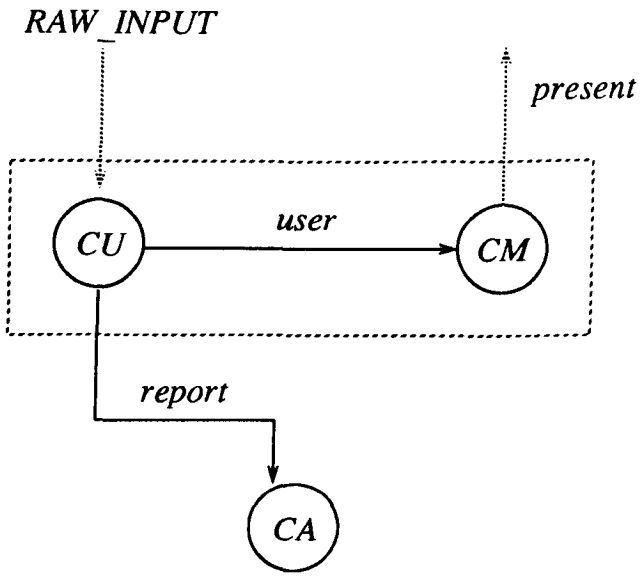
That is, some number of missed ***PICK***s, followed at least by a ***PICK*** which ***hit***s, followed by some number of ***MOVE***s, terminated by a button release and application report. One such expanded trace might be:

$$\underbrace{<P, M, M, M, M, M, M, R>}_{\textit{Surface Interaction}}$$

Such traces clearly show Surface Interaction taking place as the *ICON* is ***MOVE***d. During this time, the application is not involved. In order to achieve this, we have built the decision on *which* events to report to the application into the semantics of *CU*. In a more complex example, we would allow the surface objects themselves (and thus ultimately the application which creates them) to determine what events on them were to be reported.

4.5.4. The Communication Structure

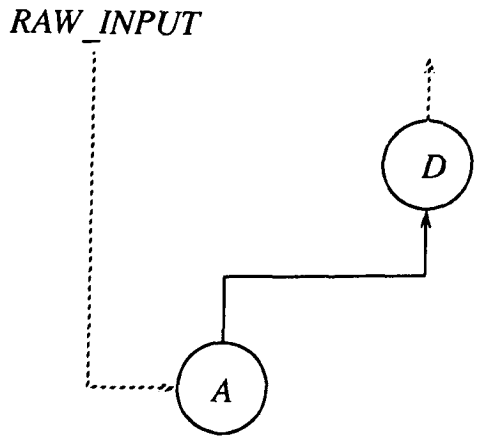
The communication structure implied by the definitions of *CM*, *CA* and *CU* can be illustrated:



This clearly conforms with the architecture for Surface Interaction. The only difference is that the *app* channel from the application directly to the medium is unused. A simple extension to the example would have *CA* not only controlling some external state, but also monitoring external state which could be reflected in the position of the *ICON*. For example, the application might read some sensors which measured the height of fluid in a tank, and the *ICON* would represent this level. In this case *CA* would spontaneously send a ***MOVE*** request to *CM* along the *app* channel. The medium, as represented by *CM*, would then more clearly be acting as the *channel* of

communication both from the user to the application, and from the application to the user.

By contrast, the communication structure of a standard window-based application is closer to:



That is, the application *A* must see all input, and manage all interaction, while the display *D* is a simple projection, and holds no state that can be accessed by *A*.

4.6. Implementation Issues

In refinement and implementation of Surface Interaction a number of issues must be considered which we have deliberately ignored in the abstract view above.

4.6.1. Performance

Efficient performance is critical to the acceptance of a user interface, both by end users and by application programmers. The restriction that every surface command should have a reply is not in fact a performance liability, as is claimed for instance by the X implementors. Most requests to the X server do not expect replies for this reason.

Surface interaction maintains performance in direct manipulation because it is precisely the mechanics of surface manipulation that applications can typically afford to ignore, for example the multiple *move* commands that make up a smooth object drag across the screen. The major difference between the surface as defined here and window manager servers like X is that the surface *retains* and *maintains* (in the

medium's model) all state relevant to the display. For this reason the surface can manage direct manipulation on its own. There is also therefore a much higher likelihood that the surface will be queried about its state, and therefore be required to give replies, than with servers that retain little other than the position of windows.

The reporting of user input first to the application is similarly offset in performance by the fact that not all input is so reported. In fact, this is a performance *improvement* on normal input handling mechanisms, in which the application receives *all* user input occurring in its window.

There may however be more extreme performance problems. Deadlock may of course occur simply because the human user is not prepared to engage in any of the events offered by the system. We cannot also exclude deadlock from occurring within a badly designed application. But it is impossible for deadlock to occur within the medium because the medium is a single process and depends on no external resources. Similarly there are no cycles of dependencies that involve the user agent.

Livelock, however, cannot be so easily avoided. If the application goes into a loop when it is reported user input, then the user agent can accept no further input, and Surface Interaction stops. For the same reason, an application which delays confirming reported input will hold up Surface Interaction. There is no way round this possibility without compromising the alternation of process input and output upon which the reliability of the communication is built. However, it is not an onerous regime to require applications to respond promptly to input - if necessary, they can delegate agents to do this for them.

4.6.2. Timing

The CSP notation does not allow the expression of any timing constraints on communication. That is, communication is considered to take place instantaneously. Clearly, if the surface and applications are distributed around a network, this may not be the case in practice. However, timing is much less critical in general user interface systems than it is in process control applications, for example, because human users are more tolerant of delays or variations in timing than machines. Indeed, delays are often deliberately introduced into interface responses to give the user the impression that the computer is pondering deeply.

One area in which explicit timing constraints do enter into the implementation of Surface Interaction is in the practice of multiple clicking of mouse buttons. In effect this is simply a way of extending the number of discrete commands that can be issued through the mouse - a less common alternative is to 'chord' the mouse buttons with keyboard keys. A multiple click is determined to have occurred if a click happens within a certain time limit of a previous click.

If multiple clicking is to be used at the surface, then it is preferable that this be implemented in the user agent rather in than the application, simply because of the unreliability of timing (in UNIX at least) over inter-process communication. The issue is whether the user agent should report *all* click events to the application, or delay in case a subsequent click follows within the time limit.

In terms of interface semantics, there would be little point in providing an object which only responded to a double click, since in this case we might as well recognise just a single click. Thus double or multiple clicking is necessarily used in cases where the object will also respond to single clicks.

The design trade-off is thus: if the user agent waits to see if there is a second click, then if the user only wishes to issue the single click command, he must wait until the double click time limit expires before any action occurs. On the other hand, if the user agent reports the first click to the application immediately, and then waits to see if there is a second click, then the object will respond to the first click (unless the application also implements a multiple click semantics), and then may immediately after have to respond to a double click.

In practice it has turned out preferable to report clicks immediately, and then possibly also signal multiple clicks. This is because in many cases the semantics of a single click are trivial (for example, the object is highlighted), and are easily cancelled or extended if a double click is deemed to have occurred.

In fact in the implementation of *Presenter* multiple clicking is used to climb the object tree (see below), but only single clicks are ever reported to the application. That is, in this system the user agent uses multiple clicking to access the medium, but does not create separate multiple click events to be reported to the application. However, the application can still attempt to impose its own timing interpretation on the clicks.

In design it would seem useful to create multiple click events, since these can always be interpreted by the application as single clicks if it does not wish to place any special importance on a multiple click. A further useful refinement would be to specify the clicking policy object by object, rather than globally, but this has not been implemented.

4.6.3. Binding User Agent and Medium

Although the user agent is defined as a separate process from the medium, in practice, because of the large amount of traffic between the two, it is convenient to bind them together into a single surface process. This is the case, for example, in the implemented system *Presenter*. It is possible to bind the two because we expect there to be only one user agent per medium.

The *disadvantages* of implementing this binding are discussed in the sections below.

4.6.4. Multiple Applications

The *UMA* architecture envisages only one application. This is a sufficient model if we only want to provide surface facilities on a per-application basis, and use windows provided by a conventional window manager as the displays for the surfaces of each application. This is the case, for example, in *Presenter*. However, by design and logically, the surface can provide a display environment for multiple applications. That is, the same surface constructs can support both inter- and intra-application displays - the global interface usually provided by a window manager, and the particular interfaces of individual applications, can both the accommodated on the surface. The advantage of this is interface consistency, and, provided the surface medium has a powerful enough model, we can avoid some of the geometric limitations of conventional window managers as outlined in Chapter 5.

Conceptually there is no problem in extending the abstract architecture to handle multiple applications, so long as we refine the model to include a notion of object ownership. That is, every object is owned by one application:

$$owner: Ob \rightarrow APP$$

(where *APP* is a *set* of applications). On the basis of this object ownership function, surface events can be reported by the user agent to the appropriate application.

The ownership function does not exclude multiple applications from *knowing about* a single surface object (i.e. it is not injective). In this way we can implement surfaces in which objects are acted on by a number of applications concurrently, for example to simulate the effects of different forces, like gravity and propulsion, on a physical object, or to allow 'groupware' [Elwart-Keys90, Lauwers90] in which a number of users cooperate on a task.

4.6.5. Fairness

Once multiple applications are allowed, the direct communication between them and the surface becomes more problematic, since an application may hog the surface by continually sending it commands. There clearly may require to be some scheduling of application requests, in order to preserve fairness between applications. Communication between the user agent and multiple applications is not a problem, other than that of livelock mentioned above, since this communication is effectively scheduled by the user.

Applications too, must be fair about processing user input promptly. This does not prevent applications from involving lengthy computation. It simply requires that an acknowledgment of a user input report should be returned quickly to the user agent. However, we make no assumptions about whether applications wait for user input reports, or on the other hand use some interrupt or polling mechanism to allow domain computation to proceed.

4.6.6. Object Structures

The formal model of Surface Interaction presented above only assumes that the surface medium consists of a set of objects. In practice, however, it is evident that surface objects are often perceived as grouped into part-whole hierarchies (e.g. character - word - sentence - text), or inheritance hierarchies, or even arbitrarily networked as in hypertext and hypermedia. It is a design issue as to which structuring is provided at the surface, or which is considered domain dependent and therefore properly to be maintained in the application. For example the implemented system *Presenter* has tree structuring of objects (see Chapter 6), whereas the formal surface model given in Chapter 8 incorporates inheritance hierarchies.

If the surface has structures, then clearly these too may be owned by particular applications. It is an unresolved design issue whether objects should be owned independently of the structures, that is, whether an object can be owned by a different application from that which owns the structure of which it is a part. This is of relevance if we allow objects to be cut and pasted between multiple applications. The issue is whether an object cut from one application's structure and pasted into that of a different application should necessarily change ownership.

4.6.7. Picking

Whereas the abstract model allows the medium to return a single object as the target of a *pick*, many systems allow the user to extend the selection of surface objects. In text this may be by dragging the cursor over a number of characters, while in graphics a common mechanism is to use a rubber selection box which selects all objects inside. A more realistic template therefore might be to return a set or list of picked objects.

In addition, the surface model may allow a single surface object to be replicated in a number of displayed images, for example to provide running headers in a document or to share a graphics primitive between a number of different locations on the display. This is possible using the model of Chapter 8. It would also be possible if the surface model were expressed in a procedural graphics language like PHIGS or PostScript in which a number of different calls can be made to the same drawing routine in the context of different transformations.

In these cases the mapping between the selection and the pick is more complex, and will involve user interface design decisions. For example, a user who selects an icon whose image is shared by other icons may intend either to select just that icon, or conversely all icons with the same image (in order, for example, to perform a global edit). The design decision is whether to allow the user direct access to objects in the surface model, or just access to their displayed instances.

4.6.8. Stylistic Binding

A full surface environment for handling multiple applications must include some frontline user interface, otherwise there would be no way for the user to get access to the system (see Section 5.1.3). The basic functions of such a user interface are

typified by an operating system command interface - the essential requirement is the ability to invoke and possibly kill applications.

Current user interface development has greatly enhanced the functionality and usability of this basic task. Within desktop or other overall metaphors, there exist interactive mechanisms like menus, scroll bars, and dialogue boxes with increasingly sophisticated 'look and feel', and a wide range of fingertip services like editors, spelling checkers, and calculators. However, at base the functionality is simply that of allowing human users to invoke applications.

The fundamental trade-off in providing sophisticated user interfaces is that between consistency and flexibility. That is, at some point the interface is bound to the environment such that it cannot be modified either by the user or by the application. It can then have the benefit of standardisation, but on the other hand it blocks the development of new interactive styles.

We wish the general model for Surface Interaction to be as free from stylistic bias as possible. Stylistic bias can enter into a refinement of the model at two points

- the medium model could be biased. For example, its constructs could consist of windows, icons and menus.

- the user agent could be extended to provide the frontline interface. For example, the user agent could request the medium to construct interactive techniques like menus etc., and interpret user input in these terms.

While the abstract specification does not exclude these possibilities, in intention, and in practice in *Presenter*, we wish to minimise the stylistic binding within both the medium and the user agent. In the medium this is a matter of providing a model with simple, generic constructs. Two such models are given in Chapters 6 and 8. Within the user agent it is a matter of confining the stylistic binding to simple one-to-one mappings of input events to medium commands. Some stylistic binding necessarily remains, however, for example the mapping of mouse buttons to classes of command. Without this, we would have to return to the situation in which the application interprets all input events. In fact, as we have shown, the communication architecture does allow an application this level of involvement, but with the loss of the benefits of Surface Interaction.

In design, we therefore expect a specialised application to exist which would generate and manage the frontline interface, for example in terms of windows or other interactive techniques, but we do not want to prescribe what these should be. This application may have to have special priority over other applications, but this an operating system issue.

The interface application will principally be active on three occasions:

- when the system starts up, it will have to initialise an appropriate surface configuration, for example an empty desktop.

- when the user indicates he wishes to start an application (by typing a command or clicking an icon provided by the interface application) it will have to invoke the process and set up the appropriate communication channels.

- when a client application is killed or dies abnormally, it will have to decide what to do with that application's surface remains, based possibly on what X calls the application's *shutdown mode*. Typically the dispossessed surface objects of a dead application would be deleted.

The benefits of separating the mechanism of interaction from the policy or style of interaction in this way has been recognised by a number of recent interface management systems, including X. Although these systems often distinguish between the windowing interface and the toolkit or widget set, from the point of view of the surface both of these are simply applications which provide a stylistic binding to the basic task of application invocation.

4.6.9. Buffering

The CSP notation assumes that no event can take place unless all processes which have that event in their alphabet are ready to accept it. Human users, however, are not normally bound by this restriction. That is, they may generate many input events (by, for example, waving the mouse, or holding down a repeating key) before the underlying application is ready to process them. As has been shown above, the user agent is blocked while it waits for an application to return confirmation from an input report. It is therefore between the input devices and the user agent that the potential backlog of input events can build up. In practice the user agent can deal with this possibility in three ways:

• it can block the user by turning off all input echoing until the application is ready to receive the next event.

• it can echo input, but lose it until the application is ready.

• it can buffer input events, and echo it either as it enters or leaves the buffer.

The first option is the least acceptable, because the user is not able, in a multi-tasking environment, to switch to a more efficient application while waiting for the input processing to complete. The second is a possible option where not all input events are critical to the interaction, and the user expects to repeat input events, like mouse clicks, that do not have an effect. The third option allows *type-ahead*, and thus is more suited to an expert user. This option would also be required for applications which need to know all input events, for example the sequence of mouse locations making up a freehand line in a draughting tool.

Communications between applications and the medium should also be buffered, if applications are not to be blocked while they wait for access. However, the model assumes that applications wait for replies from the medium. On the other hand, communications between the user agent and applications need not be buffered, since the specification *requires* the user agent to be blocked while it both waits for the attention of the application, and waits for its confirmation to continue. However, we expect the application to process reports from the user agent as a matter of priority.

4.6.10. Channels

The precise configuration of channels is to some extent orthogonal to the communications that occur between processes. This is because, at the extreme, we could establish a separate channel for each event in the common alphabet of two processes. At the other extreme, as we have seen in the case of *REPLY*s, we can do without explicit channels in some cases in the formal model because the definitions of the processes force certain communications to occur unambiguously. The *report* channel in the *UMA* architecture is in fact strictly necessary only when there are multiple applications.

Channels are required in cases where two or more processes communicate to a third using a common alphabet. Without channels, the CSP notation would only allow a communication to occur if all three processes were ready to engage in it. However, in the case for example of the user agent or applications communicating

with the medium, we clearly want these processes to communicate independently of the others.

The counterpart of this in implementation terms is whether we allow input events for example to be *despatched* to a particular owning application along its own channel or to be *broadcast* on a common channel to all applications on the expectation that only the owning application will accept the event. These issues are usually dealt with at the *framework* level, for example the X Intrinsics level. In the case of broadcasting we would formally take the view that events destined for different applications were in themselves different events. The problem in implementation is that each application would have to verify each event, and an application might erroneously accept an event not directed at it.

In practice we want to minimise the number of inter-process communication channels. UNIX pipes, for example, use file descriptors, of which there are only a limited number available. It is even possible to constrain all communication to occur along one channel (or two, if two directions are required). However, in this case the cost of multiplexing and demultiplexing the messages by tagging them with their destination increases. In implementation there is therefore a trade-off to be considered between the number of channels and the complexity of the messages.

These issues arise if the user agent and the medium are bound together, as suggested above. In this case it is possible to combine the *report* and *app* channels, and their replies (so that replies to the reports go along the *app* channel and vice versa). However these events must be clearly distinguishable. In the context of *Presenter* these issues are examined in detail in Pollard's BSc project report [Pollard89].

4.6.11. Synchronisation

We have already mentioned the benefits of a strict alternation of input and output in synchronising access to the surface medium. That is, the medium will not process a command until it has completed the previous command. However, this level of synchronisation may not be sufficient for some applications. For example, an application, upon being informed of a mouse selection, may need to construct a complex object on the medium, like a menu or a set of 'handles' around a selected object, which may require more than one instruction to the medium. Alternatively, an appli-

cation may wish to make a number of queries of the medium, for example about the positions of other objects.

We have shown how user input can be blocked by the application, simply by delaying its confirmation to the user agent. In fact this is an essential capability, since in that period the application is able to perform any number of medium commands, and be sure that the user cannot intervene. However, it is a design issue whether user input should be buffered during this period, and so allow type-ahead, for example so that a user could select a menu item before it actually appeared on the surface.

On the other hand, the application cannot be guaranteed protection by this mechanism from interference by other applications. That is, while an application is preparing to ask the medium to draw handles on an object, another application may move that object. For this reason it may be necessary to provide *acquire* and *release* operations [Hoare85 p. 200] on the medium so that an application which has critical sets of medium commands can perform these without interference.

Some window managers allow applications to grab all input, whether or not it is targetted on that application's objects or not. This capability is necessary to implement modal dialogue boxes, for example. Leaving aside the question of whether strong modality like this is a good thing, it is certainly required by some applications. It is equivalent to providing *acquire* and *release* operations on the input resources. In the implementation of Surface Interaction these would have to be supplied as operations on the user agent.

If the user agent and the medium are bound together in implementation, then these synchronisations become more difficult to manage, simply because the combined process must handle input from two or more sources: the user, and the applications. While we may want user input to be blocked while an application considers its response, we do not necessarily want access to the medium to be blocked at the same time, especially from the application which is dealing with the input.

4.6.12. Echoing

Echoing is problematic only in that many workstations offer hardware echoing of the mouse position, but rely on software to echo keyboard characters. Thus when we speak of delaying the echoing of user input until confirmation is received from the

application, this does not usually apply to mouse cursor echoing. However, Surface Interaction extends the possibilities in two ways:

- In design, the mouse cursor can be seen as an object on the surface like any other. It would be constrained to lie in the foreground, and would be owned by the user agent, but this would not preclude applications from being able to modify its contents or 'warp' its position. However, it is probably the case that most users would not want the mouse cursor to block while the user agent waited for application confirmation, so that a better scheme might be for the cursor to be owned by a special process which communicated with the user agent. The advantage of implementing the cursor within the surface rather than on top of it would be freedom from the standard 16 x 16 bitmap of the video cursor (unless the medium consisted only of 16 x 16 bit objects!).

- Character echoing can be undertaken at a variety of levels. In some cases, for example typing in a password, no echoing should take place. Applications may therefore need to be informed of keypresses each time they occur (for example in an editing application), or only after carriage return has been pressed (for example in a command processor), or perhaps not until the end of an interactive session (if the application is a document processing system which uses a separate editor). The vt100 terminal will automatically echo character presses unless 'raw' mode is set, in which case each character is first sent to the host. This is analogous to the behaviour of the user agent in Surface Interaction. The vt100 thus exhibits Surface Interaction to the extent that some echoing of user actions can occur without involving an application. Similarly this can be changed dynamically. The major difference is that in Surface Interaction this behaviour is determined by the state of the medium object, rather than by some global state in the user agent. That is, one text object may require input to be echoed automatically, another that keypresses are first reported to its owning application.

4.6.13. Pruning State

Textual output generated by applications or operating systems can be voluminous. This is no real problem on glass teletypes, since the text simply scrolls off the top of the screen and is lost. However, if the medium maintains state, then all such output would be saved.

The advantage of this is that all output can then be reviewed by scrolling, and a complete history of the interaction is always saved. The disadvantage is clearly that memory is soon filled with output text.

There therefore needs to be some mechanism whereby output state is pruned. This can either occur automatically (for example, based on age) or electively, by allowing the user an operation to flush tty windows. Alternatively, tty emulation can be strict, so that no (scrolled) text is saved.

In the long term, it might be hoped that tty-style interaction would be entirely superseded by a more direct form of textual communication.

4.6.14. Error Handling

Because the surface medium has state, two sorts of error can arise:

• An application may issue a command incompatible with the current state. For example it may ask for an object to be moved which does not exist. Clearly the medium should protect itself against this possibility, and issue an error message if it occurs, rather than crashing. This is relatively easy to implement, since the *REPLY* set can contain error messages as well as standard replies. Normally the error would be reported only to the offending application, which can then take appropriate steps. However, there is a case to be made for ease of program debugging that the error should also be displayed on the surface. An error may also be introduced if the application returns a reported user input to the user agent with an invalid object. Now it is the user agent which will make the erroneous call on the medium. In this case there has to be some additional way of informing the application of the error. We assume that the user agent itself is free from these errors, and that therefore the user cannot directly make an error on the medium.

• The medium may be put into a state which is self-consistent, but which is incompatible with the semantics of an application. For example the user may delete some important text. The easy way out of this problem is to require that wherever there is the possibility of this error, then the application should save all relevant state so that it can implement an undo operation, or alternatively that it should monitor all user actions by asking for all user input to be reported to it. However, this goes against the principle of Surface Interaction, which

guarantees that there are some user manipulations which the application need not be concerned about. There are therefore grounds for requiring the medium to maintain some history of its states, so that it can perform an undo autonomously. This however has not been implemented.

4.6.15. Logical Events

By *logical* event we mean an event that occurs within the surface itself, rather than in the peripheral input hardware. Thus logical events are distinguished from direct physically generated events. Logical events can originate in two sources:

- As an *indirect* result of user input. For example, as the user moves an object across the screen, it may come into (visual) contact with other objects. The medium could therefore generate collision events on these occasions. The range of logical events is clearly a design decision, and is dependent on the medium model. It is even possible to regard surface errors (the first type above) as sorts of logical event.

- As a result of commands from applications other than that which owns the object. Clearly if an application modifies one of its own objects, then it knows when the command to do this occurs and its effect. However, if another application modifies the object, then it is likely to be useful for the owning application to be informed. This is analogous to the situation when the user agent modifies a medium object, except in this case the user agent informs the application in advance.

Because logical events occur within the surface, they necessitate an extra communication mechanism from the surface to applications. The provision of logical events is therefore an extension of the basic architecture for Surface Interaction. They have not been implemented and their practicalities are unclear.

4.7. Conclusions

This Chapter has presented the fundamental Thesis: a *separation* can be achieved between interface and application if the interface encapsulates a generic presentation model (the medium), and also a user agent which can act independently of applications. Such an interface we call a surface. This architecture provides *direct-*

ness, since both the user and the application can access the surface objects. The architecture also *factors* a significant portion of the task of constructing a direct manipulation interface. This factoring occurs both in the encapsulated presentation mapping from the surface objects to the display, and also during Surface Interaction, when the user can manipulate surface objects without involving the application. On the other hand, the architecture allows the application, if it needs, to achieve fine-grain control over the surface, simply by specifying the input events which it wants reported.

This concludes the first part of this Thesis, which has dealt with the motivation and architecture for Surface Interaction. This part has presupposed no *semantics* for the surface. That is, we have given no *model* for the structure of the surface medium. This, however, is of critical importance to the success of Surface Interaction as a principle for separation, and we address this in the second part of the Thesis.

Chapter 5

Surface Models

This Chapter begins the second and last part of the Thesis. Having established Surface Interaction as a principle and *UMA* as an architecture for direct, separated interaction, in this part we specify two new models for the surface.

As a preparation for the new surface models described in Chapters 6 and 8, this Chapter examines current models for the surface. The first Section looks at window management and its models. The second Section examines imaging and modelling in graphics. The third Section looks at text models. The last Section covers models for documents, that is, models which incorporate both text and graphics. First of all we clarify some terms used in this Chapter.

5.0.1. Procedural and Declarative Models

In graphics terminology, a *model* is the *structure* of graphical or textual elements. In the abstract, a model can thus be represented by some connectivity relation. In practice, models can be expressed in different ways which affect the *separability* of the model.

We distinguish between *procedural* and *declarative* models of both text and graphics. A procedural model is a *program* which needs to be interpreted or compiled to generate output. Thus a description of a page in PostScript [Adobe87] or a piece of text marked up with LaTex [Lamport86] commands are both models in this sense for the final version of the page. In a procedural model the model structure is represented by the *syntactic call structure* of the program. A procedural model is thus bound early to its primary representation.

On the other hand, a declarative model presupposes some independently maintained *state* which is accessed by a set of operations. The model here is thus an Object, in the sense defined in Chapter 1. It is declarative in the following senses:

- structural and other relational constraints may be declared over components of the model, which persist and are identifiable.

- the effect of the operations is largely independent of their order of invocation.

- the implementation of the constraints is maintained by the Object, and thus hidden from the external operations. Presentation can also be seen as a constraint between the model and a display medium.

Window managers generally have a declarative model. That is, they provide operations to create and structure windows with respect to each other, while the implementation of display management is hidden from the users.

A declarative model is more separable, since both its representation and its implementation may be bound late. It is thus suited to interactive use, since the components may be created and deleted dynamically, and modified randomly. A procedural model, on the other hand, can only be used interactively by allowing the user to edit the program, and then re-executing this. This cannot be *direct*, in the sense defined in Chapter 1, since the program representation will necessarily be different from the output that it generates.

5.0.2. Marks and Media

We instantiate the notion of *binding* in a graphics context. Thus we distinguish between the output *medium* (for example a piece of paper, or a display screen, or a window), and the *marks* that may be made on that medium using primitive elements or drawing tools. We wish to think of marks as unstructured and permanent (except by overwriting), while media are structured and manipulable. Marks are thus analogous to graphics or text primitives, while media can form the components of an Object state. However, the distinction between marks and media is simply one of binding. Marks are bound to the medium, whereas media are unbound.

This results in a hierarchy of marks and media, such that media may be marks in a higher binding. Thus basic surface primitives can be seen as media with very simple marks (a solid fill, for example). These media may be structured into the rep-

resentation of a textual character, and thus be seen as marks making up the shape of the character, which is a medium at a higher level. The character media themselves may be structured as marks on a page or window of text. These page media may be structured as marks in a document or interactive interface, and so on. This notion of marks and media thus generalises the conventional graphics distinction between imaging and modelling.

The distinction between media and marks is important to this Thesis, since media, because they are late bound, are more useful than marks in interactive interfaces. It is easier to write a directly manipulable application, for example, using a Toolkit that provides many media presented as buttons, scroll bars and menus, than it is to write directly to a window simply using primitive marks like lines and text. The first section in this Chapter criticises the window manager model on the grounds that it provides only coarse media. The models presented in Chapters 6 and 8 of this Thesis on the other hand attempt to maximise the granularity and flexibility of the media, such that marks need only be used for static imaging.

Since window management provides the basic graphical and textual *model* for the majority of applications written to bitmapped workstations, we examine the window management model first and in some detail.

5.1. Window Management

Window management is a model for Surface Interaction. The standard window environment provides persistent objects (windows and icons), and operations on them (open, close, etc.), and a control abstraction whereby window management operations can be interleaved with application operations. We examine the suitability of the standard window model for Surface Interaction.

Window management is truly a bitmapped, rather than a vector, phenomenon. A few window systems have been built on vector hardware [Rosenthal81, Littlefield84], but these postdate the original conception. Window management has similarly been largely ignored by the mainstream graphics community. The proposed CORE standard, for example, had a 'synthetic camera' analogy [Rosenthal83 p.39] which allowed only a single view of the object being modelled, as opposed to the multiple views inherent in window management. In GKS, even though multiple world/device coordinate transformations are permitted, they are not *framed*: seg-

ments may be interleaved arbitrarily [Rosenthal83 p.38]. Only recently, and probably as a result of the adoption of raster hardware, has belated provision been made for window management within mainstream graphics [Voorhies88]. This, however, is at the hardware level. Other proposals opt for running GKS or CGI *within* a window [PCTE88, Hopgood86a] (there are also similar moves to provide PHIGS within X). The ANSI X3H3 group is examining a model for window management that is more closely bound to the graphics standards [Butler85b].

The seed-bed for the concept of windows and its early implementations was the Alto bitmapped workstation, built at Xerox PARC in 1973 [Thacker82]. This formed the vehicle for a number of seminal systems, notably William Newman's Officetalk, Teitelman's DLisp [Sproull79] and the interface Dan Ingalls contributed to Kay's Smalltalk-76 [Ingalls81] (Ingalls is generally credited with the invention of windows [Teitelman86 p.35]). These in turn formed the inspiration for the Star [Smith82a, Smith82b] and Lisa [Williams83] systems, and, through Warren Teitelman, the window systems for Interlisp-D, Tajo, and Cedar [Teitelman86].

All these systems, however, can be characterised as monolithic, having a single language and possibly a single address space, closely tied to the hardware, and designed for exploratory work or limited task domains. For these reasons it is often difficult to abstract the window system itself from the overall semantics of these environments. Both Star and Lisa, for example, have a large number of built-in objects like folders, files, in and out baskets, and calculators whose functionality and appearance are integral to the environment and its desktop or office metaphor. Later efforts in window management attempt to decouple the presentation and management of windows from the objects they are used to represent. Typically, these later systems have been developed to interface to an established operating system like UNIX: the Blit [Pike84], the Whitechapel MG-1's window manager [Newman85, Sweetman86], the Perq PNX's window manager [Perq84], SunView [Sun86], and Andrew's window manager [Morris86].

Most recently, window management systems have addressed three issues. Firstly, attempts have been made to abstract an underlying screen management capability from the particular windowing interface. The same 'base', 'platform', or 'substrate' window manager may thus present itself in a variety of user interface guises. X [Scheifler86 p.81], NeWS, and CSI [Williams87] are all conceived as base window systems in this sense. Secondly, distribution has been given priority. X and NeWS are designed to be network transparent - an application may display itself

through a window maintained on another machine. Lastly, the big stakes are now for standardisation. The authority of the ANSI X3H3 windowing proposals [Butler85a, Butler85b] has to a large extent been undermined by the success of X in the public domain. The latest version 11 of X is a deliberate attempt to preempt the lumbering machinery of standardisation [Laursen87].

However, it is not the intention here to present a comprehensive survey of window systems - this has been adequately undertaken by Myers [Myers88c]. Rather we wish to characterise the features and limitations of window systems as generally implemented.

5.1.1. The Model

The first step in moving away from a simple glass teletype presentation is the erection of fixed boundaries in the display space within which different streams of text can be scrolled or edited. The UNIX vi editor, for example, has a text area and separate command line implemented in this way. The fundamental model for true window management, however, sees these display partitions as discrete areas geometrically independent of each other in a 2.5D space. These areas act as 'windows' onto separate spaces under the control of different application tasks. Underlying these windows there is conventionally a 'background', usually shaded.

Most systems allow any number of windows to be open (subject to upper memory bounds), but some limit this. Star, for example, allows only up to 6 windows at once [Smith82a p.523], while Perq PNX allows 31 [Perq84] (windows in PNX are implemented as open UNIX files, upon which there is a limit imposed by the operating system).

It is interesting to note that, in spite of the vaunted 'naturalness' of direct manipulation, there is no simple physical analogue for this model of potentially overlapping windows onto larger spaces (perhaps it could be set up using mirrors).

Geometry

In the majority of systems these screen areas are rectangular. NeWS [NeWS87b p.46], however, through its PostScript imaging model, can also maintain non-rectangular areas. In Star [Lipkie82 p.119], and all other window managers that the author is aware of, the screen areas maintained are opaque. That is, it is not pos-

sible to see through them to areas notionally underneath - their RasterOp mode is a simple overwriting of the screen, rather than some combination of source window with the existing screen contents. Although documentation for both X and NeWS uses the term 'transparent', in practice their 'transparent' window or area is either a clipping rectangle onto an existing opaque window, by means of which drawing on that window is limited to the area of the rectangle, or it is an invisible 'input-only' window [Scheifler86 p.104, NeWS87a p.14, NeWS87b p.46]. NeWS in addition allows the creation of transient images (like rubber boxes for echoing movement) on a transparent *overlay* canvas [NeWS87b p.48]. As the name suggests, however, the overlay canvas is restricted to the foreground of the screen, since it is intended to be implemented by a hardware overlay plane or by the easily-reversible XOR RasterOp mode. A major contribution of the models presented in this Thesis is an integration and implementation of transparency as an attribute of any screen object, and using any image combination mode.

Restrictions may also be imposed on the position and size of windows. SunView windows, for example, cannot be moved so that any part of them is offscreen. On the other hand, Perq PNX [Perq84], Macintosh, and X windows can be moved offscreen. Most window managers allow windows to be changed in size, but Blit windows, for example, cannot change size [Hopgood86b p.134].

A major classifying division in window managers is between those that allow overlapping windows (i.e. that place no relative constraints on window size and position), and those that do not. The majority of systems that do not allow overlapping in addition *tile* their windows so that borders are contiguous. Star is the exception here: it does not allow overlapping windows [Smith82a p.523], but does not either appear to tile its windows. Bly and Rosenburg compare tiled and overlapping systems [Bly86] from the human factors point of view. Tiled systems optimise screen use, but in order to do this are relatively dictatorial about window placement. The user or the application may only be able to give 'hints' in order to specify the size or position of a window [Gosling86 p.117, Morris86 p.197]. The tiling algorithm may be more or less complex [Cohen86].

Early versions of Andrew, for example, tiled the whole screen area and realigned all windows upon a single change to one of them. This was found to be slow and confusing to the user [Morris86], and was changed so that windows existed in two columns with some background space visible at the bottom of each. Window geometry changes were therefore more localised. This scheme emulated

the fixed-format tiling algorithm of the other major tiling window manager, Cedar Viewers [Teitelman86 p.41, Teitelman84, Swinehart85] [Beach85 p.5].

It may seem strange to build tiling into the fundamental layer of a window manager, when an overlapping model is more general and may subsequently be constrained to tile, if needed. It is considerably easier, however, to implement window update and hit detection if tiling is assumed from the start. Offscreen storage for obscured windows is also not necessary. A recent system [Tanner86 p.242] tiles windows for just these reasons.

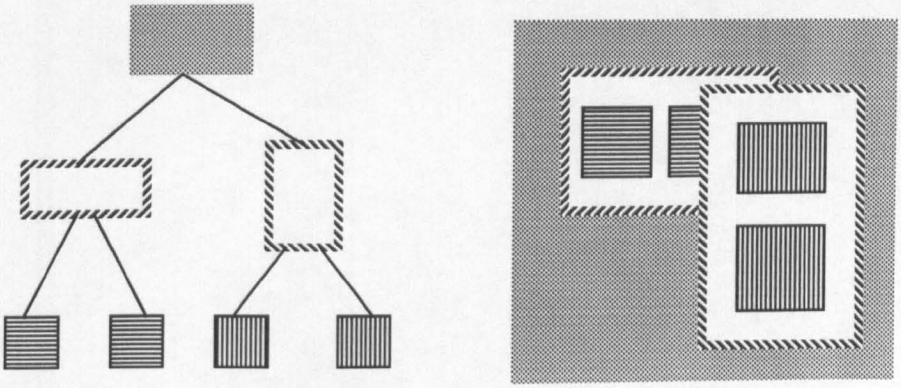
Structure

The overlapping model immediately assumes a linear ordering of the windows to determine which obscures which. Generally, a new window is created at the 'front' of the screen. Systems vary as to methods of manipulating this linear structure. In SunView, windows can arbitrarily be 'exposed' or 'hidden' (pushed to the back) - the display order of the windows can therefore be changed. On the Perq, however, windows can be cycled, but not changed in order.

A number of recent window systems maintain a tree structure of windows. In all cases this is a *geometric* hierarchy: the size and position of a child depends on the size and position of its parent. Tree-structured systems include Whitechapel MG-1 [Newman85 p.421], X [Scheifler86 p.89], and NeWS [NeWS87b p.46] In all these systems all nodes in the hierarchy are notionally of the same type (panels in MG-1; windows in X; 'canvasses' in NeWS). Some earlier systems have a kind of geometric hierarchy, but this is maintained through a hierarchy of types (viewers and subviewers in Cedar [Swinehart84]; canvasses, windows, frames, subwindows, panels in SunView [Sun86]). Of course, even the simplest window model which allows some specification of window contents can be thought of as at least a two-level geometric hierarchy.

A characteristic of most of these hierarchical models is that all nodes in the tree contain potentially displayable images. The display ordering must therefore be some deterministic traversal of the tree structure, usually preorder [Scheifler86 p.89], so that children obscure parents, and all descendents of a node obscure all

descendents of a prior sibling node. Sub-hierarchies can therefore never visually interleave:



logical view *user view*

All these systems, however, fail to exhibit geometric generality in three main ways:

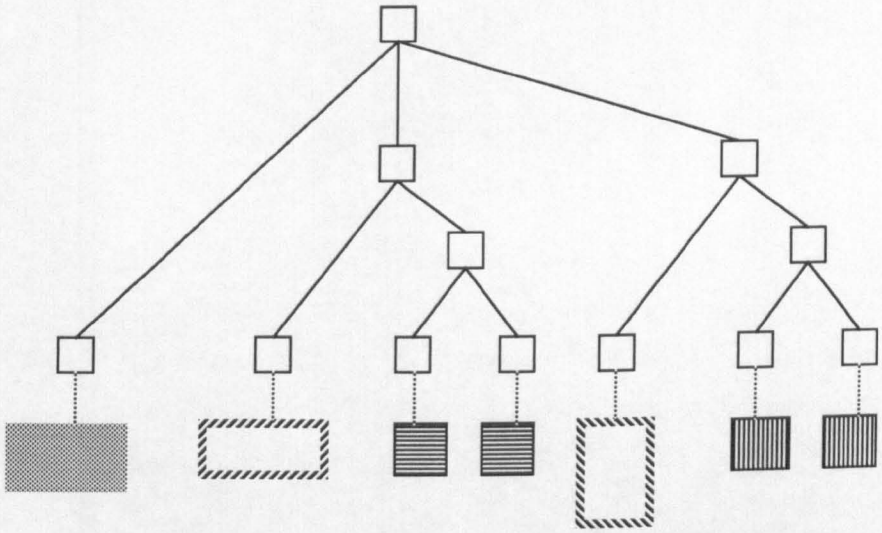
- The presence of displayable images at interior nodes means that such images cannot be manipulated *independently* of their descendents: if a parent image (representing, say, a background or border) is moved, then all its children necessarily move with it. While it may often be the case that this is what is required, it is nevertheless a restriction on the geometric manipulability of the screen objects.

- All hierarchical window managers that the author is aware of clip the display of child areas to the extent of the parent (this can be turned off in SunView, but it simply corrupts the screen). Such a strict containment scheme is obviously easier to implement, since updates are restricted to the 'refresh tree' [Rosenthal83 p.39], and hit detection can be performed recursively [Szekely88a p.40, Strubbe83 p.1035], starting with the extent of an ancestor (usually a window) and testing each level of descendants in turn against the position of the cursor. This graphical containment policy, however, is again a restriction on manipulability.

- In almost all hierarchical window systems the 'windows' form a distinguished level at the top of the hierarchy (i.e. immediately beneath the root), while lower nodes ('subwindows', 'panes', 'viewports' are common labels)

are subject to the above two restrictions. There are thus in practice at least two different display types, and it is impossible to create full windows as children of other windows, or to group already existing windows [Williams86 p.28]. (Although the model defined for PCTE+ does group windows [PCTE88 p.27], the window group is a separate type from the window, and has its own set of operations).

The models presented in this Thesis avoids all these restrictions. Regions in *Presenter* have displayable content only at the leaves of the hierarchy, so that the above user view would be generated by a logical tree such as the following:



Restricting displayable images to the leaves of the geometric hierarchy in this way is paradoxically the more general scheme, since it makes the syntactic grouping structure of the visible objects (represented by the hierarchy) orthogonal to their geometric locations. This is reinforced in *Presenter* by allowing a child region to be at any position or size with respect to its parent. This image-at-leaf model cannot be emulated by current hierarchical window systems (for instance simply by not displaying images at interior nodes), since all impose the restriction that a window that is hidden ('unmapped', in the terminology of X and NeWS) also hides all its descendents. Lastly, regions in *Presenter* are entirely independent of their position in the hierarchy: they can be cut, pasted and grouped aribitrarily. One set of operations suffices for all regions (leaf or interior).

5.1.2. Features: Icons and Menus

The notion of an icon has evolved subtly. At its conception in Smalltalk and Star, an icon was simply a compact representation for a closed window. While some later window systems do not provide icons (like Perq PNX's), most have taken up the metaphor, sometimes extending it by providing icons that are miniature copies of the associated window, or that give some visual signal when events occur within it (such as a raised flag when mail arrives). The important characteristic of icons in this conventional sense is that they are a distinguished type within the window environment, and that the relationship between windows and icons has a built-in semantics. This is reinforced by the sometimes baroque visual mechanisms that window systems indulge in to present the transition from icon to window: collapsing, exploding, or whirling rubberware, for example. The fixed relationship between windows and icons is implicit in the popular designation *WIMP* for the class of direct manipulation interfaces.

Recently, however, icons have been promoted to a more fundamental role in interface systems. In such 'iconic' interfaces [Glinert87, Gittins86] the term 'icon' is used to denote a screen object of more general applicability. Clarisse and Chang see an icon as 'a *predefined* flat pictorial symbol representing an "object" - physical or abstract' [Clarisse86 p.153]. That is, an icon in this sense is visually atomic - it has no constituent structure - and serves as a lexeme in the interaction. Korfhage and Korfhage [Korfhage86 p.210] emphasise the *symbolic* nature of such icons. In their definition, an icon is associated with a *concept* via some primarily pictorial image. Designing an interface is thus a matter of designing an iconography. This view of icons as visual tokens is thus closely associated with recent aspirations towards iconic or visual languages [Korfhage86, Haeberli88, Powell83, Chang86, Selker87, McDonald82, Myers88a, Myers86b, Cook88, Waite88, Harel88].

It is clear from this wealth of literature that an icon can be a more powerful entity than simply a 'gone for lunch' sign for a window. It may be that the term 'icon' is better reserved for the traditional use, and perhaps 'visual object' or 'atomic window' is preferable for this more general case. Certainly some of the bulk of recent literature on icons would disappear if this translation were made: Korfage's speculation about icon hierarchies, subicons, zoomable icons, and document icons [Korfhage86 p.224], for example, seems perfectly isomorphic to facilities offered by recent object/window systems.

Whatever the terminology, however, it does seem inappropriate, in view of the obvious potential of icons to form a wide class of visual interfaces, to bind the old icon/window semantics into the base level screen manager, as is the case in X [Scheifler86 p.102] and PCTE [PCTE88 p.27]. If anything, this shows the stranglehold that the window management model has on interface design.

Conventional menu facilities present a similar case. Menus are often provided at a low level in window systems. In Perq PNX, SunView, and PCTE, for example, they are available at the same level as the window itself. Even though, in display terms, their requirements are identical to windows (opaque, rectangular areas of screen), they are given a separate semantics. The appearance and behaviour of icons and menus are thus bound in to the environment.

The clarifying assumption of the model offered in this Thesis, on the other hand, is that at the base visual level there should be *no* semantics other than those associated with fundamental geometric or textual manipulation. At root the characteristic feature of icons adopted by the visual language community is their visual atomicity. This is amply carried by *Presenter*'s notion of a region and its image, and is in contrast to (and perhaps in reaction to) the conventional idea of 'window'. Similarly, at this level, a menu can simply be represented as a set of selectable text regions, leaving the semantics of selection (command invocation or option choice) to be interpreted in deep interaction.

5.1.3. The Window Interface

A clear distinction should thus be made between the base window system, which presents the screen objects and manages their manipulation, and the window *interface*, which determines the appearance of the windows and interacts with the user on the basis of some semantics (such as the association between icons and windows mentioned above, or the association between windows and processes).

It is obvious that the underlying functionality represented by the particular window abstract data type could be instantiated in a variety of styles, and using a variety of input mechanisms. Window managers vary in lexical style, for example the width of borders, presence and style of title bars, scroll bars etc.. Similarly, window operations may be invoked by direct manipulation (pushing a close button, as in the Macintosh) or via a menu (as in SunView). It is also obvious that there must exist some root process by which the whole system is bootstrapped, which allows subse-

quent application processes to be invoked and which relays system messages to the user.

It is convenient to combine this stylistic management and supra-process management in a single, (at least logically) separable window interface process. This distinction is made, for example, by Coutaz in her Box (base) and Mediator (interface) components [Coutaz85, Coutaz86], by Whitechapel's 'panel manager' (base) and window manager (interface) [Sweetman86 p.77], and at least implicitly by X, NeWS and CSI, all of which see themselves as relatively low level *protocols* upon which an interface should be constructed (although X and NeWS also supply a default interface).

The style of the window interface is often determined and made consistent via a metaphor [Carrol85]. The window itself is, of course, a powerful metaphor. It conveys the notion of viewing a wider space through a frame. More prosaic interfaces, like SunView's, at least have this unifying principle. Icons [Scheifler86 p.102], too, possibly convey some notion of symbolic designation of a large (conceptual or physical) space in a small token. More specific metaphors, like the desktop, rooms [Henderson86, Card87] (which emphasises the logical connectivity of windows), and cards in Hypercard [Hypercard89], give conceptual leverage, but possibly restrict the design space. That is, there may be a tension, as Smith points out [Smith87], between the literal and symbolic ('magical') meanings of the metaphor. The physical-world metaphor of the Alternate Reality Kit [Smith86], for example, is an attempt to maximise the literal, as opposed to the symbolic, interpretation of the metaphor so that the novice user can predict the system's functionality rather than having to be taught it. Complex metaphors (like ARK) may also require a more powerful display environment.

5.1.4. Window System Architecture

There are typically three locations for a window manager: in the client (using calls to library routines), in the kernel (using system calls), and in a server (using inter-process messaging) [Teitelman86 p.45, Gosling86 p.102]. These correspond to the internal, external, and concurrent control models outlined in section 2.1.2. Just as in those models, the server architecture seems preferable. Servers provide distribution for free [Hopgood86b p.133, Lantz84 p.24] and enable factoring and encapsulation of screen handling. Examples of server-based window systems are

NeWS [NeWS87a], X [Scheifler86 p.84], the Blit [Pike84], Andrew [Morris86], VGTS [Lantz84 p.27] and its successor TheWA [Lantz87b p.91]. Some window systems are hybrid. In the Whitechapel [Sweetman86 p.77], for example, the window manager is a server, while the panel manager resides in the kernel.

One major decision for server-based window systems is the problem of what Lantz calls *retention* [Lantz87b p.91], that is, the degree to which the server retains (and can therefore autonomously redisplay) application images. If the server retains the image, then image construction and window management functions can be decoupled. A simplistic solution is for server and client to share image memory [Hopgood86b p.133]. However, this compromises distribution, and may only be possible on single address space, or kernel-based, systems. A related issue is the nature of image encoding - whether as a bitmap, a segmented or structured display list (as in GKS or PHIGS [Salmon87 p.305, Langridge87, Langridge88]), or a display program (as in PostScript). A useful capability, at least in overlapping window systems, is for the server to be able to retain at least bitmap images. The original X system, for example, did not guarantee this [Scheifler86 p.105]. This is also the scheme adopted by the Whitechapel MG-1 manager [Sweetman86 p.78, Newman85 p.421], Rutherford's CSI [Williams87 p.25] and (in its bitmap mode) *Presenter*.

If the server retains nothing of the image, then the application has to be aware of uncovering and resizing events, and must redraw its own window [Williams86 p.27, Myers86a p.67]. This redrawing is likely to be highly redundant (portions of the window will be redrawn unnecessarily). Optimal redrawing, on the other hand, involves access to detailed information on window positions and movement, and is an unacceptable burden on the application programmer. Redrawing by the application saves on server memory space, but at the expense of increased image traffic between application and server. X, however, claims only a small performance loss in doing this [Scheifler86 p.80].

Applications may even be allowed direct access to the screen, as in SunView and X [Scheifler86 p.96]. This may be subject to heavy-handed synchronisation by blocking output when window management functions (like moving a window or popping up a menu) are invoked [Gosling86 p.101], but in other cases may simply rely on the cooperation of the application to overwrite only those areas of the screen where it is visible, or on the toleration of the user if things go wrong. In safety-critical displays this is simply not acceptable.

Issues of retention and control again place constraints on interface design using standard window systems. Further constraints on design may come from certain common implementation limitations. Cursors, for example, are invariably of fixed size. Pop-up menus are often implemented so that not only need the input queue be blocked during drawing (see section 2.2.2), but also so that all other screen update is blocked while they are on screen (because the implementation saves what they obscure) [Salmon87 p.343, Scheifler86 p.98]. In this case the application cannot present menus which the user can optionally fix on screen, as with Open Look's pushpin [Hoeber88 p.73].

Windows as Graphic Environments

A limitation of standard window systems is the use of windows to mark the boundaries of graphic environments. Windows are 'virtual terminals' [Strubbe83 p.1035, Coutaz86 p.337], 'logical display surfaces' [Cahn83 p.169], 'logical screens' [Kamran83 p.58], 'tiles' for information [Angell87 p.134], or 'viewports' for 'virtual graphics terminals' [Lantz84 p.32]. Often, in addition, a window has a one-to-one association with an application. This is the case in Strubbe's and Cahn's systems. The use of virtual terminals to link input and output is strongly criticised by Lantz [Lantz87b p.90]. It is notable that the CGI also uses virtual terminals (which it, confusingly, calls 'virtual devices' [ISO86a]) as its basic output medium.

This first-level boundary between extra- and intra-window content is the cause of many of the fundamental discontinuities in window system display and interaction. Many window systems simply assume that window content is purely in the domain of the application. There is no guaranteed consistency, therefore, between the interaction style within the window and with the window manager. While the application might deal with the window manager in terms of relatively high-level constructs like window identifiers, within its own window space it is likely to have only low level operations such as RasterOp or minimal line and text drawing. Even in window systems which provide a richer graphics environment such as structured display files [Lantz84 p.30, Cahn83 p.168], or display trees [Strubbe83 p.1035, Coutaz86 p.337], or even display languages [NeWS87a], these environments do not subsume the windows, but are subsumed by the windows. In all these systems, therefore, the window is always at the root of the display hierarchy as a distinguished type consisting of an opaque drawing surface. In the

terminology of the introduction, windows are the only media, while all other constructs are marks.

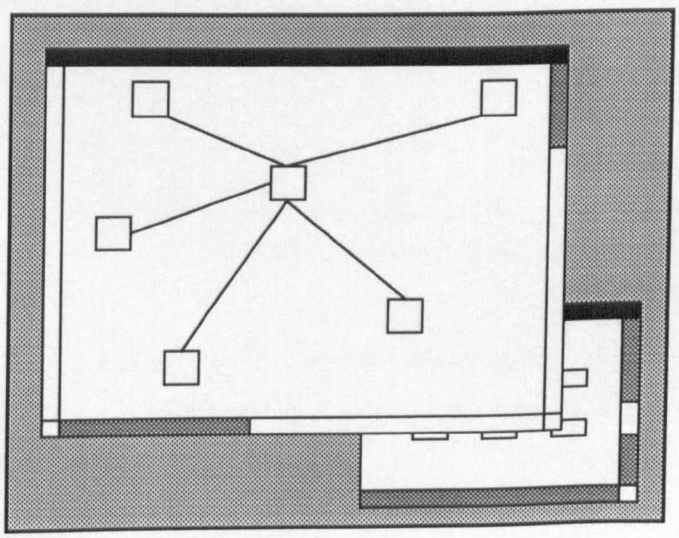
Windows as Interface Media

Additional criticisms may be levelled against the window as an interface medium.

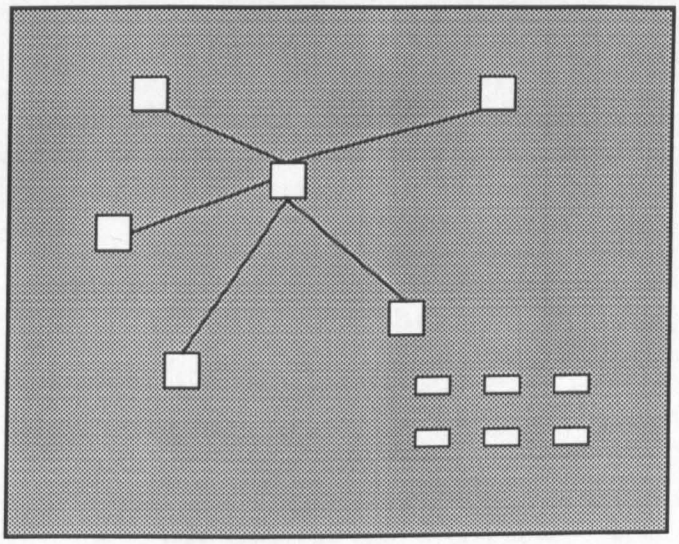
- As Henderson and Card [Henderson86 p.214] point out, overlapping windows can lead to a cluttered screen. The clutter is visually confusing and can cause 'thrashing' [Card87 p.59].

- In window interfaces which indulge in relatively wide window borders filled with functionality like scroll bars and buttons (as on the Mac) the cluttering is exacerbated, since this functionality is duplicated for each window.

- Not all objects are rectangular or opaque. In layouts of networked objects, for example, there may be opaque background around and between the objects such that some underlying objects may be obscured unnecessarily. Compare, for example the following two (artificial) screens, the first displaying a network on a standard window manager:



and the second the same network on a Presenter-generated screen:



• Unless there are special geometric constraints, it is possible to reduce the size of windows or move them out of sight (either off screen or under other windows) so that important control surfaces may become inaccessible.

A contention of this Thesis is that window managers have developed ad hoc, without a sound formal basis. They have been driven more by a virtual terminal requirement and the desktop metaphor than by a theory of screen objects.

5.2. Graphics

It is only worthwhile to make drawings on the computer if you get something more out of the drawing than just a drawing.
[Sutherland63 p.344]

In order to arrive at a more sound and general model for the surface than that provided by the windowing metaphor, we explore mainstream graphics in more detail. We examine graphics under two major categories: imaging and modelling.

5.2.1. Imaging

By image we mean an *atomic* picture. Although some structure may be used in the generation of an image, the image itself retains nothing of this structure. In Joloboff's definition [Joloboff86 p.108], an image is a 'final form representation'. The output of MacPaint [Chen88 p.17], for example, is an image (a bitmap, in this case).

Images thus consist of marks on some medium, for instance a screen or a piece of paper. Within the resolution of the display medium, an image may be arbitrarily complex.

Images may be specified completely using a simple mapping from some domain of points to a range of colours. The size of the domain will determine the resolution of the image, while the size of the range will determine the colour characteristics of the image. A monochrome image can be modelled either as a *partial* mapping onto a range of one colour, or as a total mapping onto a range of two colours (such as black and white). In a fully polychrome image the colours can also be assumed to have intensity, such that a colour in one intensity is different from the same colour in another intensity. Formally, a convenient representation is a mapping from an infinite set of points in the real plane to a set of colours in this sense (see [Mallgren83 p.8], or [Nelson85 p.237]). This is the definition used in the formal specification in chapter 6:

$$POINT == \mathbb{R} \times \mathbb{R}$$
$$IMAGE == POINT \rightarrow COLOUR$$

An image in this definition is thus two-dimensional. Although it is easy to conceive of a three-dimensional image specification (three coordinates instead of two), it is clear that in practice this would invariably be projected into two dimensions (unless we had holographic display media). Even in two dimensions, the number of possible images under this definition is infinite. If we introduce a pixel resolution (i.e. restrict *POINT* to some finite set) the information space is still enormous. The number of possible *IMAGE*s will in general be

$$\#POINT^{\#COLOUR}$$

that is, the number of pixels raised to the power of the number of colours (although clearly not every *IMAGE* will be visually distinguishable or meaningful).

The *point* is thus the most fundamental image primitive (the point is Juno's only graphical data type, for example [Nelson85]). Composing images explicitly from points, however, would be extremely tedious. That is, while in theory a

setpixelcolour (pixel, colour)

operation is capable of generating all images, in practice we need some higher image primitives in order to reduce the number of discrete operations we need to make.

Image Primitives

Ideally we would like a set of predefined higher primitives, such as lines with thickness and style or characters in particular fonts, that would still be capable of generating all possible images. Unfortunately this is impossible, since two images may differ from one another just in the colour of one point. We must accept therefore that any set of imaging primitives is an arbitrary but hopefully useful selection from the infinite set of possible marks.

There are fundamentally two approaches to making marks on the display medium using image primitives:

- The inkblock approach. The primitives are composed into an image by successive application onto the imaging medium, like an inkblock repeatedly stamped at different positions on a piece of paper. To be visible, therefore, such marks must have some thickness. The inkblock approach is used by vector-oriented graphics systems like GKS [ISO85], PHIGS [ISO87b], PIC [Kernighan81], IDEAL [Wyk82], and Juno [Nelson85].

- The stencil approach. Here there are two stages. Firstly, a *path* is created by repeated application of the primitives. Secondly, a colour or pattern is projected through the path onto the display medium. By analogy, the path cuts a stencil in some masking material, which is then inked through onto the paper. In this case, the path itself has no thickness. The stencil approach is used in PostScript's path/paint imaging model [Gosling86 p.50], Warnock's stencil/source model [Warnock82 p.314], Gargoyle's outlines and fills [Pier88 p.227], and in 'planar maps' [Baudelaire89].

The stencil approach is more general, in that it allows arbitrary *sets* of contiguous points to be coloured. Thus even a character in text can be imaged by specifying the path of its outline and a fill colour (see Pratt's work on conic splines [Pratt85]).

The two approaches converge, however, in two areas:

- It is almost as tedious to specify a character or a symbol by giving a mathematical description of its outline, as it is to specify the points of which it is

composed. Stencil-based systems therefore usually include a set of primitive paths such as character fonts or lines and circles with thickness.

• Many images require areas filled with colour. Inkblock-based systems therefore usually include an *area fill* operation.

There is no standard set of image primitives. Typically this consists of lines, curves and compositions of these into polygonal shapes. But GKS, for example, does not have spline curves [Pratt85]. Primitive shapes may notionally be transparent, as in PIC [Kernighan81] or opaque, as in PostScript [Adobe87], or allow a choice between these modes, as in IDEAL [Wyk82 p.173]. Some systems do not allow area fills, except by shading using lines or other primitives. This is true of PIC and IDEAL. GKS cannot fill curves or holed areas.

Later imaging languages like PostScript and systems like Gargoyle (which is based on Interpress [Bhushan86]) provide a richer set of style attributes than mainstream graphics. While GKS has types like 'dotted' and 'dashed', and colours and widths, PostScript and Gargoyle can in addition specify line joins (mitred, rounded, or bevelled) and line caps (squared, rounded, or butted). Here, for example, are a selection of triangles with different line joins drawn in PostScript:

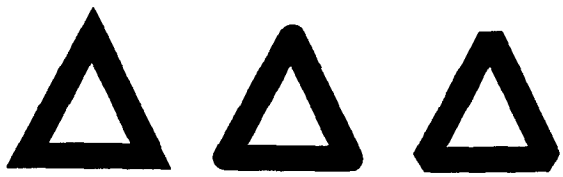


Image Transformations

In general, given the above formal definition of an image, there are three types of transformation that can be applied:

• Images may be transformed by restricting their domain of points to some subset of points (e.g. *MASK*: $\mathbb{P}$ *POINT*):

$$MASK \lhd IMAGE$$

It is conceivable that this restriction apply to arbitrary points, as in bitmap masking, but more commonly it is applied to contiguous points, as in clipping to a path.

• Images may be transformed by modifying their domain of points. That is, the colour at a particular point may be moved to another point. This can be achieved by composing the image with a geometric transformation function (e.g. *GEOMTRANS*: *POINT* → *POINT*):

GEOMTRANS ⨟ *IMAGE*

The class of affine geometric transformations such as translation and scaling are special cases of the general geometric function (see Martin [Martin82], or Bier [Bier86 p.235]), in which the function is a bijection (that is, it is invertible). There are also cases where other mappings are useful, for example in raster conversion, or 3D to 2D projections [Mallgren83 p.17].

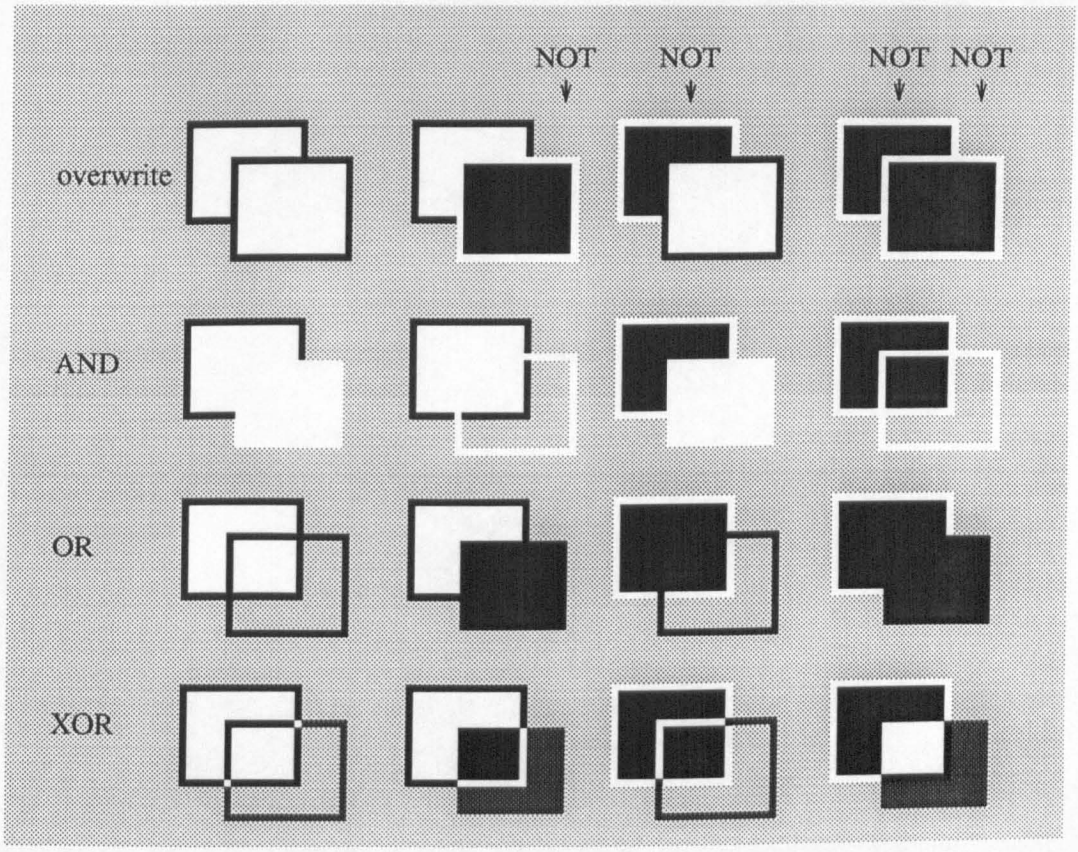
• Images may be transformed by modifying their range of colours. That is, the colours of arbitrary points may be changed arbitrarily. This can be achieved by composing the image (in the opposite order to above) with a colour transformation function (e.g. *COLTRANS*: *COLOUR* → *COLOUR*):

IMAGE ⨟ *COLTRANS*

This function can be used simply like a colour lookup table, or it could be used to threshold or filter images. More general image processing requires a function which takes account of the position of a colour, and so can perform, for example, local averaging of colours. Even in interactive graphics, colour functions like this might be required for patterning or dithering colour images on monochrome screens, or antialiasing lines during raster conversion.

Mallgren [Mallgren83 p.12] gives formal definitions for most of these transformations.

Images may also be *combined*. This involves specifying alignment (which points should be combined with which), and a colour combination. Basic RasterOp or BitBlt has sixteen different combination modes, which are a permutation of the four modes AND, OR, XOR, and overwrite, between source and destination bitmaps and their negations. The effect of these modes is shown in the following illustration. Black is equated with *true*, and white with *false*, such that black AND white, for example, is white. In each mode, the source is illustrated as overlaid on the destination:



Some combinations are equivalent, for example

(NOT source) XOR destination

and

source XOR (NOT destination)

(the overlapping portions of the second and third images on the bottom row are identical). Also, some combinations may not be particularly useful in practice. RasterOp, however, is strictly applicable only to single bit depth images. Mallgren [Mallgren83 p.25] addresses the problems of combining gray-scale and colour images, and [Porter84], [Acquah82], and [Paeth86] make some practical suggestions for combining and compositing more deeply coloured images.

Presenter, in keeping with its declarative rather than procedural nature, associates the combination mode with a region, rather than requiring the specification of source and destination regions and mode in a procedure call. Thus the destination

region changes dynamically as the (source) region is moved around the screen. One region can combine with a number of other regions in an entirely intuitive way.

Presenter also allows transparent regions to be overlaid arbitrarily. One feature of the RasterOp combination modes that has important consequences for the implementation of this is that they are not associative. For example, it is the case that

$$(1 \text{ XOR } 0) \text{ AND } 0 \neq 1 \text{ XOR } (0 \text{ AND } 0)$$

Therefore, layered transparent regions *must* be redrawn in display order, which closes out some short cuts possible if only, say, XOR mode were permitted. A useful benefit of Presenter's strict redrawing is that *any* colour combination mode could be substituted for the basic raster operations.

Within mainstream graphics, some effort has been made to accommodate the newer raster technology. The GKS 'cellular array' construct makes a token acknowledgment of bitmaps. Acquah et al [Acquah82] make an early attempt to integrate raster operations with a vector-based graphics language. This has been taken up by the proposals for the Computer Graphics Interface (CGI) [ISO86a, Salmon87 p.357] which includes a variant of RasterOp in a protocol which follows closely the form of a GKS-derived segmented display file.

The raster community, however, have not reciprocated this approach. Baudelaire and Stone [Baudelaire80], for example, attempt to provide an abstract geometric representation for raster images without reference to the emerging vector standards. Nevertheless, they still preserve the distinction between a high-level geometric representation and an encoded display file. Warnock and Wyatt's important paper [Warnock82] goes further in removing the intermediate display file altogether and relying completely on a readable, interpretable representation for object geometry. Warnock and Wyatt's work leads directly to the language PostScript [Adobe87], which is becoming a *de facto* standard for the description of raster-mappable images.

5.2.2. Modelling

Images may be generated by displaying any arbitrary segment of memory (i.e. any mapping of pixels to colours) on the imaging medium. Such a segment may for example contain a digitised photograph. Commonly, however, we wish to construct

the image out of parts either drawn directly by ourselves, or created algorithmically by the computer. We may then need to selectively modify these parts. To do this we need to create a *model* of the image.

In modelling we are concerned with the *construction* of complex images from primitive images. A model is thus a *composite* image whose structure is preserved. In mainstream graphics the notion of a model carries three connotations:

- A model is *structured.* That is, it is constructed of discrete parts which have a topology, such as a hierarchy. The model thus might correspond to some composite real world object like a plane or a chemical plant [Langridge88 p. 25].

- A model has *properties* associated with its constituent parts. The principle properties in graphical modelling are geometric relationships between parts, but properties may also determine the display of the constituent primitives in different colours, thicknesses, textures or styles.

- A model can be *viewed.* That is, it is seen from some geometrically definable viewpoint, and through some frame, and this view is projected onto the display medium.

Construction

In general, models can be constructed procedurally or declaratively. In a procedural method the sequence of commands and its call structure is itself the model, and its primitives directly mark the medium. On the other hand, a declarative method relies on an intermediate *state* whose components can be accessed randomly, and for which there exists a presentation mapping to the display medium.

5.2.3. Procedural Modelling

Procedural representations for modelling images may be

- full languages, like PostScript [Adobe87], Euler-G [Newman71], Dum [Asente87], or Metavisu's [Boullier72 p.253], Kulsrud's [Kulsrud68] or Williams' [Williams72] graphical languages;

- graphics-specific languages like PIC [Kernighan81];

• embedded as library calls in other languages. Window managers often provide a set of low-level graphics primitives in this way.

5.2.4. Declarative Modelling

We examine five types of declarative modelling: standard, direct, constraint-based, syntactic, and synthetic.

Standard Modelling

By standard modelling we refer to the mechanisms provided by the standard graphics languages like GKS and PHIGS. These are declarative to the extent that a model state is created by their operations, and the presentation mapping of this is hidden in the implementation of the package. Thus GKS allows the creation of *segments*, and PHIGS of *structures*.

However, both GKS and PHIGS have procedural qualities. GKS imposes some order dependencies. For example, the contents of a segment cannot be modified once it is closed. On the other hand, PHIGS' model structure is closely analogous to a call structure, and PHIGS' *structures* are strongly sequential, in that the interpretation of structure elements may depend on previously set attributes such as a local transformation. Modifying PHIGS structures is also in effect an *edit* of the PHIGS *program*.

Direct Modelling

Images can be generated directly by interactively manipulating a drawing device. The image is built up by leaving marks on the imaging medium using soft devices (a brush or a stylus, for example) driven by the physical drawing device (a mouse or a puck). Marks can also be left on the medium by positioning and dropping some discrete image primitive (such as a circle or an ellipse), by analogy with an ink block.

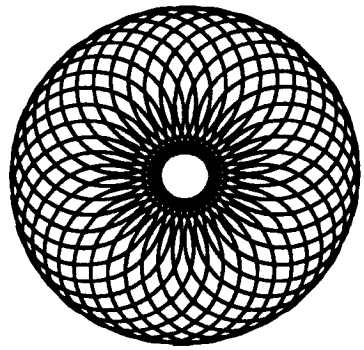
A *model* is constructed to the extent that these primitives persist and retain their discreteness, so that, for example, we can return to a square or a line already drawn, select it, and then move it or delete it without affecting the rest of the image.

However, it is often the case in direct modelling that while at one time we wish to manipulate a primitive (a line, say) on its own, at another we may want to manipulate the same primitive only as part of a larger composite object (a box, say). For

this reason, direct modelling systems (for example Framemaker [Frame87]) provide some means for the user to indicate a group of objects, for example by sweeping out a zone using the mouse. In general, this requires a tree-structured model which can be arbitrarily rearranged.

A number of systems are intended for interactive image generation: Juno [Nelson85], Tweedle [Asente87], Gargoyle [Pier88], and MacPaint and MacDraw [MacDraw88]. These, however, may vary in their *directness*.

Modelling an image directly is analogous to modelling procedurally through a machine language, in that a sequence of operations must be performed on the available image primitives. This analogy is particularly strong if the language has a notion of a current point, so that the program can generate a path by successive *move* calls. This is true of Euler-G [Newman71 p.653] and Postscript, for example, but not of GKS or Kulsrud's language [Kulsrud68]. As Chen [Chen88 p.18] and Asente [Asente87 p.2] point out, however, direct generation may not be ideal where accuracy, regularity, or recursion are required. It would be difficult to generate this PostScript image freehand, for example:



A procedural representation may also be more compact than a transcript of a user's direct actions: a large number of similar objects may be drawn at different locations, sizes, or orientations, simply by a loop which varies an attribute or a parameter to a single primitive call [Newman71 p.651]. The above diagram, for example, is generated by the PostScript line:

36{60 0 45 0 360 arc stroke 10 rotate} repeat

Constraint-based Modelling

Constraints are expressed declaratively. Constraint-based and procedural representations are necessarily equivalent in power, since we can equate the set of data types provided in the first with the set of primitives provided in the second, and can refine constraint satisfaction by procedural methods. As Borning points out, "any sort of relation can be expressed as a constraint, if a procedural test exists and some algorithm can be specified for satisfying the relation" [Borning81 p.380].

Graphical constraint-based systems can be classified according to whether the images produced are static or dynamic. Constraint-based systems like IDEAL [Wyk82], Juno [Nelson85], and Gargoyle [Pier88] produce static images (text or graphics) for inclusion into documents. On the other hand, systems like Sketchpad [Sutherland63], Thinglab [Borning81], Animus [Borning86], Cohen et al's system for windows [Cohen86], Coral [Szekely88a], and parts of the X Toolkit [Swick88 p.225] produce images which maintain their constraints under direct manipulation. As O'Callaghan says, "the need has moved from static description of pictures to maintaining integrity and invariants (constraints) under manipulation" [O'Callaghan72 p.124].

Orthogonally, graphical constraint-based systems can be classified according to whether the images are *produced* textually or by direct manipulation. In Metafont, IDEAL and Juno, and in more general language like Bertrand [Leler88], images are produced by interpreting a constraint program. This is even true in direct manipulation systems like Coral, and the X Toolkit. On the other hand, Sketchpad, ThingLab, Animus, and Gargoyle allow either the image itself, or the constraints on it, to be defined graphically. In ThingLab, for example, merges can be specified by physically moving together the points to be merged. Similarly, Gargoyle's technique of 'snap-dragging' [Bier86] allows constraints to be specified directly by the user through a ruler and compass metaphor.

Whatever the specification mechanism for the constraints, or final purpose for the images generated, in a graphical constraint system the programmer (or user), instead of giving explicit values (for example, coordinates), essentially specifies certain relations between objects. Minimally, he may constrain a variable to a particular constant value (what Borning [Borning81 p.364, Borning86 p.361] calls 'anchored' constraints). More usefully, constraints may be expressed *between* variables, such that the value of one depends on the values of others. The constraints expressible in

this way thus range from simple linear equations, like identifying the endpoints of two lines, to complex non-linear equations, like Barzel et al's mechanical system [Barzel88]. Knuth's Metafont, for example, is linear [Nelson85 p.235], whereas more general systems such as Juno, IDEAL, ThingLab, and Bertrand, can handle non-linear equations. Juno, for example, has the basic constraints of congruence and parallelism, which are capable of generating the whole of Euclidean geometry [Nelson85 p.238]. These require quadratic equations for their resolution [Nelson85 p.235]. In general, the mathematical power required to resolve constraints may or may not be provided, so that in some systems constraints may be expressible but not resolvable.

Typically, *sets* of constraints are specified in a predicate [Nelson85 p.237], all of which must be satisfied. The predicate can be regarded as a system of simultaneous equations. These may be solved by one-pass or iterative methods (for example 'relaxation' [Sutherland63 p.340, Leler86 p.26, Borning86 p.364]), depending on whether there are circular dependencies. Sutherland claims [Sutherland63 p.341] that the one-pass method is in fact successful and efficient in many problems involving geometric dependencies, but is affected critically by the order of evaluation. There are a variety of methods for determining dependency ordering, involving either local propagation of known states or degrees of freedom. In general, the number of constraints and their dependencies is limited only by efficiency considerations. Sutherland gives the example of a cantilevered bridge built of a large number of constrained beams [Sutherland63 p.343].

When using declarative constraints the relation between states may or may not be fully determined. A criticism of (textual) constraint-based systems is precisely the difficulty of ensuring that the given constraints are in fact deterministic [Bier86 p.236]. A set of constraints defining a square, for example, may request a four-sided polygon whose opposite sides are parallel and of equal length [Leler88 p.37]. The trouble is that these constraints are satisfiable by a zero-size square, which may not be what the user intended. Nelson [Nelson85 p.238] gives a similar example of non-determinism in constraint satisfaction. In general, as Borning points out [Borning81], a set of constraints may be incomplete, circular, contradictory, or contain redundancies. If the constraints are contradictory, then it is likely that the solver will not converge [Nelson85 p.242]. Bier argues that debugging a textual constraint set may be as difficult as debugging a program. Juno [Nelson85 p.238] allows the programmer to give the constraint solver 'hints' to aid in convergence. Bier [Bier86]

and Myers [Myers87b] suggest that direct manipulation is an effective means of monitoring and giving hints to the constraint satisfaction mechanism.

Finally, constraint-based modelling may be difficult to extend [Asente87 p.5], since the fundamental data types may be fixed. To some extent this is avoided in systems like IDEAL which allow data abstraction ('boxes'). However, as Leler points out [Leler88 p.87], as constraint languages become more sophisticated, they become more domain specific.

Syntactic Modelling

It is possible to generate or analyse pictures using a grammar [Stanton72, Clowes72]. Grammars generate sequences of primitive elements whose parse structure can be thought of as the model. Grammars can be multi-dimensional as well as linear. There are two important classes:

- Grammars which express topological connectivity, for example in tree or more general web structures. A general grammar for such structures is called a *plex* grammar [Gonzalez78 p.82].

- Grammars which express a picture in terms of constituent shapes. Such grammars are called *shape* grammars [Gonzalez78 p.91].

However, image *analysis* via a grammar foundered on the need but difficulty of maintaining and applying real-world knowledge to supplement the interpretation. On the other hand, image *generation* is complicated by the non-determinism of plex or shape grammars. In order to specify an image precisely, it is necessary to give some coordinate information along with the grammatical productions [Milgram72]. In this sense a picture grammar can be seen as a type of constraint system. The GREEN system [Golin90] combines a picture grammar with simple layout constraints like 'over' and 'left_of'. MicroCOSM [Barford89] allows more precise constraints expressed as attributes of the picture grammar.

More recently, grammars have been used to specify more restricted classes of images, for example software engineering diagrams [Szwillus87, Woodman87], or forms [Sugihara86]. All these recent systems in addition allow syntax-directed editing of the diagrams, for example by the use of templates. Woodman (p. 114) points out that this entails precisely the same problems of incremental update as with textual syntax editors.

Synthetic Modelling

Synthetic modelling simply assumes an image-generating transform from some underlying, non-graphical data. The major uses here are in database viewing [Friedell84, Mackinlay86, Herot80, Larson86, Garrett82] and scientific visualisation [Brown84, BrooksFP88]. We will not explore this area further.

5.2.5. Structure

Both procedural and declarative modelling result in a *structured* image. As Foley and Van Dam [Foley84a p. 328] note, there is a close analogy between a procedural call hierarchy, and a declarative object hierarchy. In either, we can say that a node on the hierarchy 'consists' of its subroutines/subobjects, and we could represent either by an acyclic directed graph structure.

We can apply two fundamental criteria to the resulting structures:

- their degree of generality
- their degree of persistence

Generality

In a procedural hierarchy it is strictly the leaves which may have displayable content (primitives), since interior nodes are by definition simply calls. In contrast, as we have seen in hierarchical window systems, there is nothing to stop a declarative object hierarchy having content at an interior node.

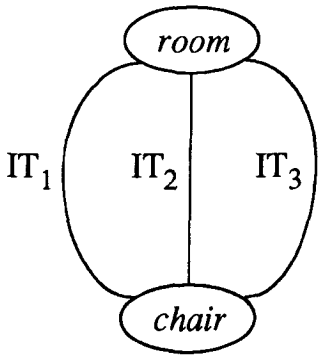
So-called hierarchical window systems are in fact usually limited to tree structures, that is, hierarchies in which children have only one parent. Thus windows are unique - they are not replicated by being shared ('called') by more than one parent.

The depth of the modelling hierarchy may also be limited. In GKS, for example, the hierarchy is limited to two levels, segments and primitives. This is clearly a restriction, and PHIGS has removed this by using a fully hierarchical structure in which segments ('structures') may include other segments. Similarly hierarchical systems are NGS [Cahn83], VGTS [Lantz84], and PostScript.

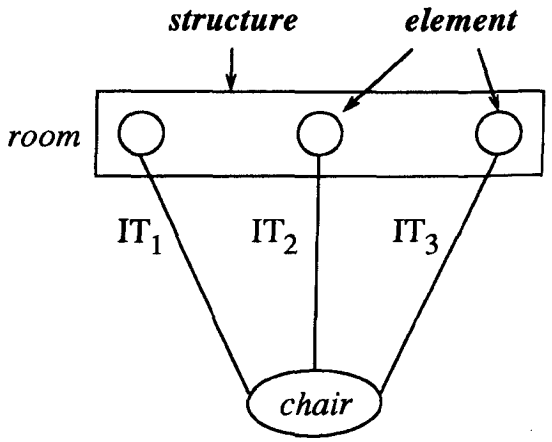
In addition, the only way to replicate a segment in GKS is to copy it. This means that in order to provide multiple views of the same object each view must be

copied and separately updated by the programmer [Little87]. In PHIGS, on the other hand, a structure may be executed repeatedly under different transformations, so that an update to the structure is automatically reflected in all the instances.

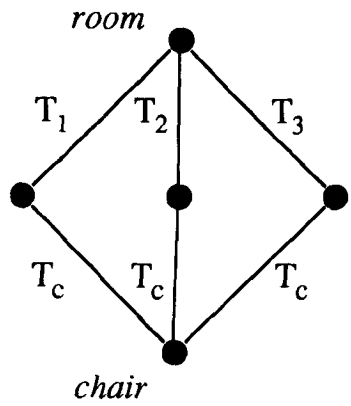
Other differences may only be virtual. Foley and Van Dam [Foley84a p. 327], for example, represent instance transformations as multiple arcs between a composite object and the instances that make it up:



Thus a room (here) consists of three chairs at different locations determined by their instance transformations. This representation, however, actually requires two different types of entity, one of which contains the other. Thus, in this example, *room* implicitly contains three calls on *chair* with different transformations. In PHIGS terminology, *room* is a *structure* which contains *elements:*



The generality of PHIGS arises from the fact that elements can either call primitives, or other structures. However, this generality can be achieved more simply, as in the models presented in this Thesis, by having only a single type of entity:



The *room* node simply represents the group of *chair* nodes. We can thus identify the transformations with the intermediate nodes: we think of a node as being of some size, position, and rotation with respect to its parent. The *chair* itself has the same transformation with respect to each of the intermediate nodes. The *compositions* of the transformations T_i and T_c, however, are all different.

As well as geometrically, a composite object may also be structured topologically (i.e. in terms of its connectivity), or by its attribute inheritance paths. [Strubbe83 p.1036] notes these three structures, but claims that in most cases they coincide and that one structure will do. The second model presented in this Thesis does not make this assumption. That is, it allows the possibility that topologically related objects may not be geometrically related, and that attributes may be inherited down separate structures that are neither geometric nor topological.

Persistence

In order to selectively modify the model, its structure must *persist*. It is impossible to manipulate primitive elements like lines or circles in a bitmap, for example, once the structure has been lost. At what level of representation the model persists will affect the efficiency of

- incremental update of the structure
- incremental redisplay of the image

A purely procedural representation, like PostScript, persists only at the level of its text, which can only be accessed serially. In order to update such a representation, its script or program must be textually edited and then re-executed. Clearly, this may result in unnecessary redisplay of some parts of the image. This, however, is not a concern for languages that are designed for document imaging, like the recent class of Page Description Languages (PDLs), such as PostScript and Interpress [Bhushan86]. These are more concerned with the efficient transmission of images than with interaction (see Chen [Chen88 p.21], Harris [Harris86], and Reid [Reid86]).

In order to optimise the redisplay of a procedural representation for interactive purposes, however, the redisplay must ideally be localised to just those areas of the screen which have changed. The problem here is that making a change at one location in the script may affect many areas of the screen, or one location on the screen may have images generated from many different places in the script. Newman points out these difficulties [Newman71 p.659]. In addition, in a direct manipulation system, screen objects may obscure, or be obscured by, other objects, such that in the worst case a minor redisplay may affect all other images [Slater88 p.7].

Procedural representations either force this low level 'damage repair' into the application, or must generate a persistent, structured representation (sometimes referred to as a *graphical database* [Lantz84 p.30]) which can be optimised for display. GKS, CGI and PHIGS for example generate intermediate segmented display lists.

A major feature of such a database is that its objects must be *identifiable* [Kulsrud68 p.248], so that they can be accessed randomly rather than serially. GKS allows the identification of segments, which can subsequently be deleted, or changed in priority or visibility, etc. However, primitives in GKS cannot be individually identified for modification, and segments cannot be updated once they are closed [Enderle84 p. 38]. The granularity of modification in GKS is thus in practice the segment, and this may be too coarse for direct manipulation graphics [Olsen85b p.194, Harke87 p.100]. PHIGS allows the identification of structures, but it also allows labels to be inserted at arbitrary points inside structures, and the elements between two labels may be deleted.

However, as Foley [Foley79] illustrates, not all systems allow an identifiable object to be modified. In GKS, for example, the PICK identifier on primitives is used

only for hit detection, while in PHIGS naming of elements is used only to filter certain input events [Brown85 p.44]. Asente, in his Tweedle system, addresses the problems of interactively editing a procedural representation for images [Asente87].

A declarative representation, on the other hand, more naturally persists at the level of the displayable objects which it creates. Ideally, these can be identified and accessed randomly, and their properties are localised. Objects can also cache intermediate display results so redrawing is efficient. For these reasons a declarative representation is more amenable to direct manipulation as well as textual modification [Pereira86, Helm86, McCabe87].

Properties

In general, image primitives consist of some essential *shape*, which is given:

- *Geometric* properties, such as size, position, and rotation
- *Rendering* properties, such as colour, thickness, and style

It is a philosophical question as to which properties may be thought to *inhere* in an object. For example, does a line have thickness? Generally speaking, however, the more properties that can be abstracted the better. An abstracted property can be *inherited* by a number of objects, thus improving modularity and reuse.

Some graphics languages exploit the abstraction of properties by preserving a set of global *attributes* which determine the properties of subsequent primitives. Since the current *state* of the attributes is thus important, these representations are commonly called *state-based*, or *state-driven*. In PHIGS, for example, one may *SET POLYLINE COLOUR INDEX* to some colour, after which all executions of the *POLYLINE* primitive inherit this colour, until it is changed. Thus one can draw a circuit consisting of a number of resistors, in different positions but the same colour (the syntax is used loosely):

```
OPEN STRUCTURE 'CIRCUIT'
    SET POLYLINE COLOUR INDEX 2
    SET LOCAL TRANSFORMATION T1
    EXECUTE STRUCTURE 'RESISTOR'
    SET LOCAL TRANSFORMATION T2
    EXECUTE STRUCTURE 'RESISTOR'
    ...
CLOSE STRUCTURE
```

(Structure *RESISTOR* contains, we assume, some *POLYLINE* calls). GKS, PHIGS, CGI, Juno [Nelson85 p.237] and PostScript are all state-driven. Similarly, X has *graphics contexts* (although Scheifler [Scheifler86 p.106] claims the X protocol is stateless).

An alternative approach is to pass properties at the point of call. If only primitives are being called, then properties can be passed as parameters. However, if there is a more general hierarchical call structure, then the *inheritance* of properties passed as parameters would rely on the programmer passing them on into the subcalls. A more automatic scheme uses special calling mechanisms. Newman's Display Procedures [Newman71], for example, pass geometric properties using explicit keywords at the point of call:

```
resistor at [50, 50] scale 1.5;
resistor at [100, 100] rot pi/2;
resistor at [0, 70] scale .5 rot -0.5;
resistor at [20, 30] trans m;
```

These properties hold for all subcalls. The advantage of this approach is that the properties are localised to the call, and previous settings can be restored upon return from a substructure. In PHIGS, attribute state is also maintained on a stack in this way. PostScript similarly maintains a stack for the graphics state, but it is necessary for the programmer to save and restore this state explicitly.

Localisation of properties also facilitates incremental editing. In a state-based system, on the other hand, the current attribute settings are *implicit*. It may not be possible to determine the precise effect of a primitive by examining the script syntactically or even statically - the current attribute state may depend on a complex execution trace. PostScript is especially difficult in this respect, since it has a large graphics state including not only line style and width, but also current path and current transformation matrix. Young also makes the point that state is not compatible with interleaved update from several cooperating processes [Young88 p.373], which in any case standard graphics does not support [Lantz84 p.46].

A fixed set of attributes, either global or local to the call, may introduce restrictions. In PHIGS, the attributes provide the only medium by which information may be passed to substructures. There thus exists what has been called the 'barber's pole' problem [Hewitt88]: since there is only one *POLYLINE COLOUR INDEX*, for

example, it is impossible to parameterise *both* colours in a substructure which draws a barber's pole. It would not be possible, therefore, to draw multiple instances of a barber's pole (or any other multicoloured structure) in which both the colours were different. PostScript is more powerful in this respect, since it can pass arbitrary numbers of parameters to subordinate procedures. In this case, however, as noted above, the inheritance mechanisms must be explicitly supported by the programmer.

Finally, there may be differences in the binding time of attributes. In GKS, for example, primitives are bound to the attributes current at definition time. In PHIGS, on the other hand, attribute binding occurs at execution time. Thus in PHIGS the inherited properties of objects can be changed dynamically.

A declarative and fully persistent model, such as the one presented in this Thesis, is not necessarily state-based, since it may not possess *global* attributes.

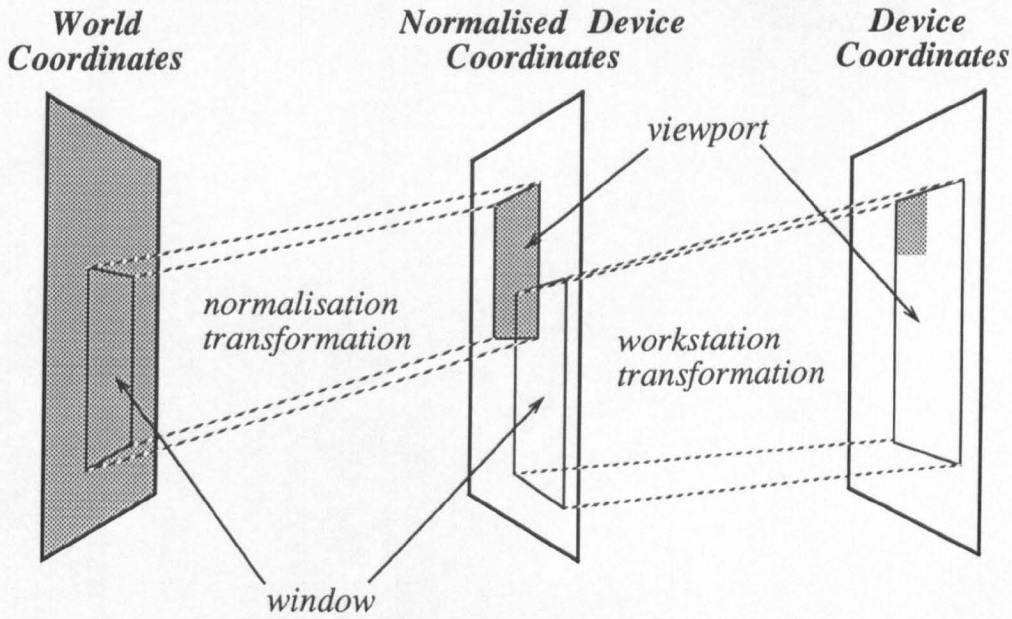
5.2.6. Viewing

Although a model may be used simply as an application database, its primary purpose is to be seen by the end user. The user's *view* of the model is necessarily limited by the display space available, the projection of the model onto the display surface, and the configuration of the model itself. The view is particularly critical in three-dimensional scenic graphics, where the projection may be parallel or perspective, oblique or orthographic [Carlbom78]. Illumination may also need to be taken into account. Even in two-dimensional schematic graphics, however, the view may be panned or zoomed inside the frame of the display surface, and components of the model may obscure other components.

Fundamentally, then, a view imposes

- a transformation
- a clipping

The window-viewport-workstation transformations of standard graphics subsumes both transformation and clipping:



That is, both the normalisation transformation and the workstation transformation are defined not by matrices, but by rectangles in WC and NDC, and in NDC and DC respectively. These rectangles may also clip the primitives. One rectangle (the window) defines *what* is to be displayed, the other (the viewport) *where* it is to be displayed.

Salmon and Slater [Salmon87] make some cogent criticisms of these distinguished coordinate spaces and transformations. They note (p. 555) that there is no reason why the chain of window/viewport pairs could not go on indefinitely. This would simply result in a sequence of transformations ending in a display surface transformation. Rosenthal sees Normalised Device coordinates as 'superfluous' [Rosenthal83 p. 42]; VGTS, for example, does without these [Lantz84 p.32]. HI-VISUAL [Monden86] implements hierarchical viewing in the context of an iconic window system. Similarly [Salmon87 p. 290], the notion of *world coordinates* as a definitive scale for objects breaks down when one considers many common real world scenes. A room, for example, may have a picture hanging on its wall which shows another scene of a room, and so on. Is the picture in world coordinates?

The hard distinction often made in mainstream graphics between the modelling system and the viewing system (e.g. [Guedj79 p.201]) is an implementation issue: if a model is clipped in a particular view, then it is more efficient to regenerate the display from an intermediate, clipped and transformed representation than it is to repeatedly clip and transform the model. Conceptually, however, there is no real need to make this distinction between modelling and viewing. Even in 3D graphics the notion that viewing transformations can be separated from modelling transformations ([Guedj79 p.193]) is invalidated by the example of a picture hanging on a wall: the relation between the scene in the picture and the picture surface is a viewing, not a modelling transformation. PHIGS does not escape this criticism: it associates the viewing transformation with a workstation [Brown85 p. 99], and therefore cannot apply viewing at arbitrary levels of the model. PHIGS in fact preserves the simple three coordinate systems of GKS - the modelling transformations are composed into the WC space, and only then are viewed.

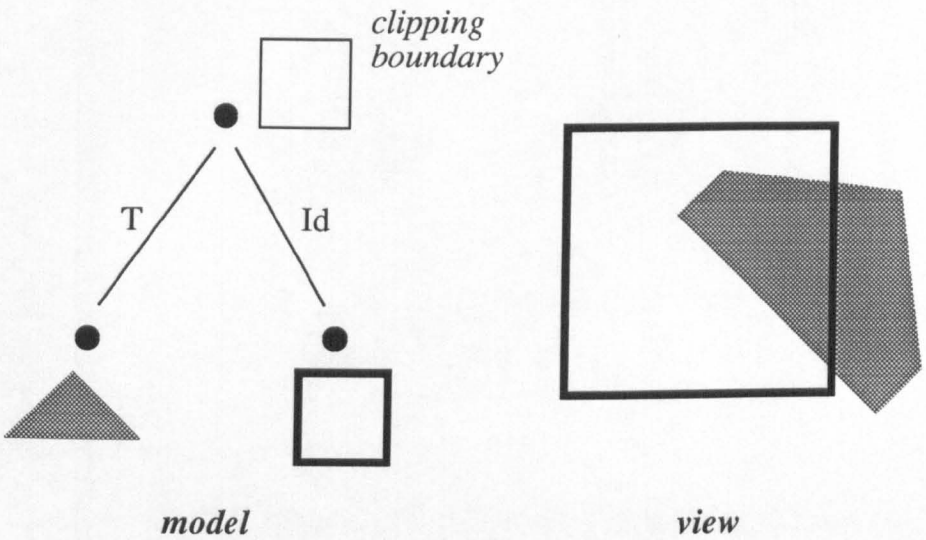
In two dimensions it is easier to avoid this issue. Here the view transformation requires no more information than the modelling transformation. PostScript, for example, makes no distinction between viewing and modelling [Salmon87 p.299]. This is also true of the model presented in this Thesis. In Presenter, the size of the root of the hierarchy is simply taken to be with respect to the display surface available.

Clipping

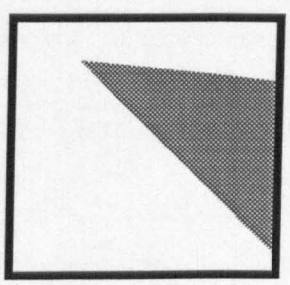
In both GKS and PHIGS clipping is treated as an integral part of viewing. However, it is equally possible to conceive of clipping as part of the model rather than the view. This is particularly the case in schematic graphics where there may be a number of nested or overlapping 'windows' onto discrete information spaces, as for instance in a standard window manager. (This is analogous to the 'picture on a wall' requirement in 3D).

A more useful approach, therefore, is to allow clipping at all levels of the model [Foley84a p. 382]. Defining clipping separately from the viewing transformation also means that arbitrary clipping paths can be set up, rather than just rectangular ones. The models presented in this Thesis generalises this by allowing arbitrary masks to be associated with any region in the hierarchy.

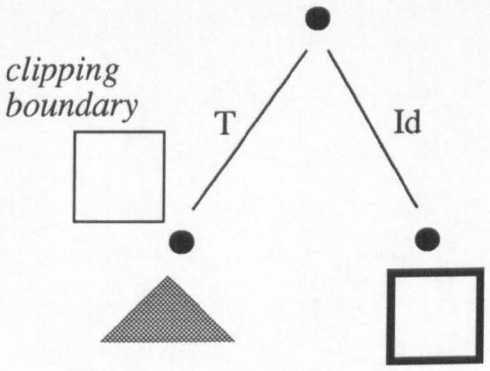
The question remains of when, during the modelling transformations, the clipping should take place. This is not just an implementation issue, since different strategies will affect the image differently. [Foley84a p.384], for example, suggests that clipping should be applied immediately, and then transformed by any further modelling transformations. In this illustration, *T* is a transformation involving rotation, and *Id* is the identity transformation:



In the model presented in this Thesis, however, the clipping is in effect delayed until the modelling transformations have been completed. This means that the boundary of a clip *coincides* for all descendants of a region with which it is associated. Using this strategy the model above would be viewed:



This is clearly of more use in windowing. However, the first view can also be achieved using this strategy simply by resiting the clipping rectangle:



That is, the clipping rectangle is affected by any modelling transformation *above* it on the hierarchy. It is therefore fully part of the model.

Dereferencing the View

Just as the objects of the model can be accessed internally by means of labels or some other persistent representation, in an interactive system the user also needs to be able to access model objects through the view. That is, there must be some *inverse view* function whereby objects can be identified from their surface representation by a process of *hit detection* or *picking*.

Dereferencing the view is problematic

- if the model and the view are separated by an intermediate, display optimised representation like a display list, since part of its optimisation is typically to lose some, if not all, of the model structure.

- if the display representation is purely procedural, since details of screen objects may not be retained, other than on the screen, after its execution. Also, as we have seen, localising exactly what constitutes an object in a procedural script may be difficult.

- if the model is hierarchical, since a model object may have a number of instantiations in the view. For example, a symbolic key on a map may have many instances [Visvalingam87]. In hierarchical systems it is thus the *path* of model objects down the hierarchy that is significant. Upon a hit, PHIGS for example

returns the entire *pick path* to the application [Brown85 p.88]. In implementation this requires rerunning the script and tracing the execution stack to determine which structure call generated the object the user is attempting to select [Toby Howard, private communication].

For these reasons the application may be forced to maintain its own database of object extents and positions in order to dereference user selections of the view. In any case, view dereferencing may be lengthy, leading to synchronisation problems between input and output [Newman71 p.659]. A declarative representation, by modularising object references, can make view dereferencing more efficient.

5.3. Text

A surface model for text is closely analogous to that for graphics. Text has content and structure, and can be composed and viewed. In addition, the requirements for content replication, attribute inheritance, multiple views and persistence are as present in text as in graphics [Took86a].

However, displayed text has a different *geometry* from graphics, if by geometry we mean a set of basic operations and their surface effects. Thus the basic operations of text are not translation, scaling, or rotation, but insertion and deletion. Even the subtle variations in character spacing required by high-quality formatting must be modelled by special inserts like kerns and fills [Knuth86 p.78]. To put it another way, the precise positions of characters in displayed text are a function of the character *sequence* and its attributes, rather than of any higher text structure. Deleting a *graphical* object, for example, does not normally affect the positions of other graphical objects, whereas this is the case with text. This view must be qualified by excluding questions of page layout, which is discussed in Section 5.4. It must also be acknowledged that the formatting of mathematical text is a special case which it may be better to model within a Cartesian, rather than a textual, geometry [Allen81 p.80, Nanard87 pp.75, 77, Arnon88 p.9].

5.3.1. Content

Just as the basic constituent of graphics is a primitive image, so displayed (roman) text consists of discrete character images. While these will have a logical representation (for example, the ASCII code), their surface presentation will be in

the form of ideographs in some font, face, and style. The design of character fonts is an arcane art [Bigelow86], but, as Knuth's box model [Knuth86 p.63] shows, for the purpose of formatting it is easy to abstract away from the aesthetics of font design to simple rectangular areas.

However, in addition to characters, text may contain other types of image. The Office Document Architecture standard (ODA) defines three types of textual content: characters, geometric graphics, and raster graphics [Appelt88 p. 95, ISO87a]. Other divisions of content are possible. [Kimura86 p.417], for example, has text, tables, equations, and figures as fundamental classes.

5.3.2. Logical Structure

The fundamental perception of text is as a sequence of elements [Wills87a p. 25] (although Kimura [Kimura86 p.418] also includes unordered compositions (sets) of objects). Orthogonal to this sequence, however, there is inevitably a further logical structure. This will at least consist of the lexical and syntactic structure of the text, that is, the grouping of characters into words and words into sentences. But it may also be extended into a structural hierarchy of nodes such as paragraphs, sections, chapters and volumes.

Simple line or screen editors may retain nothing of the logical structure of text other than is embedded in the character sequence, for example as carriage return tokens. Systems that do support structure can use this either prescriptively or descriptively. Structure-*driven* editors [Morris81, Medina-Mora82], for example, prescribe the structure using a grammar, and are designed to keep the user within the syntactic limits of a particular programming language. Other structure-*oriented* systems, such as Grif [Quint86], Pleiade [Nanard87], Quill [Chamberlin88 p.123], and IDE [Kaplan88 pp.194, 199] are designed to support more general documentation, and are descriptive in their use of structure.

Considered in the abstract, textual and graphical models have similar requirements. There will minimally be a need to group terminal elements. This results in a tree structure. A number of systems support tree structured text: Tajo [Teitelman86 p.42], Tioga [Beach83 p.131], Diamond [Crowley87 p.3], Grif [Quint86 p.202], PEN [Allen81 p.75], Etude [Hammer81 p.140], LateX [Lamport86], and Hamlet's proposed system [Hamlet86]. However, the tree structure may be restricted. Analogously to GKS, Framemaker [Frame87, Wilcox88 p.53], on which this Thesis

has been written, supports only a two-level tree structure of paragraphs and characters.

Just as in graphics, it may also be useful to *share* some terminal elements at a number of locations. Running headers or footers, for example, could be modelled efficiently in this way. This requires a full hierarchy in which a node may have more than one parent. Few systems, however, support such replication. Kimura's model [Kimura86 pp.418, 420], and IDE [Kaplan88 p.198] are exceptions.

Finally, if the nodes of the logical structure are as persistent as the text, then it is possible to support an arbitrary structure over them. This can be exploited to form a networked database by which connections can be made between otherwise unrelated pieces of text. ODA, for example, allows cross references [Horak85 p.51] while Kimura has 'links' [Kimura86 p.420]. This capability is also called *hypertext* [Nelson80, Yankelovich88 p.92, Conklin87, Brown87, Feiner82, JonesWP88, BrownP88 p.184, Ritchie89].

5.3.3. Properties

Just as in graphics, properties attached to the logical structure affect the rendering of the text primitives in terms of stylistic qualities such as font, face, and pointsize [Johnson88, Beach83, Chamberlin88, Joloboff86 p.121]. Again by analogy with standard graphics, text properties may be classified or 'bundled' into 'style rules' [Johnson88 p.34] which may be inherited down the paths of the structure, as in Kimura's model [Kimura86], or in ODA [Brown89 p.506].

5.3.4. Editing

The manipulation of text and text structures is conventionally termed *editing*. We can make exactly the same distinction between procedural and declarative models in text as we can in graphics. A procedural model is a *program* which generates some final text, while a declarative model allows random modification of the state of an Object from which there is a presentation mapping (these are elsewhere [Chen88 p.15] called *source-language* and *direct-manipulation* approaches respectively). Editing thus requires a declarative model, since it modifies the state of a text.

Procedural models usually consist of a text program which is constructed by embedding special tokens or keywords in a representation of the raw text. This is

known as *markup* [Joloboff86]. This program is then either *interpreted* by a formatting process on a workstation to produce a screen view, or *compiled* into a page description language (PDL) [Marovac87] and then *run* on a laser printer to generate a hardcopied view.

Markup itself may have either a procedural or a declarative bias [Joloboff86 p.110, Chamberlin81 p.83, Furuta82 p.460, Dam82 p.46, Chen88 p.22]. Procedural markup such as provided by troff [Lesk86] or TeX [Knuth86], gives the editor fine control over the appearance of the final text format with commands that set style, spacing, and justification. Declarative markup, on the other hand, such as in Scribe [Reid80], LaTeX [Lamport86], GML [Goldfarb81] and SGML [Chamberlin87, Stutely87, Beaujardiere88 p.86, Joloboff86 p.110], allows the editor to describe simply the structural class of components of the text, for example 'title' or 'section'. The precise formatting of these classes is left up to a separate property specification. The binding of the text to its final form is thus delayed, and may indeed be specified by someone other than the original writer. In this way device and other dependencies can be minimised [Chamberlin88 p.128], and the document may be viewed in a number of ways simply by transforming its structure [Furuta88] or its properties [Beach83]. For this reason, Joloboff calls these models 'revisable formats' [Joloboff86 p.107].

The editing of a procedural text model like these markup models is thus a separate process from its final formatting [Huu87]. The editing and formatting of procedural models, however, may vary in their concurrency. Standard markup languages are often used in a 'batched' pipeline [Furuta89] where output is generated only after editing is complete. The Tioga editor, for example, has a batch-oriented typesetter [Beach83 p.130]. However, a number of systems allow editing and formatting to run concurrently, so that the final format is displayed alongside the text program and incrementally updated after each edit of it [Chamberlin88 p.129]. The program and the final format necessarily remain distinct, so that these are often called 'two-view' editors. Examples of such two-view editors are Lilac [BrooksKP88], Janus [Chamberlin82 p.82], and its later version ICEF2 (both of which use SGML) [Chamberlin88 p.123], IDE [Kaplan88 p.194], VorTeX (which uses TeX) [Chen88 p.26], and the Andrew text editor [Morris86 p.198]. There are also similar two-view editors whose output is largely graphical, such as Juno [Nelson85 p.235], and Tweedle [Asente87].

These two-view editors can also be classified according to whether they allow the user to perform edits via either view, or just via the program view. Janus, for example, only allows edits to the program view, but VorTeX and Tweedle allow edits to both views.

Declarative models, on the other hand, *hide* the text structure, except insofar as this is visible in the presentation. The user *directly* edits the final form of the text. The text structure is implicitly accessible by cutting and pasting the displayed text. In addition, stylistic properties [Johnson88] attached to the structure may be accessible through special views such as Star's *property sheets* [Smith82a], Quill's 'looks' [Chamberlin88 p.123], or FrameMaker's *paragraph* dialogue box [Frame87].

Editors such as these are termed single-view, or WYSIWYG (What You See Is What You Get) [Chen88 p.20, Johnson88 p.33, Walker88 p.58]. They are *direct* in the definition of this Thesis because the user accesses the model *through* the final form view. Some editors, however, display only an approximation of the final formatted view [Furuta86]. Pleiade, for example, displays a 'nearly-exact' version of the document [Nanard87 p.77]. Concordia displays a 'semblance' of the final form [Walker88 p.51]. Chen and Harrison call these 'galley-oriented' editors [Chen88 p.19].

An early example of a direct editor was the Bravo editor on the Alto [Chamberlin82 p.83, Wills87a, BrooksKP88 p.146]. More recent examples are Andra [Gutknecht84], Lara [Gutknecht85], Quill (using SGML as its Object state) [Chamberlin88], Pleiade [Nanard87 p.74], Diamond [Crowley87], the Chelgraph SGML editor [Cadogan87], and Grif [Quint86].

5.4. Documents

The compositing of documents, in the general case, involves integrating both graphical and textual elements [Southall88]. The general issue is whether a Cartesian or a textual geometry applies to any particular object in a document. To be completely general, we should allow both text to be positioned using Cartesian coordinates (for example, to place a paragraph on a page), and also graphics to be positioned using textual coordinates (for example, to embed a diagram in a paragraph). This implies a fully recursive structure in which text and graphics may be arbitrarily nested, and for which the top level geometry may be either textual or

Cartesian. This section examines existing model specifications and editors against this general capability.

Conventionally, textual geometry is implemented by *formatting*, while Cartesian geometry, in the context of documents, is implemented by *pagination* [Wills 87a p.24, Clarke87], *layout* or *pasteup* (by analogy with manual document production) [Chamberlin81 p.82]. It is a contribution of the second surface model presented in Chapter 8 to make no distinction between document layout models, and more general graphical models.

5.4.1. Formatting

Formatting is essentially a function which takes a sequence of characters and a bounding box, and returns a description of the position of each character within the bounding box. In order to do this the function must have access to properties of the characters such as font, face and pointsize. These properties may be associated either with the characters themselves, or with the bounding box, or may be fixed globally in the function.

In addition, there may be properties of *sections* of the text, for example constraints on paragraphs such as spacing and justification. Thus the formatting function may also have to take note of the *logical structure* of the text. This can however be embedded in the character sequence as Carriage Returns (to indicate paragraph boundaries). More detailed structure can be embedded as markup. Alternatively, if a declarative model is used then the formatting function will reference any associated structure in the Object state.

In the general case, it is not possible for formatting to occur independently of the values of the characters. This is largely because proper formatting needs at least to break lines on word boundaries, and so must be aware of white space in the character sequence. Also, features such as hyphenation [Nanard87 p.79, Clarke87 p.208] require much more sophisticated mappings between character sequences and format. Similarly, different languages may have different formatting requirements, for example vertical rather than horizontal lines. In formatting music, the last line must always be full [Hegazy88 p.157].

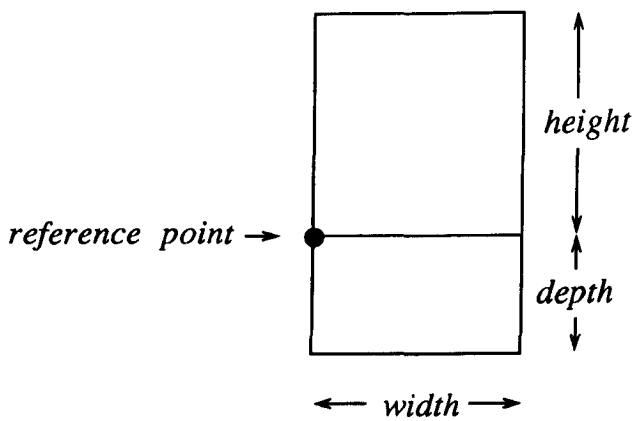
The scope of the formatting algorithm may vary. In order to reduce response time, most interactive systems format line by line [Achugbue81 p.119]. However,

TeX is capable of optimising the spacing of words paragraph by paragraph [Knuth81]. If format occurs over a number of bounding boxes, as in pagination, then paragraph breaks and spacing may also need to be taken into account, in order to avoid 'widows' and 'orphans' (single lines at the top or bottom of the page) [Nanard87 p.79].

Embedding graphics in text is accomplished simply by treating the graphics as a character. The size of the bounding box of the graphics is substituted as the character size. More sophisticated formatters, such as TeX, can in addition flow text around an irregularly shaped graphic.

5.4.2. Layout

An influential model for the layout of document structures has been Knuth's *box* [Knuth86]. A box is a rectangle with the following characteristics:

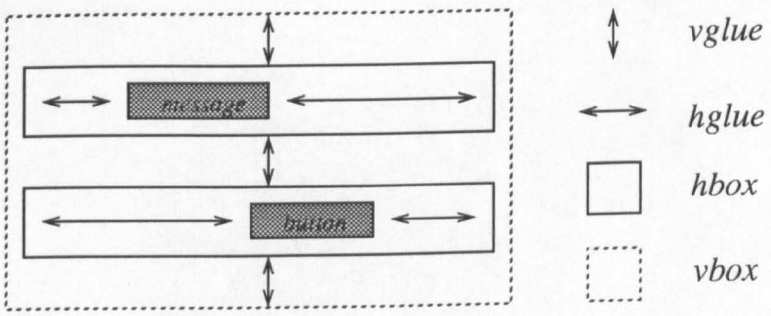


Boxes can represent characters, lines, paragraphs, diagrams or any discrete element of a document. Boxes are themselves structured into a tree structure with the constraint that child boxes are strictly nested inside parent boxes. To this extent they are similar to hierarchical window models. Boxes are also further constrained in that siblings do not overlap, and are composed either vertically or horizontally within their parent.

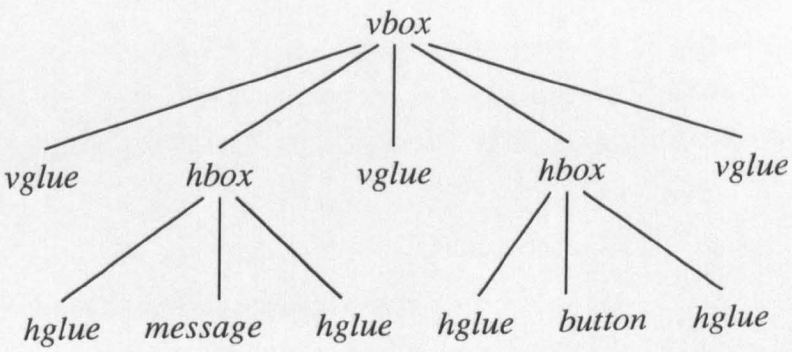
The box model gains much of its power from the notion of 'glue' [Knuth86 p.69] which may be inserted between sibling boxes. This glue controls the amount by which spaces between adjacent boxes, and between the end boxes and the enclosing parent box, can stretch or shrink. While in TeX the intention is to produce very

subtle variations in text spacing under modifications to the text or the bounding box, in fact this model is a generally applicable constraint system for the layout of two-dimensional rectangular areas.

This has been exploited by a number of systems. The following dialogue box, for example, can be constructed in Interviews [Linton89 p.13] using the box and glue model:



The constituent boxes of this dialogue box have the following tree structure:



The various pieces of horizontal and vertical glue can be given different stretching and shrinking capabilities, as well as a natural space, so that a great deal of variation can be produced in how the dialogue box behaves under changes to the size of its bounding box. Interviews also extends TeX's glue model by allowing diagonal glue between corners of boxes, and by allowing boxes to overlap.

Other systems which use variants of Knuth's box and glue model are Genie-M [Angell87 p.131] (where the boxes are called *tiles*), PEN [Allen81], Etude [Hammer81 p.140], Janus [Chamberlin82 p.87], Quill [Luniewski88] (where the boxes are called *blocks*), Grif [Quint87], and ODA's *frames* and *blocks* [Brown89

p.505]. In all these systems the boxes are strictly nested within their parent. Some of these systems, however, for example Grif, use a more explicit positioning of the boxes than glue-based spacing.

The hierarchical structuring of boxes has also been exploited by systems which express layout using a grammar [O'Callaghan72, Sugihara86 p.112, Coutaz86, Woodman87 p.113] (see also Section 5.2.4). It is a feature of these systems that positioning is often expressed relatively and loosely, using constructs such as 'above' or 'to the right of'.

One important layout which it is difficult to express using a tree structure, and which is rarely supported, is tables (tables are not supported in ODA, for example [Behrmann-Poitiers88 p.81], whereas Intermedia requires tables as a basic type alongside text and graphics [Yankelovich88 p.92]). Cells in a table belong to two structures, a row and a column. For this reason Kimura [Kimura86 p.420] believes that a cell should be shared between a row parent and a column parent, and thus that tables require to be structured as full hierarchies. However, if the hierarchy is treated as a layout structure, then the cells are essentially replicated between the columns and the rows. What seems to be needed is an orthogonal structure which maintains the table matrix, while the layout of the table can be expressed as a constrained tree structure. These constraints are detailed in Chapter 8. Issues of tabular layout are also discussed in [Kaplan88 p.200, Gutknecht84 p.98, Murrel87, Beach86].

5.4.3. Integrating Format and Layout

One of the most difficult problems of compositing is resolving the various format and layout constraints to produce an integrated document. This is because there may be subtle dependencies between the text and the graphical layout. For example, a footnote should be placed at the bottom of the page on which its callout occurs [Luniewski88 p.214]. However, if the callout is near the bottom of the page, inserting the footnote may push the callout onto the next page. Similarly, there may be 'computed format' [Chamberlin88], for example references to page numbers that are expanded in line by the formatter. In the worst case the insertion of a reference may cause the referent to move to another page. These problems may require elaborate constraints (see [Luniewski88 p.214]) or multiple-pass formatting, which can reduce interactive response.

There are two approaches to integrating format and layout: format-driven and layout-driven. These correspond to using a textual or Cartesian geometry respectively at the top of the modelling hierarchy. A format-driven approach allows the layout to be specified from within the format, for example by allowing markup, as in troff, to determine columns, spacing or page breaks. The limitation of this approach is that the layout produced is dependent on the textual content: if the pointsize changes, the resultant layout may be quite different. Joloboff notes that markup is unsuited to layout [Joloboff86 p.115], and indeed SGML does not provide support for layout [Brown89 p.505].

Alternatively, a layout-driven approach [Chen88 p.19] allows layout to be specified independently of textual content. This might well be required, for example to maintain a similar layout over a number of pages. The layout-driven approach has three consequences:

- Layout can be specified interactively, by moving and sizing rubber boxes.

- Some mechanism must be determined whereby text is distributed into laid out boxes. A common mechanism is the 'pouring' process of Interscript [Joloboff 86 p.115, Nanard87 p.75]. This also requires a record of the text *flow* (that is, a sequence of boxes), such that as text overflows from the bottom of one box it begins to pour into the next.

- Multiple text streams can be incorporated into one document, since different streams can be associated with different sequences of boxes [Hammer81 p.140, Chamberlin82 p.86, Cowan86 p.141].

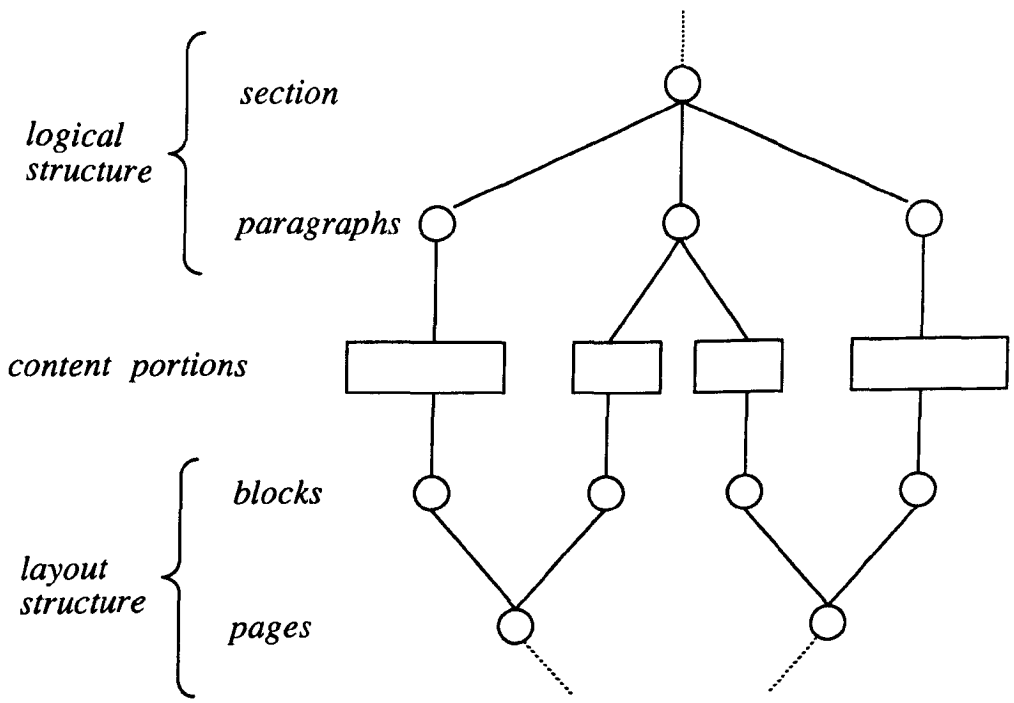
A layout-driven approach therefore seems necessary also for multimedia documents [Crowley87]. The issues here are the extent to which editors for the different media (for example, text, mathematical expressions, graphics, sound, video) are integrated [Chen88 p.16]. There may also be a separate editor for the model structure [Hammer81 p.142].

Separate editors may be highly specialised, but it may be difficult for the user to switch contexts between them. It may also be difficult to manage cuts and pastes between different sections of the document. In order to edit a maths expression in a Tioga document, for example, the expression must be extracted into the CaminoReal tool, edited, and then reinserted into Tioga [Arnon88 p.10]. Diamond [Crowley87 p.3] and IDE [Kaplan88 p.195] opt for a strongly integrated editing environment,

whereas Quill consists of a number of specialised editors [Chamberlin88 p.124, Luniewski88 p.206].

Format-driven compositing essentially delays the binding of the layout, whereas layout-driven compositing and the pouring process delays the binding of the format. It seems necessary to give one priority over the other in order to resolve the possible conflicts between layout and format noted above.

In keeping with its status as an international standard, the Office Document Architecture (ODA) allows both the logical and layout structures of documents to be described [ISO87a, Horak85, Beaujardiere88, Brown89, Joloboff86]. It is, however, layout-driven, since the format is finally determined only at the run time of the ODA processor. This results in the following dual structuring of the document content (adapted from [Joloboff86 p.120]):



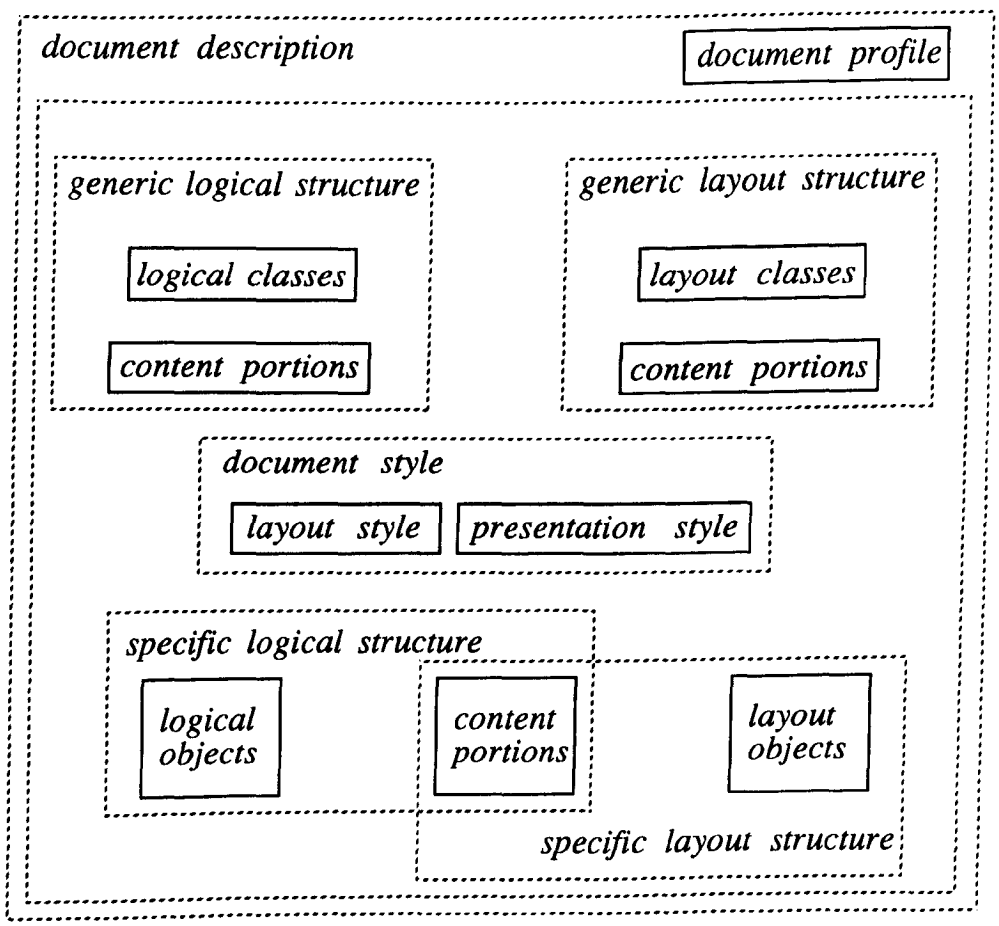
The final format is instantiated in the breakdown of the *content portions*. In this example, a logical paragraph is broken over two pages.

The power of the ODA lies in its provision not only of specific logical and layout structures as above, but also in the ability to specify *generic* logical and layout structures. The generic logical structure corresponds to the class hierarchy which

can be described for example by *elements* in SGML's *meta language* [Brown89 p.510]. Both SGML and ODA provide a simple grammar to determine the permitted sequences of terminal nodes in this hierarchy. The grammar can determine, for example, whether terminals are alternatives of each other, strictly ordered, or repeatable.

The generic layout structure is ODA's major contribution to document specification, and allows layout to be specified in terms of 'page sets' which might consist, for example, of a special title page layout, followed by an arbitrary number of similar 'continuation pages'. The precise number of pages needed is determined during layout by reference to the content portions.

This results in the following conceptual scheme for an ODA specification (simplified from [ISO87a p.20]):



It is clear from this diagram that it is the content portions which may be subject to overlapping constraints from both the logical and the layout structures. Interestingly, this is the one area that Appelt et al. do not formally specify [Appelt88 p.102].

ODA suffers from the problems of many standards: its details become baroque, and in practice its implementations are arbitrary subsets of the full specification. Harke, for example, describes an ODA-like graphics editor [Harke87]. Joloboff voices some other criticisms [Joloboff86 p.122], including fears of implementation restrictions on the transmission of ODA documents. In addition, we can note that ODA does not appear to allow arbitrary sharing of content portions, except possibly via class descriptions. Nor does it appear to allow arbitrary nesting of text and graphics.

Both of these needs are addressed by the model presented in Chapter 8. This also has a more generic geometric model for the layout of text or graphics than Knuth's strictly nested box model.

5.5. Conclusions

An ideal surface model should incorporate nested text and graphics in the most domain-independent way. Thus we should expect the surface to cope with presenting complex documents, desktops, or any of an open range of applications, without imposing a visual or interactive style.

We conclude that the model for doing this provided by most window managers is inadequate in its geometric limitations and its graphical, as opposed to textual, bias. The standard graphical models of GKS and PHIGS, while being geometrically more general, have a similar graphical bias. In common with Page Description Languages like PostScript, these models are not suited to direct interactive manipulation, due mainly to the performance limitations imposed by their programmer interface (and, in the case of standard graphics, to an over-powered set of logical input devices). Textual and document models like SGML and ODA are more geared to document transmission than direct manipulation.

The various document or desktop publishing processors, in particular those with a WYSIWYG, i.e. direct, style, come closer to fulfilling the needs of an ideal surface model. However, in almost all cases these processors are designed simply for document production, and do not allow the imposition of application semantics. That is, they cannot be used as *interface* systems.

In attempting to design an ideal surface model, therefore, some preferences remain:

- A declarative model is to be preferred to a procedural model, since it hides the representation of the model and its presentation mapping. A declarative model can be persistent, and can be constructed by *direct* manipulation.

- The model should be domain- and style-independent. In particular, its structure should be orthogonal to its content.

- The structure of the model should allow for object composition (textual or graphical), for object replication by sharing, and for object classing by inheritance.

- The operations provided by the model on its state should be closed, and any state (i.e. any surface configuration) should be reachable just using the operations. In other words, applications should not need to access the display directly, and the operations should have no domain bias.

Chapter 6

A Formal Model for the Surface Medium

6.1. Introduction

The principle of Surface Interaction, and its *UMA* architecture, presupposes no particular semantics for the surface (that is, the user agent and the medium). The medium's model could consist of any set of values and operations on these, and the user agent could encapsulate any interpretation of user input.. This chapter concentrates on the design and specification of a formal model for the surface medium which provides semantics for *M* in the *UMA* architecture.

There are three major requirements in designing a model for a separate surface medium useful to a wide range of applications:

- The medium should be *encapsulated.* That is, it should prevent access to its internal representations by providing a well-defined set of operations to update them. Also its implementation, in particular its display mechanisms, should remain hidden. Such a medium is predictable and easily distributed from the application.

- The medium model should provide constructs that are style and domain independent, and yet common to many applications.

- The medium model should *factor* a substantial portion of the application task of constructing and maintaining the interface surface. Otherwise, there would be little point in separating this.

For example, part of the commonality which the medium constructs may possess is a *structure* - many applications may need to display objects composed of multiple sub-parts. In order to support *directness* (see Section 4.2.5), both the content and the structure of the objects must persist. Simply providing a set of primitive imaging *procedures* in the medium forces the application to maintain this structure and itself implement directness. Compositional part-whole structures, however, are very generic - they may be expressed as directed acyclic graphs like trees or hierarchies. The structure of the objects is therefore ideally maintained in the medium.

Similarly the *state* of the display objects, such as their images and extents, can be maintained in the medium. This has the benefit that incremental updates to the objects can be displayed without asking applications to redraw 'damaged' objects. Perhaps more importantly, it allows the application designer to construct interfaces in terms of discrete user-level objects, rather than by means of RasterOps and other device-level operations. This construction can also take place interactively, given a suitable editing agent. This gives the application designer and the user equal, or at least analogous, control over the surface, and reinforces the role of the medium as a channel of communication between user and application, or between users, or even between applications.

It is also not necessary that the medium, or even the surface, be *programmable*, as in NeWS [NeWS87a]. This power is properly in the domain of the application. Migrating it to the surface results only in the sort of gains in interactive response that come in any case using Surface Interaction. A programmable interface still needs the same software support as a monolithic application (toolkit libraries or classes, for example), with the extra burden that the application programmer must partition his code, possibly into different languages, such as PostScript and C in NeWS.

The requirements abov are partly met by window managers and toolkits. Both provide user-level objects (windows and menus etc.) that are (ideally) application independent and common. However, the previous Chapter has examined the limitations of these in practice.

Given the premise of a *visual* environment, and the two-dimensional limitations of current display technology, the requirements above determine some features of the design of the medium model. Clearly we are limited to text and graphics (and possibly sound and video, although we do not consider these specifically). At the

same time the requirements pose a problem: is it possible to define a construct that is both *objective* and *unbound* to any semantics?

The approach therefore taken here is to provide both a set of primitive objects that are little more than coordinate spaces, *and* a structuring mechanism by which they can be composed. Both the objects and the structure are *persistent*: their identity does not change although components of their state, like size and position, might. The projection of the structured objects onto the display is managed entirely within the medium, so that the application's access to them is via an identifier rather than to their internal representation, and corresponds closely to the user's visual access.

In this way, complex objects can be composed and given textual and graphical content. Because of the orthogonal structuring mechanism, the primitive objects are not subsumed into the complex object, but remain accessible and can be changed dynamically. In a strong sense, therefore, the *objects* that can be built on the surface in this design remain *unbound*.

This design has borrowed much from standard graphics, in particular in the hierarchical structuring of primitive objects. What it has added are the notions of persistence and encapsulation. This is in addition to the advantages of surface interaction, which plays no part in standard graphics.

Specific Requirements

We wish the medium to capture as much general textual and graphical capability as possible without becoming prescriptive in style or functionality. In the last analysis, there is no clear set of generic capabilities for either text or graphics. In text, for example, the surface could provide simple text appends or backtracks (as in a UNIX Shell command line), in-line editing, screen editing with copy and move commands, or even a full-blown document processing capability with embedded graphics, spellings checkers and hardcopy facilities. It is not clear at which point have we moved away from a generic interface capability into a specific application domain.

However, in the context of a medium consisting of discrete, structured objects in accordance with the broad requirements above, we can establish some more specific requirements:

• The objects should be able to represent both text and graphics at any level of granularity.

• There should be no geometric restrictions on their visual configuration. For example, objects should not be constrained or clipped to the area of objects higher in the structure, as happens in X windows.

• All properties of the objects and their structure should be modifiable dynamically, both by the user and the application, with consistent results.

• The user should potentially be able to access, by using the mouse, exactly the same set of objects as the application can by using an internal identifier.

The model given in this Chapter is an idealisation of a previous formal specification for an implemented system *Presenter*, which is described in Chapter 7.

6.2. The *Presenter* Model

This formal description in Z specifies the model for *Presenter*, an implementation of a surface. *Presenter* is described in Chapter 7. This specification is an idealisation of a cycle of previous formal specifications and implementations. In some details, the specification here differs from the implemented system, and these are outlined in Chapter 7. Also, for reasons of space, certain details of the implementation, for example its text handling capabilities, are specified only cursorily.

The specification seeks to express clearly and precisely the underlying structures and principles of the model, and, given the requirements, to distinguish between those features of the model in which design decisions have been necessary, and those that are unavoidably determined.

6.2.1. The Specification

The Z specification of the model is in three main parts:

• the first part specifies the fundamental objects, structures and properties of the model.

• the second part specifies the derivation of a surface presentation from the primitive objects and structures.

• the third part specifies the operations available to manipulate the model.

Note that we do not need to specify directly how the operations affect the presentation, since this is implicit in the mappings defined in the second and third parts.

6.3. Objects, Structures, and Properties

6.3.1. Fundamental Objects: *REGION*s

Minimally, the fundamental objects of the model can be represented simply as atomic nodes. Without making any assumptions about what properties these nodes have, or how they are structured or visualised, we declare a fundamental type:

$$[REGION]$$

We simply assume that an unbounded number of *REGION*s are available.

6.3.2. Fundamental Representation

We need to define the ultimate representation for *REGION*s on the surface. Since we assume in this thesis a visual domain, we think of the surface here as a display area such as a screen or a sheet of paper. In order to model presentation on such a surface, we declare a set of colours (without being specific as to what the colours are):

$$[COLOUR]$$

and define a set of points on the real plane:

$$POINT == \mathbb{R} \times \mathbb{R}$$

The number of *POINT*s so defined is infinite, both in extent and resolution. At this level it is thus possible to abstract away from questions of display resolution.

We can then model any image on this surface as a mapping between *POINT*s and *COLOUR*s:

$$IMAGE == POINT \pfun COLOUR$$

Clearly, each *POINT* will have only one *COLOUR*, so *IMAGE* is a function. However, not all *POINT*s may have a *COLOUR*, so *IMAGE* is partial, and there may also be a number of *POINT*s which have the same *COLOUR*, and so *IMAGE* is not injective.

6.3.3. Fundamental Structure: Ordered Tree

We begin by defining a fundamental structure for *REGION*s in the model. This is an *ordered tree*:

ORDERED_TREE

$tree: REGION \nrightarrow iseq\ REGION$

$parent \in REGION \nrightarrow REGION$

$id\ [REGION] \cap parent^{+} = \varnothing$

$\bigcup \{s: ran\ tree \bullet ran\ s\} \subset dom\ tree$

where

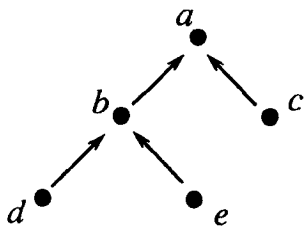
$parent = \{p, c: REGION \mid c \in ran\ (tree\ p) \bullet c \mapsto p\}$

The function *tree* is partial, and so represents one possible ordered tree out of an unbounded set of ordered trees of all sizes. *ORDERED_TREE* itself represents this set. The base of this schema is open since we wish to continue this definition into the next schema (this is a non-standard use of Z). However, the function *tree*, as defined by the predicate here, is all we need to specify the structure.

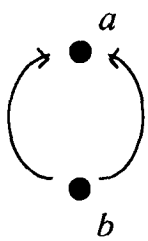
The function *tree* defines an ordered tree by mapping (parent) *REGION*s to injective sequences of (child) *REGION*s. Clearly, the child *REGION*s may themselves point to further child sequences, so that unbounded structures can be built up. For example, α is a *tree*:

$$\alpha = \{a \mapsto \langle b, c \rangle, b \mapsto \langle d, e \rangle, c \mapsto \langle\rangle, d \mapsto \langle\rangle, e \mapsto \langle\rangle\}$$

and α can be represented (using $\rightarrow$ to indicate parenthood, and left-right ordering to indicate the sequence of children):



Since the child *REGION*s are ordered by their sequence, any connected structures are also ordered (or *oriented*, in graph terminology). No child may recur in its sequence, since these are injective (*iseq*). Minimally, this excludes structures in which there is more than one path between the same two *REGION*s:



which would be represented (if it were allowed):

$$\{a \mapsto \langle b, b \rangle\}$$

The integrity of the tree structure is ensured by two conditions which are expressed in the schema predicate:

- Child *REGION*s have only one parent (the mapping between children and parents is a *function*)

- There are no cycles in the structures (a *REGION* cannot be its own ancestor, i.e. the identity mapping between *REGION*s is not represented in $parent^+$ - the closure of the *parent* function)

In addition, we make the restriction that all *REGION*s in the sequences in the range of *tree* are also represented in its domain. In other words, the domain of *tree* contains all the *REGION*s in the structure. This means that our representation for leaf *REGION*s (i.e. *REGION*s which have no children) is a mapping that points to an

empty sequence, rather than one where leaves are simply not represented in the domain of *tree*. So the representation for the small structure:

is:

$$\{a \mapsto \langle b \rangle, b \mapsto \langle\rangle\}$$

rather than simply:

$$\{a \mapsto \langle b \rangle\}$$

This is necessary, since minimally we wish to be able to represent a tree consisting of a single *REGION*:

for which the function *tree* would be:

$$\{a \mapsto \langle\rangle\}$$

Thus all *REGION*s in a particular *tree* have a sequence of children, although this sequence (in the case of leaf *REGION*s) may be empty. All leaf *REGION*s have the same empty child sequence, and so *tree* is not injective. Note also that an *ORDERED_TREE* may contain a number of disjoint trees.

In any *ORDERED_TREE*, we wish to distinguish a *root REGION*:

$$\begin{array}{|l} root: REGION \\ \hline root \in dom\ tree \\ root \notin \bigcup \{s: ran\ tree \bullet ran\ s\} \end{array}$$

The *root* is always on the *tree*, but is never part of the sequences of child *REGION*s in the range of *tree*.

6.3.4. Basic Relations

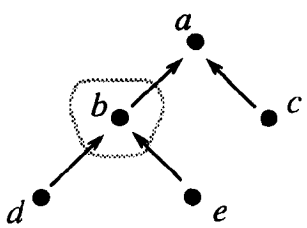
We conclude the definition of an *ORDERED_TREE* with two useful mappings derivable from a *tree*:

$$
\begin{array}{|l}
parent: REGION \nrightarrow REGION \\
family: REGION \leftrightarrow REGION \\
\hline
\forall\, r: REGION \bullet \\
\quad q = parent\ r \Leftrightarrow r \in ran\ (tree\ q) \\
\quad q \in family\ (\!|\{r\}|\!) \Leftrightarrow r \in parent^{*}\ (\!|\{q\}|\!) \vee q = r \\
\hline
\end{array}
$$

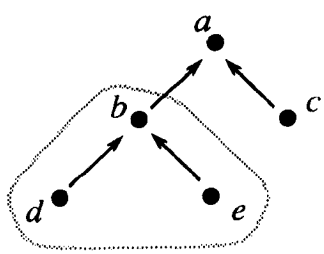
parent is a function which maps *REGION*s to their parent. It is repeated here from the first part of the schema (with a different definition) to emphasise that it is simply derived from the *tree* function.

family is a relation mapping *REGION*s to every member of their sub-tree, including themselves. A *REGION q* is in the family of a *REGION r* if *r* is an ancestor (the closure of *parent*) of *q or* (in order to include isolated *REGION*s, which cannot be represented in *parent*) if *q* is the same as *r*.

Using the *tree* α, applications of these mappings can be illustrated:



parent e = b



family $(\!|\{b\}|\!) = \{b, d, e\}$

6.3.5. Fundamental Properties

In order to exploit the *REGION* tree structure, we load the *REGION*s with *properties* by providing mappings between them and other types. These types express four fundamental properties of *REGION*s as medium objects: textual and graphical content, geometry, visualisation, and behaviour.

We model properties as tuples of named attribute values, and define particular tuples using schemas:

Tuple

$attribute_1$: $values_1$
$attribute_2$: $values_2$
...
$attribute_n$: $values_n$

Tuple thus stands for the set of all possible combinations of values for its attributes. We adopt the $\Downarrow$ notation we used in Chapter 3 for binding particular values to the named attributes in tuples:

$$t = (attribute_1 \Downarrow v_1, attribute_2 \Downarrow v_2, \ldots attribute_n \Downarrow v_n)$$

Individual attributes can thus be accessed using Z's dot notation:

$$t.attribute_1 = v_1$$

We also assume that any attribute may be *undefined*, and so have the special value $\perp$ (*bottom*) in addition to its declared values. This will allow us to deal generically with the propagation of attribute values. We thus use $\perp$ as if it had meaning, which is not *strict*. In implementation of course we would use some special null value which is well defined. We use $\perp$ in preference to *0* or *false* because of the connotations of these values.

It is thus possible to model properties by functions from *REGION*s to tuples *T* of any size:

$$PROPERTIES\ [T] == REGION \nrightarrow T$$

We use tuples for three reasons:

- Some attribute values may conflict. For example, a *REGION* cannot be both transparent and opaque. Clearly, an attribute in a schema can take on only one value at a time, so we can define attributes to have values with disjoint meanings, and be sure that no conflict will arise.

• Some attributes may be orthogonal and so may be set together. For example, a *REGION* may both be transparent *and* inverted. Orthogonal values like this can be ascribed to different attributes.

• There may be many disjoint sets of attribute values. Using tuples of attributes is a convenient packaging, just as records are in programming. The only alternative is separate mappings from *REGION*s to values for each property.

6.3.6. Content

Graphical Content

We are not concerned here to specify a rich set of imaging primitives, such as may be provided by a graphics language like PostScript. The intention in this design is to give *REGION*s the flexibility to be used as primitive constructs in themselves, rather than (as in a window manager) simply as canvasses or drawables for other primitives.

Nevertheless, in practice there is a need for some minimal set of primitives. In fact, the implemented system *Presenter* gets by with only a single line-drawing primitive, and that is all we shall define here. A line is defined by two end points:

$$LINE == POINT \times POINT$$

Graphical content will consist of a sequence of *LINE*s.

Textual Content

Although the domain of text presentation and manipulation is rich, at base the text contained in a *REGION* can be modelled as a sequence of characters. We use a basic type

$$[CHAR]$$

without further elaboration, but with obvious intention.

Text or Graphics

A *REGION* can contain either text or graphics but not both. This is logical, since the operations on the two types are quite distinct. The *composition* of text and graphics is achieved at a higher level, by the structure of the tree itself.

CONTENT

$$graphics: seq\ LINE$$
$$text: seq\ CHAR$$

$$\neg\ (graphics \neq \bot \wedge text \neq \bot)$$

CONTENT thus describes the state space in which either the component *graphics* is defined, or the component *text* is defined, but not both. We use $\bot$ for *undefined* or *unused*. *CONTENT* can thus be used as a union type, and its components accessed by the conventional dot notation. This formulation allows us to query the values to find out if they are *graphics* or *text*.

Leaves Only

Only leaf *REGION*s can have displayable *CONTENT*. This is an important feature of the model, and distinguishes it from standard hierarchical window systems, in which interior nodes may also be visible windows. However, it is similar to procedural graphics systems, in which the call structure forms a hierarchy with primitives at the leaves.

Contents

$$ORDERED\_TREE$$
$$content: REGION \nrightarrow CONTENT$$

$$\forall r: dom\ content \bullet tree\ r = \langle\rangle$$

content is a partial mapping ($\nrightarrow$), as not all *REGION*s have *CONTENT*.

Restricting content to the leaves of the tree makes content orthogonal to the *REGION* structure. We can thereby maintain the property that *any CONTENT REGION* can be manipulated independently of the others.

6.3.7. Geometric Properties

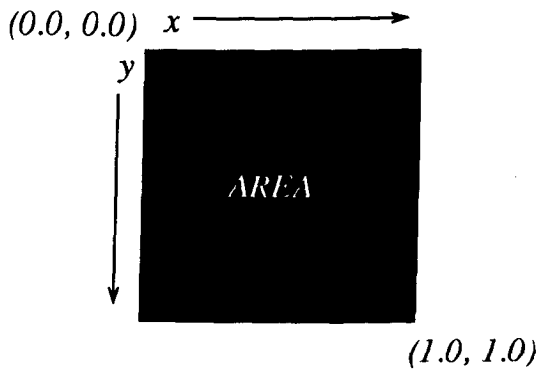
The tree structure says nothing about the eventual representation of *REGION*s and their *CONTENT*s as *IMAGE*s on the surface.

Area

In order to delimit the graphical space of a *REGION*, we give it a particular area: a set of *POINT*s on a unit square with one corner at the origin:

$$AREA == [0, 1] \times [0, 1]$$

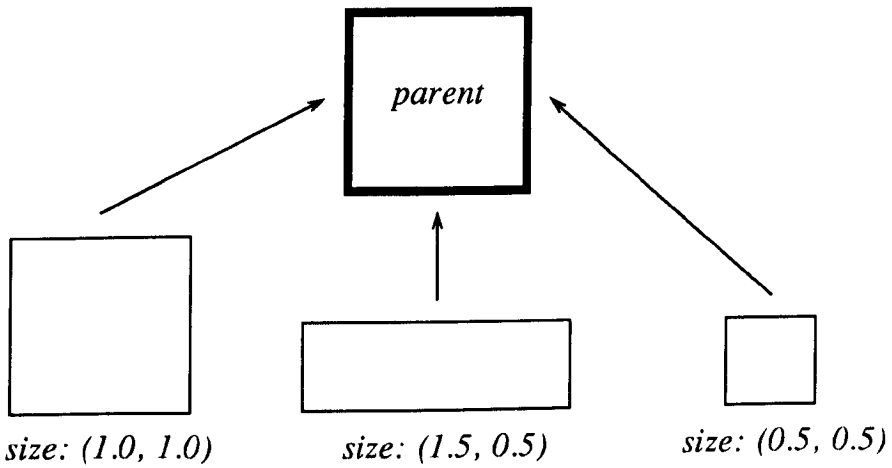
In this way we think of *REGION*s as being rectangular coordinate spaces of side *1*. The orientation of *REGION*s on the physical screen is entirely a matter of design choice: *Presenter* implements *REGION*s with their origin at the top left:



Size

The *AREA* of a *REGION* has meaning only with respect to some coordinate space. By default, we consider the *AREA* of the *root REGION* to be the same size as the display surface. However, in order not to introduce any device dependencies, the size of child *REGION*s is specified as a real proportion of their parent in both dimensions, rather than in terms of absolute coordinate lengths. Thus a *REGION* of

size *(1.0, 1.0)* is the same size as its parent. This relative sizing can be illustrated graphically:



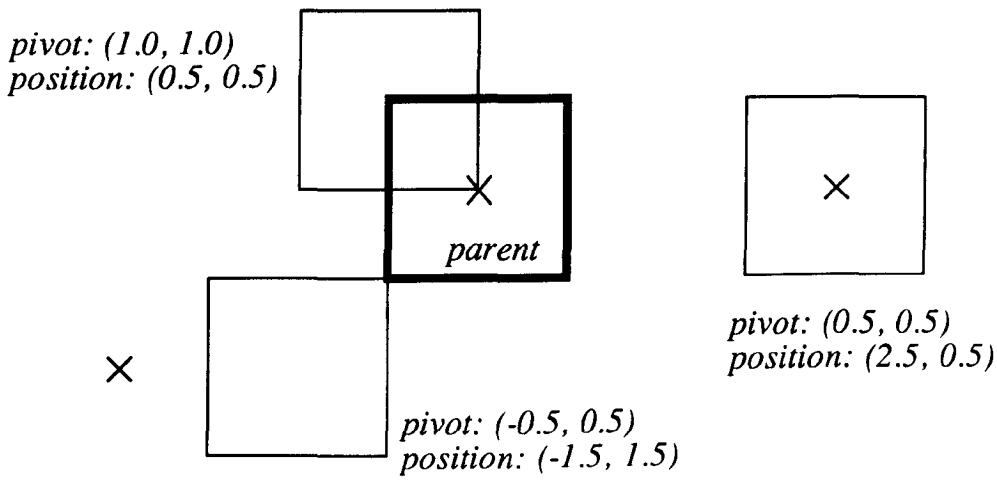
Position

The position of a *REGION* is similarly specified with respect to its parent. In order to fix the position of a *REGION*, two things are needed:

- a reference point within the child (its *pivot*).
- a *position* in the parent at which the pivot of the child will be sited.

Both of these are considered properties of the *child*, and all *REGION*s have a pivot and a position. The position and pivot of a *REGION* are real number pairs. *pivot* expresses the distance of the pivot point from the origin of the *REGION*, proportional to its width or height. Thus a pivot of *(0.5, 0.5)* is in the centre of the *REGION*. *position* expresses the distance of the pivot point from the origin of the *REGION*'s *parent*, proportional to the width or height of the parent. Thus a *REGION* with *position (1.0, 1.0)* has its pivot point at the lower right corner of its parent. The

position of *root* is taken to coincide with the display surface available. This relative positioning can be illustrated graphically:



Rotation

For completeness, we allow *REGION*s to be rotated about their pivot (although this is not implemented in *Presenter*). The angle of rotation can be expressed in some appropriate units, for example degrees.

Geometry

The total geometry of *REGION*s can be expressed as a tuple:

GEOMETRY

$size, pivot, position: \mathbb{R} \times \mathbb{R}$
$rotation: \mathbb{R}$

In practice we will never allow any of these properties to be undefined. That is, *REGION*s will always have a size, position etc. At the very least, therefore, we will need a default geometry to assign to *REGION*s on creation:

$$\begin{aligned} DEFGEOM == (&size \Downarrow (1.0, 1.0), \\ &pivot \Downarrow (0.5, 0.5), \\ &position \Downarrow (0.5, 0.5), \\ &rotation \Downarrow 0.0) \end{aligned}$$

All *REGION*s have a geometry:

Geometry

ORDERED_TREE
geometry: REGION → GEOMETRY

The informal diagrammatic interpretations of geometry above are specified formally when we derive the surface presentation of the medium in Section 6.4. Chapter 5 also compares this geometric scheme with those of PostScript, GKS, and PHIGS.

6.3.8. Visualisation

The surface presentation of the *CONTENT* of *REGION*s is clearly dependent on the sizes and positions of the containing *REGION*s. We also allow other aspects of the visualisation of *REGION*s, for example transparency, to be determined by means of a set of attributes. They are expressed as a tuple:

VISUALISATION

trans: transand | transxor | transor
invert, highlit: {true}

The definition of this tuple will be completed below. The only *defined* value for *invert* and *highlit* is *true:* we assume they may also be undefined (⊥), which will be interpreted as the opposite in each case. Similarly, a *REGION* whose *trans* attribute is undefined is taken to be opaque.

Clipping

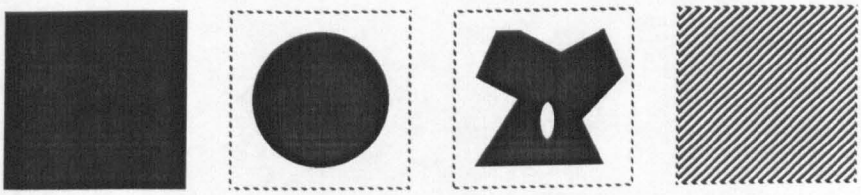
There are two aspects to clipping.

- We may wish to restrict the *IMAGE* of a *REGION* to some non-rectangular shape, for example a box with rounded corners.

- We may wish to restrict the *IMAGE*s of a set of *REGION*s to the area defined by some shape, for example to provide the effect of windowing onto a larger space containing *REGION*s.

We can provide a very general solution to these requirements by giving each *REGION* a *mask*. If the *REGION* is a leaf, then its mask clips its *IMAGE*. On the other hand, if the *REGION* is not a leaf, then its mask clips the *IMAGE*s of all its descendant leaves. It is an important feature of the model that we do not allow the mask to be affected by the geometric transformations of its children. This is explained in Section 5.2.6. The precise functionality is specified in Section 6.4.4.

The mask is modelled simply as a subset of *POINT*. For example:



If the mask equals *POINT*, i.e. the whole plane, then no clipping takes place. On the other hand, if the mask is empty, then the *IMAGE*s are effectively hidden. We leave out of consideration here *how* such masks are generated, whether by a filled path of line segments or by some other means.

The *VISUALISATION* tuple is thus completed with an attribute for the mask:

$$mask: \mathbb{P}\, POINT$$

Visualisation Properties

Visualisation attributes determine how the final contents of *REGION*s are presented as *IMAGE*s on the surface. By definition we assume that all the attributes are mutually orthogonal, so that an inverted *IMAGE* may be transparent or not. We also need a default *VISUALISATION*:

$$DEFVIS == (trans \Downarrow \bot, invert \Downarrow \bot, highlit \Downarrow \bot, mask \Downarrow POINT)$$

By default, therefore, *REGION*s will be opaque, uninverted, unhighlit, and their masks will be the whole plane and therefore clip nothing.

All *REGION*s possess a *VISUALISATION*:

Visualisation

ORDERED_TREE
visualise: REGION → VISUALISATION

6.3.9. Behaviour

The surface performance of *REGION*s under operations like changing size and position may be constrained by a set of attributes. Thus a *REGION* may be constrained to move only horizontally or only vertically. Since these attributes affect the dynamics of *REGION* manipulation, we collectively call them *behaviour*. The behaviour of a *REGION* also includes how it responds to changes in its geometric environment, to user input, and how it reports user input to its owning application. The tuple of behaviour attributes can be defined:

BEHAVIOUR

movable, sizable: horizontal | vertical | both
scale, group, permeable, selectable: {true}
reportup, reportdown, reportdrag, reportCH, reportCR: {true}

By default, *REGION*s have the following *BEHAVIOUR*:

$$\begin{aligned} DEFBEHAVE == (&movable \Downarrow both, sizable \Downarrow both,\\ &scale \Downarrow true, group \Downarrow true,\\ &permeable \Downarrow \bot, selectable \Downarrow true,\\ &reportup \Downarrow true, reportdown \Downarrow true, reportdrag \Downarrow \bot,\\ &reportCH \Downarrow \bot, reportCR \Downarrow true) \end{aligned}$$

The precise interpretation of these attributes will be defined later in Section 6.5. All *REGION*s have a *BEHAVIOUR*:

Behaviour

ORDERED_TREE
behave: REGION → BEHAVIOUR

6.3.10. The Core Model

The model of the medium can be brought together into a single schema:

MEDIUM

ORDERED_TREE
Contents
Geometry
Visualisation
Behaviour

MEDIUM specifies the core medium model completely, and includes all the information that is needed to derive a surface presentation and constrain its behaviour. To complete the picture of the model, however, two further specifications are required: a definition of precisely how a surface presentation is generated from each state of this core model; and a definition of the operations available to change the state.

6.4. Surface Presentation

It is not sufficient simply to outline the structure and properties of the model. To be convincing, we must also specify precisely how the information contained in *MEDIUM* is presented on a surface. In this section we describe how the primitive structures and their properties determine this surface presentation.

Since we are not concerned here to build up a more complex state space for the model, but simply to define some functions over *MEDIUM*, we open this model up globally by including it in an axiomatic schema [Spivey89 p.143]:

MEDIUM

We can then define the presentation functions globally in the same way.

6.4.1. Projecting the Tree

As specified, the geometry of a *REGION* is interpreted with respect to its parent *REGION*. Similarly, the effect of the visualisation attributes of *REGION*s will be

propagated from parents to children. Clearly the *paths* of *REGION*s from the root to the leaves are important in determining the presentation of the *IMAGE*s.

Since the tree is ordered, and by design we wish to exploit this order as display layering, the *sequence* of paths will also be significant.

We therefore define a function *pathseq* to produce a sequence of paths from some sequence of projection roots. (In this and later recursive definitions, we employ a pattern matching style for concision):

$$pathseq: seq\ REGION \rightarrow iseq\ (iseq\ REGION)$$

$$pathseq = pseq\ \langle\rangle$$

$$pseq = \lambda\ p, (r:sr): seq\ REGION \bullet$$

$$pseq\ p\ \langle\rangle = \langle\rangle$$

$$pseq\ p\ (r:sr) = \langle p \frown \langle r \rangle \rangle \frown pseq\ p\ sr \qquad \textit{if}\ tree\ r = \langle\rangle$$

$$pseq\ p\ (r:sr) = (pseq\ p \frown \langle r \rangle\ (tree\ r)) \frown pseq\ p\ sr \qquad \textit{otherwise}$$

The *pathseq* is defined by first defining a function *pseq*, which accumulates the paths (in *p*), given a sequence of projection roots (in *(r:sr)* where *r* is the head of the sequence, and *sr* the tail). The function shows three cases:

- The sequence of projection roots is empty ($r:sr = \langle\rangle$), else
- The head *r* of the sequence is a leaf *REGION* ($tree\ r = \langle\rangle$), else
- The head *r* of the sequence has children ($tree\ r \neq \langle\rangle$).

pathseq is generated by partially applying *pseq* to an initially empty path. Using the *ORDERED_TREE* α above as an example:

$$pathseq\ \langle a \rangle = \langle\langle a, b, d\rangle, \langle a, b, e\rangle, \langle a, c\rangle\rangle$$

Note that each of the paths produced are injective, since *ORDERED_TREE*s exclude cycles. The sequence of paths is also injective, that is, no path recurs. This is a consequence of the fact that the child sequences in *ORDERED_TREE*s are injective: no arc can be duplicated, and so no path can be duplicated. This is a useful

property since as a result it will be possible to identify a surface selection uniquely by identifying its path, as well as by its numerical position in the sequence of paths.

Although *pathseq* is defined as applying to sequences of *REGION*s, in practice we will only need to apply it to single *REGION*s which form the root of subtrees. For example, we can define a sequence of paths for the whole *tree*, by projecting from *root*:

$$paths: iseq\ (iseq\ REGION)$$

$$paths = pathseq\ \langle root \rangle$$

The *paths* produced have the *root REGION* at their head, and a leaf *REGION* as their last element. In addition, the ordering of the tree allows us to order all the projection paths. In the model this order is interpreted as screen layering.

6.4.2. Imaging

The fundamental property of *REGION* trees which allows them to have a surface presentation is that the graphical or textual *CONTENT* of leaf *REGION*s may generate an *IMAGE* which is coextensive with the *AREA* of the *REGION*. We assume, without defining it further, a function *image*:

$$image: REGION \nrightarrow IMAGE$$

$$\forall r: dom\ image \bullet tree\ r = \langle\rangle$$
$$\forall i: ran\ image \bullet dom\ i = AREA$$

We therefore abstract away from the mechanics of imaging, either in graphics or text. This is not because they are trivial, but because they are too complex and implementation-oriented for this level of description.

6.4.3. Geometric Transformations

We define the way in which *IMAGE*s are transformed by the geometric properties of the *REGION* tree. We first define operations to generate matrix

representations for the basic transformations of translation (T), scaling (S), and rotation (R):

$$T, S: (\mathbb{R} \times \mathbb{R}) \rightarrow MATRIX$$
$$R: \mathbb{R} \rightarrow MATRIX$$

$$\forall x, y, \theta: \mathbb{R} \bullet$$

$$T(x, y) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{bmatrix}$$

$$S(x, y) = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

We here assume a type *MATRIX*, of 3 by 3 matrices, and its associated arithmetic. *MATRIX* is an (injective) mapping $POINT \rightarrowtail POINT$ (that is, we assume we can invert it). We will also require an identity *MATRIX*:

$$IDMAT == \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

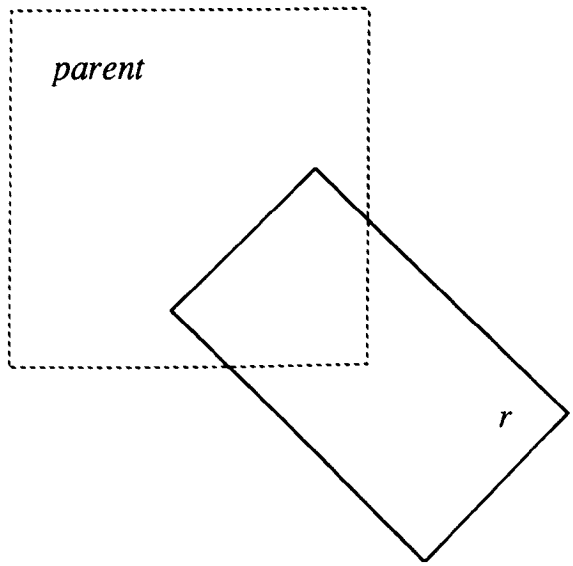
Next we define a function (M) to compose a matrix which expresses the size, position, and orientation of a *REGION* in the coordinate space of its parent, expressed as a transformation of the parent space:

$$M: REGION \rightarrow MATRIX$$

$$M = \lambda r: REGION \bullet$$
$$T(-G.pivot) \cdot R(G.rotate) \cdot S(G.size) \cdot T(G.position)$$

where
$$G = geometry\ r$$

(We assume here that (-) negates both elements of the pair). For example, if *REGION r* has *position (1.0, 1.0)*, *size (1.0, 0.5)*, *pivot (0.5, 0.5)*, and *rotate 45*, then its shape with respect to its parent (*M r*) can be illustrated:

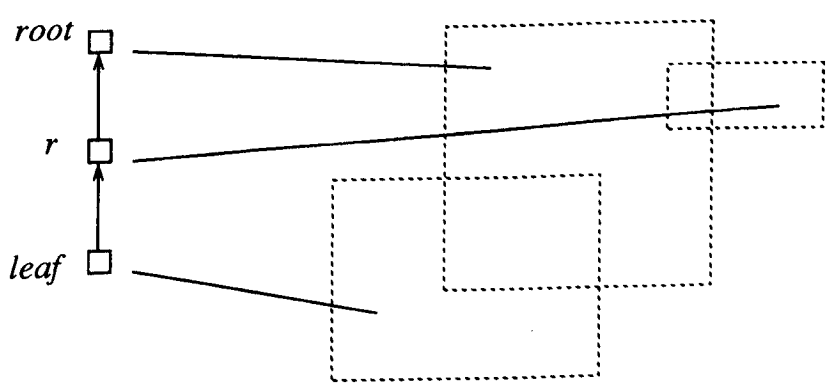


In order to ascertain the shape of a leaf *REGION* against the *root REGION* (that is, on the display surface), *all* matrices in the path of the leaf *REGION* must be composed in sequence. We define a function *C* to do this. *C* takes a path of *REGION*s and returns the composed *MATRIX*:

$$C: seq\ REGION \rightarrow MATRIX$$

$$C\ \langle\rangle = IDMAT$$
$$C\ (r:sr) = (C\ sr) \cdot M\ r$$

This can be illustrated for a short path of *REGION*s:



The transformation between the *root* and *leaf REGION*s on this path can thus be expressed:

$$C\ \langle root, r, leaf\rangle = IDMAT \cdot M\ leaf \cdot M\ r \cdot M\ root$$

Note that the matrix representation allows quite arbitrary transformation between parent and child: there are no restrictions on the size, position or orientation of the child with respect to the parent. We wish to preserve this geometric freedom throughout the model.

Finally, we must show how *POINT*s and *IMAGE*s, are transformed by the composed matrices. We define two operations: ☞, which converts a *MATRIX* into a mapping between sets of *POINT*s, and ☛, which converts a *MATRIX* into a mapping between *IMAGE*s:

$$\_ \text{☞} \_ : MATRIX \rightarrow \mathbb{P}\ POINT \rightarrow \mathbb{P}\ POINT$$
$$\_ \text{☛} \_ : MATRIX \rightarrow IMAGE \rightarrow IMAGE$$

$$\forall\ m: MATRIX;\ ps: \mathbb{P}\ POINT;\ i: IMAGE \bullet$$
$$m \text{☞} ps = \{p: ps \bullet p \cdot m\}$$
$$m \text{☛} i = \{p: dom\ i \bullet p \cdot m \mapsto i\ p\}$$

(We overload the matrix operator (·) here slightly by assuming that it converts *POINT*s to homogeneous coordinates of the form *[x y 1]*). ☛ transforms the domain of the *IMAGE*, whilst leaving the *COLOUR*s unchanged. We use *POINT*, rather than its restriction *AREA*, because child *REGION*s, and hence their *IMAGE*s, may be larger than root.

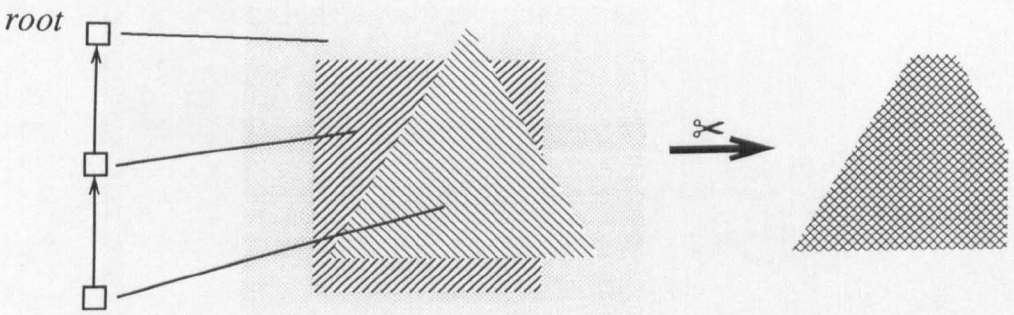
6.4.4. Clipping

In the model, every *REGION* may have a mask. Clearly, the masks will only affect the presentation of the *IMAGE*s of leaf *REGION*s. To determine the final clip-

ping mask for a leaf *REGION*, therefore, we take the intersection of all the masks on the path of the *REGION*:

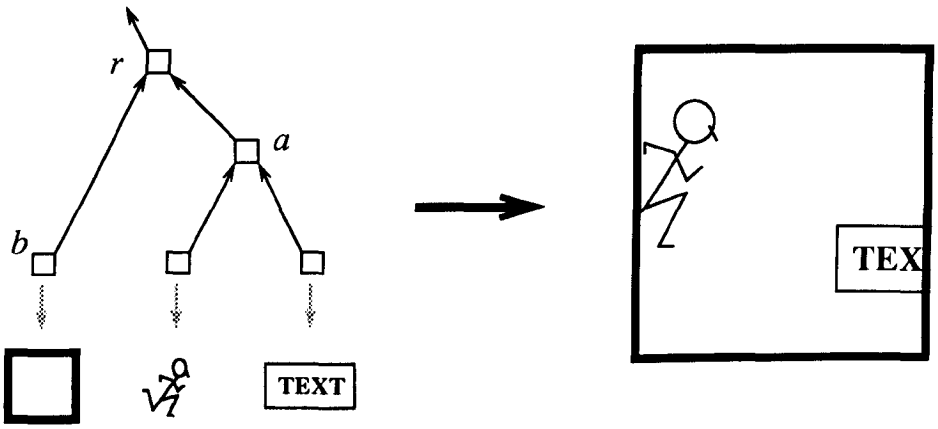
$$
\begin{array}{|l}
\hline
\text{✂}: \mathit{seq}\ \mathit{REGION} \rightarrow \mathbb{P}\ \mathit{POINT} \\
\hline
\text{✂} = \text{✂}\!\!\leftarrow\ \mathit{IDMAT} \\
\\
\text{✂}\!\!\leftarrow = \lambda\ \mathit{CTM}: \mathit{MATRIX};\ (r{:}sr): \mathit{seq}\ \mathit{REGION} \bullet \\
\quad \text{✂}\!\!\leftarrow\ \mathit{CTM}\ \langle\rangle = \mathit{POINT} \\
\quad \text{✂}\!\!\leftarrow\ \mathit{CTM}\ (r{:}sr) = (T \text{☞}\ \mathit{mask}\ r) \cap (\text{✂}\!\!\leftarrow\ T\ sr) \\
\\
\textbf{\textit{where}} \\
\quad T = \mathit{CTM}\ \cdot M\ r \\
\hline
\end{array}
$$

Again, ✂ is defined by defining an auxiliary function ✂←, and partially evaluating this with the identity matrix *IDMAT*. We can illustrate some clipping masks, and their intersection using ✂:



It is important to note in this definition that the mask is transformed (☞) *only* by the *MATRIX* formed from *REGION*s *above* it on the path (*T*). Once this has been done, the mask is not transformed further. That is, notionally, the mask has its own

orientation which is not affected by the orientation of the children of its *REGION*. This capability can be used, for example, to provide clipping windows:



Here, *b* is the same size and at the same position as *r*, and

$$mask\ r = AREA$$

and so *r* clips all descendant *REGION*s to its area. The position or size of the clipping mask is unaffected by intermediate *REGION*s, such as *a*, so that the clipping of each leaf *REGION* coincides.

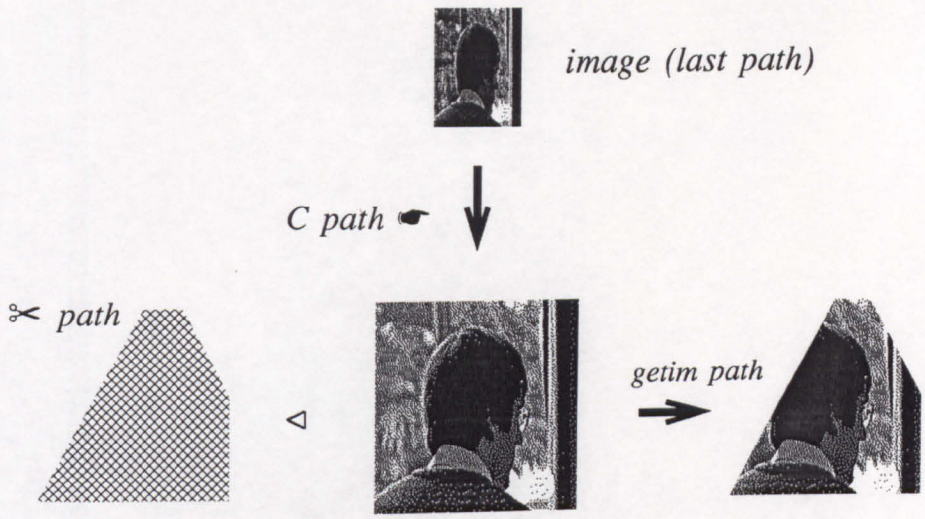
6.4.5. Combining Transformation and Clipping

We next need to combine *IMAGE* transformation with clipping. *getim* transforms the *image* of a leaf *REGION* by the composed matrix (*C*) of its path, and clips this against the combined clipping mask (✂) of the path:

$$getim: seq\ REGION \rightarrow IMAGE$$

$$getim = \lambda\ path: seq\ REGION \bullet$$
$$(✂\ path) \lhd (C\ path ☛ image\ (last\ path))$$

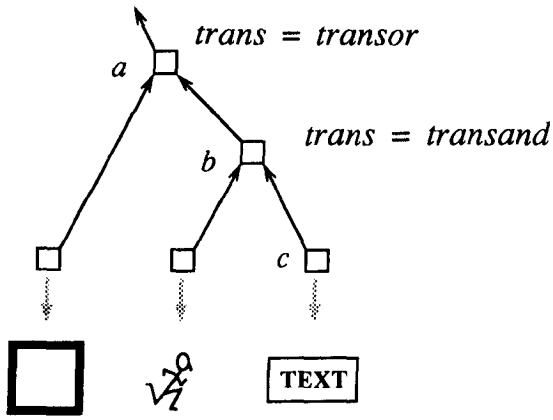
last path is clearly the leaf *REGION* which has an *IMAGE*. The clipping is performed by restricting the domain (◁) of the *IMAGE* to just the *POINT*s in the mask. Graphically:



6.4.6. Propagation of Attributes

The various *REGION* properties of content, geometry, visualisation and behaviour are determined by tuples of attributes. With all properties except content, we allow any *REGION* to possess attributes. By design, we do not allow properties to be *inherited*, in the sense that a *REGION* is not thought to possess some union of the attributes of its ancestors. However, we want the *effects* of attributes to be *propagated* down the tree, such that the *IMAGE*s of descendant leaf *REGION*s of a transparent *REGION*, for example, will be presented as transparent. In this way, the presentation of groups of *REGION*s can be modified by changing the properties of a common ancestor.

The design question that arises is, how are attributes propagated, such that they affect the presentation of the leaf *REGION*s? For example, in the following tree, which *IMAGE*s should be *transor*?:



If an *IMAGE* cannot be both *transand* and *transor*, then some scheme for resolving this conflict is necessary. The most general information we have available is the ordering of *REGION*s on the paths, and it makes sense to use this, rather than impose some external priorities (for example, that *transor* always takes precedence over *transand*).

In addition, we need to decide just how the path is exploited to prioritise conflicting attributes. We could decide that attributes higher on the path have priority over lower attributes. However, this would mean that lower settings would always be completely obscured by the higher settings. Using the above example, this strategy would result in *all* the *IMAGE*s being transor. We therefore make the obvious design decision that attributes *lower* on the path have priority over those higher on the path.

In order to define this generically, we assume a syntactic function *components* which given a particular tuple returns the set of its component attribute names. The fundamental mechanism of propagation we can therefore define for any tuple T:

$[T]$

$$\_ \blacksquare \_ : (T \times T) \rightarrow T$$

$$\forall\, t1, t2 : T;\ c : components\ T \bullet$$
$$t2.c = \bot \Rightarrow (t1 \blacksquare t2).c = t1.c$$
$$t2.c \neq \bot \Rightarrow (t1 \blacksquare t2).c = t2.c$$

Thus the function $\blacksquare$ takes two tuples, and returns a new tuple of the same type which, for each component attribute, inherits the value in the second tuple if it is defined there, and inherits the value of the component in the first tuple if it is undefined in the second tuple. The function is generic over any tuple (that is, the tuple can consist of any group of attributes).

Propagation can therefore be defined on paths of *REGION*s, so long as a default tuple and a *PROPERTIES* mapping between *REGION*s and tuples (the attribute settings for each *REGION*) are supplied:

$[T]$

$$propagate : T \rightarrow PROPERTIES\ [T] \rightarrow seq\ REGION \rightarrow T$$

$$propagate = \lambda\, default : T;\ prop : PROPERTIES\ [T];\ (sr:r) : seq\ REGION \bullet$$
$$propagate\ default\ prop\ \langle\rangle = default$$
$$propagate\ default\ prop\ (sr:r) = (propagate\ default\ prop\ sr) \blacksquare prop\ r$$

Thus *propagate* folds the $\blacksquare$ function along a sequence of tuples of attributes, starting with a default tuple of attributes. The sequence of tuples is generated when the *PROPERTIES* mapping *prop* is applied to each *REGION* in a path.

For example, taking the tree above, the *PROPERTIES* mapping may be:

$$prop = \{a \mapsto (trans \Downarrow transor),\ b \mapsto (trans \Downarrow transand),\ c \mapsto (trans \Downarrow \bot)\}$$

and

default = (trans ⇓ ⊥)

(ignoring any other attributes). The presentation of the **TEXT** *IMAGE* is then determined by propagation along its path *<a, b, c>*:

propagate default prop <a, b, c>
= ((default ■ prop a) ■ prop b) ■ prop c
= (((trans ⇓⊥) ■ (trans ⇓ transor)) ■ (trans ⇓ transand)) ■ (trans ⇓⊥)
= (trans ⇓ transand)

Thus the *IMAGE* would be transparent in AND mode.

6.4.7. Visualisation Attributes

VISUALISATION attributes, which affect the presentation of the *IMAGE*s, are propagated down the projection path in the same way. We can therefore partially instantiate *propagate* to specialise it for *VISUALISATION* attributes:

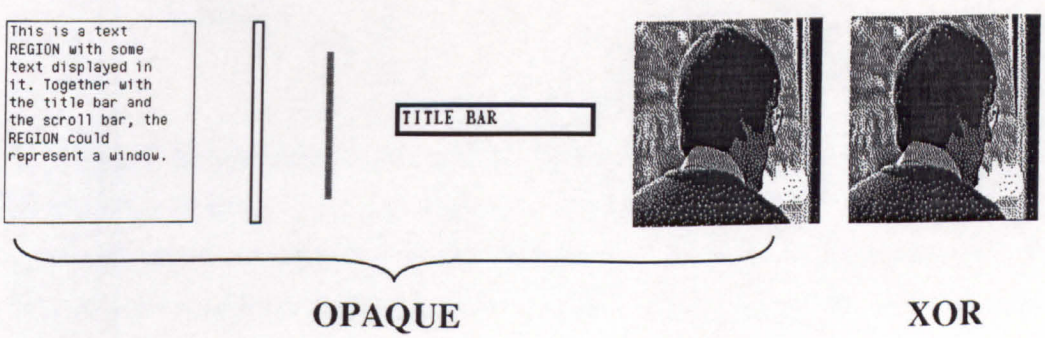
getvis: seq REGION → VISUALISATION

getvis = propagate DEFVIS visualise

6.4.8. Presentation

The overall presentation of the *REGION* tree as a single *IMAGE* on the display surface is accomplished, conceptually, in two steps:

- A sequence of transformed, clipped *IMAGE*s, paired with their propagated *VISUALISATION* attributes, is generated from the *pathseq* using *getim* and *getvis*. Such a sequence can be illustrated:



The *IMAGE*s at the front of this sequence all have their *trans VISUALISATION* attribute undefined (i.e. they are opaque). The last is transparent in XOR mode.

- Secondly, the sequence of *IMAGE*s is combined into a single *IMAGE* using compositing operators determined by the *VISUALISATION* of each *IMAGE*.

By design we instantiate the ordering of the sequence of *IMAGE*s as surface layering, that is, we mean *IMAGE*s later in the sequence to overlay overlapping *IMAGE*s earlier in the sequence. In order to present this, we need to define the ways in which *IMAGE*s with different *VISUALISATION* attributes combine.

In the case of opaque *IMAGE*s, or transparent monochrome *IMAGE*s, the combination operator is well defined and will correspond to a RasterOp mode such as simple overwrite or 'AND'ing. However, in the case of transparent colour *IMAGE*s, the combination is more contentious (see Section 5.2.1). For simplicity, we simply assume an appropriate colour combination operator, determined by the value of the *transparent* component of the *VISUALISATION* tuple, and give it the symbol ✪. We assume also that ✪ inverts or otherwise highlights the *IMAGE* appropriately if the *invert* or *highlit* attributes are true.

We define a function *present* which generates a single *IMAGE* from a sequence of paths, when provided with an appropriate background *IMAGE*:

$$
\begin{array}{l}
present: IMAGE \rightarrow iseq\ (iseq\ REGION) \rightarrow IMAGE \\
\hline
present\ s\ \langle\rangle = s \\
present\ s\ (p:sp) = \\
\quad present\ (s \oplus i)\ sp \qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ v.trans = \bot \\
\quad present\ (s \oplus \{n: dom\ i \bullet n \mapsto (s\ n) \text{✪} (i\ n)\})\ sp \qquad \textbf{otherwise} \\
\\
\textbf{where} \\
\quad i = getim\ p \\
\quad v = getvis\ p
\end{array}
$$

present accumulates the *IMAGE* in *s*. As each path *p* is taken from the head of the sequence of paths *(p:sp)*, its *IMAGE i* and *VISUALISATION v* are generated by *getim* and *getvis* respectively. If *i* is opaque (i.e. *v.trans* is undefined) then the *IMAGE* generated so far (*s*) is simply overwritten at the appropriate *POINT*s with *i*

($\oplus$ - the standard Z *function override*). On the other hand, if i is transparent, then an appropriate combination operator ($\bigstar$) is applied between the *COLOUR*s in s and i, at those *POINT*s where they coincide.

Note that what we are doing here is to associate the combination operator with the *IMAGE*, rather than *between IMAGE*s, since if an *IMAGE* is transparent, it appears to be transparent over any underlying *IMAGE*s (which have already been combined in s). It is conceivable that one could adopt an alternative scheme whereby the combination operator is specified between *IMAGE*s, such that the same *IMAGE* could appear transparent over one *IMAGE*, but opaque over another. However, this would require extra mappings to record the combination operator associated with each pair of *REGION*s. More importantly, in a highly manipulable environment this scheme would lack visual integrity.

In order to generate the display we need to define a suitable background, for example one that is all grey (we assume *grey* is a *COLOUR*):

$$BACKGROUND == \{p: AREA \bullet p \mapsto grey\}$$

A presentation of the surface display is therefore simply defined:

$$\begin{array}{|l} display: IMAGE \\ \hline display = present\ BACKGROUND\ paths \end{array}$$

The *IMAGE* sequence illustrated above would for example result in a *display* as in the illustration. Note that the last *IMAGE* in the sequence (here, furthest to the right) appears XOR transparent over *all* the underlying *IMAGE*s:

6.5. Manipulating the Model

As introduced in Chapter 3, the symbol Δ in Z indicates a change of state produced by some operation. Implicitly, including a schema with this prefix includes two copies of its components, one copy representing the state *before* the operation, and the other copy (with each component *primed* (')) representing the state *after* the operation. Thus the operation can be defined by specifying its preconditions in the unprimed state, and its postconditions as relationships between the two states. Input and output parameters are suffixed by *?* and *!* respectively.

In a large system like this, in which the state to be changed may have many components, the Δ representation of a model-based specification such as Z is perhaps more readable than functions which map between complex state spaces.

6.5.1. Initialisation

To specify the initial state of the *MEDIUM*, we define an operation with no preconditions, only postconditions:

Init_MEDIUM

Δ *MEDIUM*

tree' = *{root* $\mapsto$ *<>}*
geometry' = *{root* $\mapsto$ *DEFGEOM}*
behave' = *{root* $\mapsto$ *DEFBEHAVE}*
visualise' = *{root* $\mapsto$ *DEFVIS}*
content' = $\varnothing$

In the initial state, then, the *tree* consists simply of the *root REGION*. The *root* has a default set of attributes. We can assert that such an initial state is possible:

$\vdash \exists$ *MEDIUM* • *Init_MEDIUM*

6.5.2. Operations on the Medium

Operations affect the state of *MEDIUM* by directly accessing the *REGION* structures and their properties.

Creating *REGION*s

The following creation and copying operations affect not only the *REGION tree* itself, but also the mappings between *REGION*s and their properties.

The basic need is to create new *REGION*s. This is performed by *CREATE*. The only restriction is that the new *REGION* should not already exist on the tree:

CREATE

Δ *MEDIUM*
newreg!: REGION

$newreg! \notin dom\ tree$
$tree' = tree \cup \{newreg! \mapsto <>\}$
$geometry' = geometry \cup \{newreg! \mapsto DEFGEOM\}$
$behave' = behave \cup \{root \mapsto DEFBEHAVE\}$
$visualise' = visualise \cup \{newreg! \mapsto DEFVIS\}$
$content' = content$

The new *REGION* has a default set of attributes, but no content. It is added to the *tree* simply as an isolated node. Its identity is returned in the output parameter *newreg*.

When we *COPY* a *REGION* and its descendants, we wish to copy not only the *REGION* structure, but also any properties the *REGION*s may have, so that the *copy* projects a similar *IMAGE* to the *master*.

We define the *COPY* operation by asserting that there must exist an injective (one to one) mapping f between the *REGION*s in the family of *master* and the *REGION*s in the *copy*. In this way we can ensure that matching *REGION*s in the

master and the *copy* have the same properties. None of the *REGION*s in the *copy* (the range of *f*) are already on the *tree*:

COPY

Δ *MEDIUM*
master?, copy!: REGION

---

master? $\in$ *dom tree*
$\exists$ *f: REGION* $\rightarrowtail\!\!\!\!\!\!+$ *REGION* |
dom f = family (*{master?}*) $\wedge$ *ran f* $\cap$ *dom tree* = $\varnothing$ •
tree' = *tree* $\cup$ *{r: dom f • f r* $\mapsto$ *f* $\circ$ *tree r}*
geometry' = *geometry* $\cup$ *{r: dom f • f r* $\mapsto$ *geometry r}*
behave' = *behave* $\cup$ *{r: dom f • f r* $\mapsto$ *behave r}*
visualise' = *visualise* $\cup$ *{r: dom f • f r* $\mapsto$ *visualise r}*
content' = *content* $\cup$ *{r: dom f • f r* $\mapsto$ *content r}*
copy! = *f master?*

Since in all respects the copy is like the original, we can assert:

$\vdash$ $\forall$ *COPY* •
present BACKGROUND pathseq <master?> =
present BACKGROUND pathseq <copy!>

Constructing the *REGION* Tree

The second major group of operations allows *REGION* trees to be constructed and modified using *REGION*s that have been *CREATE*d or *COP*ied. In this way the medium *IMAGE*s can be grouped in different ways, and their layering on the surface changed arbitrarily. These *PASTE* and *CUT* operations affect only the *REGION tree* - the mappings between *REGION*s and their properties remain unchanged. The schemas are therefore restricted to specifying only the effect on *tree* by the Z schema projection operator $\upharpoonright$. We assume the other components are unaffected.

It is useful in defining these operations to assume a special null value ($\perp$) which represents lack of a *REGION*. However, $\perp$ is not a member of the *REGION* type.

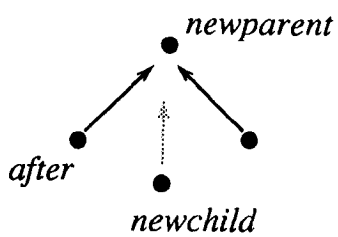
Any *existing root REGION* (i.e. one that does not have a parent) may be *PASTE*d anywhere on the tree, except into a leaf *REGION* which has content (since this then would cease to be a leaf, and only leaves can have content):

PASTE

Δ *MEDIUM* $\upharpoonright$ *(tree)*
newparent?, newchild?: REGION
after?: REGION $\cup$ $\{\perp\}$

$\{$*newparent?, newchild?*$\}$ $\subseteq$ *dom tree*
newchild? $\notin$ *dom parent*
newparent? $\notin$ *dom content*
after? $\neq \perp \Rightarrow$ *after?* $\in$ *ran (tree newparent?)*
tree' $=$ *tree* $\oplus$ $\{$*newparent?* $\mapsto$ *insert newchild? (tree newparent?) after?*$\}$

The *insert* function is defined in Appendix II. The *PASTE* operation can be illustrated:



The *PASTE* is determined by specifying the *newchild* to be *PASTE*d, the *newparent* into which it is *PASTE*d, and a *REGION* in the children of *newparent after* which the *newchild* is to be *PASTE*d. If *after* is $\perp$, then the *newchild* is *PASTE*d at the beginning of the sequence of children. It is this one case that makes it necessary to specify *newparent* as well as *after* - in all other cases where *after* is a *REGION* we could derive *newparent* from it.

Note that there is nothing to stop the *newchild* having descendants. However, we expect the restriction against cycles expressed in *ORDERED_TREE* to hold implicitly on the state after the *PASTE*. Therefore a *REGION* cannot be *PASTE*d into itself or one of its own descendants.

The arc between a *REGION* and its parent (i.e. any arc) may be *CUT*:

CUT

Δ *MEDIUM* $\upharpoonright$ *(tree)*
child?: REGION

---

oldchild? $\in$ *dom parent*
tree' = *tree* $\oplus$ *{parent child?* $\mapsto$ *remove (tree (parent child?)) child?}*

The *REGION* to be cut, *child*, must have a parent. The *CUT* is performed by simply *removing child* from the children of its parent. *remove* is defined in Appendix II. Note that this operation does not affect any children that *child* might itself have.

Deleting *REGION*s

Deleting *REGION*s clearly has an effect both on the structure of the tree, and on the sets of *REGION* properties (since deleted *REGION*s no longer have properties). If we *DELETE* a *REGION*, we must destroy all reference to it on the tree and in the property mappings. If the *REGION* has a parent, then the arc to this must be cut first:

DELETE

Δ *MEDIUM*
old?: REGION

---

old? $\in$ *dom tree - {root}*
tree' = *gone* $\lhd$ (*tree* $\oplus$ *{parent old?* $\mapsto$ *remove (tree (parent old?)) old?})*
geometry' = *gone* $\lhd$ *geometry*
behave' = *gone* $\lhd$ *behave*
visualise' = *gone* $\lhd$ *visualise*
content' = *gone* $\lhd$ *content*

Where
gone = *family* $(\!|$ *{old?}* $|\!)$

DELETE destroys the *old REGION* and all its descendants, and removes all references to these destroyed *REGION*s from *MEDIUM* (using domain subtraction ($\lhd$)). However, the *root REGION* cannot be deleted.

Inquiries on the Trees

We wish to allow applications to traverse the trees, without, however, revealing any details of the implementation structures. We therefore define tree traversal operations which deal solely in terms of the abstract construct *REGION*. In all these inquiry operations the state of the *MEDIUM* is not affected. This is specified by the schema reference prefix Ξ, which implicitly includes primed and unprimed copies of all the components, like Δ, but in addition states that they are all equal.

GETFIRSTCHILD returns the first child of *parent*:

GETFIRSTCHILD

Ξ *MEDIUM*
parent?, child!: REGION

---

child! = head (tree parent?)

Fairly clearly, *GETFIRSTCHILD* is undefined ($\bot$) if *parent* is a leaf *REGION*, or is not on the tree. Children other than the first can be found using *GETNEXTSIBLING*:

GETNEXTSIBLING

Ξ *MEDIUM*
sibling?, next!: REGION

$\exists$ *u, v: seq REGION* • *u* $\frown$ <*sibling?*> $\frown$ <*next!*> $\frown$ *v* $\in$ *ran tree*

The *next* sibling is defined so long as there exists a sequence of *REGION*s in the range of *tree* in which *next* follows the input *sibling*.

Traversing upward on the tree is simply a matter of getting the *parent* of a *REGION*:

GETPARENT

Ξ *MEDIUM*
child?, parent!: REGION

---

parent! = parent child?

The *parent* is undefined if *child* is a root, or does not exist on the tree.

Updating *MEDIUM* Contents

MEDIUM has only two types of content: text and graphics. Graphical content is updated simply by updating the *content* mapping:

DRAW

Δ *MEDIUM* $\upharpoonright$ *(content)*
reg?: REGION
newgraph?: seq LINE

---

tree reg? = <>
content' = content $\oplus$ *{reg?* $\mapsto$ *(graphics* $\Downarrow$ *newgraph?, text* $\Downarrow$ $\perp$*)}*

Textual content is similarly updated:

WRITE

Δ *MEDIUM* $\upharpoonright$ *(content)*
reg?: REGION
newtext?: seq CHAR

---

tree reg? = <>
content' = content $\oplus$ *{reg?* $\mapsto$ *(graphics* $\Downarrow$ $\perp$*, text* $\Downarrow$ *newtext?)}*

In both cases the restriction that only a *leaf REGION* can have content is maintained. Similarly, a leaf *REGION* cannot both contain text and graphics. This is

maintained brutally simply by dropping the opposing content if it exists. Clearly in practice an error report would be less destructive.

These formulations also ignore a number of important issues which are likely to be significant in an implementation:

- In the case of new leaf *REGION*s (either because they have been newly *CREATE*d, or because their children have been *CUT*) there will not exist a *content* mapping. In these cases one will have to be created, whereas in other cases an existing mapping will simply have to be updated. It is a design issue whether there should be a separate *CREATE_CONTENT* operation to set up the mapping, or whether the mapping should be automatically created when a *WRITE* or *DRAW* is requested, if it does not exist.

- Whereas the addition or deletion of a single character in text or line in graphics can be expressed simply as a substitution of whole text or graphics sequences for other sequences (in which only a small change has been made), clearly in practice these changes need to be made incrementally. That is, some *editing* mechanism must be provided.

- In order to indicate which changes are to be made, the user or the application needs to have some way of *selecting* characters or lines or insert points. This raises issues of the *persistence* and *identity* of the representation, either on the display surface, or as internal machine objects.

These issues are abstracted away from here in the interest of conciseness and clarity.

Removing Content

We may wish to remove content from a leaf *REGION* in order to paste a subtree into it. We simply specify this as removing the *REGION* from the content mapping:

STRIP

Δ *MEDIUM* $\upharpoonright$ *(content)*
reg?: REGION

---

content' = *reg?* $\lhd$ *content*

Updating and Enquiring Properties

Tuples expressing properties other than content are defined to be persistent. That is, a *REGION* will always have a *GEOMETRY*, a *VISUALISATION*, and a *BEHAVIOUR*. Since we will often wish to perform a selective update or enquiry on some attribute component of these tuples, we define generic *update* and *enquire* operations. We assume the sets *ATTRIBUTE* and *VALUE* of attribute names and their values respectively.

update produces a new tuple which is the same as the old, except that the specified *ATTRIBUTE* is updated to the specified *VALUE*:

$$[T]$$

$$update: (T \times ATTRIBUTE \times VALUE) \rightarrow T$$

$$\forall\, t: T;\ a, c: components\ T;\ v: a \bullet$$
$$c = a \Rightarrow (update\ t\ a\ v).c = v$$
$$c \neq a \Rightarrow (update\ t\ a\ v).c = t.c$$

Again we assume a syntactic function *components* which extracts the set of attribute names in a tuple. The specified *ATTRIBUTE* a must be a component of the tuple, and the specified value v must be of the type of a.

enquire returns the current *VALUE* of the specified *ATTRIBUTE*:

$$[T]$$

$$enquire: (T \times ATTRIBUTE) \rightarrow VALUE$$

$$\forall\, t: T;\ a: components\ T \bullet enquire\ t\ a = t.a$$

Without detailing each one, we assume a set of specific operations prefixed with *SET-* and *GET-* which set or enquire the value of any attribute of any *REGION* in the state. For example, in order to set the behaviour of a *REGION*:

SETBEHAVIOUR

Δ *MEDIUM* $\upharpoonright$ *(behave)*
reg?: REGION
attribute?: ATTRIBUTE
value?: VALUE

---

behave' = behave $\oplus$ *{reg?* $\mapsto$ *update (behave reg?, attribute?, value?)}*

and to enquire the current behaviour of a *REGION* in a particular attribute:

GETBEHAVIOUR

Ξ *MEDIUM*
reg?: REGION
attribute?: ATTRIBUTE
value!: VALUE

---

value! = enquire (behave reg?, attribute?)

Note that the enquiry operations do not change the state of the *MEDIUM* (Ξ).

Manipulating the *MEDIUM*

The *MEDIUM* is manipulated by updating its *GEOMETRY*. In this case the effects of an update cannot simply be defined as above, because there are also constraints that are applied by the *BEHAVIOUR*. For example, a *REGION* that is *movable* only *horizontal*ly will only allow a *position* update in that dimension. We use the operation *SETPOSITION* as an example of this class of geometric update.

The basic operation to move an object is to set its position to some location in its parent. We assume as an implicit precondition for this *SETPOSITION* operation that the target *REGION reg* already exists on the tree:

SETPOSITION

Δ *MEDIUM* $\upharpoonright$ *(geometry)*
reg?: REGION
newpos?: $(\mathbb{R} \times \mathbb{R})$

$geometry' = geometry \oplus \{reg? \mapsto update\ (old, position, pos)\}$

Where

$(behave\ reg?).\ movable = both \Rightarrow$
$pos = newpos?$
$(behave\ reg?).\ movable = \perp \Rightarrow$
$pos = old.position$
$(behave\ reg?).\ movable = horizontal \Rightarrow$
$pos = (first\ newpos?, second\ old.position)$
$(behave\ reg?).\ movable = vertical \Rightarrow$
$pos = (first\ old.position, second\ newpos?)$
$old = geometry\ reg?$

This is performed simply by updating the *position* attribute of *reg*'s *geometry*. However, setting the new value for the *position* is complicated by the need to take into account the behavioural constraints that might be set on *reg*. Thus if *reg*'s *movable* attribute is *both*, then it is simply moved to the requested position. However, if it is undefined, then its new position remains the same as its old position. If the *movable* attribute is either *horizontal* or *vertical*, then one component of the new *position* coordinate remains the same as its old value.

This is, however, not the whole story, since we must also account for the *group* behaviour in the descendants of *reg*. The intention of *group* is that, for any *REGION* on which it is *true*, that *REGION* should move if its parent moves. This is the default case, since all that is required is for the *REGION* to maintain its same *position* in its parent. If, however, *group* is undefined on a *REGION*, then that *REGION* should maintain its *absolute* position on the surface, irrespective of changes to the position of its parent. In order to achieve this, the *REGION* must effectively be moved by the same distance as the parent, but in the opposite direction. This is further complicat-

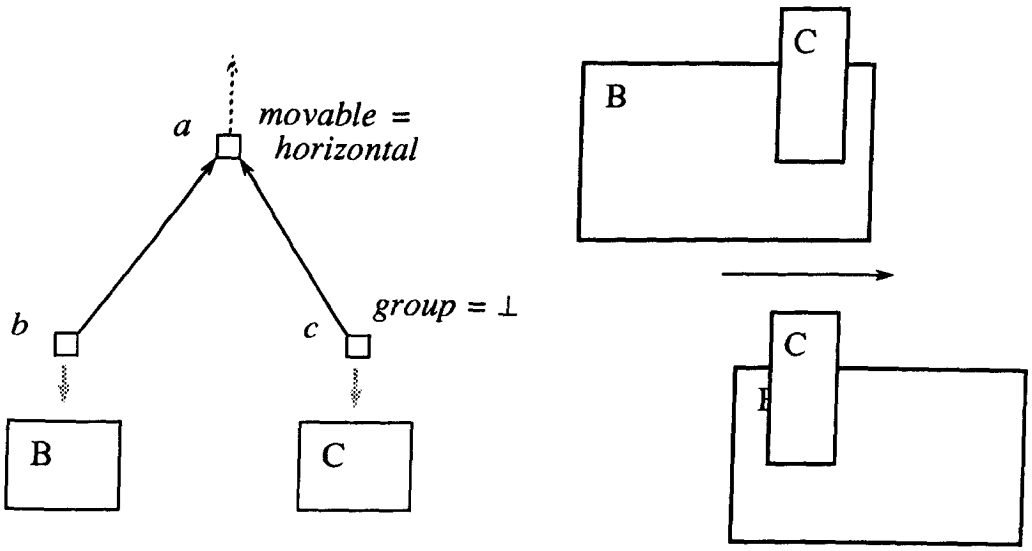
ed by the fact that the move may in fact have originated not in the *REGION*'s immediate parent, but in a higher ancestor. Since this may not be the same size as the *REGION*'s parent, the distance moved in the ancestor must be transformed into the parent's space in order to achieve the effect of leaving the non-*grouping* *REGION* unmoved.

We continue the *SETPOSITION* schema from above:

$$
\begin{array}{l}
g\ (tree\ reg?)\ (M\ reg?) \\
\\
\textbf{where} \\
\quad g <> m = true \\
\quad g\ (r{:}sr)\ m \Rightarrow \quad g\ sr\ d \wedge g\ (tree\ r)\ (m \cdot (M\ r)) \quad \textbf{if}\ (behave\ r).group = true \\
\qquad\qquad\qquad\quad\ g\ sr\ d \wedge shift \qquad\qquad\qquad\quad \textbf{otherwise} \\
\\
\quad shift = (geometry' = geometry \oplus \{r \mapsto update\ (geometry\ r, position, \\
\qquad\qquad (geometry\ r).position - (m^{\sim} \cdot pos\ -\ m^{\sim} \cdot old.position)))
\end{array}
$$

g is a recursive predicate (i.e. a boolean function) which determines that if a *REGION* does not have group behaviour, then its position is shifted with respect to its parent to cancel out the movement of its ancestor. *g* passes the transformation matrix *m* down the tree, which determines how the *REGION r* is currently transformed with respect to the parent of *reg*, until it reaches a *REGION* on which *group* is undefined. The inverse of *m* is then applied to the original points between which *reg* was moved, and *REGION r* is moved this distance in the opposite direction. The recursion does not continue to the children of *r*, since if *r* stays at the same absolute position then so will they.

The combined effect can be illustrated:



medium state *surface presentation*

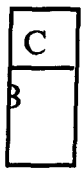
Suppose that *REGION a* is at *position (0.5, 0.5)* in its parent. Now suppose that the operation

SETPOSITION (a, (1.0, 1.0))

is invoked. Since *a* is movable only horizontally, it will in fact be moved to position *(1.0, 0.5)*, following the constraint in the first part of the schema. In the normal case, this would cause both *CONTENT REGION*s *b* and *c* to move on the surface, since they would then stay in the same position with respect to their parent *a*. However, in this example *REGION c* is set not to *group*, and so it must remain at the same absolute position on the surface, even though the rest of its group moves. To achieve this, *REGION c* must actually move in the opposite direction with respect to *a*. Now *a* moves a distance of *0.5* of the width of its parent to the right. Let us say *a* is in fact only half the width of its parent. In this case, to achieve the surface effect of staying in the same place, *REGION c* must move 2×0.5 of the width of *a*, to the left.

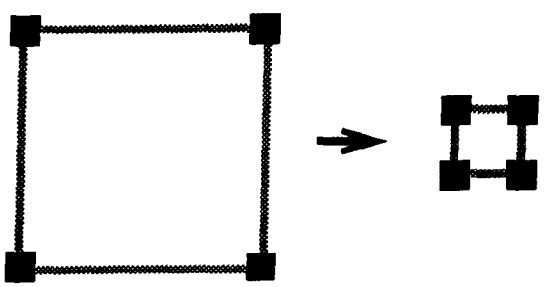
This effect is powerful, for example, in the following situation. Imagine that we wish to make *c* into a *window* onto *b*. This could be achieved by making *a* the same size as *c*, and giving *a* a mask of size *AREA*. *a* would then clip all its descendants to its own size. On the surface the effect would be that only the area of *c* would be visi-

ble. In order also to view that portion of *b* through *c*, it would be necessary to make *c* transparent:



Now if *b* were movable, then it would be possible to move it around *underneath* *c*, still viewing only that portion that was clipped by *a* to the extent of *c*. However, it is equally likely that the interface designer might require *c* to be moved about like a frame *over b*. This can simply be achieved by setting *b* rather than *c* to be non-*grouping*, and then moving *a* about. The effect is that *c* moves with *a*, so that their clipping boundaries continue to coincide, yet *b* remains in the same surface position.

The operation *SETSIZE* has a very similar definition, but we do not give this for reasons of space. The attributes *sizable* and *scale* have an analogous effect on the behaviour of *REGION*s under sizing. That is, the effect of a size can be turned off, or restricted to only the horizontal or vertical component, depending on the value of *sizable*. Similarly, setting *scale* undefined on a *REGION* has the effect that the *REGION* remains the same absolute size on the *MEDIUM* irrespective of changes to the size of its parent. This is effective in providing fixed-size components of a surface object, for example title bars or Mac-like selection handles:



Note that making a *REGION* non-*scal*ing does *not* imply that its *position* will remain unchanged under a scale of its parent.

6.5.3. Picking and Selecting

*REGION*s in the *MEDIUM* will have been *CREATE*d only by the deep processes. Thus we can expect these processes to know about, and be able to access,

MEDIUM objects by means of *REGION* identifiers. The user, on the other hand, will often only have some pointing device on the surface, such as a mouse. It is therefore necessary to reference the *MEDIUM* state in terms of the coordinates of this device. This process is called *picking*.

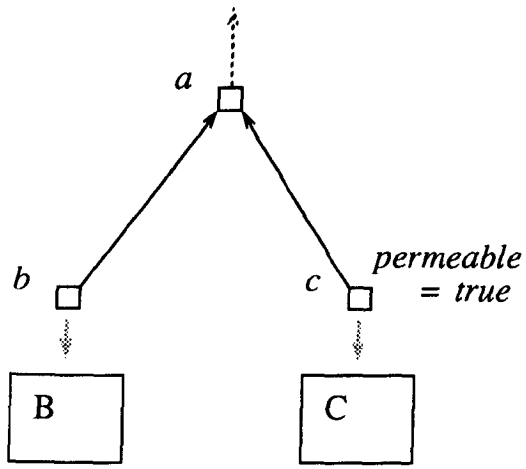
In order to abstract away from device resolution, we assume the mouse coordinates will be some *POINT* in the set *AREA*. We first define a function to perform a *pick*. This takes a sequence of paths, and a mouse coordinate, and returns a *pick*ed path:

$$pick: iseq\ (iseq\ REGION) \nrightarrow AREA \nrightarrow iseq\ REGION$$

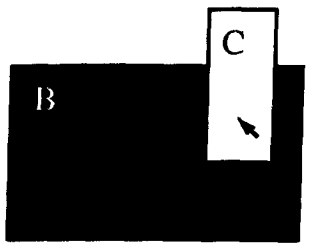
$$\begin{aligned}
&pick <> m = \bot \\
&pick\ (sp:p)\ m \quad = p \qquad\qquad \textbf{if}\ m \in dom\ (getim\ p) \wedge \\
&\qquad\qquad\qquad\qquad (propagate\ DEFBEHAVE\ behave\ p).permeable = \bot \\
&\qquad\qquad = pick\ sp\ m \qquad\qquad \textbf{otherwise}
\end{aligned}$$

Note that we intend the function to recurse *backwards* along the sequence of paths - that is, from the foreground of the display to the background. The first path that is found whose *IMAGE* contains the mouse *POINT*, and which is not *permeable*, is the path *pick*ed. This function effectively defines the semantics of the *permeable* attribute: the *IMAGE* of a path that is *permeable* allows mouse picks to drop through onto *IMAGE*s of *REGION*s underneath. The property of permeability can be propagated from ancestors.

The effect of picking and permeability can be illustrated using the example from above:



medium state



surface presentation

If *pick* is applied to the *POINT* of the cursor in this surface presentation, then the path <*a*, *b*> is returned. That is, *REGION b* is effectively picked, since *c* is permeable. This is useful, for example, if, as in the first example above, *c* were a clipping frame onto *b*, and it was necessary to move *b* through *c*.

Both mappings in *pick* are partial because in the first, not all path sequences may be pickable (all leaves may be permeable, or off screen). In the second, not all mouse coordinates will be above non-permeable *REGION*s.

Selection

This is not the whole story, however, since picking returns a *path*, rather than a single *REGION*. There needs also to be some method of discriminating between *REGION*s on the path. This is the purpose of the *selectable* attribute in a *REGION*'s *BEHAVIOUR*. Only a selectable *REGION* can be the target of a selection. However, in contrast to permeability, a non-*selectable REGION* passes the pick *up* the path,

rather than *along* the sequence of paths. A function *select* can be defined which takes a path, and returns the first *selectable REGION* up the path:

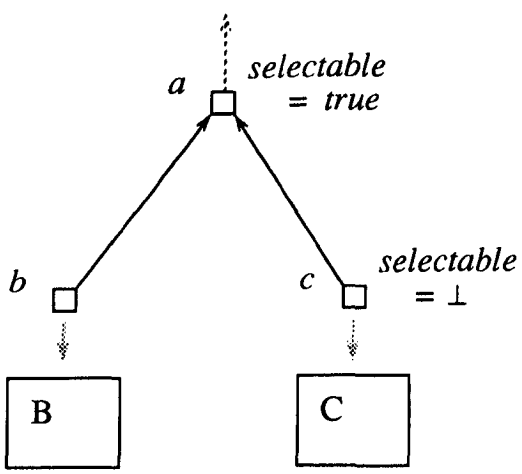
$$
\begin{array}{|l}
\textit{select: iseq REGION} \nrightarrow \textit{REGION} \\
\hline
\textit{select} <> = \bot \\
\textit{select (sr:r)} \quad = r \qquad \textit{if (behave r).selectable = true} \\
\qquad\qquad\quad = \textit{select sr} \qquad \textbf{otherwise}
\end{array}
$$

These two functions *pick* and *select* can be combined in a *MEDIUM PICK* operation:

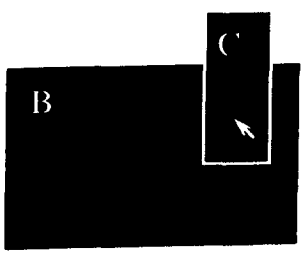
$$
\begin{array}{|l}
\textit{PICK} \\
\hline
\Xi\, \textit{MEDIUM} \\
\textit{mouse?: AREA} \\
\textit{picked!: REGION} \\
\hline
\textit{picked!} = \textit{select pick paths mouse?}
\end{array}
$$

The *paths* are the (already defined) sequence of paths projected by the *root*. The operation takes a mouse position, and returns a *REGION*. It does not change the state of the *MEDIUM*.

This combined effect can again be illustrated using the example above:



medium state



surface presentation

Here the *PICK* operation with mouse position as in the surface presentation returns *REGION a*, since the *pick*ed *REGION c* is not *selectable*. The effect on the display might be that both children *b* and *c* of *a* are highlit, or can be moved together (if *a* is movable). Thus manipulable composite objects can be constructed on the surface which the user can access as a single object.

6.5.4. The User Agent

As outlined in theory in Chapter 4, the user agent implements some mapping between sequences of raw input, and either *(input, REGION)* reports to the applications, or operation commands to the *MEDIUM*. The sencond case enables the user to access the *MEDIUM* objects 'directly' (in the sense of 'without involving the application'). To support the thesis that such direct access is possible, we wish this mapping to be as simple as possible. That is, we wish to reduce to a minimum the possibility of *semantic seepage* from deep into surface interaction. This might arise, for example, if the user agent were programmable or interpreted some dialogue specification.

We presuppose in this model (but *not* in the *principle* of Surface Interaction) that the user agent has a direct manipulation style. This excludes moded use of the keyboard for generating command strings or tokens. Thus, in the user agent mapping, keyboard input is considered simply to be text entry, and is not interpreted as commands. This leaves the mouse as the primary device for manipulating the *MEDIUM* through the surface. There is consequently a hard limit to the number of operations that can be directly invoked, simply because (normally) the mouse has three or fewer buttons.

This limitation can be avoided by incorporating *soft* buttons in some iconic environment managed by the user agent on the medium. This is in effect what happens in desktop environments such as the Mac. Such an environment may well be needed in a production system. However, in this case style and often operating system semantics are bound into the surface. In contrast, we are concerned here with providing a level of functionality which is independent of these.

The choice of which *MEDIUM* operations should be available directly to the user via the user agent, as opposed to which can only be made available via the mediation of an application, is a design issue. However, the operations which move

and size *REGION*s exploit the geometric characteristics of the mouse better than those which create or delete *REGION*s.

For this reason the implementation of the user agent makes the following default assignment of mouse buttons to medium operations (assuming a three-button mouse, and including the implemented text operations):

- left: selection in either graphics or text.
- middle: moving in graphics; cutting (button press) and pasting (button release) in text.
- right: sizing in graphics; copying (button press) and pasting (button release) in text.

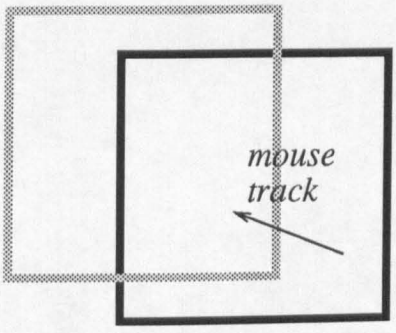
Pasting into a non-editable text or a non-text area deletes the cut or copy.

We do not give here a full formal specification of the user agent's mapping between raw user input and medium operations, because, even when the mapping is simple, such a specification rapidly approaches the complexity of an implementation. In the simple example in Chapter 4, the user agent was complex enough to require a memory of whether it was in dragging mode or not. In the present case, the user agent needs also to remember *which* object it is dragging or sizing, and to calculate, from changes in the mouse position, by how much it needs to change the position or size of the object in the space of its parent. In addition, it must determine what user input should be reported to the application, on the basis of the *BEHAVIOUR* of the selected *REGION*. We give just a flavour of the mousing issues here.

Mouse Actions

When the movement or sizing button is pressed over an object, we notionally fix the cursor to that point in the object. A moving operation will simply drag the

object so that at the end of the movement the cursor is still at the same position in the object:



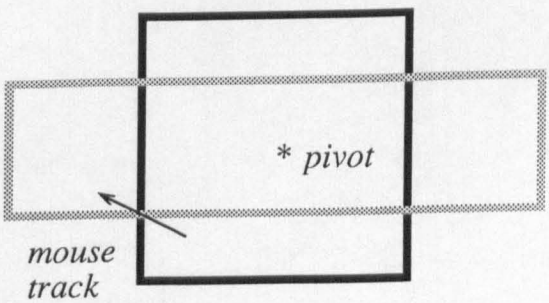
The object is a projection of a path of *REGION*s. Depending on which level of the tree has been selected (see section 6.5.3), we will actually be moving one of the *REGION*s on this path. If we call this *REGION r*, and assume that *r* is initially at *(x, y)* (the position of *r*'s pivot in the parent of *r* on the path), and that the mouse moves from *(mx1, my1)* to *(mx2, my2)* in the space of the parent, then the actual medium operation which the user agent performs will be:

SETPOSITION (r, (newx, newy))

where

newx = first ((geometry r).position) + (mx2 - mx1)
newy = second ((geometry r).position) + (my2 - my1)

Sizing objects with the mouse is very similar. We assume the cursor is fixed to a point in the space of the object, and that the *pivot* of the object (i.e. of the selected *REGION* on the path) is fixed in the space of its parent. The object will therefore alter its size (and shape) around this pivot:



Given the conditions above, an actual invocation might be:

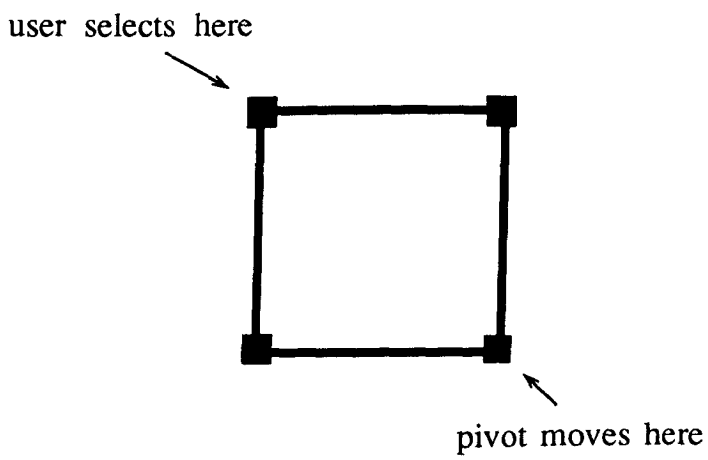
SETSIZE (r, (newwidth, newheight))

where

*newwidth = first ((geometry r).size) * (mx2 - px) / (mx1 - px)*
*newheight = second ((geometry r).size) * (my2 - py) / (my1 - py)*
px = first ((geometry r).pivot)
py = second ((geometry r).pivot)

This calculation reveals a number of design problems. The mouse cursor must not start or finish at either the horizontal or vertical axes of the pivot, or else the object will be sized to infinity or to zero respectively. Also, if the mouse track *crosses* the pivot axes, then the object will be inverted in that direction. What happens to *REGION*s containing text in this case?

In practice these are not insuperable problems. It is easy to check if the numerator or denominator evaluates to zero, and substitute some minimal value. It is also easy to rule that text *REGION*s do not invert. Slightly more difficult is the problem that sizing an object near the (invisible) pivot point results in gross size changes on the surface. Although users seem to get used to this quickly, it is also relatively easy to implement the behaviour of Mac-like 'handles' by dynamically moving the pivot point away from the mouse cursor:



The implementation also allows the application to decide (by setting an attribute here unspecified) if objects on the surface move smoothly under surface interaction. In this case a *SETPOSITION* or *SETSIZE* operation is invoked by the user agent upon every drag event. The alternative is to invoke these only at the *end* of a drag, i.e. when the relevant button is released. The object then jumps (relatively speaking) to its new position or size. This is complicated by the need to provide

some feedback during the drag - in the implementation, the user agent asks the medium to create and manipulate a *rubber box*, of the object's size, during the drag.

Thus, the basic functionality of the user agent is clear: it manipulates the medium on behalf of the user, and reports some input to the application. In practice, however, design and implementation issues soon arise, which it is not appropriate to deal with here.

6.6. Conclusions

This Chapter has presented in some detail a formal model for the surface medium. This has specified the three essential features in providing an encapsulated medium: its *state*, its *operations*, and its *presentation mapping* from the state to the display. When used as the semantics of *M* in the *UMA* architecture presented in the first part of the Thesis, this medium model underpins Surface Interaction. A particular implementation of this, *Presenter*, is described in the next Chapter.

Chapter 7

Presenter

This Chapter outlines and discusses a particular implementation of a surface, *Presenter*. This discussion, in contrast to the previous Chapter, is informal and implementation-oriented. The justification for this perspective is that constraints can arise at this level which compromise the abstract design [Took90b].

Presenter was written originally as part of the Aspect IPSE project [Hall85]. *Presenter* is written in C and runs within the SunView environment [Sun86], although, for portability, it uses only the lowest level imaging primitives from this. While it might be nice to be able to suggest that the principle of Surface Interaction came first and that *Presenter* is an instantiation of this, in fact it was the other way around. The whole burden of this thesis has in fact been to formulate and justify precisely in what way (if any) *Presenter* differs from other user interface managers. Hence the notions of surface and Surface Interaction.

Although *Presenter* was formally specified (in Z) before it was coded (and respecified after coding), the specification given in Chapter 6 is in fact an ideal rational reconstruction of a design based on *Presenter*'s tree structure (Chapter 8 outlines an alternative design based on a *hierarchical* structure). The present Chapter therefore concentrates on those areas in which the implementation differs from and refines this ideal design. It uses Surface Interaction and its *UMA* architecture to clarify these design and implementation issues. [Took90b] examines more generally the problems of refining a formal design.

We regard a practical implementation not just as a validation of the formal model, but as the model's *raison d'etre*. The model, therefore, as well as being a convenient conceptual structure, should also be expressible as an effective imple-

mentation structure. A computationally intractable model is of no *use*. (It is easy to specify such models. For example:

FERMAT

n: *integer*

$\exists\, a, b, c$: *integer* $\bullet\ a^n = b^n + c^n$

)

An implementation may also contribute significantly to the success of a design by an efficient use of temporal and spatial computer resources. However elegant the formal model, a slow or memory-prodigal implementation will compromise usability. While a great deal of effort was (successfully) put into designing the algorithms for the efficient update of the display from the model in *Presenter*, detailing these is unfortunately outside the scope of this Thesis.

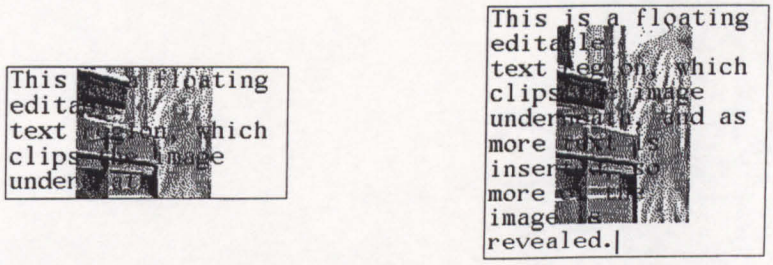
7.1. Brief Outline

[Took90a] gives an informal description of *Presenter* and Surface Interaction. There also exists a (unpublished) user/programmer manual for *Presenter*, which gives in full detail its functionality and C interface.

In many respects, *Presenter* is a faithful instantiation of the model specified in Chapter 6. It has a single construct, *region*. (In the C interface, the *only* new type that is used is **region*). Regions are created and built into tree structures using operations that are almost exact counterparts of the formal operations. There is a distinguished *root* region per workstation, from which the display is projected. Content consists of text and graphics. Only leaf regions may have content, and a region can contain *either* text or graphics, but not both.

The implementation imposes no limits on the size or position of regions with respect to each other (except that regions which are more than twice the screen area temporarily lose their visual content). There is no interference between the behaviour and visualisation properties of regions. For example an editable text region can be overlapped, clipped, cut and pasted on the tree structure, made transparent, moved and sized, and remain editable in any condition (so long as it is visible). As an illustration of this orthogonality, here is an editable text region which

is OR-transparent, floating (see below), and clips regions underneath. As the user enters text, the region expands, and more of the underlying region becomes visible:



These capabilities can all be modified dynamically, again with no restrictions. Thus if this text region were made permeable, then immediately, and with no visual change, the underlying object could be moved or sized (or edited, if it were itself an editable region) *through* the text region above it.

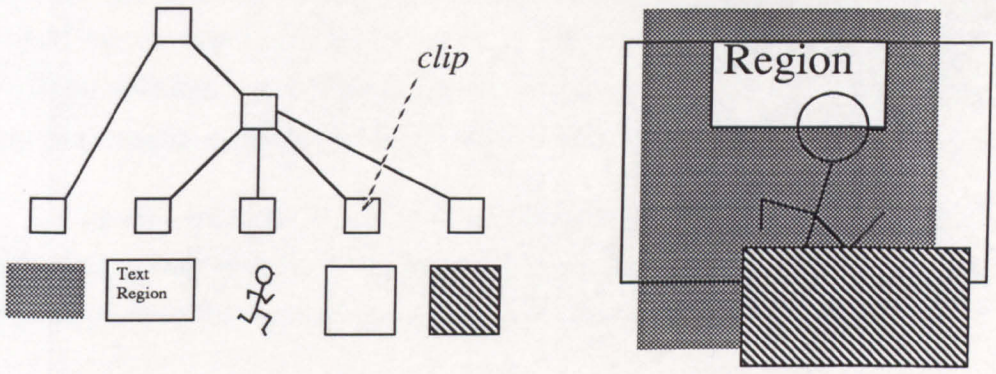
Surface Interaction is provided during *drag* events. That is, the user may change the size or position of (sizable or movable) regions by direct manipulation. The application is only informed at the beginning and end of the mouse drag (i.e. on button press and release). In addition, textual and graphical editing can be performed directly by the user as Surface Interaction. The application can elect to be informed of Carriage Return events on particular text regions. This is useful for constructing dialogue fields which the user can edit independently of the application, and then report to the application by pressing RETURN.

7.2. Differences

7.2.1. Clipping

The major *difference* between the current implementation of *Presenter* and the formal specification in Chapter 6 is in clipping. Clipping in *Presenter* has almost equivalent power to that in the formal specification, but setting it up is more awkward. The semantics of the *clip* attribute in *Presenter* are that it can only be set on

leaf regions, and that a region with *clip* set clips *prior siblings* on the tree. This can be illustrated:



logical view *surface view*

Thus the region with *clip* set clips the stick figure and the text region. It does not clip the grey background since this is not a sibling, and it does not clip the striped region since this is not prior in the ordering of the tree.

This was a design mistake. It is non-intuitive for the programmer and was extremely difficult to maintain in implementation (although it works). The motivation for this scheme was to allow clipping regions to act as windows on other regions such that

- either the window itself could be moved around over the clipped regions,
- or, if the window were also set *permeable*, underlying regions could be moved in and out of the clipping boundary.

Originally it was thought that the more intuitive scheme of simply setting clipping on arbitrary regions, and allowing a non-leaf region to clip its descendants, would not permit the first option above. That is, moving a non-leaf clipping region would also move its descendants, which is the opposite of the intention. It was not realised at that time that the (already existing) *nogroup* attribute, if set on regions which needed to stay in place, would allow the clipping ancestor to move in exactly the way required. A visible *frame* coincident with the clipping boundary could also be added as a leaf child of the clipping region, which *could* move with the clip. This therefore is the way clipping is defined in Chapter 6 (see Section 6.5.2).

The clipping scheme in the implementation is less general than that specified also in that clipping is restricted to the region boundary. Thus there is no implementation of the arbitrary *mask* of the specification. Hiding of regions, for example to create pop-up menus, is effected using a *hide* attribute. This simply removes the visual representation of a region from the screen, but does not affect any other property of the region. Clearing the *hide* attribute redisplays the region.

Presenter also provides two other attributes with the same tree scope as its *clip*: *contain* and *exclude*. A leaf region with either of these set will force prior siblings not to move either inside, or outside, its boundaries.

7.2.2. Application Confirmation of Input

In the *UMA* architecture an application confirms or modifies user input by returning the reported input back to the user agent either unchanged or changed. This is a convenient use of the event notation in CSP. An earlier version of *Presenter* did exactly this. However, it was found that as in the majority of instances the input (the selected region) was passed back unchanged, this seemed an unnecessary burden on the programmer, and sometimes led to errors of omission.

The latest version instead provides explicit operations by which the currently selected region or the currently moving or sizing regions can be altered or cancelled. Thus by default Surface Interaction goes ahead as the user planned, but can be changed by the application by explicitly changing the targets as a separate operation.

7.3. Additions

Presenter implements also some additions to the formal specification as given, simply because there was not room to define all its capabilities formally.

7.3.1. Editing

Any leaf region with content can have an *editable* attribute set. The editing operations provided by *Presenter* are bound to the mouse buttons. There are two overall principles:

• All editing operations should be performed simply using the mouse or keyboard. That is, no menus, icons or other visual prompting is built in (other than text highlighting). The intention is to minimise any predetermined style bindings at the interface. However, in this design there is nothing to stop an application interposing some more graphical representation (for example a scroll bar), and then calling on the *Presenter* operation itself on behalf of the user.

• The denotation of the mouse buttons should be (roughly) analogous in the domains of graphics and text. Thus the first mouse button selects, the second moves (in text, cuts and pastes), and the third sizes (in text, copies and pastes).

Text

If the region contains text, then the user will be able to scroll this text (using a chord of *control* and a mouse button), select sections of text, enter and delete text to the left of the selection, and cut or copy and paste text both within that region and to and from any other editable text region. Text is cut or copied and pasted simply by clicking the second or third mouse button over a selection, moving to a new location, and releasing the button. If the button is released over any screen area that is not editable text, then the cut or copy is deleted. No undo operations are provided.

Text can be in any (fixed width) font available on the system (new fonts can be loaded by the programmer). Font is determined initially by the region (different regions can have different basic fonts), but text cut from one region and pasted into another region retains its original font. Thus regions support mixed fonts. Text entered dynamically by the user takes its font from the character to its right. In the current implementation operations are not provided to change the font of a piece of text once it is set.

If a region is changed in size, either by the user or the application, then the text it contains is reformatted dynamically. By default, text regions word-wrap (lines are broken at word boundaries), although there is also an attribute which forces character wrapping. If the region becomes too small for its text, then the text runs off at the base of the region, and can be recovered by scrolling. There are no limitations on the size of text regions. They can be used to implement full size windows, but also can

be used in single line fields for dialogue boxes and the like. If the region is narrower or shorter than the font size, then the text disappears.

Finally, another attribute *float* is provided, which when set on a text region constrains the region to a size which just contains its text. If the region is in addition editable, then the region changes size as the user (or the application) inserts or deletes text. *float* can be specialised to work either horizontally or vertically, such that a region might have a fixed width but grow vertically as text is entered, or conversely a fixed height and grow horizontally.

Graphics

If the region contains graphics, i.e. line segments, then with the *editable* attribute set the user can dynamically draw straight lines in the region (using the first mouse button), and can subsequently pick these lines up and move them (using the second mouse button). As with text, if the line is moved out of the region, then it is deleted. In fact it is possible to shave bits off a line by partially moving it out of a region. However, the orientation of a line cannot be changed once drawn. Applications also can draw lines in regions, but have no access to these once drawn.

Once lines are drawn in regions then they scale proportionally as regions are changed in size. If a *flip* attribute is set, then it is possible to flip the image over either horizontally or vertically during sizing by dragging a point in the region across its pivot (*flip* can be specialised to either of these dimensions).

Bitmaps

As a practical convenience, a leaf region may also contain a bitmap. This may be loaded in from any source, for example a digitiser or an icon editor. There is no restriction on the size of the bitmap, but it must be in SUN *pixrect* format. The region is automatically scaled to the size of the bitmap upon loading. If the size of the region is subsequently changed, then the image will either be clipped (if the region is smaller), or replicated (if the region is larger).

In addition, any leaf region, either text or graphics, can dynamically be converted to a bitmap, simply by setting a *bitmap* attribute. A text region, for example, will then simply be clipped or replicated, rather than reformatted, when its size changes. A bitmap region can of course be set *transparent* or *permeable*.

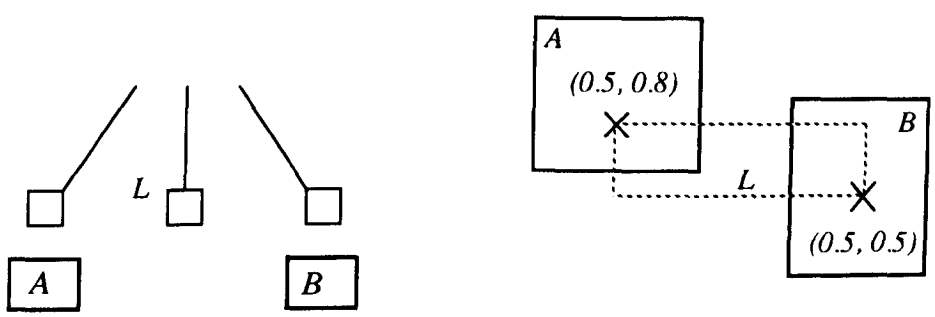
If a *bitmap* region is also set *editable*, then the user may draw pixel-wide free-hand lines on the region with the first mouse button, and erase narrow or wide bands of the image with the second or third mouse buttons respectively.

7.3.2. Linking

A powerful capability, which was not in the original design but was included under user pressure, is *linking*. On top of the region structure it is possible for the application to arbitrarily link regions by calling a *link* primitive, specifying the two regions to be linked (leaf or non-leaf), and *pin* points in the space of each region. *Presenter* then dynamically creates an empty leaf region whose size and position it maintains such that its opposing corners are always at the pin points in the two specified regions. An extra parameter also allows the application to specify whether the link should be created BELOW, BETWEEN, or ABOVE the two regions in the tree ordering. For example, a call of

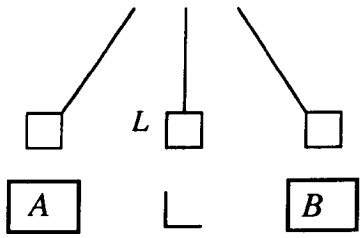
link (A, 0.5, 0.8, B, 0.5, 0.5, BETWEEN)

on regions *A* and *B*, would result in the creation of (invisible) region *L*:

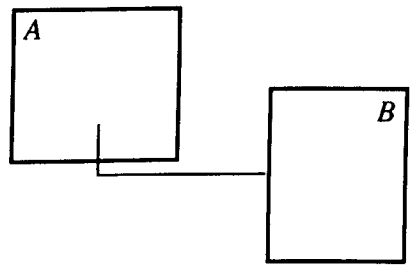


medium state *surface presentation*

The application can then insert either more regions, or content, into region *L*. For example, it could add a right angled line with the following visual result (*L* is by default *transparent*):

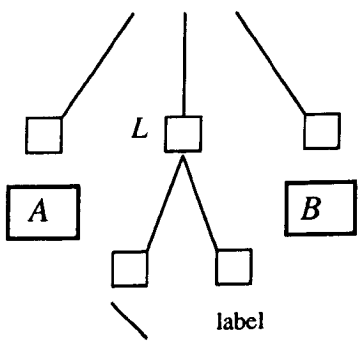


medium state

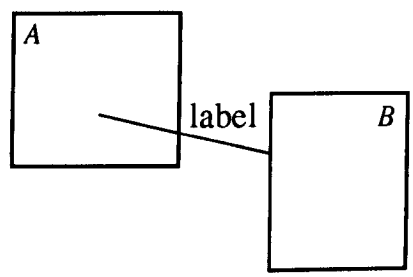


surface presentation

or to produce a diagonal line with a label the application could paste two new regions, with appropriate content, into *L*:

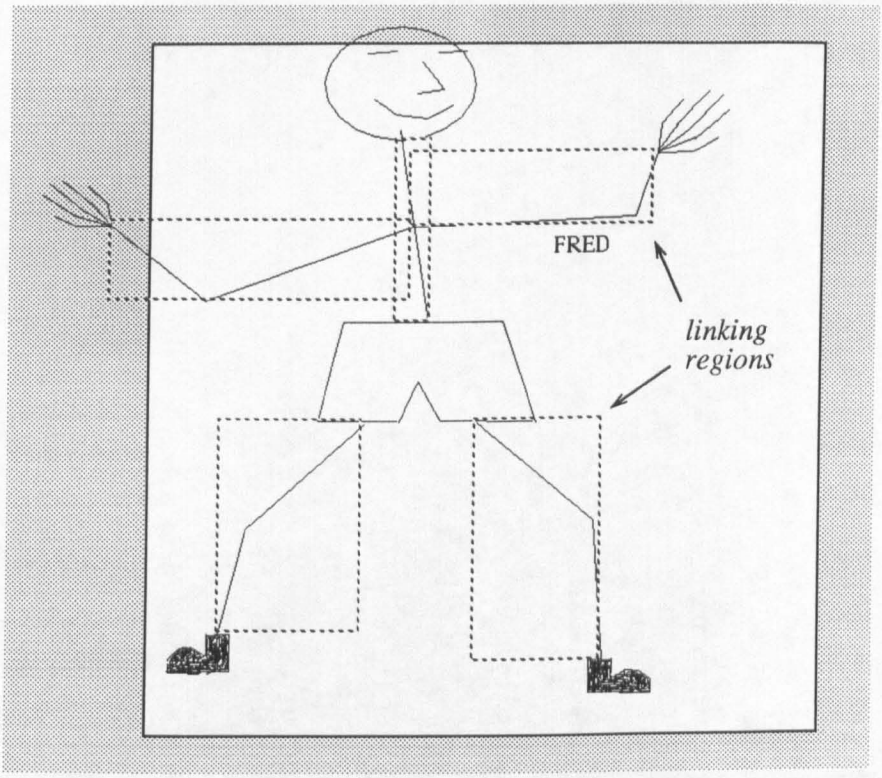


medium state

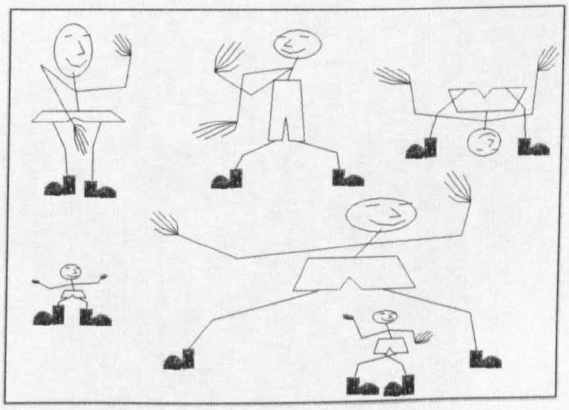


surface presentation

There is no limit to the number of links between regions. This stick man, for example, is constructed with linking regions. Note that since linking regions are in other respects normal regions, they themselves can be linked to:

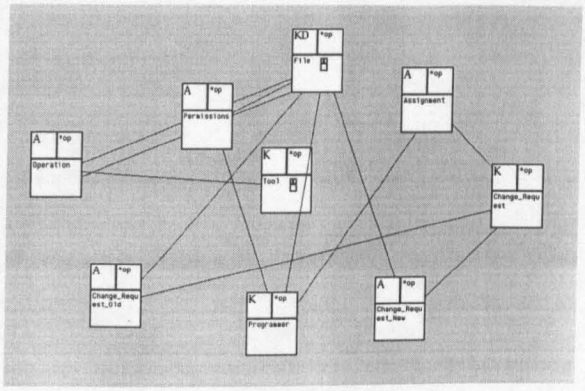


Whatever size or position changes are made to the linked regions, the linking regions and their content or subtrees are maintained by *Presenter* such that on the surface the logical connectivity continues to be represented. There is thus no limit to the manipulability of links:

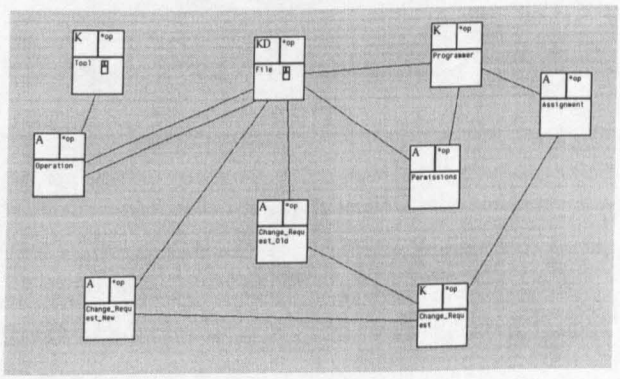


Links can model a wide range of connected diagrams, and have proved useful in software engineering diagrammatic notations, and in graphical presentations of

databases (see Appendix I). Here for example is a database schema from the Aspect project:



and the same schema rearranged by the user through Surface Interaction:



Once created, the linking structure persists with the region tree. The application may even use it as a small networked database, since operations are provided to query and traverse the links.

7.3.3. Persistence

Since the surface is not dependent on applications, its objects may persist for longer than the applications which create them. Surface objects may also be created prior to their use in an application, for example by *Presenter*'s interactive editor DoubleView [Holmes89] (see also Appendix I).

In order to commit surface objects to longer term storage, *Presenter* provides operations *save* and *load*, which write a tree structure (from any region root) out to file, and read it back. All aspects of the tree, that is, structure, content, and proper-

ties, are saved, so that upon loading back the tree and its behaviour are indistinguishable from when it was saved.

Of course, when surface objects are save and reloaded, then their internal addresses, as represented in C **region* pointers, will have changed. It is therefore necessary to associate persistently some *name* with regions in order to *re*-identify them. An operation, *namereg()*, is provided to name regions with an arbitrary length character string. Only regions of interest to the application, for example those that will expect input events, need be named. After loading the tree, the application can recover the current internal identifier from the name, using an operation *getreg()*.

7.3.4. Hardcopy

Presenter allows any region tree or subtree to be converted into a PostScript file for printing on a laser printer. The PostScript is generated within *Presenter* from its internal representation of the sizes, positions, and content of the regions. Bitmap regions are converted into PostScript *image* format, but otherwise regions are drawn explicitly using the PostScript primitives. A loose mapping is made between the SUN fonts in *Presenter*, and the fonts available within PostScript. This can sometime leads to visible differences between the face or format of text on the screen and on the hardcopy. Only transparent bitmap regions and some highlighting cannot be hardcopied successfully, due to PostScript's restrictive opaque paint model.

7.4. Refinements

This section examines details of the implementation that are present in the formal specification, but which have had to be modified or extended for practical or design reasons.

7.4.1. Presentation

The specification simply defines a presentation function *present*, which maps region trees to the display. In practice, considerations of efficiency mean that only the minimum amount of redrawing should be performed on the screen. Thus when one region out of many moves, the screen should only be redrawn at that region's old and new positions.

In addition, it may be that an application wishes to construct a complex object, consisting of a number of nested and overlapping regions. If each region were drawn on the screen as it was created, the result would be unacceptable flicker.

For these reasons, *Presenter* provides an operation *present()*, which redraws a region and its subtree, if any. Thus a programmer should always choose to *present* the minimum subtree which covers the changes he has made to the regions, and should use it *after* all the current changes he wished to make have been completed. *Presenter* always adopts this strategy when managing Surface Interaction.

This very slight consideration is offset by the convenience of *present*. In order to move an object around the screen, the application programmer need only change the position of its region, and then call *present* on the region. All redrawing, both in order to remove the object from its old position and create it at the new position, at whatever visual depth or location on the screen, is handled automatically within *Presenter*. There are no exceptions to this. Thus the programmer never has to be aware of the sometimes very complex redrawing procedures, for example for regions that might be transparent and overlapped by other transparent and clipping regions.

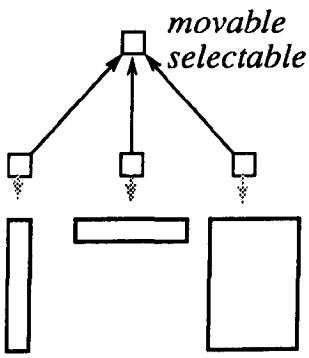
7.4.2. Selection

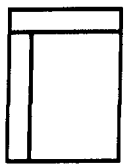
By design, *Presenter* maintains a single text selection and a single region selection orthogonally. That is, the selected text does not have to be in the selected region. As soon as another region or text is selected, the previous selection is cancelled. As outlined above, selected text can be cut or copied.

Presenter always searches up the tree from a hit leaf region for the first *selectable* region. Selection of non-leaf regions can thus be achieved when leaf regions are set non-*selectable*, or by multiple clicking. An editable region always registers the first click as a selection of its textual or graphical content. A double click is required to select the editable region itself.

Selectability is also used as a guard on the other operations of Surface Interaction. Thus in order to move a region, it must be both *movable* and *selectable*, and a *move* request takes effect on the first *selectable* and *movable* region up the path from the hit region, unless a higher region has actually been selected. This scheme allows

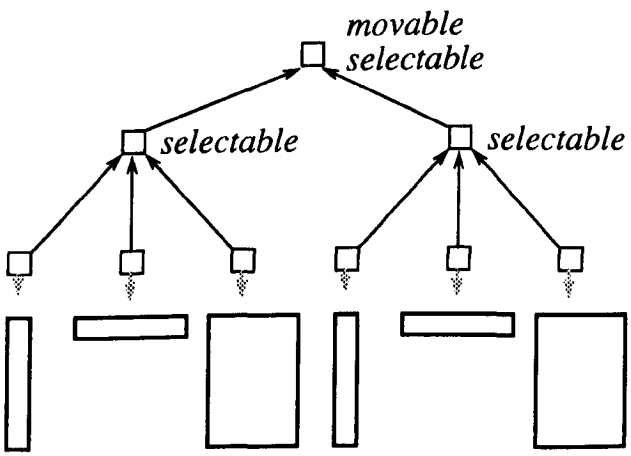
composite objects to be moved as a group without first selecting the common ancestor region:

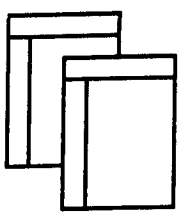




medium state *surface presentation*

By acting as a guard on movement or sizing, however, selectability can also prevent large composite regions, which may need to be moved only exceptionally, from being moved inadvertently because the user has attempted to move a non-movable component. By placing *selectable* on lower regions, a move on the higher object is blocked unless it has first been explicitly selected (by multiple clicking):





medium state *surface presentation*

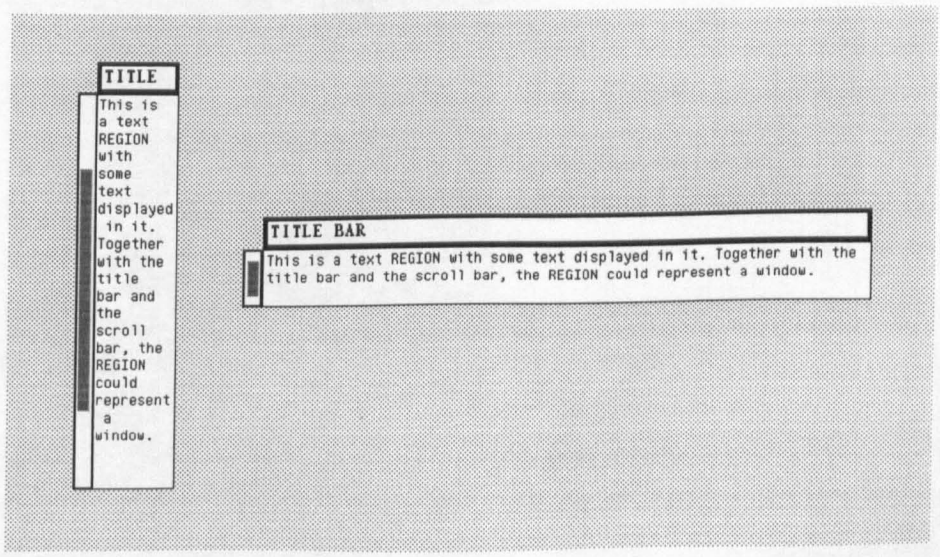
By making the intermediate regions *selectable*, but not *movable*, we block a move request. That is, we prevent situations where the user attempts to move one of the smaller objects, but finds he has unintentionally moved the whole environment. With these attribute settings, the user must explicitly select the higher region first (by clicking up the tree) before being able to move it.

7.4.3. Highlighting

A selected region is highlit by default (another attribute turns highlighting off). A surprising number of application built on top of *Presenter* have needed to extend the default highlighting. Doubleview, for example, requires to highlight two objects at once, one in its view of the medium tree structure, and a corresponding one in its surface view. Many applications also use highlighting to represent the progress of an operation. For example, a print option might be chosen from a menu, which is then highlit. When the operation finishes the highlight is turned off. *Presenter* therefore supplies operations by which the region can be highlit or de-highlit by the application.

7.4.4. Grouping and Scaling

Like other attributes mentioned above, the attributes that affect whether a region changes size or position if an ancestor does have be specialised to have an effect possibly in one dimension only. In scaling a window composed of a text region and a separate region for the title bar, for example, we should expect the title bar to remain the same width as the window as this changes, but to retain its height. Similarly a scroll bar attached to a window conventionally changes height with the window, but remains a constant width. This can be achieved by setting *noscale* specialised to have an effect only horizontally or vertically (in addition, the pivot point of the noscaling region has to be set on the edge that adjoins the window, to prevent overlapping or a gap opening as the parent region contracts or expands):



7.5. Deficiencies

This section outlines ways in which *Presenter* has been found to be deficient, either because it has not fully or effectively implemented the formal specification, or because the specification itself is limited.

7.5.1. Text and Graphics

The major deficiency of *Presenter* as it stands is in its text handling capabilities. While the user can arbitrarily insert, delete, and scroll text, the application cannot. This means that applications that require to implement some specialised form of text editor, for example a structure-oriented editor with templates, cannot easily be catered for.

It is also not possible to embed graphics in text such that the graphics is subject to formatting constraints, nor is it possible to flow text from one region to another. Both of these capabilities would be required for document processing. These issues are taken into account in the alternative design in the next chapter.

The graphics primitives provided in *Presenter* are deliberately simple. It was no part of the design goals to provide a full set of linewidths, spline curves etc., but instead to take these from the available environment. The PostScript primitives for example would form an adequate set. However, a certain amount of work would be needed to provide these interactively.

7.5.2. Input Masks

The principle criterion of Surface Interaction, that some input should be handled autonomously by the user agent within the surface, is only crudely implemented in *Presenter*. As mentioned above, Surface Interaction takes place during object drags, and during most text editing. The only input mask under application control that has been implemented is in fact the choice to report Carriage Return events. Nevertheless, it is easy for applications to ignore input reports which do not concern them, and very few applications have needed to have drag events reported. One exception is an application to implement gesture-based diagram editing, and this required a special version of *Presenter* (actually a simple update). It is again in the area of text manipulation that finer control over the reporting of editing events might be beneficial to specialised applications.

7.5.3. The User Agent

The critical contribution of the *UMA* architecture is the *user agent*, which manages Surface Interaction on behalf of the user. In *Presenter*, the user agent is bound into the code which maintains the surface. This is in fact advisable, since there is only one of each, and the speed of their communication is critical to the usability of the surface (see Section 4.6.3).

However, *Presenter* was written before this architecture was clearly formulated, and although the functionality of the user agent is there, it is not cleanly modularised. As we have seen, some default stylistic bindings must be built into the user agent (like mouse button mappings), and so it would be useful to be able to plug in different user agent modules.

7.5.4. Client-Server Working

In spite of the *UMA* architecture, *Presenter* is in fact bound into the application. Thus only one application is possible at one time, and *Presenter* strictly exhibits external, rather than concurrent control (see Section 2.1.2). Nevertheless, the interface between *Presenter* and the application is simple (the application must provide one routine, *presevent()*, to which *Presenter* passes in an event and the region in which it occurred), and few applications have found it restrictive.

As a separate project, *Presenter* has been split off into a separate server process [Pollard89]. This caused few, if any, problems. A telling test example was a stick man who waved his hand under application control. The user was able concurrently to pick up and move the hand, and while moving it continued to wave. This was because commands both from the user agent and the application were being interleaved in the medium. The satisfying result was that this caused no screen drawing synchronisation problems, even though no changes were made to the *Presenter* code (the interprocess communication was implemented by a special application simply bound in to the standard *Presenter*).

7.5.5. SunView Dependence

While *Presenter* runs within a SunView window as a matter of convenience, it was deliberately implemented to be as independent of this environment as possible. Thus apart from the initial request for a drawing space from SunView, it only uses

raster operations, the SunView line drawing primitive, and the SunView fonts. It does not use any other SunView facilities like text or subwindowing constructs. As a result, it has recently been fairly successfully ported to the X environment [Bramley90]. This port necessarily adopted the same strategy as *Presenter* itself in SunView. That is, it simply asked X for a drawable and constructed regions out of raster operations on this. It would not be possible to implement *Presenter* regions one-to-one with X windows, simply because of the geometric and visualisation restrictions of the latter (see Chapter 5).

This port encountered some problems in matching fonts, and in moving bitmap regions. There was also considerable loss of performance. This was perhaps to be expected as *Presenter* provides services at a similar level to X - i.e. 'mechanism rather than policy'.

7.5.6. Memory Limitations

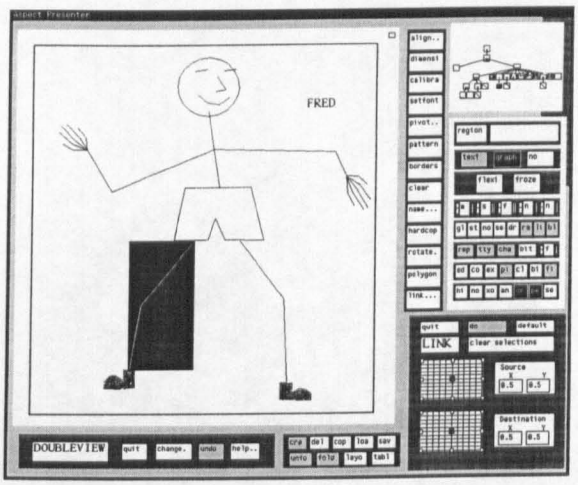
The high manipulation efficiency of *Presenter* relies on offscreen bitmaps for each content region. Clearly there will be a limit on these depending on the available memory. However, even though many applications have been written on *Presenter*, some employing hundreds of regions (see Appendix I), memory limitations have not been a problem (except with applications which themselves use large amounts of memory).

Presenter implements some memory-saving devices: the bitmaps of regions which are wholly obscured or wholly outside the screen area are freed, and then recreated as soon as the regions become visible - there is no noticeable delay in doing this. Nevertheless, an application domain which might have memory problems using *Presenter* is one which uses objects which are very much larger than the screen, for example large maps or engineering diagrams which the user needs to window around. It would probably be necessary to break these up into smaller regions and have the application manage their selective display.

It should be emphasised, however, that this is only an implementation problem. Logically, regions can be any size and have any content. Nor does the formal specification of the surface model imply an implementation in terms of persistent bitmaps. It would equally be possible to regenerate the image from the logical content each time it needed to be redisplayed. The trade-off is between speed of manipulation and memory use, and this is a pragmatic issue.

7.5.7. Manipulation Efficiency

The speed and smoothness with which any *Presenter* object can be moved about the screen is often noted. For example, although a rubber box is the default outline during the moving or sizing of a region, for leaf objects or simple composite objects it is much more visually effective to move or size the object smoothly by setting the attributes *glide* or *stretch* on them. With either mechanism, however, *Presenter* interfaces are unusual in that (if the application allows it) *any* discrete component can be moved and sized by the user in Surface Interaction. For example, here is an interactive rearrangement of the default DoubleView surface (compare the picture in Appendix I):



There is, however, a disadvantage to this power in the implementation of *Presenter*. The designers of some complex objects, for example menus, dialogue boxes, or windows do not expect their components to be rearranged. The power to do so is therefore redundant (it can be switched off simply by clearing the *movable* and *sizable* attributes). Even so, *Presenter* treats the component regions as separate objects, and if the *whole* object needs to be moved or sized, then the default recursive drawing of its tree structure comes into play, such that its components actually move one at a time. With simple objects this is not noticeable, but in more complex objects it can be slower and visually diverting.

What is clearly needed is some way to *coalesce* those groups of regions which can be thought of as single objects, such that while retaining their logical identity, in display representation they simply form part of a larger bitmap which contains the whole composite object. In standard window systems this is in fact the usual scheme, simply because child objects are constrained to be within the extent of their

parent. These systems therefore lack the manipulability noted above. The problem of implementing such a scheme here is that the surface is more geometrically general: child regions may be *outside* the area of their parent. They therefore, in the general case, cannot utilise the same bitmap.

7.5.8. Constraints

Very deliberately, *Presenter* cannot be *programmed* (see Section 2.4.4). Similarly, the constraints it provides, such as *noscale* and *contain*, are specified simply by using a set of attributes, rather than by *declaring* explicitly that some relationship hold. While the existing set of attribute constraints have in fact proved to be surprisingly powerful for generating a wide range of interfaces and diagrammatic notations, it is *always* possible to think of a more specific constraint. For example, an application may want a region's movement constrained to some diagonal path, rather then horizontal or vertical.

A possible extension to *Presenter*, and to the formal specification of the surface, may be to allow the declaration of constraints over the properties of its objects. However, there are a number of classes of constraint power (see Section 5.2.4), of varying computability, and the appropriate class to be supplied at the surface is not obvious. A danger in pursuing this power at the surface is that it becomes *over-powered* [Took90b], and it becomes difficult for the programmer to access its simpler functionality. It seems appropriate to exploit the *UMA* architecture to allow applications which wish to impose exotic constraints to do so directly, by taking over surface management from the user agent.

Nevertheless, some extensions to the constraint set in *Presenter* seem desirable. It would be nice to be able to constrain the aspect ratio of a region to some constant irrespective of size, and possibly also to impose some maximum and minimum sizes on regions.

7.5.9. Higher Constructs

Higher level constructs, such as windows, menus, scroll bars etc., are not provided by *Presenter* (nor in its formal specification), to avoid stylistic or domain bindings in the surface. However, it is clear that in many cases these are useful, and relatively general, constructs. An obvious extension to *Presenter* would be the provi-

sion of a toolkit, along conventional lines, constructed out of regions. Such a toolkit has in fact been built [Jones89].

However, it is not appropriate for this sort of toolkit to be bound in to the surface, for the reasons above. Nevertheless, there may exist constructs at a higher level than the region tree which are yet sufficiently general that they could form part of the surface. These constructs will essentially consist of a *structure* orthogonal to the tree, and a set of *constraints* that are specialised to the structure. Linking is one example of such a higher level construct (see above). Another, which is not provided in *Presenter*, is *tables*. Tabular arrangements underlie the presentation of many visual structures, for example matrices, dialogue boxes, menus, scroll bars, documents, and (tiled) windows. The constraints and structures necessary in a tabular construct are examined informally in Chapter 8.

7.6. Issues

This Chapter concludes with some (slightly) more general issues which arise out of the design and implementation of *Presenter*. Issues of wider generality are examined at the end of Chapter 4 with respect to the *UMA* architecture.

7.6.1. Empty Leaves

What representation on the surface should empty leaves have? It seems logical that they should be invisible, on the basis that only *content* is displayable. Early versions of *Presenter* implemented this. However, it is sometimes useful to be able to select and manipulate a leaf region, even though it has no content. DoubleView, for example, allows region trees to be constructed top down - the user therefore may well want to change the size or position of a region *before* he adds further child regions or content.

In early versions of DoubleView therefore, it was necessary to create temporary visible content, manipulate the region, and then delete the content in order to add further regions. To avoid this, later versions of *Presenter* made empty leaf regions visible as empty white (or black) areas, so that they could be selected and manipulated.

This is a practical solution, but produces some unwanted side effects. For example, an application may wish to temporarily remove some subtree of regions from the screen by *cut*ting the root from its parent. However, if the parent has no other children, such that after cutting it becomes an (empty) leaf, then the cut regions will not simply disappear from the screen, but will be replaced by a blank area of the size of the parent. In most cases this is not wanted.

7.6.2. Access to Text

Providing the application access to text is problematic. An application might need such access to perform pattern searches through text, followed by subsequent text replacement. What is required are unique, persistent text identifiers. Simple character values are clearly not unique, since there is likely to be more than one instance of a character in a text. Identifying a character by its logical position in a text sequence is not persistent, since prior characters may be inserted or deleted. Similarly, identifying a character by physical position in terms of *(line number, character number)* coordinates is not persistent, since the text may be reformatted. Dix [Dix88a] examines these problems with his notion of 'pointer spaces'. The only likely solution is to identify characters by their internal addresses (or some protected mapping to this from a name space), just as regions are identified.

More problematic is the persistence of *format*. Should text *lines*, for example, be considered objects? This is the case on a vt100 terminal, which provides operations to insert and delete lines, and to move the cursor up and down a line. However, on a surface like *Presenter*'s, in which the user can reformat text arbitrarily simply by changing the size of its region, and do this independently of the application through Surface Interaction, lines clearly have little persistence.

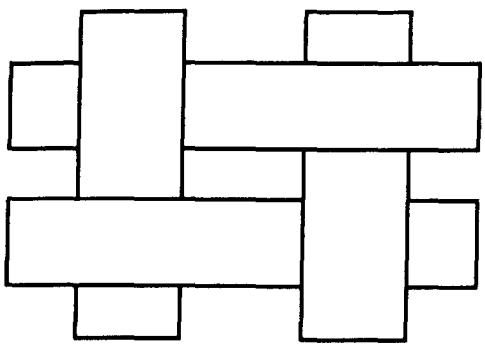
7.6.3. Rectangularity

All *Presenter* regions are rectangular, and aligned with the screen axes. This restriction is clearly driven by implementation considerations. It would not be insurmountable to provide both polygonal regions and rotation (as specified), or even other types of graphical transformation such as shearing. NeWS, for example, provides these through its PostScript imaging mechanism. Non-rectangular regions like boxes with rounded corners can in fact play a large part in giving a distinctive 'look and feel' to interfaces.

The problems arise in deciding the effects of these transformations on contained text. Should text always be horizontal, or should it rotate with a region? Should rotated text be editable? This is possible, for example, in MacDraw. If a region is non-rectangular, should the contained text be formatted to its borders with the flexibility of, for example, TeX, whilst retaining interactive efficiency? There may even be some applications, for example advertising graphics, where text should be subject to the same (possibly non-affine) transformations as the region which contains it. These requirements are more easily stated than implemented!

7.6.4. Dimensionality

The formal model in Chapter 6, and the implementation of *Presenter*, are two-dimensional. There is a crude third dimension in the ordering of the tree, which maps to display layering, such that systems like this are sometimes called $2^1/_2$ D. The objects themselves cannot have depth, and thus one object cannot be interleaved with another, and groups like the following cannot naturally be represented:



It is a question whether higher dimensionality needs to be provided in systems which are primarily intended for *schematic* rather than *scenic* modelling (see Chapter 1), since one of the motivations for diagrammatic and other schematic representations is that they abstract away from real world imagery. Nevertheless, the author has seen a database prototype from the American organisation MCC which provides three-dimensional *schematic* database representations through which user can navigate as if he were flying a plane.

Three dimensional objects can also be used simply to provide graphic realism to essentially two dimensional interfaces. The X toolkit *Motif* provides pseudo-solid objects through skillful use of shading, but the author has also seen examples of but-

ton boxes that rotate in 3D space to reveal further buttons underneath. Obviously these representations are limited in their interactive efficiency by current hardware, but it would be unwise to reject the third dimension simply on the basis of these arguments.

Upgrading the surface to provide three dimensional objects is not simply a matter of adding another dimension to the specification. Parameters such as viewpoint, projection plane, lighting, and surface shading have then also to be specified. If the design of two dimensional interfaces is difficult, then certainly providing a third dimension will not solve any problems.

7.7. Conclusions

This Chapter has described the implementation of *Presenter*, a particular model for Surface Interaction. The description has concentrated on the areas in which Presenter differs from the formal specification. The Chapter attempts to account for these differences either in terms of faulty design, or implementation constraints. It highlights those areas where the ideal design is unavoidably compromised by implementation considerations.

Chapter 8

Future Work: An Alternative Model for the Surface

An alternative model for the surface medium has been designed and specified formally (although not yet implemented). This has been partly in response to perceived deficiencies in the *Presenter* model (see Chapter 7). However, this new surface model is not just a fix of *Presenter* (which has its own domain of use). The major motivation for this alternative model is to encompass a new paradigm for the surface and Surface Interaction: the *document*. The distinguishing feature of the document, as here perceived, is to arbitrarily nest graphics and text, such that the top level structure may be either text or graphics. In a document, that is, both text and graphics are first class objects. In *Presenter*, on the other hand, text is a second class object.

Certainly an application interface in any particular state can be thought of as a document (it may be hardcopied, for example). A document conversely may be interactive: hypertext systems [Conklin87] are beginning to realise this possibility. It seems highly desirable to be able to present application interfaces with the same aesthetic considerations that go into the production of (static) documents, as well as to be able to display documents containing active components which can be directly manipulated or invoked by the reader.

We here give the basic design requirements for this new model, and describe it informally. In addition to this fundamental motivation, our design goals attempt to include a range of capabilities that are both abstract and powerful, and have not been provided before in user interfaces. Specifically, we wish to capture, in addition to the fundamental properties of textual and graphical content and geometry:

• generic ordering: textual objects will occur in some sequence, and (two dimensional) graphical objects will have some overlapping priority.

• generic composition: composite objects can be constructed out of parts, and these out of other parts, and so on. A graphical object may be constructed from smaller subparts which are held together in some geometric relationship, while a textual object such as a document may be composed of chapters, sections, and paragraphs. This requires at least a tree structure on objects.

• replication: essentially the same object can be presented in different locations, such that modifying one instance modifies all instances. This requires a hierarchical structure on objects.

• inheritance: rendering or visualisation properties may be inherited from other objects, and these objects may or may not form part of the composite structure. In this way, inherited properties may be modified independently of the structure of the objects.

• nesting: text may be framed by graphical areas (as in a window), or graphics may be embedded in text (as in an illustration). The nesting should be unbounded, since diagrams in text may contain labels or further text, and so on.

• persistence: we do not wish the structure simply to be an execution trace, as in graphical languages like PHIGS and PostScript, but to have a permanent existence as a single linked entity which can be incrementally updated, copied, or saved. Persistence also means that the same structure may be projected through a number of different roots simultaneously. Since the nodes are persistently identifiable, further structures, for example a hypertext network, may be supported on top.

A motivation for the new formal specification is to reduce the model to the *minimum* number of constructs capable of satisfying the above requirements. The surface model presented here has a single generic hierarchical structure, which is instantiated with only two node types, covering text and graphics. It is difficult to see how this could be reduced further.

8.1. An Informal Description of the Model.

All surface objects have a geometry. That is, they have a shape, and there exist operations on them which result in certain spatial transformations. On this basis, two types of surface geometry can be distinguished:

- Cartesian geometry, which has shapes such as lines and circles, and operations such as translation and rotation

- *textual* geometry, which has characters as shapes, and operations such as insertion and deletion.

It is, however, possible for a character to be manipulated graphically, as well as for a set of line segments to form a character or a diagram embedded in text and so be manipulated textually. The distinction is thus between geometries rather than objects: objects are textual or graphical depending on the geometry applied.

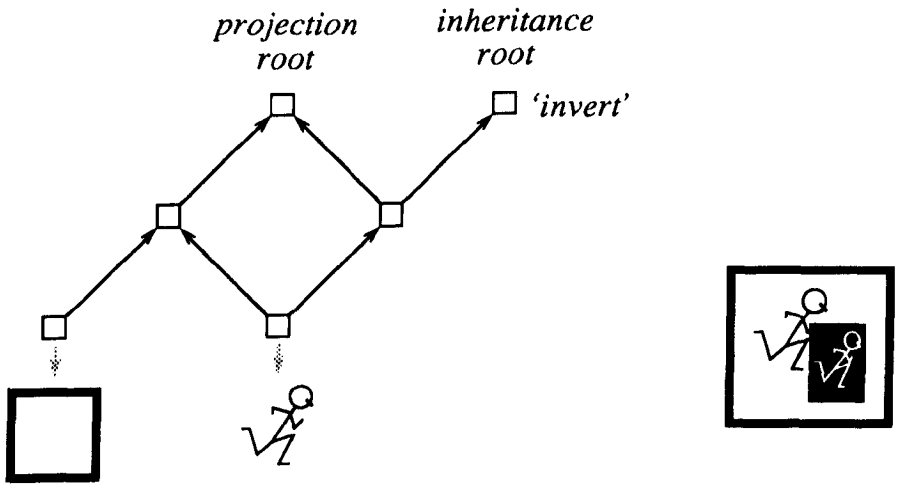
The model therefore consists of two fundamental constructs, *REGION* and *BLOCK*. *REGION*s provide a graphical, or Cartesian, coordinate space; *BLOCK*s provide a textual coordinate space. The model also contains one common structure, the *hierarchy*. A hierarchy is a directed, acyclic graph in which nodes may have more than one parent. Both *REGION*s and *BLOCK*s can be built up into hierarchies. *REGION* hierarchies model graphics, *BLOCK* hierarchies model text. In order to do this, these abstract structures are *loaded* with content and properties, either graphical or textual. In addition, in order to provide the required nesting, *BLOCK* hierarchies may be *framed* by *REGION*s, and *REGION* hierarchies may be *embedded* in *BLOCK*s.

Only leaf nodes in the hierarchies may have visible content. Leaf *REGION*s may contain graphical images, leaf *BLOCK*s may contain characters. In this way it is possible to manipulate any object independently of any other, while at the same time it is possible to manipulate groups of objects by manipulating interior nodes in the hierarchy.

Graphical properties consist primarily of geometry, that is, the size and position of each node with respect to any of its parents, and attributes which affect the visualisation of the content of the leaf nodes. Visualisation attributes, such as transparency and inversion, are inherited down the hierarchy, with lower settings

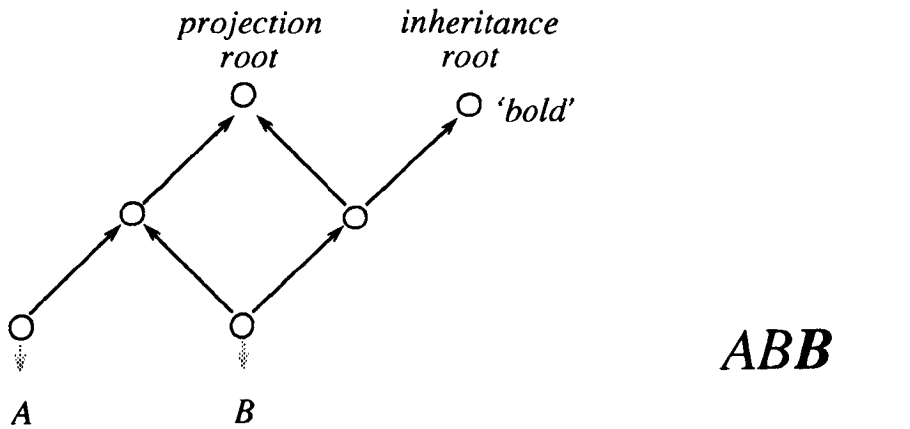
overriding those from above. Textual properties consist of rendering attributes which are similarly inheritable, and affect the pointsize, font, face etc. of characters.

In order to present these loaded structures on the display surface, the hierarchies are *projected* from some root. In the projections, it is the sequence of *paths* from the root to the leaves which are important in generating discrete surface images. Attributes may be inherited not only down the projection paths, but also down any intersecting paths from a group of inheritance roots. The net result is a set of visible surface objects. A typical *REGION* structure and its projection can be illustrated:



structure ***surface projection***

and so can a similar *BLOCK* structure:



structure ***surface projection***

These examples illustrate:

• ordering: the logical sequence of paths maps either to graphical overlapping or to textual sequencing. Clearly the textual example is trivial, but it is simply a matter of expanding the structure in width and depth to model a large document with a complex structure (ignoring implementation limitations).

• composition: the graphical 'window' illustrated here is composed of a background rectangle and two images of a man; similarly, the characters 'ABB' form a group. Modifications to the properties of ancestor nodes affect all descendant leaf children, which strengthens the user's perception of these as a group. For example, if a common ancestor is moved, then all surface images related to that ancestor move as a group.

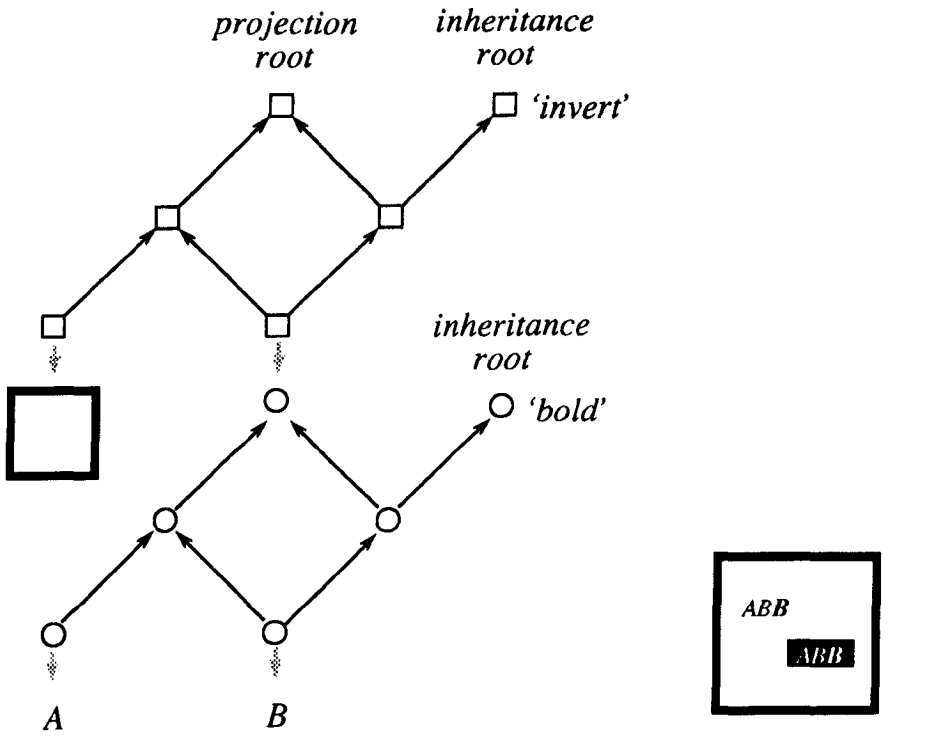
• replication: the image of the man, and the character 'B' are shared between two different paths. In the case of the graphics, the two replicated men are at different sizes and positions determined by *REGION*s which are *not* common in the two paths. In the case of the text, the replicated 'B's are distinct simply through the ordering of the paths. Clearly, if the logical image of the man, or the logical 'B', were changed, then *both* replications would change on the surface.

• inheritance: the attribute 'invert', and the attribute 'bold', are inherited into *one* of the paths of the replicated graphics or text. In general, by exploiting the hierarchical structure, attributes can be inherited independently of the projection structure, so that, for example, a number of structurally dispersed objects in the projection (for example, all chapter titles in a document) could inherit the same characteristics, and this could be modified in one operation.

• persistence: the structure can be projected from different nodes simultaneously. For example, a document can have a global projection, displaying all the text, but there may also be a 'contents page' projection which is simply linked in to the chapter headings, or to the subtitles of figures. These could then be projected onto separated pages.

• nesting: the model also allows text *BLOCK*s to be framed in a sequence of graphical *REGION*s, and *REGION*s to be embedded in *BLOCK*s, to any depth. However, no cycles must be formed. This is capable of modelling a very wide

class of document and application interface layouts. We can extend the illustration to show an example of this mechanism:

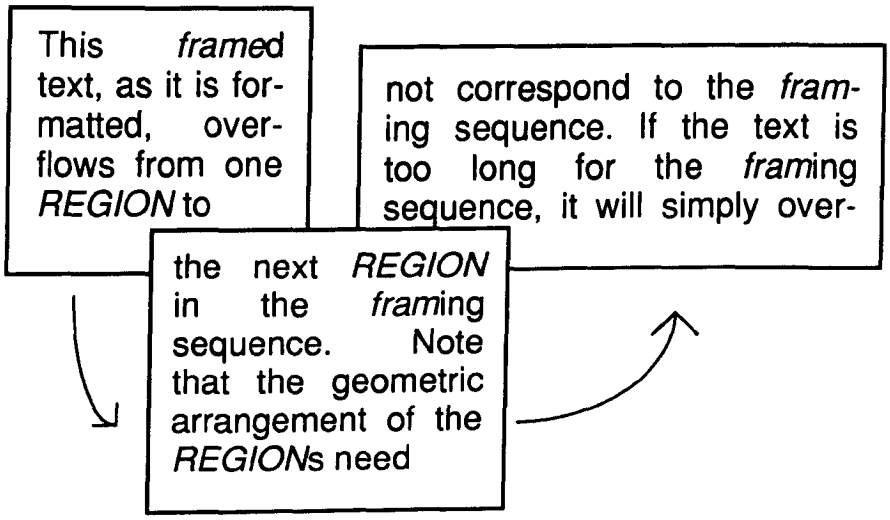


structure ***surface projection***

8.1.1. Framing

The framing of *BLOCK*s of *TEXT* in *REGION*s is extended by specifying an order to the framing *REGION*s. Using this structure it will be possible to model the

presentation of a large *BLOCK* of text over a number of *REGION*s, such that as text overflows from one, it runs on into the next:

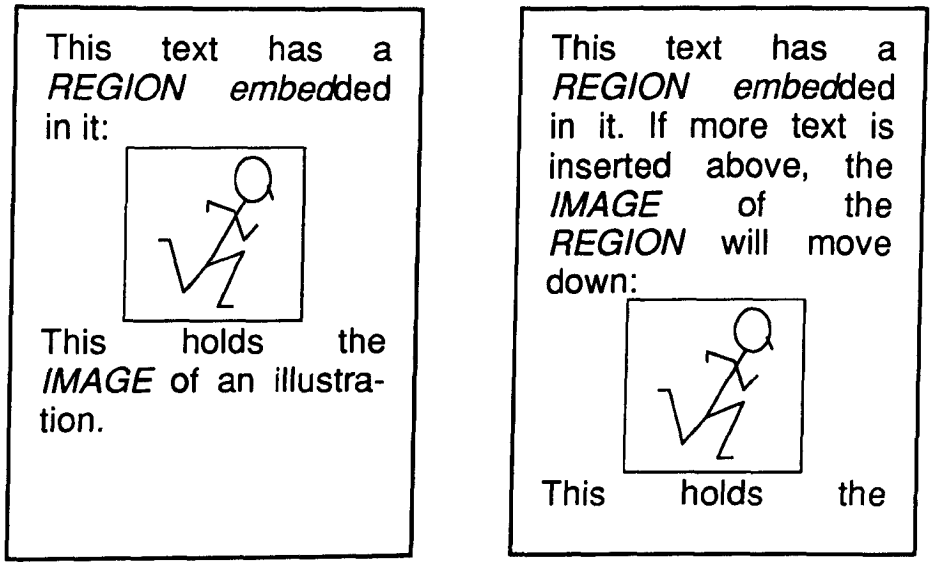


In this way, pages, and areas of text on pages, can be presented. The framing order is ideally orthogonal to the ordering of the *REGION*s on their hierarchy (as in the example).

8.1.2. Embedding

The model allows graphics in the form of *REGION* hierarchies to be embedded in *BLOCK*s of text. In this way we indicate that textual rather than Cartesian geome-

try applies to these graphical objects, so that as the text flows under formatting, they maintain their position in the text sequence:

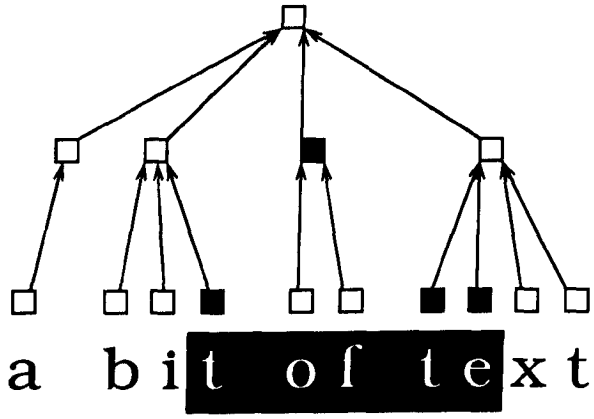


8.1.3. Multiple Inheritance

Presenter provides a form of single inheritance (although we have been careful to call this *propagation* in the formal specification). Single inheritance has two major limitations:

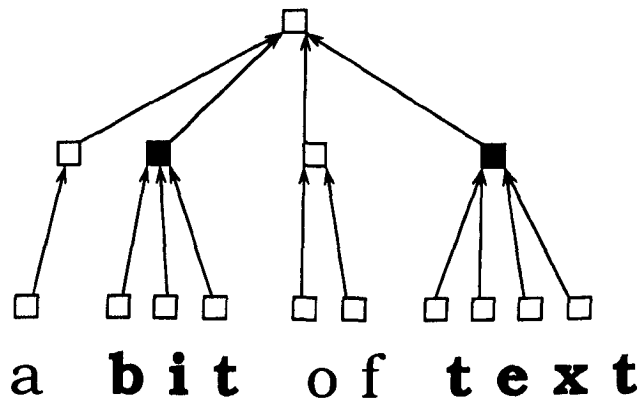
- The scope of the inheritance is tied to the structure of the object. For example, if we used a tree structure to model a document, with interior *NODE*s representing chapters, sections, paragraphs etc., then it would only be possible to set rendering attributes on elements of this structure. If we wished to model the highlighting of *arbitrary* selections of the document by using attributes (and

why not?), then in many cases we would have to make multiple attribute settings to get the effect of a single highlight. For example:



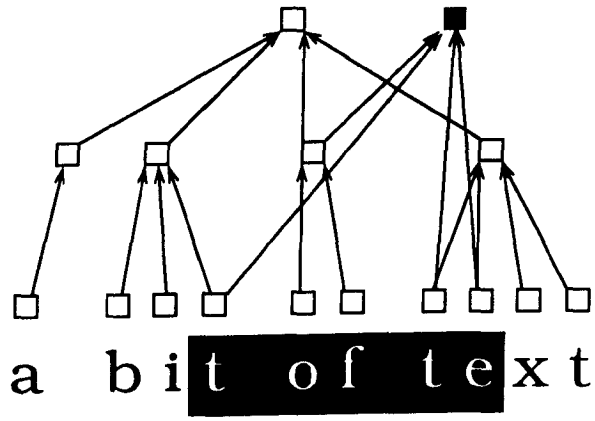
The filled squares represent the setting of a highlighting attribute, and show the minimum number of settings that could made to highlight this section of text.

- It is not possible to *abstract* inheritance from the structure. That is, it is not possible to bring together similar attribute settings scattered through the structure, in order, for example, to perform global editing on them:

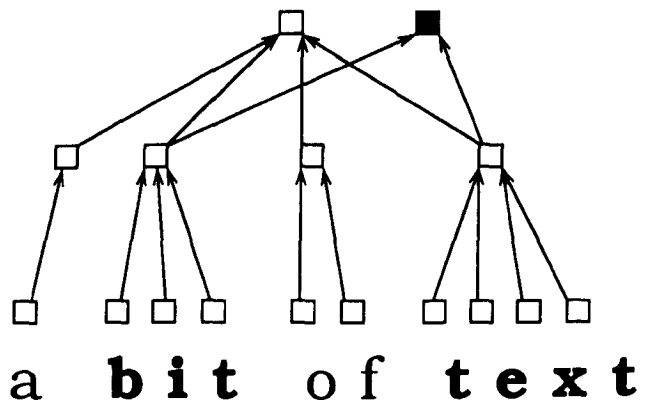


In this example the filled squares represent the setting of a *bold* attribute. In order to remove the *bold* from *bit* and *text*, two separate modifications to the structure must be performed.

We can exploit the hierarchical structure of the new model to remove both of these limitations, so long as we allow *NODE*s to inherit attributes along more than one path. This enables us to build structures like the following:



In this way, arbitrary sections of the leaf sequence can inherit from a single attribute setting.



In this way, the scattered emboldening can be modified from a single *NODE*.

Note that these extra arcs do not have any effect on the projection from the original root. We call the extra *NODE*s *inheritance roots*. In the general case, there may be many inheritance roots for any one projection root. We call a projection through a particular group of inheritance roots a *presentation* of the hierarchy. It is important to note that there is no difference between projection roots and inheritance roots, except that, in a particular presentation of the hierarchy, we simply choose to project via one *NODE* and inherit extra attributes multiply from the *pathseq*s of some

other *NODE*s. Another presentation of the same hierarchy could just as easily reverse these interpretations.

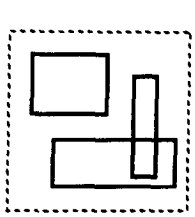
Clearly, there may be conflicts if opposing attribute settings are inherited down different paths into a single projection path. We therefore prioritise the inheritance paths by sequencing them. That is, a particular *presentation* of the hierarchy consists of a projection root, and a sequence of inheritance roots. (in the formal specification we conveniently represent a presentation as a single sequence, in which the projection root forms the head). Inheritance *later* in the sequence has priority.

8.1.4. Constraints

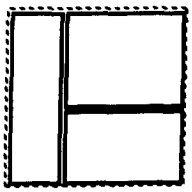
Constraints have proved powerful in *Presenter*. The new model proposes some new graphical constraints. The children of a *REGION* with:

- *contain* set are all contained within the extent of the *REGION*.
- *tile* set are tiled in the extent of the *REGION*.
- *exclude* set are excluded from *each other*.
- *align* set to *row* are aligned in a row (along their pivots).
- *align* set to *column* are aligned in a column (along their pivots).

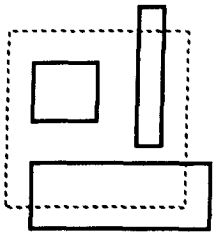
These constraints can be illustrated (the dotted square is the extent of the parent *REGION*):



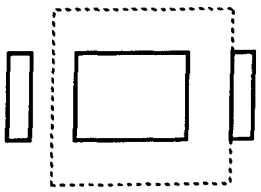
contain



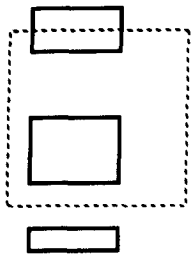
tile



exclude



align = row



align = column

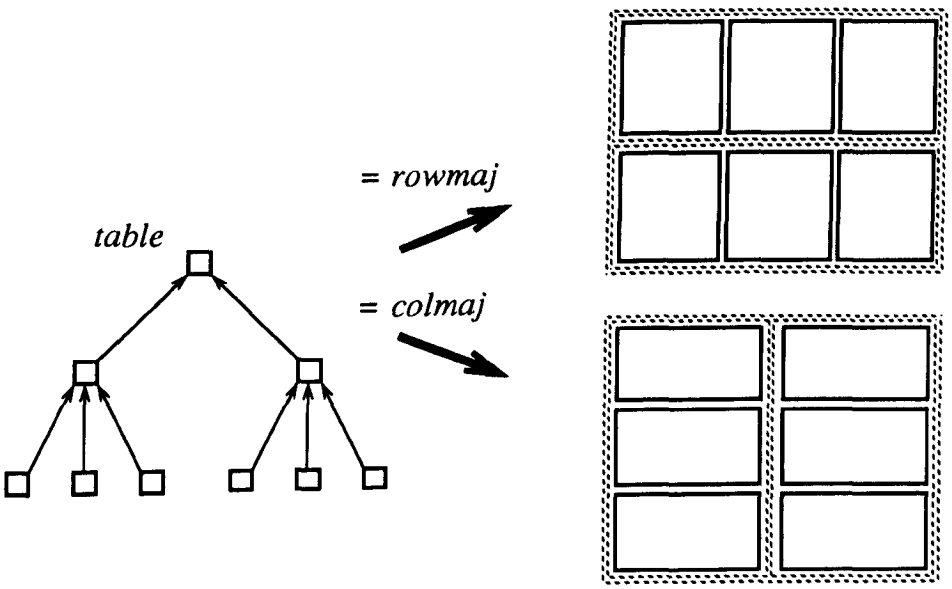
Note (semi-formally) that:

- ¬ (containment ⇒ exclusion) (contained *REGION*s may overlap)
- ¬ (exclusion ⇒ containment) (excluded *REGION*s may extend beyond the parent)
- tiling ⇒ (containment ∧ exclusion) (but not the reverse implication)
- alignment ⇒ exclusion
- ¬ (alignment ⇒ containment)

8.1.5. Tables

Chapter 7 noted that *Presenter* was deficient in higher constructs, with the exception of its *linking* facilities. We briefly give a design for a tabling construct in the new model. Tabling is a graphical construct, although of course tables may contain framed text. As noted in Chapter 7, a higher construct requires both an orthogonal structure, and a set of constraints on its objects.

We imagine (there is no implementation) tables to be established by setting a *table* attribute on a *REGION*. The *table* attribute may take two values: row major (*rowmaj*) and column major (*colmaj*). Tabular presentation of *REGION*s necessitates a structure consisting of two-level subhierarchies:



Depending on the value of the *table* attribute, such hierarchies are presented either with the first level subdivided into columns and the second level into rows (row major), or vice versa (column major). The table structure itself may be anywhere in the hierarchy. That is, the grandchildren may themselves contain further *REGION*s, and may even be *tabled* themselves.

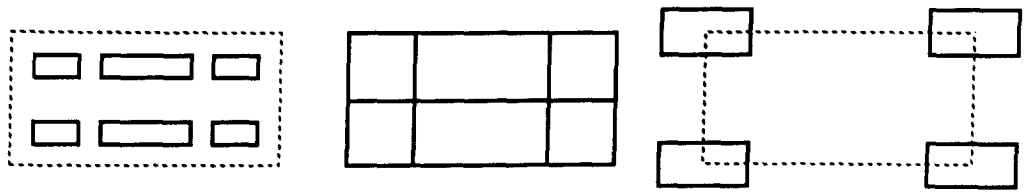
The major structural *constraint* is that the numbers of grandchildren must be equal. Without this, the grandchildren would not be able to possess the dual row/column membership which is the essence of a table. The geometric constraints are that, in a row major table, the children of the *tabled REGION* must be in a column and each of the sequences of grandchildren must be in a row. In addition, all the *i*th grandchildren must be in a column. A column major table has a similar definition.

Note that this definition does not say anything about tiling or containment. That is, semi-formally

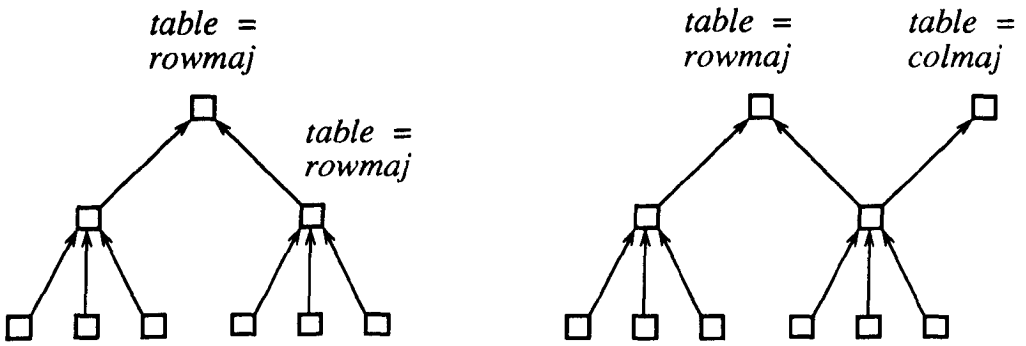
- $\neg$ (tabling $\Rightarrow$ containment)
- $\neg$ (tabling $\Rightarrow$ tiling)

There are equally no constraints on widths in rows or heights in columns.

Tables may therefore look like any of these with respect to their parent *REGION*:



There are clearly opportunities here for constraint conflict, for example if a parent and child are both set to be row major tables, or if a grandchild is part of two different tables via dual ancestry on the hierarchy:



While tabling appears to be a powerful and useful construct, it is not clear how many special cases like these might exist, especially under user manipulation in Surface Interaction. This will affect how readily tabling can be incorporated into the hierarchical structure

8.2. Conclusions

This Chapter has described informally a new, unimplemented model for the surface which uses the notion of a *document* as the motivating paradigm. Documents can nest text and graphics. When installed within a *UMA* architecture, this document model for the surface can be used to present a very wide range of interfaces, from desktop publishing to arcade games.

Clearly, while the model is capable of modelling a complex interrelated structure, it is just as possible to use these features sparingly, or not at all, in any particular document or application. The model may easily be restricted to text or graphics only, to a tree structure rather than a hierarchy, or to a single level modelling a single sequence of text or graphics. The power of the model is that the more complex features do not require any extra constructs.

Chapter 9

Conclusions

9.1. The Thesis of Surface Interaction

This Thesis presents an alternative architecture and models for the separation of interface and application. The motivation has been twofold:

- Separation, in its full sense of both abstraction and late binding, allows design, programming and computation costs to be factored from applications. This improves productivity in the construction and running of applications and their interactive interfaces. Separation also allows some measure of application-independent user control over the interface at run time. This may improve usability. It at least allows *interactive* design of the interface, and brings the roles of user and interface designer closer.

- Separation has conventionally concentrated on abstracting either *dialogue* or *devices* from the application. We have tried to show (Chapters 2 and 3) that dialogue abstraction (in *linguistic* architectures) has not been as successful as hoped, because dialogue must be bound early to the functionality it drives. On the other hand, device abstraction in the newer *agent* architectures, object-oriented or not, typically binds *presentation* early to application functionality by specialising generic interface tools with application semantics, and thus also is not optimally separable.

The Thesis therefore attempts an alternative separation based on the abstraction not of the *form* of interaction (dialogue syntax), nor of the *devices* of interaction, but simply of the *medium* of interaction. We have shown that:

- It is possible to *abstract* the medium, since we have given a formal model of an Object whose encapsulated state expresses generic textual and graphical structure and content. This is generated entirely by well defined operations provided externally, while the internal representations and implementation, in particular the presentation mapping to the display, remain hidden.

- It is possible to *separate* the medium, since the Object state can be persistent, and a dedicated *user agent* can be constructed to invoke medium operations on behalf of the user, independently of the application. This also preserves *directness* at the surface, since the objects of the medium can be the targets not only of output application manipulation, but also, via the *pick* function, of input user manipulation.

The combination of medium and user agent is called a *surface*. The *UMA* architecture defines, without placing any constraints on style or semantics, the possible communications between the user and the application via the surface. The *UMA* architecture is capable of supporting either fully application-determined objects at the surface (as is usually the case in window-managed environments) or allowing a degree of user-determination of surface objects during *Surface Interaction.*

Surface Interaction is the fullest exploitation of the benefits of such an architecture. In Surface Interaction, not only is the surface *medium* separated from the application as a data abstraction, but also surface *control*, as managed by the user agent, is separated as a *control* abstraction. The control abstraction of Surface Interaction *extends* the traces of the raw application functionality. That is, under Surface Interaction, the user is able to *directly* manipulate the surface objects without necessarily involving the application. This is conceptually possible since we have shown (in Chapter 4) that there may exist surface operations whose invocation is entirely independent of application semantics. In contrast, the control abstraction provided by dialogue management in UIMSs seeks to *restrict* the traces of application functionality. We have shown (in Chapter 3) that this is incompatible with full separation.

Surface Interaction is effective as a user interface service because applications are thereby relieved of much of the fine grain of surface control, without losing expressive power over the surface. This is because an application can either rely on Surface Interaction to allow the user to manipulate the surface objects (possibly under application-specified constraints), or it can capture input events and manage the (*deep*) interaction itself by sending commands to the surface.

Surface Interaction thus *factors* a significant portion of the task of providing highly interactive interfaces, as well as allowing the user and the interface designer a great deal of expressive control over the appearance of the surface. In addition, Surface Interaction provides the interface designer with *user-level objects*, rather than with constructive screen operations like RasterOp. This forces the interface designer to consider end-user rather than device capabilities.

The implementation of the Surface Interaction system *Presenter*, and the range of applications which it has supported (see Appendix I), demonstrate that Surface Interaction is practically, as well as theoretically, possible.

9.1.1. Surface Models

The degree to which the separation of Surface from deep Interaction is actually of benefit to application writers and users is critically dependent on the quality of the surface model. A command-line editor, for example, implements a simple textual surface in this definition, but it clearly provides little support for anything other than batch-oriented applications (whose interaction is limited to the single invocation of their command).

At the very least, a general surface model should integrate graphics and text [Sproull83 p.146]. The Thesis has examined in detail the surface models provided by window management, graphics standards, and text standards. It proposes a *document* model as the most powerful and general surface model.

9.2. Contributions of the Thesis

The major contributions of the Thesis are:

- To recognise that *separation* of interface and application is fundamental to providing economy with quality in the production of applications, and to recognise the importance of both abstraction and late binding in this separation.

- To recognise the importance of *directness* in manipulable interfaces, and to see as the essential feature of directness the use of the *same* surface object for both output and input.

- To categorise existing interface systems in terms of *dialogue* and *device* abstractions, and to show precisely the limitations of these as architectures for user interface separation.

- To show the possibility of an alternative separation of the interface *medium* by modelling this not just as a view projection from application state, but as a semantic domain in its own right.

- To give the user direct control over the medium through a dedicated user agent, thus forming a surface.

- To show the limitations of conventional window management as a model for the surface.

- To propose a new data abstraction (model) for the surface which integrates document and interface presentation without stylistic bias, and which takes account of current requirements and standards in text and graphics.

- To propose an architecture (*UMA*) for separating the surface as an Object with its own state and operations, and for allowing both the user (via the user agent) and the application direct access to this surface.

- To recognise the possibility and usefulness of Surface Interaction in not only enabling directness, but in factoring the management of direct manipulation interfaces from applications.

Although *UMA* is a 3-component architecture, it is quite different from the MVC or Seeheim architectures (see Chapter 2). These, and all other 3-component architectures that the author is aware of, tend to reduce to a *linguistic* model of lexical (presentation), syntactic (dialogue or control), and semantic (application) components. In contrast, the *UMA* architecture *partitions* the semantics of applications into surface and deep components. That is, aspects of both control and state are split between the application and the surface. At the same time there is no explicit dialogue component: the behaviour of the user agent (see Chapter 4) is independent of application functionality and does not change.

The *UMA* architecture also differs from 3-component architectures which are intended to be used recursively as device abstractions (for example PAC), in that the surface is integrated into a single component. Since all these architectures are

typically bound in with their applications (via libraries or classes), they cannot provide the persistence and application independence of Surface Interaction.

The *UMA* architecture is closer to the models of [Ciccarelli85] and [Scofield85], in that these propose separate presentation and application *databases*, and have no explicit dialogue component. However, in both these architectures there is a close mapping between the presentation and application structures, which is maintained by an *editing* component. This component must therefore know about both the presentation and application data types, and these are clearly bound early here. The *UMA* architecture, on the other hand, assumes nothing about the application state or data types, and only communicates with the application using *messages*, which are open to any application interpretation.

The *UMA* architecture also differs from more abstract models of interaction such as PIE [Dix88a], which see interaction as a function between sequences of input and semantic effects. These abstract models allow properties such as predictability or observability to be expressed over dialogue. In contrast, *UMA* is less expressive but more pragmatic, taking account of separation between a number of applications and the interface, and the need for directness.

There are, however, grounds for claiming that the *UMA* architecture is equally generally applicable. That is, it can form a reference model against which the architectures of all interactive systems can be evaluated. This is based on the observation that interactive systems consist of users, a task semantics, and some medium of communication. For example, the *UMA* architecture could support a linguistic division if the user agent were expanded to include a dialogue interpreter, and the surface were restricted to a presentation or view mapping. We do not explore this further.

9.3. Limits of Surface Interaction

It is not at all the case that Surface Interaction guarantees good user interfaces. It is just as possible to make a bad interface (in terms of usability etc.) with Surface Interaction as a good one. However, Surface Interaction reduces considerably the turnaround time for iterative design of interactive interfaces, and so reduces the cost of *achieving* a good interface.

One of the central problems of interface design is that of indicating to the user the potential and actual effects of his actions, or, in other words, providing predictability and feedback. This necessitates a mapping between surface and deep interaction. Thus not all interface design issues can be encapsulated in the surface.

There is also an unavoidable trade-off between input device independence and semantic freedom. That is, although we wish to minimise semantic seepage from the application into the surface by passing (some subsequence of) input events (and their *picked* selections) raw to the application, this binds the application to a particular input device configuration. On the other hand, abstracting a common input language from the possible range of input devices inevitably involves the use, even at the low level attempted here, of logical devices which encapsulate their own state and (possibly) output. This reduces the application designer's expressive freedom. However, in contrast to the variety of devices addressed by early mainstream graphics, modern workstations are relatively standardised in their input devices. Even here, though, there are variations which cannot be overlooked, for example in the number of mouse buttons.

Surface Interaction is strictly an enabling platform upon which interfaces can be built. It thus sits most nearly at the level of current base window management systems in that it provides a framework for filtering and routing input and constructs for output. However, its models for both input and output are more integrated and general.

Surface Interaction does not presuppose any top-level interface, such as a desktop. Clearly such interfaces have to be built in order to use Surface Interaction to front an operating system. In addition, Surface Interaction does not exclude the interposition of Toolkit or UIMS layers between the surface and the application. In practice, however, the ease with which the implemented system *Presenter* has been used directly by many applications supports the claim that Surface Interaction is in itself an effective factoring of the application task.

9.4. Postlude

The notion of Surface Interaction has been approached by a few authors. Wills [Wills87b p.10] uses the terms surface and deep, but implies a continuum of functionality. Bennett sees the user interface as "a surface through which data and

control is passed back and forth between computer and user" [Bennett87 p.102]. Draper notes that input and output is "communication via a shared medium" [Draper86].

The emphasis in this Thesis is on the surface as a basis for *separating* interface and applications. Lantz [Lantz84 p.29] recognises the usefulness of 'high-level short-circuiting' between input and output (that is, providing a control abstraction at the surface). He also notes the consequent need for the manager to be supplied with high-level information about the model to be displayed, rather than just an image of the model. However, he does not distinguish clearly between surface objects and model objects.

The conclusions of the 1982 UIMS conference [Thomas83 p.24] also places high-level linkage ('image modification with application concurrence') at the head of its list of UIMS capabilities, but is similarly vague about the characteristics of surface objects. In the 1987 UIMS conference, Lantz et al proposed a model for the 'Workstation Agent' which was free of application semantics [Lantz87b p.89] and retained a persistent representation for its objects [Lantz87b p.91]. However, again no precise details for the surface objects were given, and the representations suggested were procedural (segmented or structured display files) rather than declarative.

Perhaps the last word should go to Sutherland, who thought that

> *one of the largest untapped fields for application of Sketchpad is as an input program for other computation programs.*
>
> [Sutherland63 p.343]

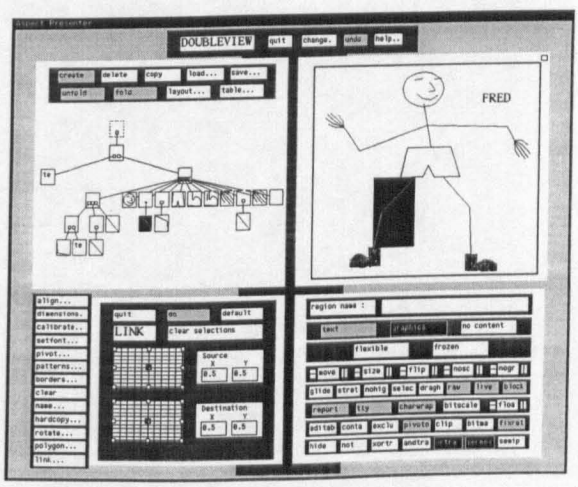
Sutherland perceived Sketchpad, that is, as a medium for Surface Interaction.

Appendix I

Presenter Applications

We here give a brief illustration of the range of applications which have already been written to *Presenter*, and the range of styles in which they have been implemented. Although Surface Interaction does not preclude UIMS or Toolkit layers between the surface and the application, in practice *Presenter* has tended to be used directly by applications because of the ease with which interfaces can be prototyped and fine tuned. These applications have covered a broad range of disciplines.

An important application has been *Presenter*'s interactive interface editor DoubleView [Holmes89]. Here is a typical surface during the editing of Fred (a linking region is highlit):

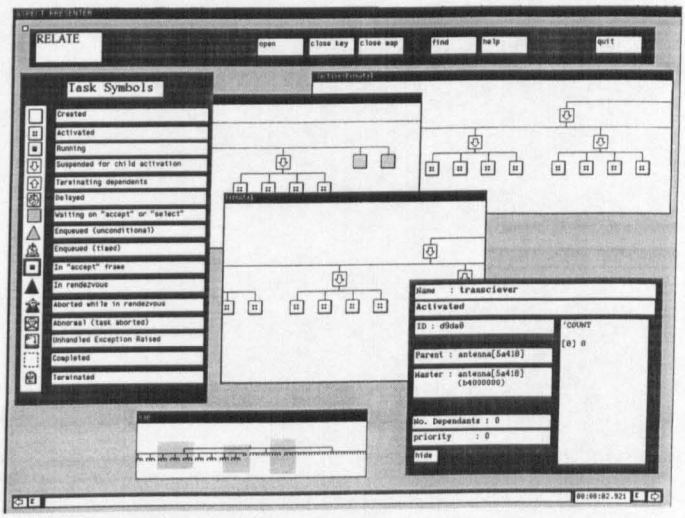


The left hand view shows the logical region structure, while the right shows the user's view. The component regions can be accessed through either representation. The bottom right box gives the attribute state for the selected region, while the left box invokes *Presenter* operations. Each of the distinguishable objects here,

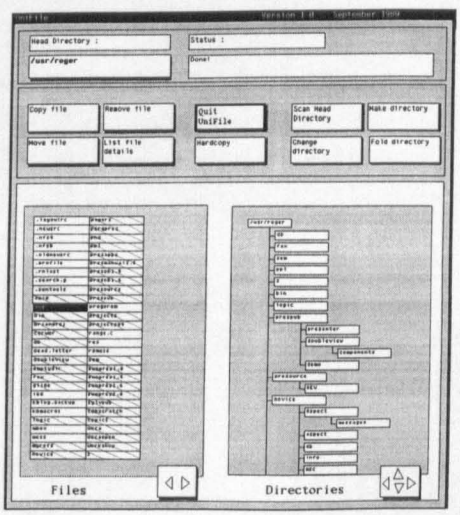
including each node on the tree, are separate regions. To illustrate the manipulability of *Presenter* interfaces, compare this configuration of the surface with the one in Section 7.5.7, which was derived from it simply by direct manipulation of the objects.

The following are a selection of further application interfaces created on *Presenter*:

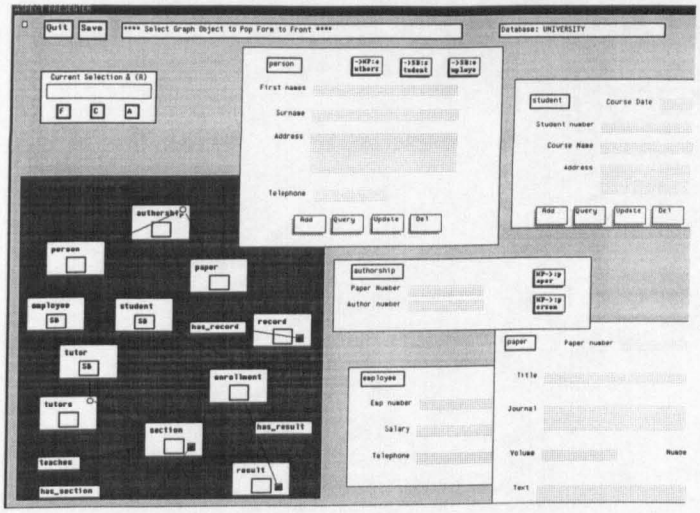
- An Ada debugger [Cobbett89]:



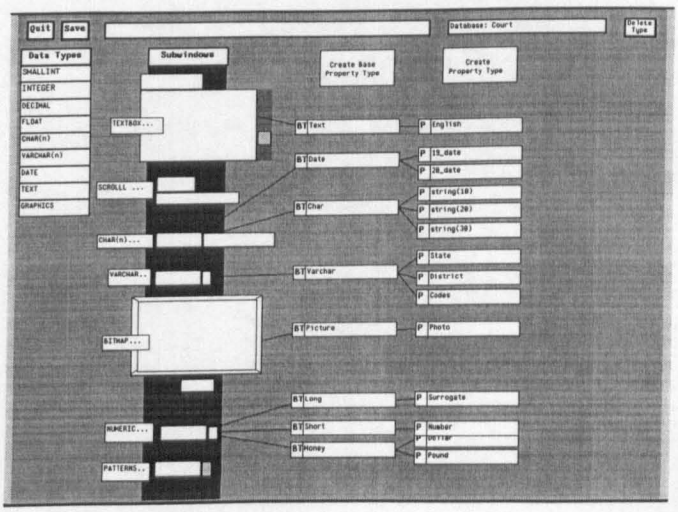
- A UNIX file manipulation system [Davies89]:



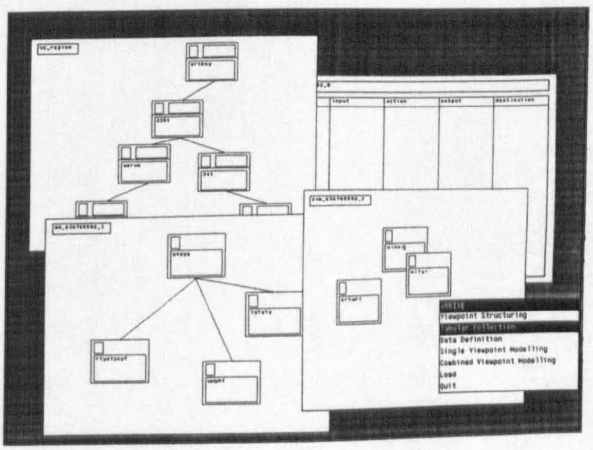
• An interactive database management system [Daly89]:



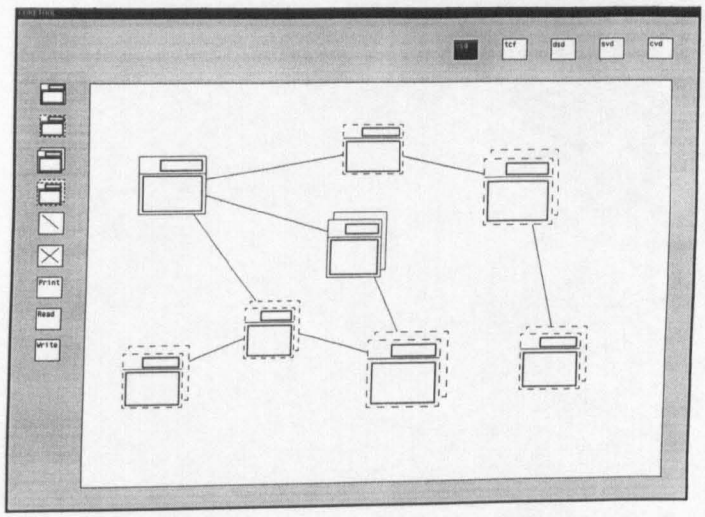
This is a second screen from the same application:



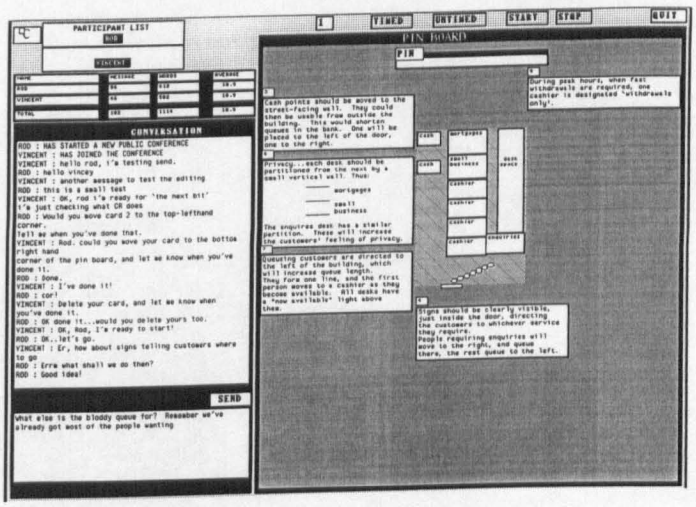
• A software engineering diagramming application:



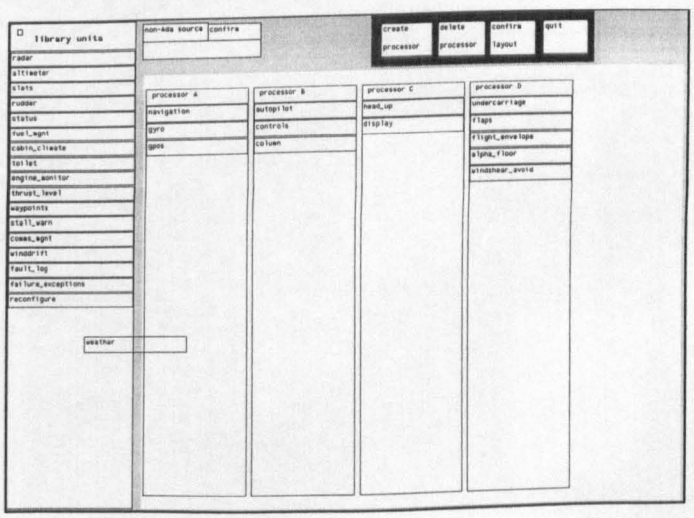
• Another software engineering diagramming application:



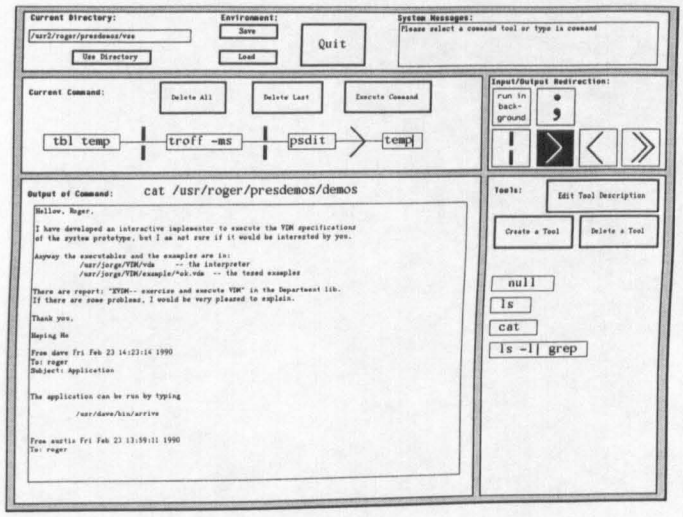
• An Interactive conferencing system [McCarthy90]:



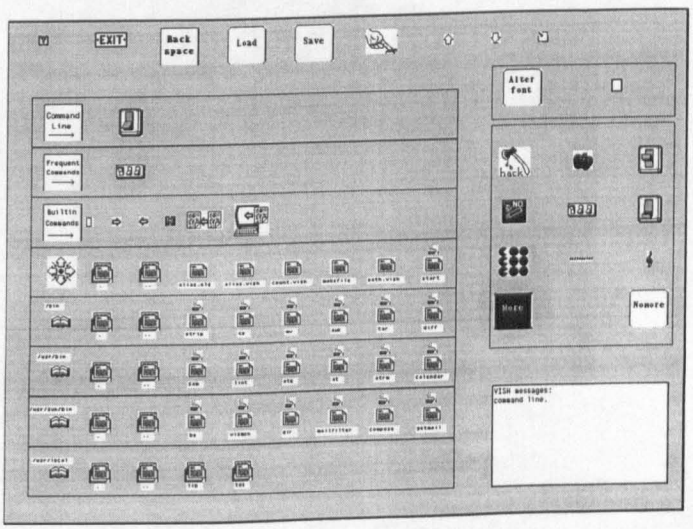
• An Ada program distribution tool [Hutcheon90]:



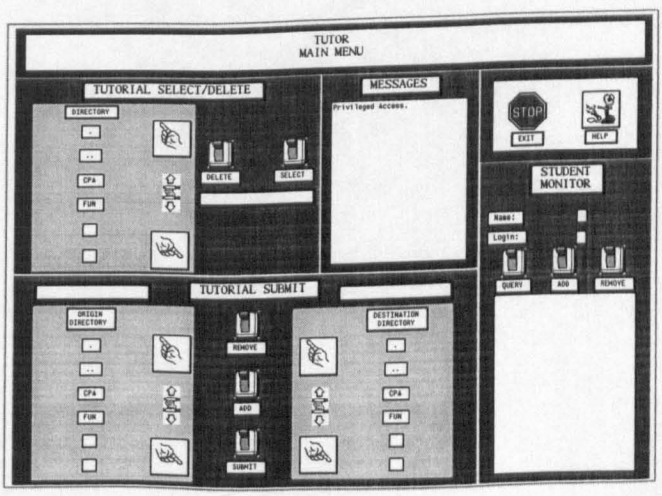
• A visual Shell [Stoddart90]:



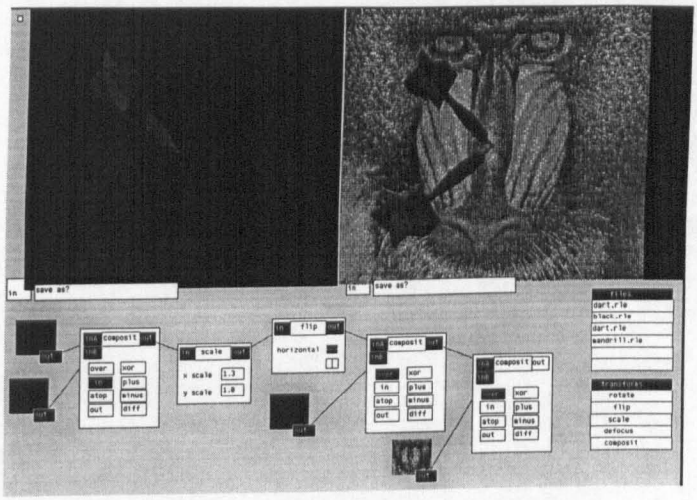
• Another visual Shell [Chapman90]:



• A tutorial system [Fisher90]:



• A direct manipulation image processing application {Smith90]:



Presenter has also provided interfaces for a Mascot 3 Paintbox [Whiteley88], and a CORE requirements method workstation [BAe88]. Current projects at York are using *Presenter* as the front end for a Prolog graphics database, and for image processing applications. A project in Hull is using *Presenter* as the interface to the Lisp-based object-oriented system Loops, in order to prototype interfaces to photo-copiers.

Appendix II

Generic Functions

We define two generic functions used in Chapter 6.

First, *insert* inserts an element into a sequence of elements, *after* a specified element. If *after* is the special value $\perp$, then the new element is prepended to the sequence:

[X]

$$insert: seq\ X \rightarrow X \rightarrow X \rightarrow seq\ X$$

$$\forall\ s, s1, s2: seq\ X;\ after, i: X \mid s = s1 \frown s2 \bullet$$
$$after = last\ s1 \Leftrightarrow insert\ i\ s\ after = s1 \frown \langle i \rangle \frown s2$$
$$after = \perp \Leftrightarrow insert\ i\ s\ after = \langle i \rangle \frown s$$

Secondly, *remove* removes an element from a sequence, and returns the shortened sequence. We define this recursively:

[X]

$$remove: seq\ X \rightarrow X \rightarrow seq\ X$$

$$remove\ \langle\rangle\ r = \langle\rangle$$
$$remove\ (x:xs)\ r = xs \qquad \textbf{if}\ r = x$$
$$remove\ (x:xs)\ r = x \frown remove\ xs\ r \qquad \textbf{otherwise}$$

Appendix III

Glossary

The following is a summary of those operations of the Z mathematical language, and others, which are used in this Thesis. The Z schema language is presented in Chapter 3.

Sets and Types

$S \cup T$ $(S \cap T)$	the union (intersection) of the sets S and T.
$\cup \{S, T\}$ $(\cap \{S, T\})$	the distributed union (intersection) of the set of sets $\{S, T\}$, i.e the union (intersection) of all their elements.
$S \subseteq T$	the set S is a subset of the set T.
$S \subset T$	the set S is a proper subset of the set T, i.e. some elements of T are not in S.
$s \in S$	s is an element, or member, of the set S.
$\mathbf{P}\, S$	the powerset of the set S, i.e. the set of all subsets of S.
$\# S$	the cardinality of the set S, i.e. the number of elements in S.
$\mathbb{R}$	the set of all real numbers, e.g. $\pi \in \mathbb{R}$
$\mathbb{N}$	the set of all natural numbers, i.e. $\{0, 1, 2, ...\}$.

$\mathbb{N}^+$	the set of all strictly positive natural numbers, i.e. $\{1, 2, 3, ...\}$.
$m \,..\, n$	the set of natural numbers between and including m and n.
$\varnothing$	the empty set.
$s: S$	a type declaration - s is a variable of type S, i.e. s can take on any value in the set S.
$T ::= c_1 \mid ... \mid c_n \mid t_1 \langle\langle T_1 \rangle\rangle \mid ... \mid t_m \langle\langle T_m \rangle\rangle$	a free type definition. The c_i are constants, the t_j are tags, or constructors, and the T_j are type expressions. The tags serve to distinguish different values which may be constructed from the same T_js. The type definition may consist of zero or more constants and tagged types, with the restriction that they must all be disjoint. The T_js may contain T, so that the definition can be recursive, e.g. binary integer trees: $T ::= leaf \langle\langle \mathbb{N} \rangle\rangle \mid branch \langle\langle T \times T \rangle\rangle$
$\{declaration \mid predicate \bullet term\}$	set comprehension, e.g. $\{x: \mathbb{N} \mid x < 4 \bullet x^2\}$ (the set of squares of x where x is a natural number less than 4). If *term* is the same as *declaration*, then *term* can be omitted, e.g. $\{x: \mathbb{N} \mid x < 4\}$ (the set of natural numbers less than 4).

Pairs and Tuples

(s, t)	an ordered pair

$s \mapsto t$	a single mapping (also called *maplet*) between s and t. $s \mapsto t$ is exactly the same as (s, t).
$S \times T$	the Cartesian (or cross) product of the sets S and T, i.e. the set of all ordered pairs such that the first element of the pair is a member of S and the second is a member of T, *i.e.* $\{s: S; t: T \bullet (s, t)\}$.
$T_1 \times T_2 \times \ldots \times T_n$	the set of n-tuples whose components have the types $T_1, T_2, \ldots T_n$, in that order.

Relations and Functions

$S \leftrightarrow T$	the set of all relations whose source is the set S and whose target is the set T. $S \leftrightarrow T = \mathbb{P}(S \times T)$
$s\ R\ t$	the relation R holds between s and t, i.e. $s\ R\ t \Leftrightarrow (s, t) \in R$
$R\ (\!\mid S \mid\!)$	the image of the set S through the relation R. $R\ (\!\mid S \mid\!) = \{t: T \mid (\exists s: S \bullet s\ R\ t)\}$.
$Q \fatsemi R$	the forward composition of the relations Q and R. $Q \fatsemi R = \{(u, w) \mid (\exists v \bullet u\ Q\ v \wedge v\ R\ w)\}$.
R^{-1}	the inverse of the relation R. $s\ R\ t \Leftrightarrow t\ R^{-1}\ s$
$id\ [S]$	the identity relation on the set S. $id\ [S] = \{s: S \bullet (s, s)\}$
$dom\ R$	the domain of relation R, i.e. the set of first components of all the pairs in R. $dom\ R = \{(s, t): R \bullet s\}$

$ran\ R$

the range of relation R, i.e the set of second components of all the pairs in R.

$ran\ R = \{(s, t): R \bullet t\}$

$S \pfun T$

the set of partial functions from the set S to the set T.

$S \pfun T \subseteq S \rel T$.

$S \pfun T = \{R: S \rel T \mid R^{-1} \fatsemi R \subseteq id\ [T]\}$

$S \fun T$

the set of total functions from the set S to the set T, i.e. in which the domain of the function is the same as its source.

$S \fun T = \{f: S \pfun T \mid dom\ f = S\}$

$S \pinj T$

the set of partial injections from the set S to the set T, i.e. in which each value in the domain maps to a different value in the range.

$S \pinj T \subseteq S \pfun T$

$\forall f: S \pinj T \bullet \bigcap \{s: S \bullet \{f\ (s)\}\} = \varnothing$

$\wedge f^{-1} \subseteq T \pinj S$

$S \domres R$

domain restriction: restrict relation R to those mappings whose first elements are in the set S.

$s\ (S \domres R)\ t \Leftrightarrow s\ R\ t \wedge s \in S$

$S \domsub R$

domain subtraction: subtract all mappings from the relation R whose first elements are in the set S.

$s\ (S \domsub R)\ t \Leftrightarrow s\ R\ t \wedge s \notin S$

R^*

transitive closure of relation R.

$R^* = \bigcup \{n: \mathbb{N} \bullet R^{\,n}\}$

R^+

non-reflexive transitive closure of relation R.

$R^+ = \bigcup \{n: \mathbb{N}^+ \bullet R^{\,n}\}$

λ	a function abstraction, e.g. the square function is $\lambda x \bullet x^2$
$f(a), f\,a$	application of function f to argument(s) a. Brackets are omitted where the sense is clear, or where a may itself be a function.
$f \oplus g$	override function f with function g, i.e. the resultant function has all the mappings in g, plus any mappings in f whose domain is not defined in g. $f \oplus g = g \cup ((dom\ g) \lhd f)$

Sequences

$seq\ T$	the set of all sequences of elements of type T. Sequences are thought of as functions from initial segments of the positive natural numbers to elements, i.e. $seq\ T = \{f: \mathbb{N}^+ \pfun T;\ n: \mathbb{N}^+ \bullet dom\ f = 1\ ..\ n\}$
$iseq\ T$	the set of all injective sequence of type T, i.e. those whose elements are not repeated. $iseq\ T = \{f: \mathbb{N}^+ \pinj T;\ n: \mathbb{N}^+ \bullet dom\ f = 1\ ..\ n\}$
$e{:}es$	a general sequence template for use in pattern matching, i.e. a sequence is an element e at the head of a sequence of elements es.
$<>$	the empty sequence.
$S \frown T$	sequence S concatentated with sequence T, e.g. $<a, b> \frown <c, d> = <a, b, c, d>$
$head\ (S)$	returns the first element in sequence S, i.e. $head\ (S) = S(1)$. $head\ (e{:}es) = e$.

tail (S)	returns sequence *S* less its head, i.e. $tail\ (S) = \lambda\ n: 1\ ..\ \#S - 1 \bullet S\ (n + 1)$ $tail\ (e:es) = es$
last (S)	returns the last element in sequence *S*, i.e. $last\ (S) = S(\#S)$
front (S)	returns sequence *S* less its last element, i.e. $front\ (S) = \{\#S\} \lhd S$
S **in** *T*	*S* is a contiguous subsequence of *T*, e.g. $\langle b, c \rangle$ **in** $\langle a, b, c, d \rangle$

Processes (CSP)

$P = e \rightarrow Q$	*P* is a process which engages in event *e* (the *guard*) followed by the events of process *Q*. The definition may be recursive, e.g. $P = e \rightarrow P$ defines a process *P* which engages in an infinite sequence of events *e*.
$c!v$	an output event consisting of a communication of value *v* on channel *c*.
$c?v$	an input event consisting of a communication of value *v* on channel *c*.
αP	the alphabet of process *P*, i.e. the set of all events in which it is able to engage.
traces (P)	the set of sequences of events in which *P* can engage.
P **sat** *S*	process *P* satisfies specification *S*, i.e. $\forall\ tr \bullet tr \in traces(P) \Rightarrow S$
$P = (e \rightarrow Q \mid g \rightarrow R)$	*P* is a (deterministic) process which engages either in the event *e* followed by the events of *Q*, or the (different) event *g* followed by the events in *R*. The choice between these two

courses of action is determined by what events are offered in P's environment. If both events e and g are offered simultaneously, then the choice is arbitrary. P cannot refuse to engage in any event that is offered and that it is ready to engage in.

$P = (e \rightarrow Q) \sqcap (g \rightarrow R)$

P is a non-deterministic process which will engage either in the event e followed by the events of Q, or the event g followed by the events in R. The choice between these two courses of action is determined by the process itself, that is, it can *refuse* to engage in an offered event it is ready to engage in.

$\mu X: A \cdot F(X)$

process abstraction: X is a process with alphabet A, such that $X = F(X)$, where F returns some guarded process expression involving X.

$P = Q \parallel R$

process P is defined as process Q in parallel with process R. P can engage in any events of Q that are not in the alphabet of R, and in any events of R that are not in the alphabet of Q, but events in the alphabets of both Q and R require the simultaneous participation of both Q and R. This is the mechanism for process communication.

Metasymbols

$I == C$

an abbreviation definition, i.e. identifier I is bound to constant expression C. I is global (its scope is the rest of the specification document).

$I ::= T$

a free type definition, i.e. identifier I is bound to type T (see above).

References

Abowd89 G. Abowd, J. Bowen, A. Dix, M. Harrison, R. Took, User Interface Languages: a survey of existing methods, Oxford University Programming Research Group, Tech. Rep. No. PRG-TR-5-89, May 1989.

Abowd90 G. Abowd, Agents: Communicating Interactive Processes, in *Proc. Interact '90 - Human-Computer Interaction*, August 1990.

Achugbue81 J. O. Achugbue, On the Line Breaking Problem in Text Formatting, *ACM SIGPLAN Notices*, **16**(6), pp. 117-122, June 1981.

Acquah82 J. Acquah, J. Foley, J. Sibert, P. Wenner, A Conceptual Model of Raster Graphics Systems, *ACM Computer Graphics*, **16**(3), pp. 321-328, July 1982.

Adobe87 Adobe Systems Inc., PostScript Language Reference Manual, Addison-Wesley, 1987.

Agha85 G. Agha, C. Hewitt, Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism, The Artificial Intelligence Laboratory, MIT, A. I. Memo No. 865, October 1985.

Agha86 G. A. Agha, Actors, MIT Press, 1986.

Akscyn88 R. Akscyn, E. Yoder, D. McCracken, The Data Model is the Heart of Interface Design, in *Proc. CHI '88*, pp. 115-120, ACM, Washington, May 1988.

Alexander87 H. Alexander, Formally-Based Tools and Techniques for Human-Computer Dialogues (PhD Thesis), University of Stirling, 1987.

Allen81 T. Allen, R. Nix, A. Perlis, PEN: A Hierarchical Document Editor, *ACM SIGPLAN Notices*, **16**(6), pp. 74-81, June 1981.

Alty84 J. L. Alty, Use of Path Algebras in an Interactive Adaptive Dialogue System, in *Proc Interact '84*, **1**, ed. B. Shackel, pp. 321-324, IFIP, September 1984.

Alty87 J. L. Alty, J. Mullin, The Role of the Dialogue System in a User Interface Management System, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 1007-1012, North-Holland, 1987.

Angell87 I. O. Angell, Y. P. Low, A. R. Warman, Genie-M, A Generator for Multimedia Information Environments, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 129-143, Springer-Verlag, 1987.

Anson80 E. Anson, The Semantics of Graphical input, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 115-126, North-Holland, 1980.

Anson82 E. Anson, The Device Model of Interaction, *ACM Computer Graphics*, **16**(3), pp. 107-114, July 1982.

Appelt88 W. Appelt, R. Carr, G. Richter, The Formal Specification of the Document Structures of the ODA Standard, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 95-108, Cambridge University Press, 1988.

Arnon88 D. Arnon, R. Beach, K. McIsaac, C. Waldspurger, Caminoreal: An Interactive Mathematical Notebook, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 1-18, Cambridge University Press, 1988.

Asente87 P. J. Asente, Editing Graphical Objects Using Procedural Representations, Digital Western Research Laboratory, Palo Alto, WRL Research Report 87/6, November 1987.

ASPECT87 ASPECT: Specification of the Public Tool Interface, System Designers PLC, 1987.

BAe88 Assessment Report for the Aspect HCI, British Aerospace PLC, Preston, Tech. Rep. No. BAe-WSD-R-ASP-SWE-1563, January 1988.

Bailey88 W. A. Bailey, S. T. Knox, E. F. Lynch, Effects of Interface Design Upon User Productivity, in *Proc. CHI '88*, ed. E. Solloway, D. Frye, S. B. Sheppard, pp. 207-212, Addison Wesley, May 1988.

Barford89 L. A. Barford, B. T. V. Zanden, Attribute Grammars in Constraint-based Graphics Systems, *Software - Practice and Experience*, **19**(4), pp. 309-328, April 1989.

Barth86 P. S. Barth, An Object-Oriented Approach to Graphical Interfaces, *ACM Trans. on Graphics*, **5**(2), pp. 142-172, April 1986.

Barzel88 R. Barzel, A. H. Barr, A Modeling System Based on Dynamic Constraints, *ACM Computer Graphics*, **22**(4), pp. 179-188, August 1988.

Bass85 L. J. Bass, A generalised user interface for applications programs (II), *Comm. ACM*, **28**(6), pp. 617-627, June 1985.

Baudelaire80 P. Baudelaire, M. Stone, Techniques for Interactive Raster Graphics, *ACM Computer Graphics*, **14**(3), pp. 314-320, June 1980.

Baudelaire89 P. Baudelaire, M. Gangnet, Planar Maps: An Interaction Paradigm for Graphic Design, in *Proc CHI 89*, pp. 313-318, ACM, April 1989.

Beach82 R. J. Beach et al., The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program, *ACM Computer Graphics*, **16**(3), pp. 277-287, June 1982.

Beach83 R. Beach, M. Stone, Graphical Style: Towards High Quality Illustrations, *ACM Computer Graphics*, **17**(3), pp. 127-135, 1983.

Beach85 R. J. Beach, Experience with the Cedar Programming environment for computer graphics research, Xerox, Tech. Rep. No. CSL-84-6, July 1985.

Beach86 R. J. Beach, Tabular Typography, in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, pp. 18-33, Cambridge University Press, 1986.

Beaujardiere88 J-M. de la Beaujardiere, Well-Established Document Interchange Formats, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 83-94, Cambridge University Press, 1988.

Behrmann-Poitiers88 J. Behrmann-Poitiers, H. Keil, H. Loebl, Hard Copy Rendition of ODA Documents, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 71-82, Cambridge University Press, 1988.

Ben-Ari82 M. Ben-Ari, Principles of Concurrent Programming, Prentice-Hall, 1982.

Benest79 I. D. Benest, A Review of Computer Graphics Publications, *Computers and Graphics*, **4**, pp. 95-136, Pergamon Press, 1979.

Benest85 I. D. Benest, R. K. Took, Interactive Techniques in Software Engineering Environments, in *IEE Colloqium Digest* No. 1985/102, pp. 3/1-3/15, 1985.

Bennett87 J. L. Bennett, Collaboration of UIMS Designers and Human Factors Specialists, *ACM Computer Graphics*, **21**(2), pp. 102-105, April 1987.

Benyon84 D. Benyon, Monitor. A self-adaptive user interface, in *Proc. Interact '84*, **1**, pp. 307-313, September 1984.

Bhushan86 A. Bhushan, M. Plass, The Interpress Page and Document Description Language, *IEEE Computer*, **19**(6), pp. 72-77, 1986.

Bier86 E. A. Bier, M. C. Stone, Snap-Dragging, *ACM Computer Graphics*, **20**(4), pp. 233-240, August 1986.

Bigelow86 C. Bigelow, K. Holmes, The Design of Lucinda, an Integrated Family of Types for Electronic Literacy, in *Text Processing and Document Manipulation*, ed. J. C. Van Vliet, pp. 1-17, Cambridge, April 1986.

Bly86 S. A. Bly, J. K. Rosenberg, A Comparison of Tiled and Overlapping Windows, in *Proc. CHI '86 Human Factors in Computing Systems*, pp. 101-106, Boston, April 1986.

Bohm66 C. Bohm, G. Jacopini, Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, *Comm. ACM*, **9**(5), pp. 366-371, May 1966.

Borning81 A. Borning, The Programming Language Aspects of ThingLab, a Constraint- Oriented Simulation Laboratory, *ACM Trans. Programming Languages and Systems*, **3**(4), pp. 353-387, October 1981.

Borning86 A. H. Borning, R. A. Duisberg, Constraint-based Tools for Building User Interfaces, *ACM Trans. Graphics*, pp. 345-374, October 1986.

Bos78 J. Van den Bos, Definition and Use of Higher-Level Graphics Input Tools, *ACM Computer Graphics*, **12**(3), pp. 38-42, August 1978.

Bos80 J. van den Bos, High-Level Graphic Input Tools and their Semantics, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 159-169, North-Holland, 1980.

Bos83 J. van der Bos, Whither device independence in interactive graphics?, *Intl. J. Man-Machine Studies*, **18**, pp. 89-99, 1983.

Boullier72 P. Boullier, J. Gros, P. Jancene, A. Lemaire, F. Prusker, E. Saltel, METAVISU: A General Purpose Graphics System., in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 244-267, North-Holland, 1972.

Bramley90 A. Bramley, Porting Presenter to X (BSc Project Report), Dept. Computer Science, University of York, 1990.

BrooksKP88 K. P. Brooks, A Two-View Document Editor with User-definable Document Structure, Digital Systems Research Centre, Technical Report 33, November 1988.

BrooksFP88 F. P. Brooks Jnr., Grasping Reality Through Illusion - Interactive Graphics Serving Science, in *Proc. CHI '88*, ed. E. Solloway, D. Frye, S. B. Sheppard, pp. 1-11, Addison Wesley, May 1988.

Brown84 M. H. Brown, R. Sedgewick, A system for Algorithm Animation, *ACM Computer Graphics*, **18**(3), pp. 177, July 1984.

Brown85 M. D. Brown, Understanding PHIGS, Template, Megatek Corporation, San Diego, 1985.

Brown87 P. Brown, Presenting Documents on Workstation Screens, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 122-128, Springer-Verlag, 1987.

BrownMH88 M. H. Brown, Perspectives on Algorithm Animation, in *Proc. CHI '88*, pp. 33-38, ACM, Washington, May 1988.

BrownP88 P. Brown, Hypertext: The Way Forward, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 183-191, Cambridge University Press, 1988.

Brown89 H. Brown, Standards for Structured Documents, *The Computer Journal*, **32**(6), pp. 505-514, December 1989.

Brown90 A.W. Brown, R.K. Took, W.G. Daly, Design and Construction of Graphical User Interfaces using Surface Interaction, in *Proceedings of the 8th British National Conference on Databases (BNCOD-8)*, ed. A.W. Brown, P. Hitchcock, Pitman Publishing Ltd., July 1990.

Burbeck87 S. Burbeck, Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC), Softsmarts Inc., 1987.

Butler85a J. Butler, Proposal for Project to Develop New X3 Standard, OEM Group, Microsoft, Bellevue, Wa., January 1985.

Butler85b J. Butler, Windows Subgroup Report (presentation to ANSI X3H3 Plenary Session), Palm Bay, Florida, January 1985.

Buxton83 W. Buxton, Lexical and Pragmatic Considerations of Input Structures, *ACM Computer Graphics*, **17**(1), pp. 31-37, January 1983.

Buxton86 W. Buxton, There's More to Interaction Than Meets the Eye: Some Issues in Manual Input, in *User Centered System Design*, ed. D. A. Norman, S. W. Draper, pp. 319-337, Lawrence Erlbaum, 1986.

Cadogan87 P. H. Cadogan, The Chelgraph SGML Structured Editor, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 190-195, Springer-Verlag, 1987.

Cahn83 D. U. Cahn, A. C. Yen, A Device-Independent Network Graphics System, *ACM Computer Graphics*, **17**(3), pp. 167-174, July 1983.

Call87 L. S. Call, D. L. Cohrs, B. P. Miller, CLAM - An Open System for Graphical User Interfaces, in *Proc. USENIX C++ Workshop*, pp. 305-325, Santa Fe, NM, 1987.

Card87 S. K. Card, A. Henderson Jnr., A Multiple, Virtual-Workspace Interface to Support User Task Switching, in *Proc CHI + GI 1987*, ed. J. M. Carroll, P. P. Tanner, ACM SIGCHI Bulletin, 18(2), pp. 53-59, April 1987.

Cardelli85 L. Cardelli, R. Pike, Squeak: A Language for Communicating with Mice, *ACM Computer Graphics*, **19**(3), pp. 199-204, July 1985.

Cardelli87 L. Cardelli, Building User Interfaces by Direct Manipulation, Digital - Systems Research Center, Palo Alto, California, Tech. Rep. No. 22, October 1987.

Carlbom78 I. Carlbom, J. Paciorek, Planar Geometric Projections and Viewing Transformations, *Computing Surveys*, **10**(4), pp. 465-502, December 1978.

Carroll85 J. M. Carroll, R. L. Mack, Metaphor, Computing Systems, and Active Learning, *Int. J. Man-Machine Studies*, **22**, pp. 39-57, 1985.

Chamberlin81 D. D. Chamberlin et al, JANUS: An Interactive System for Document Composition, *ACM SIGPLAN Notices*, **16**(6), pp. 82-91, June 1981.

Chamberlin82 D. D. Chamberlin et al., JANUS: An Interactive Document Formatter Based on Declarative Tags, *IBM Systems Journal*, **21**(3), pp. 250-271, 1982.

Chamberlin87 D. D. Chamberlin, C. F. Goldfarb, Graphic Applications of the Standard Generalised Markup Language (SGML), *Computers and Graphics*, **11**(4), 1987.

Chamberlin88 D. D. Chamberlin, H. F. Hasselmeier, D. P. Paris, Defining Document Styles for WYSIWYG Processing, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 121-137, Cambridge University Press, 1988.

Chang86 S. K. Chang, Introduction: Visual Languages and Iconic Languages, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 1-7, Plenum Press, New York, 1986.

Chapman90 S. Chapman, VISH: A Visual Shell (BSc Project Report), Dept. Computer Science, University of York, 1990.

Chen88 P. Chen, M. A. Harrison, Multiple Representation Document Development, *IEEE Computer*, **21**(1), pp. 15-31, January 1988.

Ciccarelli85 E. C. Ciccarelli, Presentation Based User Interfaces (PhD Thesis), MIT, 1985.

Clarisse86 O. Clarisse, S. -K. Chang, VICON: A Visual Icon Manager, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 151-190, Plenum Press, New York, 1986.

Clarke87 M. Clarke, Back to Basics - Simple But High-Quality Text Pagination Systems, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 203-211, Springer-Verlag, 1987.

Claybrook85 B. G. Claybrook, A. M. Claybrook, J. Williams, Defining Database Views as Data Abstractions, *IEEE Transactions on Software Engineering*, **SE-11**(1), pp. 3-14, January 1985.

Clowes72 M. B. Clowes, Scene Analysis and Picture Grammars, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 70-82, North-Holland, 1972.

Cobbett89 A. P. Cobbett, I. C. Wand, The Debugging of Large Multi-Task Ada Programs, in *Proc Ada UK Conference*, York, September 1989.

Cockton88a G. Cockton, Interaction Ergonomics, Control and Separation: Open Problems in User Interface, Scottish HCI Centre, Heriot-Watt University, Tech. Rep. No. AMU8811/03H, February 1988.

Cockton88b G. Cockton, Generative Transition Networks: A New Communication Control Abstraction, in *People and Computers IV: Proc. HCI '88*, ed. D. M. Jones, R. Winder, pp. 509-527, Cambridge, September 1988.

Cohen86 E. S. Cohen, E. T. Smith, L. A. Iverson, Constraint-based tiled windows, *IEEE Computer Graphics and Applications*, **6**(5), pp. 35-45, May 1986.

CohenB86 B. Cohen, W. T. Harwood, M. I. Jackson, The Specification of Complex Systems, Addison-Wesley, 1986.

Conklin87 J. Conklin, Hypertext: An Introduction and Survey, *IEEE Computer*, **20**(9), pp. 17-41, September 1987.

Cook88 S. Cook, S. Masnavi, Visual Programming of User Interfaces, in *Proc. BCS Conf. Graphics Tools for Software Engineering: Visual Programming and Program Visualisation*, ed. A. C. Kilgour, R. A. Earnshaw, pp. 1-10, BCS, London, March 1988.

Coutaz84a J. Coutaz, A Paradigm for user interface architecture, CMU, Tech. Rep. No. CMU-CS-84-124, May 1984.

Coutaz84b J. Coutaz, The Box, a layout abstraction for user interface toolkits, CMU, Tech. Rep. No. CMU-CS-84-167, December 1984.

Coutaz85 J. Coutaz, Abstractions for User Interface Design, *IEEE Computer*, **18**(9), pp. 21-34, September 1985.

Coutaz86 J. Coutaz, Abstractions for User Interface Toolkits, in *Foundation for Human-Computer Communication*, ed. K. Hopper, I. A. Newman, pp. 335-354, North-Holland, Amsterdam, 1986.

Coutaz87 J. Coutaz, PAC, an Object Oriented Model for Dialog Design, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 431-436, North-Holland, 1987.

Coutaz89a J. Coutaz, Architecture Models for Interactive Software, in *Proc ECOOP '89*, 1989.

Coutaz89b — J. Coutaz, UIMS: Promises, Failures and Trends, in *People and Computers V - Proc HCI '89*, ed. A. Sutcliffe, L. Macaulay, pp. 71-84, Cambridge, 1989.

Coutaz90 — J. Coutaz, L. Bass, Requirements for UIMSs, in Proc Esprit/Eurographics Workshop on User Interface Management Systems and Environments, Lisbon, June 1990.

Cowan86 — D. D. Cowan, G. De V. Smit, Combining Interactive Document Editing with Batch Document Formatting, in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, pp. 140-153, Cambridge, April 1986.

Crampton87 — C. Crampton, A Portable Object-Oriented Toolkit, Presented at *Eurographics Workshop on Higher Level Tools for Window Managers*, Amsterdam, August 1987.

Crowley87 — T. Crowley et al, The Diamond Multimedia Editor, in *Proc USENIX Conf.*, pp. 1-18, June 1987.

Daly89 — W. Daly, A Graphical Management System for Semantic, Multimedia Databases (PhD Thesis), University of York, 1989.

Dam82 — A. Van Dam, N. Meyrowitz, Interactive Editing Systems, in *Document Preparation Systems*, ed. Nievergelt J, Coray G, Nicoud J-D, Shaw A C, pp. 21, North Holland, 1982.

Dance87 — J. R. Dance et al., Report on the Run-Time Structure of UIMS-Supported Applications, *ACM Computer Graphics*, **21**(2), pp. 97-101, April 1987.

Davies89 — A. Davies, A Report on the Design and Implementation of UniFile, a Graphically-based File System Manager for the UNIX Operating System (MSc Project Report), University of York, September 1989.

Dix85 — A. J. Dix, C. Runciman, Abstract Models of Interactive Systems, in *British Computer Society Conference Proc., "People and Computers: Designing the User Interface"*, ed. P. Johnson, S. Cook, pp. 13-22, Cambridge University Press, 1985.

Dix86 — A. J. Dix, M. D. Harrison, Principles and Interaction Models for Window Managers, in *British Computer Society Conference Proc., "People and Computers: Designing for Useability"*, ed. M. D. Harrison, A. Monk, pp. 352-366, Cambridge University Press, 1986.

Dix87a — A. J. Dix, M. D. Harrison, C. Runciman, H. Thimbleby, Interaction Models and the Principled Design of Interactive Systems, in *Proc European Software Engineering Conference*, ed. H. Nichols, D. S. Simpson, pp. 127-135, Springer Lecture Notes, 1987.

Dix87b — A. J. Dix, M. D. Harrison, Formalising Models of Interaction in the Design of a Display Editor, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 409-414, North-Holland, 1987.

Dix88a — A. J. Dix, Formal Methods and Interactive Systems: Principles and Practice (PhD Thesis), University of York, Dept. of Computer Science, 1988.

Dix88b — A. Dix, Abstract, Generic Models of Interactive Systems, in *People and Computers IV: Proc. HCI '88,* ed. D. M. Jones, R. Winder, pp. 63-77, Cambridge, September 1988.

Draper86 — S. W. Draper, Display Managers as the Basis for User-Machine Communication, in *User Centered System Design*, ed. D. A. Norman, S. W. Draper, pp. 339-352, Lawrence Erlbaum, 1986.

Duce90 — D. A. Duce, R. van Liere, P. J. W. ten Hagen, An Approach to Hierarchical Input Devices, *Computer Graphics Forum*, **9**, pp. 15-26, North-Holland, 1990.

Eckhouse79 — R. E. Eckhouse, L. R. Morris, Minicomputer Systems, Prentice/Hall, 1979.

Elwart-Keys90 — M. Elwart-Keys, D. Halonen, M. Horton, R. Kass, P. Scott, User Interface Requirements for Face to Face Groupware, in *Proc CHI90*, pp. 295-302, Addison Wesley, April 1990.

Encarnacao79 — J. Encarnacao, G. Nees, Recommendations on Methodology in Computer Graphics, in *Methodology in Computer Graphics*, ed. R. A. Guedj, H. A. Tucker, pp. 87-92, North-Holland, 1979.

Enderle84 — G. Enderle, K. Kansy, G. Pfaff, Computer Graphics Programming: GKS - The Graphics Standard, Springer-Verlag, 1984.

Feiner82 — S. Feiner, S. Nagy, A. Van Dam, An Experimental System for Creating and Presenting Interactive Graphical Documents, *ACM Trans. Graphics*, **1**(1), pp. 59-77, January 1982.

Fischer89 — G. Fischer, Human-Computer Interaction Software: Lessons Learned, Challenges Ahead, *IEEE Software*, **6**(1), pp. 44-52, January 1989.

Fisher90 — A. Fisher, The Computer as an Assistant to Learning (BSc Project Report), Dept. Computer Science, University of York, 1990.

Foley74 — J. D. Foley, V. L. Wallace, The Art of Natural Graphic Man-Machine Conversation, *Proc. IEEE*, **62**(4), pp. 462-471, April 1974.

Foley79 — J. D. Foley, Picture Naming and Modification: An Overview, in *Methodology in Computer Graphics*, ed. R. A. Guedj, H. A. Tucker, pp. 105-117, North-Holland, 1979.

Foley80a J. D. Foley, The Structure of Interactive Command Languages, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 227-234, North-Holland, 1980.

Foley80b J. D. Foley, Methodology of Interaction, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce, pp. 55-57, North-Holland, 1980.

Foley84a J. D. Foley, A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, 1984.

Foley84b J. D. Foley, V. L. Wallace, P. Chan, The Human Factors of Computer Graphics Interaction Techniques, *IEEE Computer Graphics and Applications*, **4**(11), pp. 13-48, November 1984.

Frame87 FrameMaker Reference Manual 1.0, Frame Technology Corporation, 2240 Lundy Avenue, San Jose, Ca 95131, 1987.

Fraser80 C. W. Fraser, A Generalised Text Editor, *Comm. ACM*, **23**(3), pp. 154-158, 1980.

Friedell84 M. Friedell, Automatic synthesis of graphical object descriptions, *ACM Computer Graphics*, **18**(3), pp. 53, July 1984.

Fuks84 D. B. Fuks, V. A. Rokhlin, Beginner's course in topology, Springer, 1984.

Furuta82 R. Furuta, J. Scofield, A. Shaw, Document Formatting Systems: Survey, Concepts and Issues, *ACM Computing Surveys*, **14**(3), pp. 417-472, September 1982.

Furuta86 R. Furuta, An Integrated, but not Exact-Representation, Editor/Formatter, in *Text Processing and Document Manipulation*, ed. J. C. Van Vliet, pp. 246-259, Cambridge, April 1986.

Furuta88 R. Furuta, P. D. Stotts, Specifying Structured Document Transformations, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 109-120, Cambridge University Press, 1988.

Furuta89 R. Furuta, An Object-based Taxonomy for Abstract Structure in Document Models, *The Computer Journal*, **32**(6), pp. 494-504, December 1989.

Gangopadhyay82 D. Gangopadhyay, A Framework for Modelling Graphic Interactions, *Software - Practice and Experience*, **12**, pp. 141-151, 1982.

Garrett82 M. T. Garrett, J. D. Foley, Graphics Programming Using a Database System with Dependency Declarations, *ACM Trans. Graphics*, **1**(2), pp. 109-128, April 1982.

Gentleman81 W. M. Gentleman, Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept, *Software - Practice and Experience*, **11**(5), pp. 435-466, May 1981.

George72 J. E. George, A Graphical Meta System, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 83-109, North-Holland, 1972.

Gittins86 D. Gittins, Icon-Based Human-Computer Interaction, *Int. J. Man Machine Studies*, **24**, pp. 519-543, 1986.

Glinert87 E. P. Glinert, J. Gonczarowski, A (Formal) Model for (Iconic) Programming Environments, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 283-290, North-Holland, 1987.

Goldberg83 A. Goldberg, D. Robson, Smalltalk-80, The Language and its Implementation, Addison-Wesley, 1983.

Goldfarb81 C. F. Goldfarb, A Generalised Approach to Document Markup, *ACM SIGPLAN Notices*, **16**(6), pp. 68-73, June 1981.

Golin90 E. J. Golin, S. P. Reiss, The Specification of Visual Language Syntax, *Visual Languages and Computing*, **1**, pp. 141-157, Academic Press, 1990.

Gonzalez78 R. C. Gonzalez, M. G. Thomason, Syntactic Pattern Recognition, Addison-Wesley, 1978.

Gosling86 J. Gosling, SunDew - A Distributed and Extensible Window System, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 47-52, Springer-Verlag, Berlin, 1986.

Gosling86 J. Gosling, Partitioning of Function in Window Systems, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 101-106, Springer-Verlag, Berlin, 1986.

Gosling86 J. Gosling, D. Rosenthal, A Window Manager for Bitmapped Displays and UNIX, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 116-128, Springer-Verlag, Berlin, 1986.

Green85a M. Green, The University of Alberta User Interface Management System, *ACM Computer Graphics*, **19**(3), pp. 205-213, July 1985.

Green85b M. Green, Report on Dialogue Specification Tools, in *User Interface Management Tools*, ed. G. E. Pfaff, pp. 9-20, Springer-Verlag, Berlin, 1985.

Green86 M. Green, A Survey of Three Dialogue Models, *ACM Trans. on Graphics*, **5**(3), pp. 244-275, July 1986.

Green87 M. Green, Directions for User Interface Management Systems Research, *ACM Computer Graphics*, **21**(2), pp. 113-116, April 1987.

Guedj79 Methodology in Computer Graphics, ed. R. A. Guedj, H. A. Tucker, North-Holland, 1979.

Guedj80 Methodology of Interaction, ed. R. A. Guedj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce, North-Holland, 1980.

Gutknecht84 J. Gutknecht, W. Winiger, Andra: The Document Preparation System of the Personal Workstation Lilith, *Software - Practice and Experience*, **14**, pp. 73-100, 1984.

Gutknecht85 J. Gutknecht, Concepts of the Text Editor Lara, *Comm. ACM*, **28**(9), pp. 942-960, September 1985.

Guttag78 J. V. Guttag, J. J. Horning, The Algebraic Specification of Abstract Data Types, *Acta Informatica*, **10**, pp. 27-52, 1978.

Haeberli88 P. E. Haeberli, ConMan: A Visual Programming Language for Interactive Graphics, *ACM Computer Graphics*, **22**(4), pp. 103-111, August 1988.

Hagen85 P. J. W. ten Hagen, J. Derksen, Parallel Input and Feedback in Dialogue Cells, in *User Interface Management Systems*, ed. G. E. Pfaff, pp. 109-124, Springer-Verlag, 1985.

Hall85 J. A. Hall, P. Hitchcock, R. K. Took, An Overview of the ASPECT Architecture, in *Integrated Project Support Environments*, ed. J. McDermid, pp. 86-99, Peter Peregrinus, London, 1985.

Hamlet86 R. Hamlet, A Disciplined Text Environment, in *Text Processing and Document Manipulation*, ed. J. C. van Vliet, pp. 78-89, Cambridge, April 1986.

Hammer81 M. Hammer et al., The Implementation of Etude, An Integrated and Interactive Document Production System, *ACM SIGPLAN Notices*, **16**(6), pp. 137-146, June 1981.

Harel88 D. Harel, On Visual Formalisms, *Comm. ACM*, **31**(5), pp. 514-530, May 1988.

Harke87 U. Harke, M. Burger, Dr. Gall, Embedding Graphics into Documents by using a Graphics Editor, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 87-101, Springer-Verlag, 1987.

Harris86 D. J. Harris, An Approach to the Design of a Page Description Language, in *Text Processing and Document Manipulation*, ed. J. C. Van Vliet, pp. 58-64, Cambridge, April 1986.

Harrison90 M. Harrison, A. Dix, A State Model of Direct Manipulation in Interactive Systems, in *Formal Methods in Human-Computer Interaction*, ed. M. Harrison, H. Thimbleby, pp. 129-151, Cambridge, 1990.

Hartson89 R. Hartson, User-Interface Management Control and Communication, *IEEE Software*, **6**(1), pp. 62-70, January 1989.

Hayes83 P. J. Hayes, P. A. Szekely, Graceful Interaction through the COUSIN Command Interface, *Int. J. of Man-Machine Studies*, **19**(3), pp. 285-305, September 1983.

Hayes84 P. J. Hayes, Executable Interface Definitions Using Form-Based Interface Abstractions, CMU, Tech. Rep. No. CMU-CS-84-110, March 1984.

Hegazy88 W. A. Hegazy, J. S. Gourlay, Optimal Line Breaking in Music, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 157-169, Cambridge University Press, 1988.

Hekmatpour87 S. Hekmatpour, M. Woodman, Formal Specification of Graphical Notations and Graphical Software Tools, Open University, Milton Keynes, Tech. Rep. No. 87/7, 1987.

Helm86 R. Helm, K. Marriott, Declarative Graphics, in *Proc. Third Int. Conf. on Logic Programming*, ed. E. Shapiro, pp. 513-527, Springer Verlag, Berlin, 1986.

Henderson86 D. A. Henderson Jnr., S. K. Card, Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface, *ACM Trans on Graphics*, **5**(3), pp. 211-243, July 1986.

Herot80 C. F. Herot, Spatial Management of Data, *ACM Trans. Database Sys.*, **5**(4), pp. 493-514, 1980.

Hewitt88 W. T. Hewitt, R. Hubbold, T. Howard, D. Finnegan, T. Arnold, M. Patel, K. Wyrwas, Interactive Computer Graphics - Course Notes, Computer Graphics Unit, Dept. Computer Science, University of Manchester, June 1988.

Hill86 R. D. Hill, Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction - The Sassafras UIMS, *ACM Trans. on Graphics*, **5**(3), pp. 179-210, July 1986.

Hill87a R. D. Hill, Event-Response Systems - A Technique for Specifying Multi-Threaded Dialogues, in *Proc. SIGCHI+GI '87: Human Factors in Computing Systems*, pp. 241-248, Toronto, Canada, April 1987.

Hill87b R. D. Hill, Some Important Features and Issues in User Interface Management Systems, *ACM Computer Graphics*, **21**(2), pp. 116-120, April 1987.

Hoare85 C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall International, 1985.

Hoeber88 T. Hoeber, Open Look Design Gaols, Sun Technology, pp. 63-75, Sun Microsystems, September 1988.

Holmes89 S. Holmes, Overview and User Manual For Doubleview, University of York, Tech. Rep. No. YCS109, 1989.

Hopgood80 F. R. A. Hopgood, D. A. Duce, A Production System Approach to Interactive Graphic Program Design, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 247-263, North-Holland, 1980.

Hopgood86a F. R. A. Hopgood, A Graphics Standards View of Screen Management, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 87-95, Springer-Verlag, Berlin, 1986.

Hopgood86b *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, Springer-Verlag, Berlin, 1986.

Horak85 W. Horak, Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardisation, *IEEE Computer*, **18**(10), pp. 50-60, 1985.

Hudson87 S. E. Hudson, UIMS Support for Direct Manipulation Interfaces, *ACM Computer Graphics*, **21**(2), pp. 120-124, April 1987.

Hudson90 S. E. Hudson, Adaptive Semantic Snapping - A Technique for Semantic Feedback at the Lexical Level, in *Proc CHI 90*, pp. 65-70, 1990.

Hurley89 W. D. Hurley, J. L. Sibert, Modeling User Interface - Application Interactions, *IEEE Software*, **6**(1), pp. 71-77, January 1989.

Hutcheon90 A. D. Hutcheon, A. J. Wellings, The York Distributed Ada project, in *Distributed Ada*, ed. J. Bishop, pp. 71-108, Cambridge University Press, 1990.

Hutchins86 E. L. Hutchins, J. D. Hollan, D. A. Norman, Direct Manipulation Interfaces, in *User Centered System Design*, ed. D. A. Norman, S. W. Draper, pp. 87-124, Lawrence Erlbaum Associates, 1986.

Huu87 L. Van Huu, An Environment for SGML Document Preparation, in *Proc USENIX Conf.*, pp. 43-52, June 1987.

Hypercard89 Macintosh Hypercard - User's Guide, Apple Computer Inc., Cupertino, Ca 95014, 1989.

Ingalls81 D. Ingalls, The Smalltalk Graphics Kernel, *Byte*, **6**(8), pp. 168, August 1981.

Ingalls88 D. Ingalls, W. Wallace, Y. Chow, F. Ludolph, K. Doyle, Fabrik, a Visual Programming Environment, in *Proc OOPSLA 88*, September 1988.

ISO85 ISO, Information Processing Systems - Computer Graphics - Graphics Kernel System (GKS) functional description, ISO Central Secretariat, Geneva, ISO 7942, 1985.

ISO86a ISO, Information Processing Systems - Computer Graphics - Interfacing Techniques for Dialogues with Graphical Devices (CGI), ISO Central Secretariat, Geneva, ISO/TC97/SC21/WG2 N356, May 1986.

ISO86b ISO, Standard Generalised Markup Language, ISO DIS 8879, 1986.

ISO87a ISO, Information Processing - Text and Office Systems: Office Document Architecture (ODA) and Interchange Format, ISO DIS 8613 part 1-8, July 1987.

ISO87b Information Processing - Computer Graphics - Programmers Hierarchical Interactive Graphics System (PHIGS), ISO DIS 9592-1:1987(E), October 1987.

Jacob86a R. J. K. Jacob, A Visual Programming Environment for Designing User Interfaces, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 87-107, Plenum Press, New York, 1986.

Jacob86b R. J. K. Jacob, Using Formal Specifications in the Design of a Human-Computer Interface, in *Software Specification Techniques*, ed. N. Gehani, A. D. McGettrick, pp. 209-222, Addison-Wesley, 1986.

Jacob86c R. J. K. Jacob, A Specification Language for Direct-Manipulation User Interfaces, *ACM Trans. on Graphics*, **5**(4), pp. 283-317, October 1986.

Johnson88 J. Johnson, R. J. Beach, Styles in Document Editing Systems, *IEEE Computer*, **21**(1), pp. 32-43, January 1988.

Joloboff86 V. Joloboff, Trends and Standards in Document Representation, in *Proc. Conf. Text Processsing and Document Manipulation*, pp. 107-124, Nottingham, 1986.

Jones-Ng86 L. Jones-Ng, The Spy Editor, Rutherford Appleton Laboratory, March 1986.

JonesWP88 W. P. Jones, A. C. Kay, W. Kintsch, R. H. Trigg, A Critical Assessment of Hypertext Systems, in *Proc. CHI '88*, pp. 223-227, Addison Wesley, May 1988.

JonesDW88 D. W. Jones, How (Not) to Code a Finite State Machine, *SIGPLAN Notices*, **23**(8), pp. 19-22, March 1988.

Jones89 M. Jones, A Presenter Toolkit (MSc Project Report), Dept. Computer Science, University of York, September 1989.

Josephs88 M. B. Josephs, A State-Based Approach to Communicating Processes, *Distributed Computing*, **3**, pp. 9-18, 1988.

Kamran83 A. Kamran, M. B. Feldman, Graphics Programming Independent of Interaction Techniques and Styles, *ACM Computer Graphics*, **17**(1), pp. 58-66, January 1983.

Kamran85 A. Kamran, Issues Pertaining to the Design of a User Interface Management System, in *User Interface Management Systems*, ed. G. E. Pfaff, pp. 43-48, Springer-Verlag, 1985.

Kantorowitz89 E. Kantorowitz, O. Sudarsky, The Adaptable User Interface, *Comm ACM*, **32**(11), pp. 1352-1358, November 1989.

Kaplan88 M. Kaplan, Abstraction and Integration in IDE, an Editing and Formatting Environment, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 193-204, Cambridge University Press, 1988.

Kasik82 D. J. Kasik, A User Interface Management System, *ACM Computer Graphics*, **16**(3), pp. 99-106, July 1982.

Kasik89 D. J. Kasik, M. A. Lund, H. W. Ramsey, Reflections on Using a UIMS for Complex Applications, *IEEE Software*, **6**(1), pp. 54-61, January 1989.

Kass88 R. Kass, T. Finin, A General User Modelling Facility, in *Proc CHI 88*, pp. 145-150, ACM, May 1988.

Kernighan81 B. W. Kernighan, PIC - a language for typesetting graphics, *ACM SIGPLAN Notices*, **16**(6), pp. 92-98, June 1981.

Kimura86 G. D. Kimura, S Structure Editor for Abstract Document Objects, *IEEE Trans. Software Engineering*, **SE-12**(3), pp. 417-435, March 1986.

Knuth81 D. E. Knuth, M. F. Plass, Breaking Paragraphs into Lines, *Software - Practice and Experience*, **11**, pp. 1119-1184, 1981.

Knuth86 D. E. Knuth, The TeX Book, Addison-Wesley, Reading, Mass., 1986.

Korfhage86 R. R. Korfhage, M. A. Korfhage, Criteria for Iconic Languages, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 207-231, Plenum Press, New York, 1986.

Kulsrud68 H. F. Kulsrud, A General Purpose Graphic Language, *Comm. ACM*, **11**(4), pp. 247-254, April 1968.

Lamport86 L. Lamport, LATEX - A Document Preparation System, Addison-Wesley, September 1986.

Lamport89 L. Lamport, A Simple Approach to Specifying Concurrent Systems, *Comm ACM*, **32**(1), pp. 32-45, January 1989.

Langridge87 R. Langridge, B. Bryant, PHIGS, A Universal Graphics Standard: Is It All a Pipedream, in *Proc BCS Conf on Future of Graphics Software*, ed. R. A. Earnshaw, London, October 1987.

Langridge88 R. Langridge, B. Bryant, PHIGS: Is It a Universal Graphics Standard?, *Computing Techniques*, **3**(5), pp. 25-29, May 1988.

Lantz84 K. A. Lantz, W. I. Nowicki, Structured Graphics for Distributed Systems, *ACM Trans. Graphics*, **3**(1), pp. 23-51, January 1984.

Lantz87a K. A. Lantz, On User Interface Reference Models, in *Proc CHI + GI 1987*, ed. J. M. Carroll P. P. Tanner, ACM SIGCHI Bulletin, 18(2), pp. 36-42, April 1987.

Lantz87b K. A. Lantz et al, Reference Models, Window Systems, and Concurrency, *ACM Computer Graphics*, **21**(2), pp. 87-97, April 1987.

Lantz87c K. A. Lantz, Multi-Process Structuring of User Interface Software, *ACM Computer Graphics*, **21**(2), pp. 124-130, April 1987.

Larson86 J. A. Larson, Visual Languages for Database Users, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 127-147, Plenum Press, New York, 1986.

Laursen87 D. Laursen, Why X? A Standard for Window Systems, *Tekniques*, **10**(4), pp. 20-23, Tektronix, Wilsonville, Oregon, 1987.

Lauwers90 J. C. Lauwers, K. A. Lantz, Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems, in *Proc CHI90*, pp. 303-312, Addison Wesley, April 1990.

Leler86 W. Leler, Specification and Generation of Constraint Satisfaction Systems (PhD Thesis), 1986.

Leler88 W. Leler, Constraint Programming Languages - Their Specification and Generation, Addison-Wesley, Reading, Ma., 1988.

Lesk86 M. E. Lesk, Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff, Bell Laboratories, Murray Hill, New Jersey 07974, April 1986.

Lieberman85 H. Lieberman, There's More to Menu Systems Than Meets the Screen, *ACM Computer Graphics*, **19**(3), pp. 181-189, July 1985.

Liere87 R. van Liere, P. J. W. ten Hagen, Resource Management in DICE, Presented at *Eurographics workshop on Higher Level Tools for Window Management*, Amsterdam, August 1987.

Linton87 M. A. Linton, R. R. Calder, The Design and Implementation of InterViews, in *Proc. USENIX C++ Workshop*, pp. 256-267, Santa Fe, NM, 1987.

Linton89 M. A. Linton, J. M. Vlissides, P. R. Calder, Composing User Interfaces with Interviews, *IEEE Computer*, pp. 8-22, February 1989.

Lipkie82 D. E. Lipkie, S. R. E. Evans, J. K. Newlin, R. L. Wissman, Star Graphics: an object oriented implementation, *ACM Computer Graphics*, **16**(3), pp. 115, July 1982.

Little87 C. T. Little, Graphics and GKS at the UK Meteorological Office, in *Proc BCS Conf on Future of Graphics Software*, ed. R. A. Earnshaw, London, October 1987.

Littlefield84 R. J. Littlefield, Priority Windows: A Device Independent, Vector Oriented Approach, *ACM Computer Graphics*, **18**(3), pp. 187, July 1984.

Luniewski88 A. W. Luniewski, Intent-Based Page Modelling Using Blocks in the Quill Document Editor, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 205-221, Cambridge University Press, 1988.

MacDraw88 MacDraw User Manual, Claris Corporation, Mountain View, Ca 94043, 1988.

Mackinlay86 J. Mackinlay, Automating the Design of Graphical Presentations of Relational Information, *ACM Trans. Graphics*, **5**(2), pp. 110-141, April 1986.

Mallgren83 W. R. Mallgren, Formal Specification of Interactive Graphics Programming Languages, MIT Press, 1983.

Manheimer89 J. M. Manheimer, R. C. Burnett, J. A. Wallers, A Case Study of User Interface Management System Developement and Application, in Proc *CHI '89*, pp. 127-132, Austin, Texas, May 1989.

Marcus84 A. Marcus, Corporate Identity for Iconic Interface Design: The Graphic Design Perspective, *IEEE Computer Graphics and Applications*, **4**(12), pp. 24-32, December 1984.

Marovac87 N. Marovac, Page Description Languages, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 14-26, Springer-Verlag, 1987.

Martin82 G. E. Martin, Transformation Geometry, Springer-Verlag, 1982.

McCabe87 F. G. McCabe, Denotational Graphics, Dept of Computing, Imperial College, London, July 1987.

McCarthy90 J. M. McCarthy, V. C. Miles, Elaborating Communication Channels In Conferencer, in *Proc IFIP WG8.4 Conference on Multi-User Interfaces and Applications*, May 1990.

McDonald82 A. McDonald, Visual Programming, *Datamation*, **28**(11), pp. 132-140, October 1982.

Meads72 J. A. Meads, A Terminal Control System, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 271-287, North-Holland, 1972.

Medina-Mora82 R. Medina-Mora, Syntax-directed editing: Towards Integrated Programming Environments, CMU, March 1982.

Meyrowitz86 N. Meyrowitz, Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework, *ACM SIGPLAN Notices*, **21**(11), 1986.

Milgram72 D. L. Milgram, A. Rosenfeld, A Note on "Grammars with Coordinates", in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 187-191, North-Holland, 1972.

MIT88 MIT X Window System Manual Set Version 11, Release 2, IXI Ltd., Wellington Court, Cambridge CB1 1HZ, 1988.

Monden86 N. Monden, Y. Yoshino, M. Hirakawa, M. Tanake, T. Ichikawa, HI-VISUAL, A Language Supporting Visual Interaction in Programming, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 233-259, Plenum Press, New York, 1986.

Moran81 T. P. Moran, The command language grammar: a representation for the user interface of interactive computer systems, *Intl. J. Man-Machine Studies*, **15**, pp. 3-50, 1981.

Moreland87 J. Moreland, Graphics in Presentation Management, in *Proc BCS Conf on Future of Graphics Software*, ed. R. A. Earnshaw, London, October 1987.

Morris81 J. M. Morris, D. S. Mayer, The Design of a Language-Directed Editor for Block-Structured Languages, *ACM SIGPLAN Notices*, **16**(6), pp. 28-33, June 1981.

Morris86 J. H. Morris et al, Andrew: A Distributed Personal Computing Environment, *Comm. ACM*, **29**(3), pp. 184-201, March 1986.

Murrel87 S. L. Murrel, D. De Baer, An Interactive WYSIWYG Table Editor, in *Proc USENIX Conf.*, pp. 19-29, June 1987.

Myers83 B. A. Myers, Incense: a system for displaying data structures, *ACM Computer Graphics*, **17**(3), pp. 115, July 1983.

Myers86a B. Myers, Issues in Window Management Design and Implementation, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 59-69, Springer-Verlag, Berlin, 1986.

Myers86b B. A. Myers, Visual Programming, Programming by Example, and Program Visualization: A Taxonomy, in *Proc ACM SIGCHI '86 Conf.*, pp. 59-66, ACM, New York, April 1986.

Myers87a B. M. Myers, Gaining General Acceptance for UIMSs, *ACM Computer Graphics*, **21**(2), pp. 130-134, April 1987.

Myers87b B. A. Myers, Creating Interaction Techniques by Demonstration, *IEEE Computer Graphics and Applications*, pp. 51-60, September 1987.

Myers88a B. A. Myers, The State of the Art in Visual Programming and Program Visualization, CMU, Tech. Rep. No. CMU-CS-88-114, February 1988.

Myers88b B. A. Myers, Tools for Creating User Interfaces: An Introduction and Survey, CMU, Tech. Rep. No. CMU-CS-88-107, January 1988.

Myers88c B. A. Myers, Window Interfaces - A Taxonomy of Window Manager User Interfaces, *IEEE Computer Graphics and Applications*, pp. 65-84, September 1988.

Myers89 B. M. Myers, Encapsulating Interactive Behaviors, in *Proc CHI 89*, pp. 319-324, ACM, April 1989.

Nanard87 J. Nanard, M. Nanard, G. Cottin, PLEIADE, A System for Interactive Manipulation of Structured Documents, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 73-86, Springer-Verlag, 1987.

Nelson80 T. H. Nelson, Replacing the Printed Word: A Complete Literary System, *IFIP Proc.*, pp. 1013-1023, October 1980.

Nelson85 G. Nelson, Juno, A Constraint-based Graphics System, *ACM Computer Graphics*, **19**(3), pp. 235-243, July 1985.

Newman68 W. M. Newman, A system for interactive graphical programming, *AFIPS Conf procs (SJCC)*, **32**, pp. 47-54, 1968.

Newman71 W. M. Newman, Display Procedures, *Comm. ACM*, **14**(10), pp. 651, 1971.

Newman79 W. M. Newman, R. F. Sproull, Principles of Interactive Computer Graphics, Mcgraw-Hill, 1979.

Newman83 W. M. Newman, T. Mott, Officetalk-Zero: An Experimental Integrated Office System, in *Integrated Interactive Computer Systems*, ed. P. Delgano, E. Sandewall, pp. 315-331, North-Holland, Amsterdam, 1983.

Newman85 W. Newman, N. Stephens, D. Sweetman, A window manager with a modular user interface, *Proc HCI '85*, pp. 415-426, CUP, September 1985.

Newman87 W. M. Newman, Designing Integrated Systems for the Office Environment, McGraw-Hill, 1987.

Newman88 W. M. Newman, The Representation of User Interface Style, in *People and Computers IV: Proc. HCI '88*, ed. D. M. Jones, R. Winder, pp. 123-143, Cambridge, September 1988.

NeWS87a NeWS Manual, Sun Microsystems, 1987.

NeWS87b NeWS Technical Overview, Sun Microsystems Inc., Tech. Rep. No. 800-1498-05, March 1987.

Nielsen86 J. Nielsen, A Virtual Protocol Model for Computer-Human Interaction, *Intl. J. of Man-Machine Studies*, **24**, pp. 301-312, 1986.

O'Callaghan72 J. F. O'Callaghan, Use of a Picture Language to Generate Descriptions of Line Drawings, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 123-141, North-Holland, 1972.

Olsen83a D. R. Olsen, E. P. Dempsey, SYNGRAPH: A Graphic User Interface Generator, *ACM Computer Graphics*, **17**(3), pp. 43-50, July 1983.

Olsen83b D. R. Olsen, Automatic Generation of Interactive Systems, *ACM Computer Graphics*, **17**(1), pp. 53-57, January 1983.

Olsen84 D. R. Olsen, Pushdown automata for user interface management, *ACM Trans. Graphics*, **3**(3), pp. 177-203, July 1984.

Olsen85a D. R. Olsen, Presentational, Syntactic and Semantic Components of Interactive Dialogue Specifications, in *User Interface Management Systems*, ed. G. E. Pfaff, pp. 125-133, Springer-Verlag, 1985.

Olsen85b D. R. Olsen Jnr., E. P. Dempsey, R. Rogge, Input/Output Linkage in a User Interface Management System, *ACM Computer Graphics*, **19**(3), pp. 191-197, July 1985.

Olsen86 D. R. Olsen, MIKE: The Menu Interaction Kontrol Environment, *ACM Trans. Graphics*, **5**(4), pp. 318-344, October 1986.

Olsen87a D. R. Olsen, Larger Issues in User Interface Management, *ACM Computer Graphics*, **21**(2), pp. 134-137, April 1987.

Olsen87b D. R. Olsen (Chairman), ACM SIGGRAPH Workshop on Software Tools for User Interface Management (introduction), *ACM Computer Graphics*, **21**(2), pp. 71-72, April 1987.

Olsen90 D. R. Olsen, Propositional Production Systems for Dialog Description, in *Proc CHI 90*, pp. 57-63, April 1990.

Paeth86 A. W. Paeth, K, S, Booth, Design and Experience with a Generalized Raster Toolkit, in *Proc Graphics/Vision Interface '86*, pp. 91-97, 1986.

Palay88 A. J. Palay et al., The Andrew Toolkit - An Overview, in *Proc. USENIX Conf.*, pp. 9-21, February 1988.

Payne84 S. J. Payne, Task-Action Grammars, in *Human-Computer Interaction, Interact '84*, ed. B. Shackel, pp. 527-532, North-Holland, 1984.

PCTE88 PCTE User Interface Specification, Yard Ltd., Tech. Rep. No. C1211/3, March 1988.

Pereira86 F. C. N. Pereira, Can Drawing Be Liberated From The Von Neumann Style?, in *Logic Programming And It's Application*, ed. M. Van Caneghan, D. H. D. Warren, pp. 175-187, Ablex Publishing, Norwood, New Jersey, 1986.

Perq84 Perq - Guide to PNX, ICL, 1984.

Pfaff85 User Interface Management Systems, ed. G. E. Pfaff, Springer-Verlag, Berlin, 1985.

Pier88 K. Pier, E. Bier, M. Stone, An Introduction to Gargoyle: An Interactive Illustraction Tool, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 223-238, Cambridge University Press, 1988.

Pike84 R. Pike, The Blit: a multiplexed graphics terminal, *AT&T Bell Labs technical Journal*, **63**(8), pp. 1607, October 1984.

Pike85 R. Pike, B. Locanthi, J. Reiser, Hardware/Software Tradeoffs for Bitmap Graphics on the Blit, *Software - Practice and Experience*, **15**(2), pp. 131-151, February 1985.

Pike89 R. Pike, A Concurrent Window System, *Computing Systems*, **2**(2), pp. 133-153, January 1989.

Pollard89 W. Pollard, Providing Client-Server Interaction in a Highly Interactive Graphics Environment (BSc Project Report), Dept. Computer Science, University of York, March 1989.

Porter84 T. Porter, T. Duff, Compositing Digital Images, *ACM Computer Graphics*, **18**(3), pp. 253-260, July 1984.

Powell83 M. L. Powell, M. A. Linton, Visual Abstraction in an interactive programming environment, *ACM SIGPLAN Notices*, **18**(6), pp. 14-21, June 1983.

Pratt85 V. Pratt, Techniques for Conic Splines, *ACM SIGGRAPH*, **19**(3), pp. 151-160, July 1985.

Prime90 M. Prime, User Interface Management Systems - A Current Product Review, *Computer Graphics Forum*, **9**(1), pp. 53-76, March 1990.

Quint86 V. Quint, I. Vatton, Grif: An Interactive System for Structured Document Manipulation, in *Text Processing and Document Manipulation*, ed. J. C. Van Vliet, pp. 200-213, Cambridge, April 1986.

Quint87 V. Quint, I. Vatton, An Abstract Model for Interactive Pictures, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 643-647, North-Holland, 1987.

Rao87 R. Rao, S. Wallace, The X Toolkit - The Standard Toolkit for X Version 11, in *Proc USENIX Conf.*, pp. 117-129, June 1987.

Reid80 B. K. Reid, Scribe: A High-Level Approach to Document Formatting, *Proc 7th Symposium on Principles of Programming Languages*, Las Vegas, January 1980.

Reid86 B. K. Reid, Procedural Page Description Languages, in *Text Processing and Document Manipulation*, ed. J. C. Van Vliet, pp. 214-223, Cambridge, April 1986.

Reiser88 B. J. Reiser et al., A Graphical Programming Language Interface for an Intelligent Lisp Tutor, in *Proc. CHI 88*, Washington, May 1988.

Reisner81 P. Reisner, Formal Grammar and Human Factors Design of an Interactive Graphics System, *IEEE Trans. Software Engineering*, **SE-7**(2), pp. 229-240, March 1981.

Ritchie89 I. Ritchie, HYPERTEXT - Moving Towards Large Volumes, *The Computer Journal*, **32**(6), pp. 516-523, December 1989.

Roberts88 W. T. Roberts, A. Davison, K. Drake, C. E. Hyde, M. Slater, P. Papageorgiou, NeWS and X, Beauty and the Beast?, in *Proc. EUUG*, pp. 265-310, Cascais, October 1988.

Rosenthal80 D. S. H. Rosenthal, Models of Interactive Graphics Software, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 361-370, North-Holland, 1980.

Rosenthal81 D. S. H. Rosenthal, 'Methodology in Computer Graphics' Re-examined, *ACM Computer Graphics*, **15**(2), pp. 152-162, July 1981.

Rosenthal82 D. S. H. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, M. Sabin, The Detailed Semantics of Graphics Input Devices, *ACM Computer Graphics*, **16**(3), pp. 33-38, July 1982.

Rosenthal83 D. S. H. Rosenthal, Managing Graphical Resources, *ACM Computer Graphics*, **17**(1), pp. 38-45, January 1983.

Rosenthal87 D. S. H. Rosenthal, A Simple X11 Client Program, or How hard can it really be to write "Hello World"?, Sun Microsystems Inc., 1987.

Salmon87 R. Salmon, M. Slater, Computer Graphics - Systems and Concepts, Addison-Wesley, 1987.

Scheifler86 R. W. Scheifler, J. Gettys, The X Window System, *ACM Trans. Graphics*, **5**(2), pp. 79-109, April 1986.

Schmucker86 K. J. Schmucker, MacApp: An Application Framework, *Byte*, pp. 189-193, August 1986.

Scofield85 J. A. Scofield, Editing as a Paradigm for User Interaction (PhD Thesis), Computer Science Dept., Univ. of Washington, Seattle, Tech. Rep. No. 85-08-10, August 1985.

Scott88 M. L. Scott, S. -K. Yap, A Grammar-Based Approach to the Automatic Generation of User Interface Dialogues, in *Proc. CHI '88*, ed. E. Solloway, D. Frye, S. B. Sheppard, pp. 73-78, Addison Wesley, May 1988.

Selker87 T. Selker, C. Wolf, L. Koved, A Framework for Comparing Sytems with Visual Interfaces, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 683-688, North-Holland, 1987.

Shaw80 A. C. Shaw, On the Specification of Graphics Command Languages and their Processors, in *Methodology of Interaction*, ed. R. A. Geudj, P. J. W. ten Hagen, F. R. A. Hopgood, H. A. Tucker, D. A. Duce., pp. 377-392, North-Holland, 1980.

Shaw83 M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, R. Pausch, Descartes: A Programming-Language Approach to Interactive Display Interfaces, *ACM SIGPLAN Notices*, **18**(6), pp. 100-111, ACM, 1983.

Shneiderman82 B. Shneiderman, The Future of Interactive Systems and the Emergence of Direct Manipulation, *Behaviour and Information Technology*, **1**(3), pp. 237-256, 1982.

Shneiderman83 B. Shneiderman, Direct Manipulation: A Step Beyond Programming Languages, *IEEE Computer*, **16**(8), pp. 57-69, 1983.

Sibert85 J. Sibert, R. Belliardi, A. Kamran, Some Thoughts on the Interface Between User Interface Management Systems and Application Software, in *User Interface Management Systems*, ed. G. E. Pfaff, pp. 183-192, Springer-Verlag, 1985.

Sibert86 J. L. Sibert, W. D. Hurley, T. W. Bleser, An Object-Oriented User Interface Management System, *ACM Computer Graphics*, **20**(4), pp. 259-268, August 1986.

Simoes87 L. P. Simoes, J. A. Marques, Images - An Object Oriented UIMS, in *Human-Computer Interaction - Interact '87*, ed. H. -J. Bullinger, B. Shackel, pp. 751-756, North-Holland, 1987.

Singh89 G. Singh, M. Green, A High-Level User Interface Management System, in *Proc CHI '89*, pp. 133-138, Austin, Texas, May 1989.

Slater88 M. Slater, A. Davison, M. Smith, Liberation from Rectangles: A Tiling Method for Dynamic Modification of Objects on Raster Displays, *Proc. Eurographics '88*, 1988.

Smith82a D. C. Smith, C. H. Irby, R. B. Kimball, E. Harslem, The Star User Interface: An Overview, *Proc AFIPS Conf.*, **51**, pp. 515-528, NCC, 1982.

Smith82b D. C. Smith, C. Irby, R. Kimball, B. Verplank, Designing the Star User Interface, *Byte*, 7(4), pp. 242-282, April 1982.

Smith86 R. B. Smith, The Alternate Reality Kit - An Animated Environment for Creating Interactive Simulations, in *Proc IEEE Workshop on Visual Languages*, pp. 99-106, Dallas, Texas, June 1986.

Smith87 R. B. Smith, Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic, *IEEE Computer Graphics and Applications*, pp. 42-50, September 1987.

Smith90 S. Smith, Compo: A Prototype for a Direct Manipulation Interface for Image Processing (MSc Project Report), University of York, 1990.

Southall88 R. Southall, Visual Structure and the Transmission of Meaning, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 35-45, Cambridge University Press, 1988.

Spivey89 J. M. Spivey, The Z Notation - A Reference Manual, Prentice Hall International, 1989.

Sproull79 R. F. Sproull, Raster Graphics for Interactive Programming Environments, *ACM Computer Graphics*, **13**(2), pp. 83-93, August 1979.

Sproull83 R. F. Sproull, Challenges in Graphical user interfaces, *Man-Machine Interaction*, pp. 145-152, University of Newcastle Computing Laboratory, September 1983.

Stanton72 R. B. Stanton, The Interpretation of Graphics and Graphic Languages, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 144-159, North-Holland, 1972.

Stefik86 M. Stefik, D. G. Bobrow, K. M. Kahn, Integrating Access-Oriented Programming into a Multi-Paradigm Environment, *IEEE Software*, **3**(1), pp. 10-18, January 1986.

Stoddart90 A. Stoddart, A Report on the Visual Shell Environment Manager (VSEM) - A Visual Shell for UNIX (BSc Project Report), Dept. Computer Science, University of York, 1990.

Stroustrup87 B. Stroustrup, What is "Object-Oriented Programming"?, in *Proc. USENIX C++ Workshop*, pp. 159-180, Santa Fe, NM, November 1987.

Stroustrup88 B. Stroustrup, What is Object-Oriented Programming?, *IEEE Software*, **5**(3), pp. 10-20, May 1988.

Strubbe83 H. J. Strubbe, Kernel for a Responsive and Graphical User Interface, *Software - Practice and Experience*, **13**(11), pp. 1033-1042, 1983.

Stutely87 R. Stutely, The Standard Generalised Markup Language, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 176-189, Springer-Verlag, 1987.

Sufrin90 B. Sufrin, J. He, Specification, Analysis and Refinement of Interactive Processes, in *Formal Methods in Human-Computer Interaction*, ed. M. Harrison, H. Thimbleby, pp. 153-200, CUP, 1990.

Sugaya84 H. Sugaya, E. S. Biagioni, Input/Output Functions for a Bitmapped Raster Graphics Terminal, Brown Boveri Forschungszentrum, CH-5405 Baden-Dattwil, Tech. Rep. No. KLR 84-94 C, June 1984.

Sugihara86 K. Sugihara, J. Miyao, M. Takayama, T. Kikuno, N. Yoshida, A Visual Language for Forms Definition and Manipulation, in *Visual Languages*, ed. S. -K. Chang, Tadao Ichikawa, P. A. Ligomenides, pp. 109-125, Plenum Press, New York, 1986.

Sun86 SunView Programmer's Guide, Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Ca 94043, 1986.

Sutherland63 I. E. Sutherland, Sketchpad: A Man-Machine Graphical Communication System, in *Proc SJCC*, pp. 329-346, 1963.

Sweetman86 D. Sweetman, A Modular Window System for UNIX, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 73-78, Springer-Verlag, Berlin, 1986.

Swick88 R. R. Swick, M. S. Ackerman, The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire, in *Proc USENIX Conf.*, pp. 221-228, February 1988.

Swinehart84 D. C. Swinehart, P. T. Zellweger, R. B. Hagman, The structure of Cedar, Xerox PARC, 1984.

Swinehart85 D. Swinehart, Cedar User Facilities, in *Proc User Interface Design Methodology Workshop*, Cosener's House, Oxford, September 1985.

Szekely87 P. Szekely, Modular Implementation of Presentations, in *Proc CHI + GI*, ed. J. M. Carroll P. P. Tanner, *ACM SIGCHI Bulletin*, **18**(2), pp. 235-240, April 1987.

Szekely88a P. A. Szekely, B. A. Myers, A User Interface Toolkit Based on Graphical Objects and Constraints, in *Proc OOPSLA '88*, pp. 36-45, ACM, September 1988.

Szekely88b P. Szekely, Separating the User Interface from the Functionality of Application Programs (PhD Thesis), CMU, Tech. Rep. No. CMU-CS-88-101, January 1988.

Szwillus87 G. Szwillus, GEGS - A System for Generating Graphical Editors, in *Human-Computer Interaction - INTERACT '87 (Participants' Edition)*, ed. H. -J. Bullinger, B. Shackel, pp. 135-141, North-Holland, 1987.

Tanner86 P. P. Tanner, S. A. MacKay, D. A. Stewart, M. Wein, A Multitasking Switchboard Approach to User Interface Management, *ACM Computer Graphics*, **20**(4), pp. 241-248, August 1986.

Tanner87 P. P. Tanner, Multi-Thread Input, *ACM Computer Graphics*, **21**(2), pp. 142-145, April 1987.

Teitelman84 W. Teitelman, A Tour Through Cedar, *IEEE Software*, **1**(2), pp. 44-73, 1984.

Teitelman86 W. Teitelman, Ten Years of Window Systems - A Retrospective View, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 35-46, Springer-Verlag, Berlin, 1986.

Tesler81 L. Tesler, The Smalltalk environment, *Byte*, 6(8), pp. 90, August 1981.

Thacker82 C. P. Thacker, E. M. McCreight, B. W. Lanpson, R. F. Sproull, D. R. Boggs, Alto - A Personal Computer, in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, A. Newell, pp. 549-572, McGraw-Hill, New York, 1982.

Thimbleby80 H. Thimbleby, Dialogue Determination, *Intl. J. of Man-Machine Studies*, 13, pp. 295-304, 1980.

Thimbleby84 H. W. Thimbleby, Generative user-engineering principles for user interface design, in *Human-Computer Interaction - INTERACT '84*, ed. B. Shackel, pp. 102-107, North-Holland, 1984.

Thimbleby85 H. Thimbleby, User Interface Design: Generative Engineering Principles, in *Fundamentals of Human-Computer Interaction*, ed. A. Monk, Academic Press, 1985.

Thomas83 J. J. Thomas, G. Hamlin, Graphical Input Interaction Technique Workshop Summary, *ACM Computer Graphics*, **17**(1), pp. 5-30, January 1983.

Took86a R. K. Took, Text Representation and Manipulation in a Mouse-Driven Interface, in *People and Computers: Designing for Usability*, ed. M. D. Harrison, A. F. Monk, pp. 386-401, Cambridge University Press, 1986.

Took86b R. K. Took, The Presenter - A Formal Design for an Autonomous Display Manager, in *Software Engineering Environments*, ed. Ian Sommerville, pp. 151-169, Peter Peregrinus, 1986.

Took90a R. K. Took, Surface Interaction: A Paradigm and Model for Separating Application and Interface, in *Proc CHI '90*, pp. 35-42, ACM, April 1990.

Took90b R. K. Took, Putting Design into Practice: Formal Specification and the User Interface, in *Formal Methods in Human-Computer Interaction*, ed. M. Harrison, H. Thimbleby, pp. 63-96, Cambridge, 1990.

Turner84 J. U. Turner, A programmer's interface to graphics dynamics, *ACM Computer Graphics*, **18**(3), pp. 263, July 1984.

Visvalingam87 M. Visvalingam, Problems in the Design and Implementation of a GKS-Based User Interface for a Graphical Information System, University of Hull, Cartographic Information Systems Research Group, Discussion paper 2, September 1987.

Voorhies88 D. Voorhies, D. Kirk, O. Lathrop, Virtual Graphics, *ACM Computer Graphics*, **22**(4), pp. 247-253, August 1988.

Waite88 K. W. Waite, Visualising Abstract Data Types, in *Proc. BCS Conf. Graphics Tools for Software Engineering: Visual Programming and Program Visualisation*, ed. A. C. Kilgour, R. A. Earnshaw, BCS, London, March 1988.

Walker88 J. A. Walker, Supporting Document Development with Concordia, *IEEE Computer*, **21**(1), pp. 48-59, January 1988.

Wallace76 V. L. Wallace, The Semantics of Graphic Input Devices, *ACM Computer Graphics*, **10**(1), pp. 61-65, April 1976.

Warnock82 J. Warnock, D. K. Wyatt, A Device Independent Graphics Imaging Model for Use with Raster Devices, *ACM Computer Graphics*, **16**(3), pp. 313-320, 1982.

Wasserman85 A. I. Wasserman, Extending state transition diagrams for the specification of human-computer interaction, *IEEE transactions on software engineering*, **SE-11**(8), pp. 699-713, August 1985.

Whiteley88 K. Whiteley, M. J. Birch, A. Parker, A Mascot 3 Paintbox for Aspect, in *Proc. Software Engineering 88*, Liverpool, July 1988.

Wiecha89 C. Wiecha, W. Bennett, S. Boies, J. Gould, Generating Highly Interactive User Interfaces, in *Proc CHI 89*, pp. 277-282, ACM, May 1989.

Wiederhold86 G. Wiederhold, Views, Objects, and Databases, *IEEE Computer*, pp. 37-44, December 1986.

Wilcox88 L. D. Wilcox, A. L. Spitz, Automatic Recognition and Representation of Documents, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 47-57, Cambridge University Press, 1988.

Williams72 R. Williams, A General Purpose Graphical Language, in *Graphic Languages*, ed. F. Nake, A. Rosenfield, pp. 334-349, North-Holland, 1972.

Williams83 G. Williams, The Lisa Computer System, *Byte*, 8(2), pp. 33-50, February 1983.

Williams86 A. S. Williams, A Comparison of Some Window Managers, in *Methodology of Window Management*, ed. F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, A. S. Williams, pp. 16-32, Springer-Verlag, Berlin, 1986.

Williams87 A. S. Williams, C. M. Crampton, C. A. A. Goswell, Unix Window Management Systems Client-Server Interface Specification, Rutherford Appleton Laboratory, Tech. Rep. No. RAL-87-017, March 1987.

Wills87a A. Wills, Document Processing Review, University of Manchester, Dept. of Computer Science, Ipse2.5 Report 060/acw00048/1.1, February 1987.

Wills87b A. Wills, Structure of Interactive Environments, Manchester University Dept. of Computer Science, May 1987.

Woodman87 M. Woodman, D. Ince, J. Preece, G. Davies, A Grammar Formalism as a Basis for the Syntax-Directed Editing of Graphical Notations, in *Workstations and Publication Systems*, ed. R. A. Earnshaw, pp. 102-116, Springer-Verlag, 1987.

Woods70 W. A. Woods, Transition Network Grammars for Natural Language Analysis, *Comm ACM*, **13**(10), pp. 591-606, October 1970.

Wyk82 C. J. van Wyk, A High-Level Language for Specifying Pictures, *ACM Trans. Graphics*, **1**(2), pp. 163-182, April 1982.

Yankelovich88 N. Yankelovich, B. J. Haan, N. K. Meyrowitz, S. M. Drucker, Intermedia: The Concept and the Construction of a Seamless Information Environment, *IEEE Computer*, **21**(1), pp. 81-96, January 1988.

Young88 M. Young, R. N. Taylor, D. B. Troup, C. D. Kelly, Design Principles Behind Chiron: A UIMS for Software Environments, in *Proc. 10th Intl. Conf. Software Engineering*, pp. 367-376, IEEE, Singapore, April 1988.

Young89 R. M. Young, T. R. G. Green, T. Simon, Programmable User Models for Predictive Evaluation of Interface Designs, in *Proc CHI 89*, pp. 15-19, ACM, May 1989.

Zellweger88 P. T. Zellweger, Active Paths through Multimedia Documents, in *Document Manipulation and Typography*, ed. J. C. van Vliet, pp. 19-34, Cambridge University Press, 1988.