[Bea1991]

Programming of Application Interface and Image Access Made Simple

Author: **Beaulieu** Jean-Marie

Conference: Canadian Conference on Electrical and Computer Engineering

Quebec, Quebec, Canada

Sept., 1991, p. 23.1.1-4

Abstract: A good user interface is an important aspect of software products. We should also consider the effort and time spent for its programming. We present a “C” language extension facility which allows the definition of menus and fill-in forms. All the flexibility of the “C” programming language is preserved while making user interfaces programming easier. Following an object-oriented approach, we have also defined a set of functions and macros for image manipulation.

Programming of Application Interface and Image Access Made Simple,
Beaulieu Jean-Marie,
Canadian Conference on Electrical and Computer Engineering, Quebec, Quebec, Canada, Sept., 1991, p. 23.1.1-4.
[Bibtex]

Congrès canadien en génie électrique et informatique
Canadian Conference on Electrical and Computer Engineering
Québec, (Qué), Canada, September 25-27 Septembre 1991

PROGRAMMING OF APPLICATION INTERFACE AND IMAGE ACCESS MADE SIMPLE

Jean-Marie Beaulieu
Département d'Informatique
Université Laval, Québec, QC, G1K-7P4

Abstract

A good user interface is an important aspect of software products. We should also consider the effort and time spent for its programming. We present a "C" language extension facility which allows the definition of menus and fill-in forms. All the flexibility of the "C" programming language is preserved while making user interfaces programming easier. Following an object-oriented approach, we have also defined a set of functions and macros for image manipulation.

1. Introduction

The execution of application programs is much easier if we have an appropriate interactive environment. Hence, user interface systems based upon windows, menus, icons and mouses are now available for microcomputers and workstations. However, all this versatility may be not required for some applications. For example, the execution of image processing programs requires generally few selections from a set of alternatives and the input of few parameters before starting the program. The programming environment should be adapted to the application domain.

We should also consider the difficulty of the programming task. A complex interactive environment makes its programming more difficult [2],[3]. In our research, we have selected the basic elements of window and menu interfaces, while keeping these as simple as possible. This was originally developed to operate into a classical environment of terminals (VT100) connected to main frame computers (Vax) [1].

An appropriate interactive environment is needed to give more control to the user over the execution of application programs. We propose "C" language extensions to make the programming of menus and fill-in forms easier. The extensions are viewed as a set of new language instructions to define the static and dynamic parts of forms. Hence, forms which set the values of many variables could be programmed with few instruction lines. This tool is appropriate for scientific application development, as it provides appropriate interactive facilities with minimal programming cost in a standard programming environment.

The static part of forms is defined by a set of contiguous program lines which are an exact copy of what will appear on the terminal screen. Special characters in the text indicate positions where variable values will be written. Instructions, similar to read or write instructions, are used to read and/or write the values of variables to these screen positions which could be easily referenced by identification numbers. In the active part, the programmer defines the instructions that will be executed when the cursor moves to a new position or when the user modifies a variable value. Actions could also be associated with keyboard control keys by simple "IF_" instructions. Complex responses to user inputs could thus be programmed.

Modular programming can be used in the definition of large forms. A form can be composed of embedded sub-forms. The cursor can move freely from form to sub-form with the corresponding transfer of the program control from the main routine to the subroutine. The forms are hierarchically organized. The cursor can also be bound to form windows that overwrite the previous form. The user must leave (close) the form to get back to the previous one.

Following an object-oriented approach, we have also defined a set of functions and macros for image manipulation. An object consists first of a descriptor and a memory space. A module is responsible for the management of the memory space and descriptors. We view a file as main memory space, this simplifies the software programming. For the "image" object, operations to define (name, type, size, ...), to open, to close, to access and modify the pixel values are provided. New object definitions could be easily added (e.g., histogram, data base).

2. Fill-in forms

Fill-in forms and menus are displayed on the terminal screen and control the content of parts of the screen. A fill-in form description is composed of a static part, which defines the annotation text or background, and of a dynamic part which controls the output and modification of variable values. The dynamic fields are named fels (form elements). The user can move to any fel by using the cursor keys, enter a new value, and then start the program execution by pressing the "DO" key.

For example, the following program acquires the values of parameters and executes (calls) an image contrast enhancement function.

```
 1   # include "uiform.h"
 2   main()
 3   {
 4             char input[30], output[30];
 5             int  wsize; float coef;
 6      FORM( 1, 1,1,0,0, "%&", UiFullScreen )
 7      "         *** CONTRAST ENHANCEMENT ***       ",
 8      "                                            ",
 9      " input file  :  %1 &                        ",
10      "                                            ",
11      " window size :  %2 &                        ",
12      " coefficient :  %3 &                        ",
13      "                                            ",
14      " output file :  %4 &                        "
15      FORM_ACTION
16      POS( 1) FEL_STRING( 1, input,  UiRead,"%s", 30);
17      POS( 2) FEL_INT(   2, &wsize, UiRead,"%d", 1,50 );
18      POS( 3) FEL_FLOAT( 3, &coef,UiRead,"%5.2f",0.,99.);
19      POS( 4) FEL_STRING( 4, output, UiRead,"%s", 30 );
20      IF_KEY( UiKeyDo )
21          contrast_enhanc( input, output, wsize, coef);
22      FORM_END
23   }
```

The output screen for this program is:

```
 ____________________________________________
|                                            |
|        *** CONTRAST ENHANCEMENT ***        |
|                                            |
| input file  :  image01                     |
|                                            |
| window size :  5                           |
| coefficient :  2.00                        |
|                                            |
| output file :  image02                     |
|                                            |
|____________________________________________|
```

The C language preprocessor allows the addition of new language constructs by performing macro or symbol substitutions. Hence, we have developed appropriate language constructs to define fill-in forms. Three macros must be used, FORM, FORM_ACTION and FORM_END, with text or instructions included between them to customise the forms.

The character strings between FORM and FORM_ACTION define the static part. Each string corresponds to one line of the output screen. Hence, in the example program, lines 7-14 give an exact description of what the output screen will look like. Thus, even with a simple and conventional program development system, we get a good evaluation of the output results, without compiling and executing the program. The dynamic elements are the missing parts in this text, i.e. the variable values. Special characters, % and &, are used in the character strings to mark the positions of these dynamic elements or fels. In order to make referencing each of these positions easier in the following part of the program, a label or number is assigned to each one. They are the numbers following the % character, and are called fel-id. The arguments of the FORM macro are the form id, the position and size of the form, the fel delimitation characters and the form type (embedded or not).

The program lines between the FORM_ACTION and FORM_END macros define the dynamic operation of the form. Any C language instruction could be used here. However, functions are provided to write or read (i.e. wait for user input) variable values to a fel (dynamic field) position. For example, the FEL_STRING function, at line 16, outputs the value of string "input" and waits for user modification. The arguments are 1) the fel-id to identify the screen position, 2) the address of the variable, 3) the flag to indicate the type of operation (output only or output with user input), 4) the formatting string, and 5) the length of the string variable. For integer or real variables, a warning message appears if the value is outside the specified min-max range. Functions for button and multi-choice selection variables are also provided. When the read flag is used, the functions return only after a new value is provided, or a cursor or control key is pressed. Then, the UiNew variable can be tested to know if a new value has been provided. Other form status variables could also be used by the programmer.

The instruction lines between the FORM_ACTION and FORM_END macros are divided into groups by POS, IF_ and IF_KEY macros which define when the instructions are executed. The FORM_ACTION and FORM_END macros generate, in fact, a while loop. The POS(id) macro tests if the current cursor position is located at the fel id. If so, the following group of instructions is executed. At each iteration, only instructions from one POS macro are executed, while instructions from many IF_ macros can be executed. The argument of the IF_ macro is any valid C logical expression. The IF_KEY macro is used to test if the user has pressed a cursor or control key. In the example program, at lines 20-21, the "DO" key is used to start the execution of the contrast enhancement function. An "exit" key is also defined to exit and terminate the form (go to line 23).

The cursor can be positioned only on fels designated for value input (read). Cursor keys are employed to move from one fel to another. Cursor position is shown by highlighting the fel value. Combinations of bold, underline and reverse video are used to distinguish string, number, button or multi-choice screen fields. Complex interactive responses could be programmed by writing the appropriate C instructions after POS or IF_ macros. Hence, the modification of one fel (variable) value could required the updating of many other form fields. We can output new fel values with the flag UiWrite in FEL_ functions. Before the interactive phase (with cursor movement), an initialisation phase performs a scanning of the position to evaluate and display the fel values (without waiting for user input). We can assign value to the UiOpcode variable to perform operations like exit or update the form.

A menu is viewed as a special and simpler fill-in form. The dynamic screen fields (fels) correspond to buttons, which should be depressed in order to initiate actions. The actions are defined by the instructions following the corresponding POS macros. The macros MENU, MENU_ACTION and MENU_END are adapted from the form macros.

This fill-in form utility could easily be extended to provide mouse support, pop-up and pull-down menu, run time window modification of position and size, border and colour control ...

3. Modular composition of fill-in forms

A fill-in form can be embedded into another form. After the FORM_ACTION macro, any C instruction can be used, even the group of instructions that define a form, or a call to routines that define forms. Thus, sub-forms can be defined inside the instructions of a form, producing a hierarchy of forms. We distinguish two types of cooperation between a form and its sub-forms: embedded and overwrite sub-forms.

An overwrite sub-form takes control over its parent, and replaces the whole or part of the screen with its output. The cursor can only move inside the sub-form and the control returns to the parent form only when the user exits the sub-form. In this case, forms can be viewed as different screen windows and only one is active at a time.

An embedded sub-form is viewed by the user as forming an integral part of the parent form. The cursor can move freely between sub-form fels (dynamic fields) and parent fels. The cursor movement control routines automatically exit or enter sub-forms when needed. Embedded forms are useful for form programming. Their use allows the modular composition of forms. Hence, a regularly used part of a form has not to be copied each time. Instead, it is converted to a sub-form, included in a routine which is called when required.

For example, in image processing, we often need the image identification parameters: name, size, type. It is advantageous to define a sub-form and include it into a routine. Then, any application which requires an image identification can call the routine inside its form to embed the sub-form. The following program is an example of embedded sub-forms.

```
# include "uiform.h"
main()
{
        int  img_id, coef;
   FORM( 1, 1,1, 9,41, "%&", UiFullScreen )
   "*****************************************",
   "*      *** IMAGE PROCESSING ***         *",
   "*                                       *",
   "* input image                           *",
   "*   %1 &                                *",
   "*                                       *",
   "*                                       *",
   "* coefficient :  %5 &                   *",
   "*****************************************"
   FORM_ACTION
   POS( 1) get_image( &img_id );
   POS( 5) FEL_INT( 5, &coef, UiRead,"%d", 0,99);
   IF_KEY( UiKeyDo )
         process_image( img_id, coef);
   FORM_END
}
```

```
get_image( int * img_id_p )
{
            char name[30];
            int  npix, nlin;
      FORM( 2, 0,0,  2,32, "%&", UiEmbedded )
      "++ name: %2&                  ++",
      "++ dimension: %3& pix %4& lin ++"
      FORM_ACTION
      POS( 2) FEL_STRING(2, name,UiRead, "%s",30);
              if( UiNew)
                { *img_id_p = open_image( name);
                  UiOpcode  = UiUpdate;        }
      POS( 3) npix = number_pixel( *img_id_p );
              FEL_INT(3, &npix, UiWrite,"%d",0,0);
      POS( 4) nlin = number_line( *img_id_p );
              FEL_INT(4, &nlin, UiWrite,"%d",0,0);
      FORM_END
}
```

The output screen for this program is:

```
*****************************************
*       *** IMAGE PROCESSING ***        *
*                                       *
* input image                           *
*   ++ name: image01             ++     *
*   ++ dimension: 128 pix 128 lin ++    *
*                                       *
* coefficient :  10                     *
*****************************************
```

4. Accessing image data

Computer programs basically access and modify data stored in memory. However, data files are also used for non-volatile storage, to preserve data between program executions. Memory data can be accessed and manipulated more easily than file data: file data should be first copied into memory in order to access or modify them.

Following an object-oriented approach, we have defined a set of functions and macros for image manipulation. An object consists basically of a descriptor and a memory space. The Data Store module is responsible for the management of object descriptors and memory space. It provides a uniform view of objects that are stored either in central (in-core) memory only or in disk files (in-file). In-core objects are temporary and vanish when the program terminates. In-file objects are permanent. When the object is opened, the file content is mapped directly in central memory space. The file content can then be directly addressed by data pointers or memory addresses. Standard file operations, such as reading or writing of data lines, are not needed. File mapping makes file creation or opening similar to dynamic memory allocation, and provides the same versatility for data access. File mapping is not supported by all computer systems. It can be simulated by file loading and saving.

The Data Store module also provides header spaces in front of the data spaces to store data descriptive parameters. Therefore, the main functions are open/create/close data stores and read/write header data. This module is small, with only 600 lines of code.

Using the Data Store module, new modules to define and manipulate different types of objects can be easily created. Hence, an image module, with 900 lines of code, provides basic image access and manipulation functions: 1) definition, creation or opening of multispectral images (in-file or in-core), 2) direct access to pixel values, 3) image region selection, image copy with data type conversion. An image descriptor structure contains the information to access and manipulate the image data. This information is updated when an image is opened or created. Part of it is stored into the data store header. The image data is viewed as a vector and any pixel value can be directly accessed with the macro ImPixelValue.

5. Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada.

6. References

[1] F.J. Dixion, "Simplifying Screen Specifications: the Full Screen Manager Interface and Screen Form Generating Routines", The Computer Journal, Vol. 28 (2), pp. 117-127, 1985.

[2] M. Shaw, "An Input-Output Model for Interactive Systems", CHI'86 Conference, pp. 261-272, April 1986.

[3] B.A. Myers, "User-Interface Tools: Introduction and Survey", IEEE Software, pp. 15-23, January 1989.