Definitive Interfaces as a Visualisation Mechanism

Meurig Beynon
Yun Pui Yung
Department of Computer Science
University of Warwick
Coventry CV4 7AL, UK

Abstract

The specification of the display interface for visualisation is regarded as analogous to devising a mechanical linkage. A definitive (definition-based) approach to display specification is described, illustrated and evaluated.

Résumé

On conçoit la spécification de l'interface visuelle servant à la visualisation comme analogue à l'invention d'une articulation mécanique. On décrit une approche, fondée sur l'emploi de définitions, de la spécification visuelle. On donne un exemple et une évaluation de cette approche.

Keywords: interactive techniques, visualisation, user-interfaces, windowing systems, display management

Introduction

The execution of an interactive program is a sequence of transitions carried out by the computer or the user. The current state of the execution is represented internally by the values currently assigned to memory locations, but this is not in general an appropriate representation for the user. The user-interface serves a dual purpose: giving the user a comprehensible view of the current internal state and dictating the protocol by which the user can alter this state.

This paper considers a new programming paradigm for visualisation. A definition-based or "definitive" programming style is adopted, whereby the relationship between the screen display and the internal state is represented by a set of variables whose values are defined by formulae that may be changed during program execution. In effect, the screen is regarded as an exotic variable whose value is defined - in a flexible manner with reference to potential transitions from the current state - as a function of the internal state. This generalises the use of a script to define a static environment for evaluation in a functional programming system [16]: a "script" is instead used to represent state and can be dynamically modified during execution to reflect the prevailing relationship between the screen state and the internal state.

The screen display will in general be composed of several windows, each of which has special characteristics according to its role in the visualisation process. Some windows may display text, others graphical images of various different kinds. Previous papers [2,3,4,5] have considered the design and implementation of special-purpose notations based on definitive principles that are suitable for particular types of display. These include the interactive graphics systems DoNaLD and ARCA [3,4] for instance. This paper applies definitive principles to display management, describing and illustrating a general framework within which to integrate the diverse application-oriented components required for effective visualisation (cf [9]).

The paper is in 5 sections. §1 motivates and introduces a definitive notation for screen layout. §2 explains why a definitive screen interface resembles a mechanical linkage. §3 discusses the advantages of definitive screen layout specification and explains the important role of the user protocols that must complement the display specification. §4 outlines appropriate techniques for specifying such protocols. §5 discusses the current status of the project and indicates directions for further research.

1. Motivation and Basic Principles

The set of definitions that specifies both the nature and current state of the display will be formulated using SCOUT, a *definitive notation* for screen layout. A definitive notation is a simple programming language in which a program is a sequence of variable declarations and definitions. Each definition associates a formula with a declared variable. This formula either specifies the value of the variable explicitly (as in "a=3") or implicitly in terms of the values of other variables (as in "a=2*b+c"). The same variable may be redefined many times in the course of a program; such redefinition effectively overwrites the previous definition of the variable. A definition is invalid only if it introduces direct or indirect self-reference (as in "a=3*a" or "a=3*b; b=a+c").

The semantics of a definitive notation is determined by an *underlying algebra* of data types and operators. The underlying algebra in effect specifies the domain of possible values for variables and the operators that can be used in the

defining formulae. The type of a variable is specified on declaration. A program over a definitive notation is interpreted by regarding the prevailing set of definitions (i.e. that determined by the most recent definitions of each of the declared variables) as specifying a state. Each new declaration, definition or redefinition then effects a change of state. The state information associated with a set of definitions reflects the current values of variables - where defined - and the relationships between these values; it typically represents the current state of a designed object (e.g. the screen display). A change of state may reflect redesign (e.g. the relocation of a display window by the designer) or a change in the state of the object (e.g. a screen update).

Such *definitive principles* are exploited in their simplest form in spreadsheets; they were also applied to graphics in [15]. Recent research at the University of Warwick has investigated generic techniques for designing and implementing application-oriented definitive notations as a medium for interactive programming [2,3,4,5]. As one example, an interactive graphics system based on a definitive notation for line-drawing DoNaLD over an underlying algebra of **points**, **lines** and **shapes** comprising sets of **points** and **lines** has been developed [4].

A full description of DoNaLD is beyond the scope of this paper; a DoNaLD script to describe a room layout similar to the picture "floorplan" that appears in Figure 1 is given in [4]. By way of illustration, the plan of the door in [4] is generated by the following set of definitions:

```
point NW = {100,900}      # the NW corner of the room
openshape door
within door {
    real    width = 200
    bool    open
    line    door = [hinge, lock]
    # the line segment joining points hinge and lock
    point   hinge = ~/NW + {15,-10}
    point   lock = hinge + if open then {0, -width}
                                 else {width,0}
}
```

When used in conjunction with notations such as DoNaLD, SCOUT addresses the need to construct a display from several images, each of which has a specialised function. This is desirable for instance in an architectural design environment where a plan may be accompanied by perspective drawings, symbolic diagrams and textual annotations.

The principal features of SCOUT will be illustrated by example. Figure 2 shows a screen display comprising several components: a room layout consisting of a door, a desk and a table bearing a table lamp; an auxiliary window in which a selected area of the room can be shown in greater detail; a set of buttons indicating menu options that can be used to simulate actions such as opening and closing the door and altering the subregion selected for magnification; additional text windows that are introduced as appropriate to indicate particular error conditions. Both the room layout and the auxiliary window are specified in DoNaLD (they are in fact different views of the same DoNaLD picture); the contents of other windows of the display are specified using variables representing textual strings.

The data types in SCOUT are designed for display management. A *screen* is a variable of type **display**, where **display** = *list of* **window**. The **window** data type is a union of several different types, reflecting the variety of ways in which special-purpose definitive notations may be used to generate components of the screen display. The essential ingredients common to each window subtype are information about the location, content and attributes of the window. In the example, window locations are specified using the types **integer**, **point**, **box** and **frame**, where:

point = **integer** × **integer**,
box = **point** × **point**,
frame = *list of* **box**.

The syntax used for specifying values of type **point** and **box** is illustrated in Figure 1. For instance, [p2,q2] designates the box whose opposite corners have the coordinates {275,100} and {475,300} respectively. The location of a textual window is specified by a **frame**, and that of a DoNaLD window by a **box**. The content of a textual window is specified by a **string**, and that of a DoNaLD window by the graphical interpretation of a specified file of DoNaLD definitions (this file is omitted from Figure 1 – for details of the DoNaLD specification, see [4]). The principal attribute of a **window** is its type; additional attributes are used to select a background colour or border.

The detailed interpretation of Figure 1 is a simple exercise to the reader. The definition of a variable of type **window** makes use of constructors such as

"{ *frame*: ..., *string*: ..., *border*: ..., ... }"

that synthesise a value of a **window** subtype from its constituent fields, with the convention that unspecified fields are defined by default values. Semantic links between windows are set up by using common variables, such as the boolean *DoorHitsTable*.

2. Sets of Definitions as Mechanisms

In the visualisation process, the external display has to be closely coupled to the internal state of the system. The implied analogy with a mechanical system in which a coupling is established through direct physical connection between components is instructive and will be explored in detail. For instance, we should like to view the correlation of changes in internal and display state in the same way that we conceive the interdependent synchronised movement of levers in a linkage.

The characteristic behaviour of a mechanical device is associated with the communication of change from one component of a system to another: when gear A rotates clockwise, gear B rotates anti-clockwise with twice the angular displacement. The propagation of change is directional: turning gear A turns gear B. Though a construction such as a gear-pair allows propagation in either direction this is not the case in general: a block placed on a table will move when the table moves, but not vice versa. State changes within a mechanical device can also affect the framework for propagation: as when a new gear is selected in a gearbox.

The communication of change from the internal state of a system to the display interface follows a closely analogous pattern. It is commonplace for the external representation of internal parameters to depend functionally upon their values. In a direct manipulation interface there are symmetric relationships that enable the user to change internal parameters as if through the display interface, but in fact changes propagate from the internal state to the display rather than vice versa. When the mode of operation of an executing program changes, entirely different relationships between external and internal values may be established.

The mechanical analogy can be usefully extended. A common practice in machine design is to develop simple mechanisms that can be composed by linkages. These submechanisms are themselves designed in a similar manner. The hierarchical decomposition of a machine into component parts resembles the decomposition of a system into objects in an object-oriented paradigm. The role of definitions in specifying the screen interface in Figure 1 is in many respects similar. Some of the definitions determine the geometric relationships in the application, others the relationships between the elements of the screen display. These sets of definitions can be developed independently and linked through bridging definitions (such as

integer *DoorIsOpen* = **boolean** *door/open*

where *DoorIsOpen* is a SCOUT variable and *door/open* is a DoNaLD one) expressing the dependency between pictorial elements and parameters in the application.

A set of definitions serves as an effective computational device for representing the propagation of change. In modelling mechanical systems, definitions are better adapted for describing the relationships between objects than orthodox message passing techniques [14]. They can represent the relationship between coupled components in such a way that the computation associated with maintaining the relationship is invisible. The computational abstraction resembles that exploited in pure functional programming, where function evaluation is invisible. A definitive approach has the advantage of also representing the current state of the components. Other approaches to handling state change propagation represent state information explicitly, but do not use abstraction to capture coupling relationships. For instance, in Borning and Duisberg's approach to building user-interfaces [8], explicit methods for constraint satisfaction are introduced in an object-oriented paradigm. Their emphasis is then upon convenient means of specifying the procedures that propagate change.

3. The Merits of Definitive Specification

A controversial issue in human-computer interaction is the degree to which separation of output from the application is possible ([1]p27,[13]). Ideally we should like to be able to edit the application and the display functions independently within a computational framework that conveys the coupling between changes of state in the application and the display in execution. In a procedural paradigm, maintenance of the display typically means that display actions must be invoked after each significant change to the internal state. In a conventional program, this means that the procedural actions in the application and the display actions are intertwined in the text. Delegating the task of maintaining consistency between the display and the internal store to objects (cf [8]) allows application and display functions to be edited independently, but introduces a degree of indirection into the execution. A definitive description of the screen display addresses this problem, since the relationship between the screen definition and the application state is established through symbolic links, whilst changes in state to parameters within the application are communicated to the display in a conceptually indivisible fashion.

A definitive approach to screen display realises several benefits of separation that have been widely cited by previous researchers [1,13], viz the power to separate applications and interface development, to reuse an interface in a similar application and to experiment with different interfaces for the same application. The fact that a set of definitions explicitly describes the data dependencies means that the propagation of change can be targeted, leading to greater efficiency when selective updating of the display is possible. The declarative nature of the description of the screen display state can also assist program development, obviating the need for reasoning about sequences of display actions.

Definitive principles provide a means of constructing executable specifications for the display manager. A definitive specification of the screen state is an idealisation in that the formulae that define the screen state in terms of the internal parameters must be repeatedly re-evaluated in order to keep the screen updated. Effective implementation techniques that address this problem have been described in detail elsewhere [4]. Implementing SCOUT specifications using a conventional software package such as X Windows can be reduced to implementing the basic operators of the SCOUT underlying algebra using the standard basic procedures. This technique is useful for rapid prototyping purposes. An alternative approach that might be appropriate in developing interfaces for a particular target firmware configuration would involve substituting an underlying algebra tailored to the most efficient primitives available.

The advantages of using a definitive paradigm for interface design are fully appreciated only when the protocols governing redefinition are considered. To pursue the analogy introduced above, the function of a mechanical device can only be understood with reference to possible user input. The control a driver has over an engine is confined to altering certain parameters, such as the depression of the accelerator, and certain functional relationships, such as the gear ratio. Formerly — in the days of the hand-crank — it encompassed means to drive the pistons manually. The set of definitions in Figure 1 must be interpreted with similar regard for what privileges the user has to redefine variables. These in turn will be determined by the intended role of the interface and its designer's conception.

These issues underlie the use of definitive principles for integrating modelling and animation in Chmilar and Wyvill's animation system [10], as can be illustrated with reference to Figure 1. The interface described in Figure 1 may be

designed for a simple educational tool that simulates the relocation of furniture and objects within the room. In that case, the user will be allowed to open and close the door, but not to relocate it. At the discretion of the interface designer, the location of monitoring messages that indicate error conditions, such as the obstruction of the door, might also be under user control. In designing the interface for use by an architect, the options available to the user would include scope for both simulation of the room in use and redesign. The representation of state by a set of definitions makes it readily possible to model the characteristic privileges to change state associated with each agent.

4. Dialogue and Automatic Mechanisms

The SCOUT notation is primarily concerned with issues of display. In the light of the above discussion, the role of SCOUT in user-interface management and visualisation can only be fully understood with reference to the protocols for interaction that surround the display interface. At one level of abstraction, it is sufficient to complement a screen display specification with appropriate information about how and when the user is privileged to redefine variables. At another level, it is essential to consider the means by which the user affects such a change. In Figure 1, for instance, the user may be entitled to redefine the variable *open*, but the user action to be performed is a menu selection. It may be that the error message "Table obstructs door" is only to be displayed when the user attempts to open or close an obstructed door. It may be that a menu selection invokes a complex sequence of actions involving simulated movement of the furniture, perhaps in such a way that the user can intervene.

The need to extend the specification methods so far described is most apparent if Figure 1 is considered to be part of the display specification for a fictitious interactive game that might appropriately be called *Poltergeist*. In *Poltergeist*, the human player must rearrange furniture to satisfy an objective such as enabling a trapped elephant to escape from the room whilst the elephant and the furniture are simultaneously being moved around by the computer. The mechanical analogy in this case is with machines that have an autonomous behaviour.

The full consideration of display protocols for such applications clearly requires a more general computational model than simple redefinition of SCOUT scripts provides. For the most effective application of definitive principles to display management, this model must permit compatible specification of related aspects of the interface. For instance, the specification in Figure 1 cannot otherwise be refined to incorporate menu selection or to describe the responses required of the *Poltergeist* display. The concepts being developed for this purpose will be briefly described; for more discussion, see [6].

The user protocol is specified by the values to which the user responds, the variables which the user can conditionally redefine, and the enabling condition associated with a redefinition. In the context of Figure 1, the user might be permitted to open or close the door only when it is unobstructed. Abstractly, this means that the boolean condition "*open* **and not** *DoorHitsTable*" is the enabling condition that determines whether the door can be closed. In specifying *Poltergeist*, the protocol for redefinition to be observed by the computer might be identified most effectively by conceptually distributing the computation between two agents: the elephant and the poltergeist, and specifying their protocols independently. The LSD notation has been designed for such a specification role (cf [6,7]).

In LSD, the interface through which each agent interacts with the system is viewed as analogous to a generalised spreadsheet whose state can be changed both by the user and by external agents. There are variables whose values are visible to the agent, but are subject to change beyond its direct control - *oracle* variables. There are variables that can be conditionally redefined - *state* variables - subject to a protocol specifying the enabling conditions for redefinition. The values of other variables - *derivate* variables - are given in terms of these by appropriate definitions.

The LSD specification models the way in which an agent can respond to changes in its environment reflected in the perceived values (as typically represented via oracle and derivate variables) by making reciprocal changes in the state of its environment (as represented via state variables). In itself, such a specification cannot be interpreted operationally: the fact that an agent is privileged to respond according to a particular protocol does not give sufficient information about which actions it chooses to perform, the speed with which it responds to changes in its environment or how fast it executes relative to other agents. LSD does provide a suitable framework in which to model the different characteristics of agents however. For example, the specifications of elephants and poltergeists might respectively reflect knowledge of the immediate local environment and comprehensive knowledge of the entire room layout (cf [7]).

Specifying agent privileges for conditional redefinition of variables is the first step towards a full implementation of a required interface. The complete specification is most appropriately expressed using the framework provided by the Abstract Definitive Machine (ADM) [5,6]. In this computational model, the current state of the execution is recorded by a set of definitions and each transition consists of a set of parallel redefinitions. In *Poltergeist*, this set might represent simultaneous actions on the part of the user, the elephant and the poltergeist, for instance. The set of redefinitions to be performed in a single transition is determined with reference to boolean conditions that serve as guards. The explicit representation of data dependency in the ADM model can be used to address problems of interference (cf [5]).

Since all transitions in the ADM model are represented by redefinitions, some interpretation for user input is required. The signal generated by clicking the mouse button in a particular location can be interpreted as a change of internal state initiated by the user. This state change can be represented consistently by conceiving the internal state as specified by a set of definitions encompassing both application-oriented variables such as *open* and variables to represent e.g. the status and position of the mouse. It is such variables that the user is able to redefine. Within the ADM

model, these redefinitions can be used in conjunction with guards to transform a primitive user input into a redefinition as required e.g. for menu selection or direct manipulation.

5. Status and comparison

Our prototype system runs under UNIX on a SUN workstation. It makes use of a pipeline consisting of SCOUT and DoNaLD filters to generate intermediate code in the hybrid definitive/procedural language EDEN [4] (cf the use of tbl and eqn preprocessors to creat nroff code). The display output is generated by executing the EDEN interpreter in parallel with an interface to X Windows. The screen displays in Figure 2 were generated by converting a window dump into a Postscript file. The system can be used in conjunction with the ADM for animation purposes, as illustrated in [5]. Our present prototype is too slow to be used directly to drive practical interfaces, but could be used by the interface designer to study different modes of data presentation, for instance.

Our approach will be briefly reviewed with reference to Olsen's discussion of the state of the art in UIMS's, as reported in [11].

Successful user-interface management relies upon combining good data abstraction, such as object-oriented programming provides, with control over the logic of the sequence, as is best captured in state-transition models. The approach to specification described in [1] aims to treat sequencing and data transformation as separate concerns, but it is evident that dialogue control is often directed by data values. Definitive principles may be an appropriate solution: they support good data abstraction and powerful state-transition models; they make it possible to relate the sequencing and effect of actions to the current data values, and also offer methods for relating synchronisation to stimulus and response [7].

The distinctive feature of a definitive approach is that it makes it possible to represent the total events indivisibly associated with particular actions. Notice that it is not generally necessary to explicitly update all dependent variables when performing a redefinition in order to guarantee this - it is enough that the values of implicitly defined variables are calculated from their current definitions as and when they are required. When one redefinition is made, the values of several variables are changed in what is conceptually a single indivisible transition.

The special characteristics of definitive principles can be exploited in several ways in specifying interface. The representation of state by sets of definitions makes 'rubout' actions easy to implement [11]. The indivisible effect of actions can be context-dependent, as is appropriate when specifying interface modes. Because consistency between values is rigorously maintained, it is relatively easy to enable the user to interrupt execution and to interpret the computational state.

The most significant implications of definitive principles relate to the design and development of interfaces. Because the event associated with an action is determined by its context, the designer can effectively specify and modify the effect of an action retrospectively. It is also possible to deal simultaneously with issues at different levels of abstraction in the user-interface, for example, to establish connections between the syntactic and semantic levels as required. These considerations suggest that a definitive approach can support a process of software development in which requirements analysis and implementation are intertwined [7]; this accords with Olsen's view of the interface development process as far more hectic and iterative than is described by a simple "waterfall" model [11,12].

Conclusion

The SCOUT system provides a framework within which several special-purpose definitive tools can be used in conjunction. It makes it possible to establish relationships between dissimilar component parts of the display whilst retaining the advantages of conceptually disjoint application-oriented underlying algebras. This illustrates one approach to the effective visualisation of objects that can be viewed in several orthogonal abstract ways (cf [3]).

In its present form, SCOUT only addresses some of the significant issues in user-interface management. Further research is required e.g. to deal with mouse input and dialogue control. SCOUT is also in certain respects a low-level language. Methods for describing generic features of a display (such as the concept of an "error monitor window") would be essential for convenient use. In this context, the issues for further research are similar to those raised elsewhere in connection with developing definitive interactive graphics systems for large scale applications [5].

The ultimate objective of the definitive programming project is to demonstrate the feasibility of addressing all aspects of software specification in an integrated and consistent manner by applying definitive principles [6]. The role of definitive interfaces in this research programme is potentially as significant as that played by mechanical analogy in general engineering.

Acknowledgements

We are indebted to Yun Wai Yung for invaluable help in developing the prototypes described in this paper and to William Beynon for the loan of Donkey Kong II.

References

[1] H Alexander, Formally-based tools and techniques for human-computer dialogues, Computers and their Applications, Ellis Horwood, 1987

[2] W M Beynon, Definitive notations for interaction, Proc. hci'85, "People and Computers: Designing the Interface", ed Johnson and Cook, CUP 1985, 23-34

[3] W M Beynon, Definitive principles for interactive graphics, NATO ASI Series F:40, Springer-Verlag 1988, 1083-1097

[4] W M Beynon, Y W Yung, Implementing a definitive notation for interactive graphics, New Trends in Computer Graphics, Springer-Verlag 1988, 456-468

[5] W M Beynon, Evaluating definitive principles for interactive graphics, New Advances in Computer Graphics, Springer-Verlag 1989, 291-303

[6] W M Beynon, M T Norris, S B Russ, M D Slade, Y P Yung, Y W Yung, Software construction using definitions: an illustrative example, CS RR#147, Univ of Warwick 1989

[7] W M Beynon, M T Norris, R A Orr, M D Slade Definitive specification of concurrent systems, Proc UKIT'90, Southampton, March 1990 (to appear)

[8] A Borning, R Duisberg, Constraint-based Tools for Building User Interfaces, ACM Transaction on Graphics, Vol 5 No 4, 1986, 345-374

[9] D W Brown, C D Carson, W A Montgomery, P M Zislis, Software specification and prototyping technologies, AT&T Tech Journal, July/August 1988, 33-45

[10] M Chmilar, B Wyvill, A Software Architecture for Integrated Modelling and Animation, New Advances in Computer Graphics, Proc. of CGI'89, 257-276

[11] K Ehrlich, Report on Seminar "UIMS: State of the Art" by Dan Olsen, SIGCHI Bulletin, July 1989

[12] W L Johnson, Deriving Specifications from Requirements, Proc 10th Int Conf on Software Engineering, Singapore, 428-438, 1988

[13] H R Hartson, D Hix, Human-Computer Interface Development: Concepts and Systems for Its Management, ACM Computing Surveys, 21(1), 1989, 5-92

[14] T Tomiyama, Object-oriented programming for intelligent CAD systems, in Intelligent CAD systems 2: Implementation Issues, Springer-Verlag 1989, 3-16

[15] B Wyvill, An interactive graphics language, PhD Thesis, Univ of Bradford, 1975

[16] The miranda manual, Research Software Ltd, 1987

```
# Windows Showing DoNaLD Pictures
point p2 = {275, 100};                                  # top left corner of the window
point q2 = {475, 300};                                  # bottom right corner
point zoomPos = {500, 500};                             # zooming position in DoNaLD coordinates
point zoomSize = 500;                                   # size of the picture in DoNaLD coordinates
window don2 = {
        type:     DONALD,                               # it is a DoNaLD picture
        box:      [p2, q2],                             # area in which the DoNaLD picture is shown
        pict:     "floorplan",                          # name of the DoNaLD picture
        xmin:     zoomPos.1 - zoomSize/2,               # defining the portion of the DoNaLD
        ymin:     zoomPos.2 - zoomSize/2,               #   picture to be displayed in the box
        xmax:     zoomPos.1 + zoomSize/2,               #     defined above; zoomPos.1 returns the
        ymax:     zoomPos.2 + zoomSize/2,               #         first coordinate of zoomPos
        border:   ON                                    # there is a border around the box
};
...


# Error Message Windows
integer DoorHitsTable = DONALD boolean DoorHitsTable;           # a bridging definition
window monDoor = {
        frame:    ([monDoorPos, 1, strlen(monDoorStr)]),
                  # A string will be put into a frame containing only one box
                  # that has top left corner monDoorPos  and is 1 row by strlen(monDoorStr) columns
        string:   monDoorStr
};                                                      # by default, windows are of type TEXT
string monDoorStr = if DoorHitsTable then "Table obstructs door" else "" endif;
point monDoorPos = {25, 50};
...


# Menu Buttons
point tblMenuRef = {100, 400};                          # the centre of the table-menu
point tblUpPos = tblMenuRef – {(strlen(tblUpMenu)/2).c, 2.r};       # the location of the table-up button
        # .c and .r serve as units, they can be read as 'columns' and 'rows' respectively
window tblMenus = {
        frame:    ([tblUpPos, 1, strlen(tblUpMenu)], [tblDownPos, 1, strlen(tblDownMenu)],
                   [tblLeftPos, 1, strlen(tblLeftMenu)], [tblRightPos, 1, strlen(tblRightMenu)]),
                  # a list of 4 boxes
        string:   tblUpMenu // tblDownMenu // tblLeftMenu // tblRightMenu,    # // – string concatenation
        border:   ON
};
string tblUpMenu = "UP", tblDownMenu = "DOWN", tblLeftMenu = "LEFT", tblRightMenu = "RIGHT";

integer DoorIsOpen = DONALD boolean door/open;          # a bridging definition
point miscMenuRef = {250, 400};
point doorButtonPos = miscMenuRef + {strlen(plugMenu).c /2, 1.r};
window doorButton = {
        frame:    ([doorButtonPos, 1, strlen(doorMenu)]),
        string:   doorMenu,
        border:   ON
};
string doorMenu = if DoorIsOpen then "Close Door" else "Open Door" endif;
...


# Forming Display
display basicScreen = <
        tblHeader / tblMenus / zoomHeader / zoomMenus / plugButton / doorButton / don1 / don2
>;      # if windows overlap,  tblHeader overlays tblMenus etc.
display scr = if DoorHitsTable then insert(basicScreen, 1, monDoor) else basicScreen endif;
display screen = if CableIsShort then insert(scr, 1, monCable) else scr endif;
        # 'screen' is a special variable representing the actual screen
```

Figure 1: Extracts from the SCOUT specification for Figure 2a

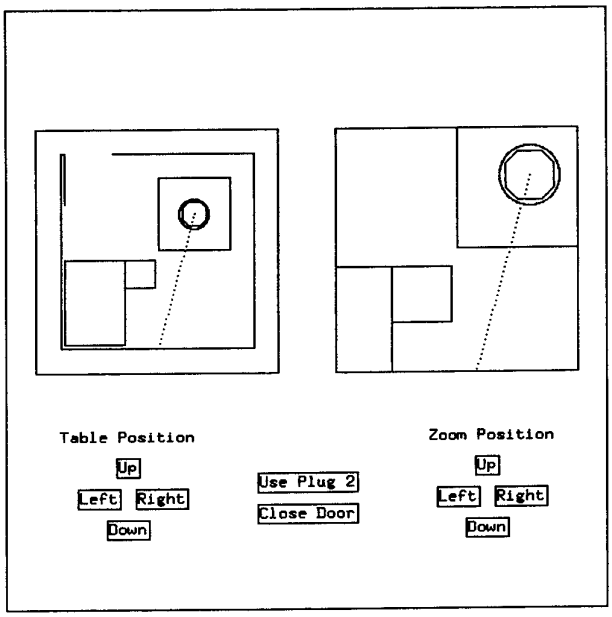

Figure 2a: The display associated with Figure 1

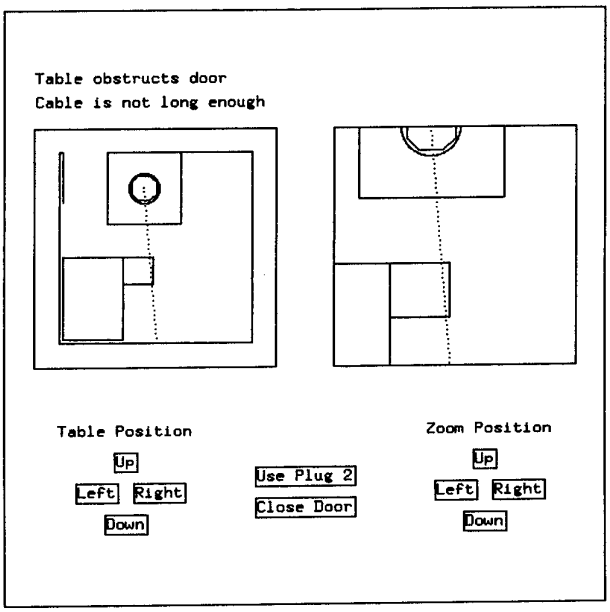

Figure 2b: The display after the table position has been redefined

Figure 2: Two sample screen displays