An Editing Model for Generating Graphical User Interfaces

by
Dan R. Olsen Jr.

Computer Science Department
Brigham Young University

Abstract

A basic architecture for a User Interface Management System is presented. The problem of updating a display in response to interactive commands is discussed. The basic architecture is then extended to include basic editing and browsing processes on arbitrary data structures. Editing templates are presented as a technique which embodies the entire manipulation process for a particular data structure / data display combination. Such templates in conjunction with the User Interface Management System are able to automatically provide a majority of the code required in an interactive application.

Introduction

Within the graphics community the concept of a User Interface Management System (UIMS) has come into usage[THO83]. Such systems have been developed to overcome the high cost of implementing interactive graphics programs with quality human-computer interfaces. Most such systems have concentrated on the problem of input dialogue management [KAM83, GRE85, JAC83, VAN83]. Having developed three such systems in our laboratory[OLS83, OLS85a, OLS85b, OLS85c], we have become concerned with the problem of data display in an interactive program. In implementing interactive programs we have found that the input dialogue can be programmed in a matter of hours or days, using our tools, but the code to update the display after each modification to the application data structure takes months to implement.

In attacking this display update problem we approached it from the point of view of an intelligent display processor. Our original architecture for an interactive program is shown in Figure 1.

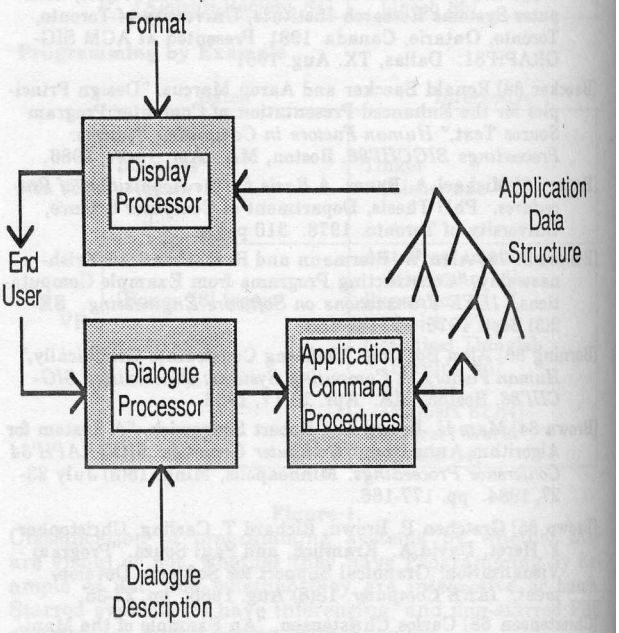

Figure 1.

In this architecture the input events are parsed according to the dialogue description and, based on the input, one or more of the application command procedures is invoked. It is the responsibility of these application command procedures to update some application data structure. It is the role of the display processor to create a graphical presentation of the application data according to the specified format. There are a wide range of possible formats for displaying the data which will only lightly be touched upon here. This architecture is somewhat similar to that proposed at the Seeheim workshop on user interface management [PFA85]. The key problem of interest in this paper is how the graphical image should be updated whenever the application data is changed. The obvious solution is to simply redraw the entire image. This is an extremely poor solution if one desires reasonable response time.

After some experience with the above model we determined that a closer relationship between the data editing commands and the display update functions is essential. A more acceptable system architecture is shown in Figure 2.

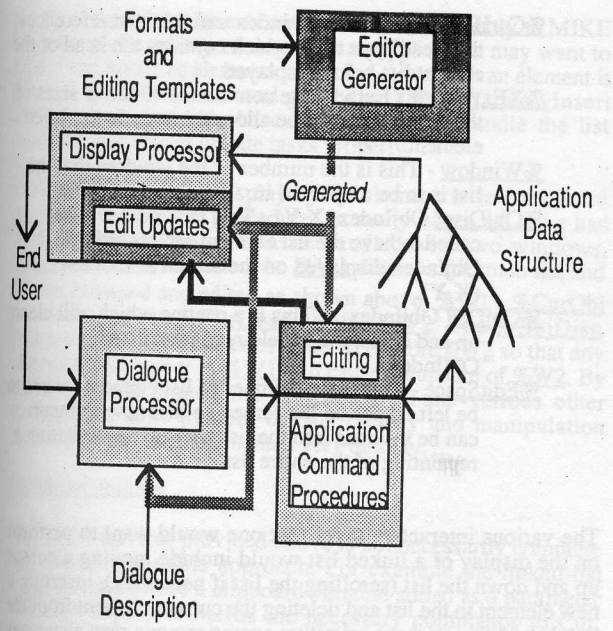

Figure 2.

In this architecture a large portion of the interactive dialogue is viewed as a process of browsing and/or editing the application data structure. Based on this view, a description of the application data is taken together with a selection of browsing and editing methods to generate the dialogue definition, display update procedures and application command procedures for the selected operations. Note that this does not generate the entire interactive program but simply the editing portion which is the most display update intensive. Any analysis, file management or help facilities can all be added. In our experience it is the editing and browsing operations with dominate an interactive program.

This paper will then proceed in the following fashion. First a data management facility called STUF (STrUctured Files) will be reviewed so as to define the possible data structures that might be represented in this UIMS model. This will be followed by a short description of MIKE (Menu Interaction Kontrol Environment) which is our input dialogue system. Having set the stage for our editing environment, editing templates will be described which provide the screen update facilities that we are interested in.

STUF

The STUF package was developed as a data model specifically for interactive applications. Our desire was to create a data model which would behave equally well in main memory or on secondary storage. We also imposed the requirement that STUF data must be accessible either relationally (as in a relational database) or as a linked structure (as with pointers in Pascal). The reason for this is that relational data models are very powerful but in many cases are too inefficient for practical graphics use.

Each STUF file has a specific *filetype* which is defined by a set of *datatypes* . A datatype is defined to be either a Record or a Union. A datatype consists of a list of fields each of which has a *fieldtype*. A Record is defined to have all of the fields in the list. A Union is defined to have one of the fields in the list (as determined by a tag field at run time). A Union is similar in capability to a Pascal variant record. A fieldtype is either an Integer, Real, Char, Boolean or a reference to an object of some other datatype. A Field can also have dimension so as to create fixed length arrays.

The links or references to other datatype objects pose a problem. In a relational model, tuples are linked to each other by matching key values. In programming languages this linkage is handled with pointers. The pointer solution provides the efficiency that we require but when pointers are written to secondary storage they lose all meaning. In addition the pointer model does not provide the flexibility found in the relational model. The STUF solution is to create a variable length array of tuples for each datatype in the file. A reference to a tuple is then stored as an index into this array. New and Dispose procedures are provided to allocate and deallocate tuples within a data type. In addition, the undeleted tuples in one of the arrays can be treated as a relation in the sense of a relational database to provide an associative access method for tuples. Since tuple indices do not lose their meaning when saved to disk or passed to another program we have our needed data facility.

One more point to consider is that of cursors for editing. When one is editing it is usually performed by moving some cursor or *current data object pointer*. Editing and browsing operations are then performed relative to this current data object pointer. When editing a STUF data file, such a cursor is represented as a tuple index. Many of the editing templates described below will be defined in terms of such cursors.

MIKE

MIKE is our dialogue handling system which is described in more detail elsewhere [OLS85c]. The important point to understand is how MIKE models the input dialogue. The basic interactive unit in MIKE is the command. A MIKE command is simply a Pascal procedure or function. MIKE accepts as its initial dialogue description a set of such procedures and then generates a compiled interface to these procedures. Interactively all procedures are presented in a menu and the selected menu item becomes the current command. Having selected a command, MIKE then prompts with a menu of all functions whose result type is the same as the type of the command's first parameter. These types can be any type from the application program. In addition to the functions in the menu, MIKE can supply primitive inputs for integer, real, string, function key and point types. MIKE continues accepting inputs until a complete command expression has been parsed. The command expression syntax is very similar to Pascal procedure invocation syntax with the exception that no punctuation, such as commas and parenthesis, is actually input. MIKE's interactive use of functions and procedures is similar in many ways to Smalltalk's interactive use of methods [GOL83].

Given such a primitive interface, a profile editor can be used to improve it. The profile editor allows menus to be restructured, prompts and echos changed, icons drawn, function buttons mapped to commands and help texts written. This fleshes out and enhances the user's view of the interface but it does not change the underlying model of command procedures and functions. The profile editor serves a similar role to Buxton's MenuLay [BUX83] in editing external presentations of dialogues.

This command model has proven to be much simpler to use than the state machine and grammar approaches that we have used previously. For the purposes of our editing model of interaction we can characterize all changes to the application data structure as Pascal procedures or functions. That is for each kind of change to be made to the data structure we generate a procedure which makes the change and performs

any necessary display updates. The generator then informs MIKE of the procedures' names and parameter types. From this information MIKE can create the necessary user input dialogue interface.

Editing Templates

The editor generation concept is based on the idea of editing templates. An editing template is designed as a presentation of a particular general class of data structures. Linked lists, symbol tables and trees are examples of such structure classes. An editing template then consists of a set of routines to do the following:

a. display application data from a STUF file using the model,
b. provide data structure traversal commands for browsing through the data image being displayed by the model, and
c. provide the editing commands for creating, deleteing and/or modifying the data presented using the model.

It should be noted that the commands provided must, in addition to performing their intended tasks, also update the display to reflect the results of their tasks. It is the display update which is most important to our discussion here.

In addition to the services that an editing template provides, each template also has a set of parameters which are used when creating an instance of a template. In this sense an editing template can be thought of as a macro except that the parameters may be lists and other data structures rather than simple text to be substituted.

In Smalltalk an editing template would be defined as a class. The services that it provides would be methods of the class and the parameters would be methods of the objects that the template is manipulating. In ADA an editing template could in most cases be represented as a generic package[GEH84]. Our work has been done in Pascal using a macro preprocessor but the concepts are the same. An editing template then is characterized by the data structure that it represents. An example is given below of an editing template for linked lists.

Linked-List Editing Template

As a first example of how editing templates function a linked list is appropriate. Since an editing template is meant to be a generic capability it must know a number of things about the linked list that it is to display. For example it must know how to find the head of the list, what field is used as the link for the list, how to display one of the elements of the list and how much display space to allocate to each element. This kind of information is provided by the following set of parameters. These parameters are all preceded by a percent sign so that their text is easily recognized in the generated routines.

%ObjType - The data type of the list elements to be displayed.
%Link - The field that is used to link elements of ObjType together.
%CurObj - The tuple index variable that is to be used as the cursor in moving up and down the linked list. This is an index into ObjType's array.
%HeadType - The type of object where the head of the list is stored.
%HeadField - The field in HeadType that points to the head of the linked lists being displayed. This field must reference data type ObjType.
%CurHeadObj - The tuple index variable that references the HeadType tuple which contains the head of the current list being displayed.
%XExt and %YExt - These contain the X and Y sizes of the screen space to be allocated to each displayed element.
%Window - This is the number of the window that the list is to be displayed in.
%ObjDisp(ObjIndex; X,Y) - This is a routine to be called to have the list element referenced by ObjIndex displayed on the screen at location (X,Y).
%ObjDel(ObjIndex) - This is a routine which will clean up and delete the list element reference by ObjIndex.
%EditSpace - This is the number of list element spaces to be left empty on the screen so that new elements can be inserted into the list without necessitating a repainting of the entire list.

The various interactive tasks that one would want to perform on the display of a linked list would include moving a cursor up and down the list (scrolling the list if necessary), inserting a new element in the list and deleting the current element from the list. In addition to these primitive operations one may also want to select a current item from the list using some other criteria such as a name. To accomplish all of these the template would provide the following command procedures directly to MIKE.

%WindowUp
{ move the list cursor up one element scrolling if necessary }
%WindowDown;
{ move the list cursor down one element scrolling if necessary}
%WindowLeft and %WindowRight
{ if, because of the size and shape of the window, the list is displayed in multiple columns then this will move the cursor left or right as the case may be and update %CurObj appropriately }
%WindowPageUp and %WindowPageDown
{ if the list is longer than will fit in the window then this will move the cursor backward and forward through the list one page at a time}
%WindowDelete;
{ delete the current object pointed at by %CurObj }

In addition to these command procedures which are exposed to MIKE the following additional service routines are generated.

Restore%Window
{ This completely refreshs the window whenever the window itself changes or the value of %CurHeadObj changes. }
%WindowUpdateCur;
{ This will simply update the display of the current object due to some modification to it by some other command }
%WindowInsert(ObjIdx)
{ This will insert the specified object into the linked list immediately after %CurObj and make it the current object updating the display appropriately }
%WindowChangeCursor(ObjIdx)
{ This will change %CurObj to the value of ObjIdx and update the screen appropriately }

Note that all of the names of the generated routines are parameterized by %Window so as to make them unique. Note

also that there is no insert command exposed directly to MIKE because of the variety of ways that an application may want to create and initialize elements of the list. After such an element is created by some command procedure the %WindowInsert service procedure can be called which will handle the list insertion and screen update tasks properly.

Note that the actual display of an element is left up to an application supplied procedure. In many cases we have had lists of lists to display. This is handled with two windows, %W1 and %W2 for example. %W1 displays the main list and can be browsed and edited as shown above. %W1's %CurObj cursor is also the %CurHeadObj for %W2. The %ObjDisp procedure for %W1 contains a call to Restore%W2 so that any change of the cursor in %W1 will cause an update of %W2. By combining the linked list template with the various other templates a large number of data display and manipulation techniques can be implemented very quickly.

Other possibilities

The set of services provided above is not necessarily complete nor the only possible approach. For example the linked-list template could have a clipboard variable added as a parameter and then supply to MIKE the necessary commands to Cut, Paste and Copy list segments to and from the window's clipboard. An additional feature might be an element from the list with a mouse rather than scrolling through the list.

Other data structuring mechanisms that we have implemented include unordered and sorted associative lists. Such lists are accessed by name or some other criteria. These two techniques view a given STUF datatype as a table of tuples for a relation and allow scrolling through and editing of such tables based on application supplied sort orders and selection criteria.

These structure editing templates are being combined with a forms editor which handles the element display functions. Other templates which we are still working on would handle network or schematic type displays such as is shown in Figure 3.

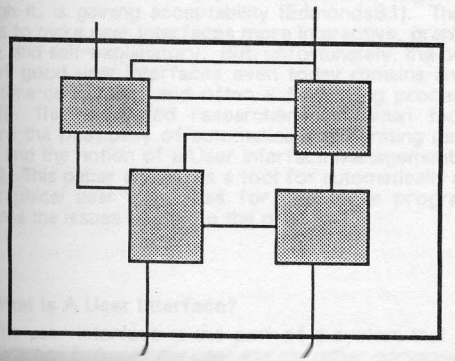

Figure 3.

Such display forms are more difficult because of the layout connection routing aids that one would want to provide automatically.

Summary

Our User Interface Management System therefore consists of a *command* based dialogue editor called MIKE which provides quick easily learned prototyping of dialogues along with refinement of the dialogue via a profile editor. The interaction is viewed as an editing of a file in STUF format which has the expressive power of both relational and linked data structures. Display tasks are handled by formats which describe how data should appear along with editing templates which provide generic data editing and screen update procedures for various data display techniques.

The main difference between this approach and other UIMS approaches is that the interaction is viewed as a data editing process rather than simply as an input dialogue parsing problem or a screen management problem. Having adopted this view one can then identify generic classes of display and editing techniques for various data organizations. Given such a class a template is developed which carefully links the input dialogue with the display management functions to provide vastly improved interactive response. Such a class can then be applied to specific data editing problems simply by binding the parameters.

Assuming that one has an application whose data organization can suitably use the editing templates provided (which is an openended set) then an interactive browser / editor can be created in a matter of days rather than months. For those parts of such an application which do not match one of the existing templates new code can be written and easily integrated with the templates. In fact after such code has been written it should be examined for its potential to become a new template itself.

References

[BUX83] Buxton, W. "Towards a Comprehensive User Interface Management System." Computer Graphics 17, 3 (July 1983).

[GEH84] Gehani, N. Ada: An Advanced Introduction. Prentice-Hall, 1984.

[GOL83] Goldberg, A. and Robson, D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.

[GRE85] Green, M. "The University of Alberta User Interface Management System." Computer Graphics 19, 3 (July 1985).

[JAC83] Jacob, R.J.K. Using Formal Specifications in the Design of a Human-Computer Interface. Communications of the ACM 26, 4 (April 1983).

[KAM83] Kamran, A. and Feldman, M.B. "Graphics Programming Independent of Interaction Techniques and Styles." Computer Graphics 17, 1 (Jan 1983).

[OLS83] Olsen, D. R. and Dempsey, E. P. SYNGRAPH: A Graphical User Interface Generator. Computer Graphics 17,3 (July 83).

[OLS85a] Olsen, D.R., Dempsey, E.P. and Rogge, R.A. "Input/Output Linkage in a User Interface Management System." Computer Graphics 19, 3 (July 1985).

[OLS85b] Olsen, D.R. "Pushdown Automata for User Interface Management." ACM Transactions on Graphics 3, 3 (July 1985).

[OLS85c] Olsen, D.R. "User's Manual for MIKE-2.0." Tech. Report 85-1, Computer Science Department, Brigham Young University, Provo, UT.

[PFA85] Pfaff, G. and ten Hagan, P.J.W. Seeheim Workshop on User Interface Management Systems. Springer-Verlag, Berlin, 1985.

[THO83] Thomas, J. J. and Hamlin, G. Graphical Input Interaction Technique Workshop Summary. Computer Graphics 17,1 (Jan. 1983).

[VAN83] van den Bos, J., Plasmeijer, M.J. and Hartel, P.H. "Input-Output Tools: A Language Facility for Interactive and Real-Time Systems." IEEE Transactions on Software Engineering SE-9, 3 (May 1983).