EVENT DRIVEN GRAPHICAL MENU INTERACTION

Ted Crane and Jon H. Pittman
Cornell University
Ithaca, New York

ABSTRACT

A software package to facilitate the development of computer graphics control and menu display software independent of application code is described. This package is also useful for the development of interactive, but non-graphic, programs.

A graphics program is discussed as a finite state automaton. The program is treated as a set of menus (states) with actions (transitions) between menus based on the input received while in a given state. Virtual devices embedded in the menu and control structure are employed to supply input.

Program development tools based on these concepts and run-time software tools which provide the control structure, interaction mechanisms, and menu display for a graphics program are described.

The steps to be followed in the development of an interactive graphics program are outlined. Finally, an example of use of this package is provided.

Introduction.

Using current methods, application programmers must write code to display graphic menus, interact with these menus, and organize the flow of control within a program. This is usually done before the application portion of the program is written. Programming the menu and flow of control thus results in a large expenditure of time and effort before the programmer may directly address the issue at hand. This approach to the development of an interactive graphics program has a number of drawbacks:

1. The application programmer is forced to deal with the implementation of program control and interaction (i.e. syntax of the user interface) rather than with the application (i.e. semantics of the user interface).

2. Menu display and control structure coding is spread throughout the program and may even be duplicated in several different places within a program.

3. The applications programmer "reinvents the wheel" by writing new menu display and control code for each new program (or worse, adapts menus and control structure from a previously developed program without understanding the reasoning behind their design).

4. The potential exists for a lack of consistency in menu design within a program and between programs, thus resulting in poor user interfaces.

5. The lack of a common means by which interaction is implemented results in inefficient use of the underlying graphics software.

To address these problems and provide the applications programmer with an efficient means of creating a good user interface, several things must be done. First, menuing and control functions must be isolated from the application program. Second, the application programmer must be provided with a means of quickly defining and modifying the menu and control structure and an easy means of integrating application modules with the menu and control structure. Third, program modularity should be encouraged by separating application routines from the program structure and allowing them to operate as independent entities. Finally, a set of "primitive" interaction devices which handle both input and syntactic feedback functions must be provided. Previous attempts to address these issues have included menu design programs and software that implements virtual devices. The concepts and limitations of these two approaches are discussed below.

Menu design programs generally produce graphics display data in the form of source language "include" (or "require") text files which are inserted into the application program and then compiled. They may also produce actual language-specific source code that is compiled and performs program control flow functions. The files produced by such menu generators are usually edited by the application programmer to incorporate tests and coding specific to the application. While addressing some of the issues mentioned above, this approach leaves a final version of the program that is a mixture of menu display, graphic interaction, and program control code.

Virtual devices are library software procedures that deal with physical devices and simulate physical devices graphically. For example, a virtual pushbutton may actually be a graphic menu switch that is controlled by a stylus and tablet. Such virtual devices provide the means to easily program individual instances of menu interaction, but do not incorporate an overall menuing and flow of control structure for the program.

Proposed Solution.

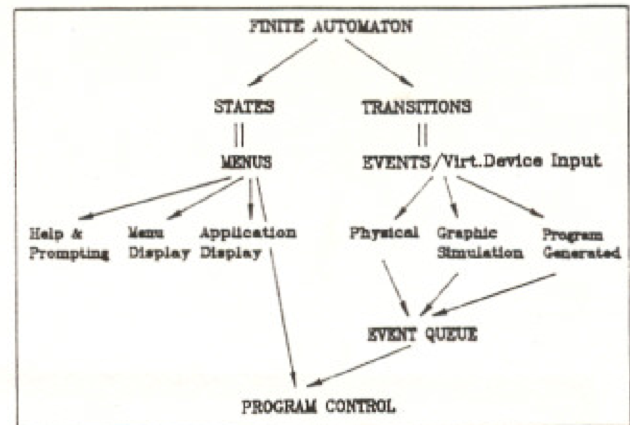
We have tried to address these problems by designing a system that allows the user to describe menus and control structure through a menu description language. The language is translated and program control tables which drive the application program through a runtime software package are generated. The application code consists of a set of action routines which are invoked in response to the menu and control system's recognition of a syntactically valid input.

This system allows the programmer to describe the menu interaction and program control structure as a finite state machine. Each state in the finite state machine is equivalent to a graphic menu and transitions between states (menus) correspond to valid inputs received from virtual devices.

For each menu, a set of valid virtual device inputs is defined. The interactive menu display is composed of a set of virtual devices. Some of the virtual devices such as buttons, valuators, and selectors may be displayed graphically while others, such as timers and keyboard inputs, have no graphical representation.

This system is event-driven. It has the capability to queue individual events (virtual device inputs) and process them on a first in, first out basis.

Whenever a valid input for a particular menu is found, a transition to another menu may occur and an application-specific action routine may be called. In this way, the application program is driven by the menu and control structure.



Components of Menu Interaction System

Finite State Machine.

The graphical menu interaction system is based on the finite-state machine concept. A finite automaton has been defined by Hopcroft and Ullman:

> "A finite automaton (FA) consists of a finite set of states and set of transitions from state to state that occur on input symbols from an alphabet Σ. For each input symbol, there is exactly one transition out of each state (possibly back to the state itself). One state, usually denoted q0, is the initial state, in which the automaton starts. Some states are designated as final or accepting states."

In a finite state machine, the current state is the result of the previous set of transitions. Thus, the current state is the result of the stream of all input symbols to the machine.

A finite automaton can be represented by a directed graph, called a transition diagram. In such a graph, the verticies of the graph denote the states and the arcs of the graph denote the transitions. The transition diagram can, in turn, be represented by a table of individual states with their allowable inputs and the actions to be taken upon transition from one state to the next.

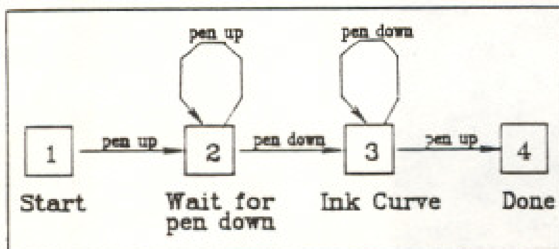
Finite automata are used in many traditional programming applications, such as compiler design and text editors, but are a useful means of structuring a graphics program as well. Since graphics control code is often repetitive code. (a set of case or goto statements), the control structure can be represented as a transition diagram in table form. A general purpose routine can be provided to direct the flow of control along paths specified in the transition table. The same table can also contain information regarding the treatment and display of virtual devices.

The following example illustrates the concept of a graphics program as a finite state machine. It is an inking program which accepts input and draws a line as if it were being inked with a fountain pen. Inking is a fairly standard graphics input technique and can easily be represented as a finite state machine.

First, we show how an inking routine is written using traditional methods.

```
 nking loop, sample coding
! Note application-specific code (underlined).
!
   POINT_COUNT = 0
   do
       begin
       ! display menu - may be lots of code
       ! read new pen value - via graphics package
       end
   until PEN_SWITCH eqlu DOWN
   do
       begin
       POINT_COUNT = POINT_COUNT + 1
       POINT[POINT_COUNT,X] = PEN_X
       POINT[POINT_COUNT,Y] = PEN_Y
       ! display existing inked line
       ! display menu - duplicate of above
       ! read new pen value - via graphics package
       end
   while PEN_SWITCH eqlu DOWN
```

This program can be represented by the following finite state diagram:



Sample Finite State Machine – Inking Loop

The finite state diagram can, in turn be represented by a finite state machine table:

Current State	Input Pen Token UP	DOWN
	New State	
1 Start	2	invalid
2 Wait Pen Down	2	3
3 Ink Loop	4	3
4 Done	none	none

FSM Table to Describe Inking Loop

Although this example is a fairly trivial one, it is clear that the finite state machine can be an efficient way to represent a graphics program. Of course, as graphics programs increase in size and complexity, the economy of dealing with menus and graphic interaction through a finite state mechanism increases greatly.

Implementation.
The graphical menu interaction system consists of several components:

* a collection of virtual devices;
* an event queue for handling virtual device inputs;
* a vocabulary for defining menus and control structure;
* a translator to convert the menu and control description to machine-readable code which may be linked with an application program;
* runtime software to handle menu display, virtual devices and menu interaction, and control flow.

Each of these components will be described in detail below.

Virtual Devices.
The virtual devices in the graphical menu interaction system process user input tokens (i.e. a button push, a keyboard string, a control character, a digitized value, a timer expiring) and emit "events" which are placed in a queue to be processed sequentially by the runtime software. A virtual device may be an interface to a physical device or it may simulate a physical device using a combination of software and another physical device.

Six virtual device types are included in the initial design of the graphical menu interaction system: keyboard, button, clock, valuator, and selector.

1. **Keyboard.** A keyboard handles a string of characters or a control character that is input from a physical keyboard device. A string of characters is defined as a sequence of characters that is terminated by a control character (a character with an ASCII value between 0 and 31). A control character may itself be an input token. If an input string is terminated with any control character other than <CR> (ASCII 13), the input string and the control character are placed on the event queue as two separate events: the string and the control character.

   The keyboard definition contains a list of valid tokens. A token may be a string, a numeric expression, a filename, a date, or a control character. When keyboard input is encountered, the list of valid tokens is searched for a match. If one is found, the transition associated with it is taken and the associated action routine, if any, is called. A facility for checking the range of numeric values and dates is provided and several discrete ranges may be specified.

If a token is received that is not in the list of valid tokens, an optional "otherwise" directive can describe a default transition to make.

Before accepting input, the keyboard virtual device displays a prompt which is defined in the keyboard virtual device specification.

The current design of the keyboard virtual device assumes a physical keyboard device. It is possible that a graphic simulation of a keyboard will be incorporated. This will allow for such features as software selectable QWERTY or DVORAK keyboards and the construction of special-purpose keyboards for a particular application.

2. **Button.** A button may be one of two types: momentary contact or toggle. A momentary contact button always generates a 'closed' event, i.e. it causes the same thing to happen every time it is hit. A toggle button inverts its previous state (saved from a previous input), and can be used to switch back and forth between two options.

   Buttons may be either physical or graphical. A physical button may be a device such as a button box or keyboard program function key. A graphical button is simulated by a sensitive box area on the screen.

   A string may be specified as the label for a button. This string is displayed in the graphical representation of the button and some physical button devices allow the display of a label in an LED display near the button.

   Since each button causes a transition to a new menu, help text, and action routine may be associated with each button device.

3. **Clock.** A clock virtual device is one that generates an input token after some specified time interval has elapsed. The graphical menu interaction system incorporates two types of clock virtual devices: repeating global clocks and one-shot local timers.

   A repeating global clock generates an event every "n" seconds. Its action is not affected by the particular state (menu) which the program is currently in, hence the designation as "global". This type of clock is useful for handling continuously changing displays, gathering program statistics, etc.

   A one-shot local timer is restricted in its action to a specific menu. When the state (menu) is entered, the clock begins its count and, if the menu is still active after "n" seconds, the timer generates an event. This type of timer is useful to prompt for input or cause a timeout to go to a different state.

   As with any other virtual device, both clock types generate input tokens. Therefore, a new menu and an action routine may be associated with a clock input. Several different clocks with different time increments may be associated with a given program or menu.

4. **Valuator.** A valuator is a virtual device that returns a value or set of values within a specified range. A valuator may be one, two, or n-dimensional. A one-dimensional valuator (physically, a slide or dial potentiometer) returns one value within a specified range. A two-dimensional valuator (physically, a joystick or digitizing tablet) returns two values within a specified range. The graphic menu interaction system also has the capability to deal with n-dimensional valuators which return n values.

   A valuator may return absolute or relative values. An absolute value is a number within a range, a relative value is the amount of change since the last time the valuator returned a value. The range from which the valuator is to return its values must be specified for each dimension of the valuator. A function can be specified for each value. This function modifies the value thus allowing scaling or nonlinear output from the valuator.

   Like buttons, a valuator may be a physical device or a graphic simulation. A valuator generates an event with each use, thus an action routine, help, and a new menu may be associated with a valuator.

5. **Selector.** A selector virtual device allows one to select "one of many" in a set. A selector presents a list of options and lets one choose from that list. It can be viewed as an array of buttons but, unlike a collection of individual buttons, the selector acts as a single device. A selector may be momentary or toggle, just like a button. A toggle selector allows only one member of the list of choices to be active at any given time, whereas the momentary selector allows one to choose a member of the list for a given action but does not "remember" its state.

   A selector is defined by specifying a screen position and number of columns. The graphic menu interaction system generates the bounds of each sensitive area with the selector array.

   For each selector choice, the choice definition may specify a display label, help text, an action routine, and a new menu state.

6. **Keypad.** A keypad virtual device simulates a calculator and permits the user to input numeric values. It is, in effect, a special-purpose keyboard with the additional capability to do arithmetic computations. This functionality could be implemented by an application programmer. but would require a significant amount of effort. The keypad device is one of many higher level virtual devices that could be available to simplify the application programmer's task.

   The keypad functions as a whole unit, and an event is generated only after the user hits the "enter" key on the keypad. Until that time, the keypad functions may be used without invoking any other program components.

The programmer defines a keypad by indicating a position, an optional default value which is placed in the numeric area when the menu is entered, an optional message which is placed in the keypad's message area, a new menu, help text, and an action routine.

The keypad has the following components:

1. **Message area.** The message area displays the message string specified in the keypad definition.

2. **Numeric area.** The numeric area displays the numeric value being calculated.

3. **Numeric keys.** The 0-9, ".", and "E" keys allow the entry of numeric values.

4. **Operator keys.** The "+", "-", "*", "/", and "=" keys perform the the usual arithmetic calculations.

5. **DELETE key.** The delete key deletes the last digit typed on the keypad.

6. **CLEAR key.** The clear key sets the numeric value to 0.0.

7. **ENTER key.** The ENTER key causes an event to be placed on the event queue and passes the keypad input (the value of the expression currently in the numberic area) to an action routine.

A menu is composed of a collection of these virtual devices. In addition, a menu may contain several other entities such as:

* a prompt to direct the user to take action;
* a timed prompt to direct the user to take action after a specified time interval has expired;
* a set of display ports in which user data is displayed;
* a set of message ports in which help text is displayed;
* groups of virtual devices;
* submenus;
* graphic entities such as text, lines, and boxes.

Overlapping virtual devices are displayed and selected based on the order in which they are defined — the first device is "in front" and obscures devices behind it. Message ports have a higher priority ("in front of") than virtual devices but a lower priority in any selection tests. Application display ports have lower priority ("behind") than virtual devices.

Although the current system design has only these six virtual device types, it is open-ended enough that more virtual device types could be added at a later date.

<u>Event Queue.</u>
The virtual devices implemented by runtime software generate events and place them on an event input queue. When an event is available on the input queue, the menu software dequeues it and evaluates it in the context of the current menu. Unexpected events (touching an inactive button, for example) may be rejected and, if requested by the application designer, a warning issued. When a match occurs between the event and a transition defined for the current menu, the menu software may call an action routine (so the application can make use of the input) and/or the program may change state to a new menu. Once the event has been processed, the queue entry is deleted.

The runtime software is divided into two independent portions which communicate through the event queue. The "virtual device input" portion of the software runs at an elevated priority level (in the VAX/VMS environment, AST driven) relative to the "menu control" software. The "menu control" software is normally idle (in the VAX/VMS environment, hibernating or waiting for a local event flag), waiting for an event to be placed on the event queue. Sufficient memory must be allocated to the event queue so that the potential combination of a fast input device (such as a free running digitizer) and a slow transition action routine (such as a complex engineering analysis on every digitized point) will not overflow the queue. When events are missed, a warning message will be displayed.

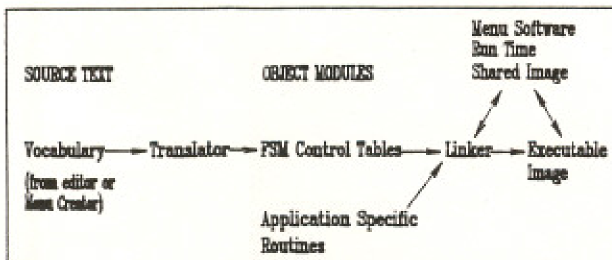
The runtime software also keeps a backwards chain of transitions. In the menu description it is possible to specify a "RETURN" transition. If such a transition is taken, the program goes to the previous state in the chain.

<u>Menu Vocabulary.</u>
The menus and control structure of the program are specified by a keyword-based language. This language describes the various components that make up the menu and the actions to be taken in response to the manipulation of these components. This language consists of text strings written in a sequential file.

The complete language description is too large to list here, but a representative portion of it is presented below. This portion contains definitions for the menu construct and some virtual devices.

```
<menu>       := MENU(
                NAME = <name>
                [, VIEWPORT( <position_value> ) ]
                [, HELP_LIBRARY = <filename> ]
                [, HELP_TEXT = <help_text> ]
                [, INIT_ROUTINE = <routine_name> ]
                [, DONE_ROUTINE = <routine_name> ]
                [, PROMPT = <help_text> ]
                [, TIMED_PROMPT = <help_text> ]
                [, PROMPT_TIMER = <number> ]
                [, OPTIONS = { WARN | NOWARN } ]
                [, MESSAGE_PORT( <position_value> ) ]
                [, <submenu> ] 0-n
                [, <display_port> ] 0-n
                [, <virtual_device> ] 0-n
                [, <group> ] 0-n
                [, <graphic_entity> ] 0-n )
```

```
<keyboard>   :=  KEYBOARD(
                 PROMPT = <string>
                 [, <token> ] 1-n
                 [, OTHERWISE( NEW_MENU = <new_menu> [,
                    ACTION = <routine_name> ] ) ] )

<token>      :=  TOKEN(
                 { TEXT = <string>
                   [, MATCH_CHARS = <number> ] |
                   CONTROL_CHAR( <number> ) |
                   NUMBER( { <number> | <range> } ) |
                   FILENAME |
                   DATE }
                 [, NEW_MENU = <new_menu> ]
                 [, HELP_TEXT = <help_text> ]
                 [, ACTION = <routine_name> ] )

<button>     :=  BUTTON(
                 { MOMENTARY | TOGGLE },
                 { GRAPHIC [( <position_value> )] |
                   PHYSICAL( <name> ) }
                 [, LABEL = <string> ]
                 [, NEW_MENU = <new_menu> ]
                 [, HELP_TEXT = <help_text> ]
                 [, ACTION = <routine_name> ] )

<valuator>   :=  VALUATOR(
                 { GRAPHIC [( <position_value> )] |
                    PHYSICAL( <name> ) },
                 { ABSOLUTE | RELATIVE }
                 [, NUM_DIMENSIONS = <number> ]
                 [, RANGE( { <range> |
                    ( <range> )
                    [, ( <range> ) ] 1-n } ) ]
                 [, FUNCTION = <routine_name> ]
                 [, NEW_MENU = <new_menu> ]
                 [, ACTION = <routine_name> ] )

<selector>   :=  SELECTOR(
                 { MOMENTARY | TOGGLE }
                 [, POSITION( <position_value> ) ]
                 [, NUM_COLUMNS = <number> ]
                 [, <choice> ] 1-n )

<keypad>     :=  KEYPAD(
                 POSITION( <position_value>
                 [, DEFAULT_VALUE = <number> ]
                 [, MESSAGE = <string> ]
                 [, NEW_MENU = <new_menu> ]
                 [, HELP_TEXT = <help_text> ]
                 [, ACTION = <action_routine> ] )
```



Menu Software Implementation

Menu Software Implementation.

In a VAX/VMS environment, the graphical menu interaction software has three major components:

* **Translator/Compiler.** The translator is a standalone utility which translates the source language describing the program finite state machine into tables. These tables are subsequently interpreted by the runtime system to generate graphic menu displays and control program execution. The tables are produced in the form of object modules, i.e. lists of commands and data to be processed by the VAX/VMS linker. The finite state machine may be described in one source file or several separate modules since individual menus are labeled by a unique global symbol which can be resolved by the linker.

* **Runtime support.** The runtime support system consists of a collection of library routines gathered together in a VAX/VMS shared library image. Since the routines in this image are intended to control the actual program execution and VAX/VMS shared images do not contain a program transfer address, it is necessary for the main routine of an application image to call the "main" menu startup routine. This may be achieved by a small amount of coding in the application program, but it is preferable to specify one of the menus as the "main" menu. The translator then generates a minimal routine which calls the runtime support library and emits object code which instructs the linker to use the address of that routine as the program transfer point.

* **Menu Creator.** The menu creator is an application program that can be used to interactively generate the menu vocabulary source files, thus allowing interactive design and layout of menus and control structure.

In addition, the menu software relies on the actual vocabulary (described above) for describing menus and a database of virtual devices.

Steps for Use of the System.

To use the graphical menu interaction system, one would follow these steps.

1. Define states and transitions necessary to implement the application program. The definition may initially be small and grow during program development. This is the paper planning stage.

2. Write dummy routines to represent the initial application code. Compile these routines.

3. Design a graphic menu, if any, to implement each state. The menu creator may be used to speed this process. Commonly used submenus may be defined and used within several major menu displays. Some menus, such as those which allow input only from physical devices (such as a keyboard), may not require graphic menus.

4. Get the menu vocabulary source files in good shape.

5. Translate menu description into object code (using the menu translator).

6. Link prototype program to test menu concepts, interaction, and graphic representation. If changes are desired, return to menu creator or text editor.

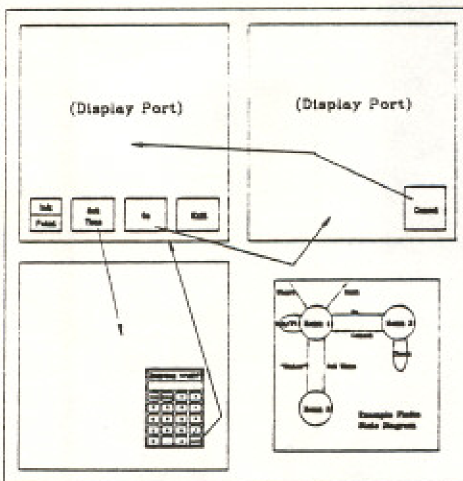
Having done all the hard part...

7. Replace dummy action routines with working application routines and create a working application program.

<u>Example.</u>
To illustrate how the graphical menu interaction system works, we have developed a simple graphics program as an example. This program permits a user to ink (or rubber-band) a path around around an object (such as a cube). The program then generates a walk-around view of the object by moving the eyepoint along the inked path.

This example consists of three menus (states). The first state allows one to set a time increment for the path to be followed, select between inking and rubber banding, move around the cube, and exit from the program. There are additional states for setting the time increment and moving about the cube.

A state diagram of this program is shown below. The states (menus) are represented by circles, and the transitions (virtual device inputs) are represented by arcs.



The first menu is the main menu of the program. It has a main display port, a valuator for inking and rubber banding, a selector for inking or rubber banding, a "go" button, a "set time" button, and an "exit" button. The main display port displays a plan view of the cube. The inking and rubber banding valuator is in the same location as the display port and allows one to define a path around the cube. The selector for inking or rubber banding allows one to choose the input mode for defining the path. The "go" button transfers to a menu which uses a repeating clock event to initiate display of successive views of the cube along the inked path.

This menu includes a "cancel" switch which returns to the main menu. The "set time" button transfers to a menu with a calculator to enter the number of seconds it will take to traverse the path. Finally, the "exit" switch ends execution of the program.

The menu definition language description of the program is as follows:

```
program( name = CUBEWALK,
                                 ! action routines
 external( DISPLAY_MAIN,         ! display main viewport
           GET_PATH,             ! read path data from
                                 ! valuator
           SET_INK,              ! set inking mode
           SET_RUBBER_BAND,      ! set rubber band mode
           DEFINE_TIME,          ! read time from
                                 ! calculator
           DISPLAY_WALK,         ! display walk around
                                 ! cube
           CHANGE_POSITION       ! change position along
                                 ! walk

 menu( name = MAIN MENU,
       display_port( viewport( bottom = .2 ),
                     DISPLAY_MAIN
                   ),
       valuator( graphic( bottom = .2 ),
                 absolute,
                 num_dimensions = 2,
                 range( (min=0.,max=1.),
                        (min=0.,max=1.)
                      ),
                 new_menu = MAIN_MENU,
                 action = GET_PATH
               )
       selector( toggle,
                 position( left=.05, right=.2,
                           bottom=.05, top=.2
                         ),
                 num_columns = 1,
                 choice( label = "Ink",
                         new_menu = MAIN_MENU,
                         action = SET_INK
                       ),
                 choice( label = "Rubber Band"
                         new_menu = MAIN_MENU,
                         action = SET_RUBBER_BAND
                       )
               )
       button(   momentary,
                 graphic( left=.25, right=.45,
                          bottom=.05, top=.2
                        ),
                 label = "Go",
                 new_menu = DO_IT
             ),
       button(   momentary,
                 graphic( left=.5, right=.7,
                          bottom=.05, top=.2
                        ),
                 label = "Set time",
                 new_menu = SET_TIME
             ),
       button(   momentary,
                 graphic( left=.75, right=.95,
                          bottom=.05, top=.2
                        ),
                 label = "Exit",
                 new_menu = return
```

```
menu( name = SET_TIME.
      display_port( viewport( bottom = .2 ),
                    DISPLAY_MAIN
                  ),
      keypad( position( left=.75,right=.95,
                        bottom=.05,top=.45
                      ),
              default_value = 5.,
              message = "Enter number of seconds",
              new_menu = MAIN_MENU,
              action = DEFINE_TIME( VALUE )
            )
    )
menu( name = DO_IT.
      display_port( viewport(),
                    DISPLAY_WALK
                  ),
      clock( global,
             time_increment = 1.,
             new_menu = DO_IT,
             action = CHANGE_POSITION
           )
      button( momentary,
              graphic( left=.75, right=.95,
                       bottom=.05, top=.2
                     ),
              label = "Cancel",
              new_menu = MAIN_MENU
            )
    )
)
```

These tables completely define the menu design and interaction. Note the seven action routines declared external. They are the only source code that need be written to complete this program. Using this method, it is possible to generate the menus and control structure before writing the application code.

Conclusions.

The graphical menu interaction system provides an alternative to the traditional approach to application program development. It provides a tool for the application programmer to develop an application system faster and more efficiently. In computer programming circles, there is an increasing awareness of the need to provide programmers with tools to improve their productivity. It is our feeling that the graphical menu interaction system is such a tool.

The system, as currently designed, still requires the programmer to spend an inordinate amount of time designing graphic interaction. However, it does allow the programmer to think about graphic interaction in fairly high level terms. Additional work in the area of software tools for menu and interaction design certainly seems warranted to further reduce the design time and expertise required to produce a graphics program.

While designing this system, we have received generally positive interest from the programmers and implementors of applications software in our laboratory. In addition, many of these individuals have, at one time or another, had similar ideas. This indicates to us that the ideas are certainly worth exploring. We hope that this preliminary design provides the impetus for further work in this area.

Acknowledgements.

The authors would like to thank Carlos I. Pesquera, whose "Menu Creator" program was used to draw the figures in this paper. We would also like to thank the Program of Computer Graphics at Cornell University for providing the facilities to prepare this paper and the figures associated with it.

References.

1. Brown, James W. "Controlling the Complexity of Menu Networks". Communications of the ACM. July 1982. Vol. 25 7 Pg. 412-418.

2. Computer Graphics. Quarterly report of ACM/SIGGRAPH. January 1983 Vol.17 1.

3. Foley, J.D. and Van Dam, A. Fundamentals of Interactive Computer Graphics Addison-Wesley 1982.

4. Hopcroft, John E. and Ullman, Jeffrey D. Introduction to Automata Theory, Languages, and Computation. 1979 Addison-Wesley. Reading, Mass.

5. Rubel, A. "Graphic Based Applications Tools to Fill the Software Gap" Digital Design. July 1982.