SOFTWARE FOR DEVICE-INDEPENDENT GRAPHICAL INPUT

by

Griffith Hamlin
Los Alamos National Laboratory

ABSTRACT

This paper discusses a three-level model and a graphics software structure based on the model that were developed with the goal of making graphical applications independent of the input devices. The software structure makes graphical applications independent of the input devices in a manner similar to the way the SIGGRAPH CORE proposal makes them independent of the output devices. A second goal was to provide a convenient means for application programmers to specify the user-input language for their applications.

The software consists of an input handler and a table-driven parser. The input handler manages a CORE-like event queue, changing input events into terminal symbols and making their terminal symbols available to the parser in a uniform manner. It also removes most device dependencies. The parser is table driven from a Backus-Naur form (BNF) grammer that specifies the user-input language. The lower level grammar rules remove the remaining device dependencies from the input, and the higher level grammar rules specify legal sentences in the user-input language.

Our implementation of this software is on a table-top minicomputer. Our experience with retrofitting existing applications indicates that we can find a grammar that removes essentially all the device dependencies from the application proper.

Key words: device-independence; graphical input; user interface.

I. INTRODUCTION

The proposed SIGGRAPH CORE standard provides a large measure of device-independent graphical output [1]. It represents a synthesis of many years experience in producing graphics output on various devices. The same measure of device-independent graphical input, however, does not exist. The CORE proposal does eliminate application program dependence upon specific physical input devices, provided the physical devices can be cast into one of several logical-device classes (pick, button, valuator, keyboard, locator). However, logical-device dependence of the application program is not addressed. Also, it is hard to see where some input devices (for example, a voice recognition unit) fit into the logical-device classification scheme. Van den Bos has described an alternative to the logical device model [2]. This paper presents a model that can incorporate the Graphics Standards Planning Committee (GSPC) logical-device model, but adds another layer between the logical devices and the application.

II. MODEL OF USER INPUT

The literature identifies three different types of processing of the user's input: lexical, syntactic, and semantic (see Fig. 1). In looking at several existing programs at the Los Alamos National Laboratory, we observed that device dependence is usually introduced into the

application at the middle level. At the first (lowest) level of the model in Fig. 1, users select and use physical-input devices. Software provides them with low-level (lexical) feedback. Examples of this are tracking a table or light pen and echoing of text. This processing is device dependent but application independent, although the device may be able, through subroutine calls, to specify one of several alternative types of lexical-level feedback. This level of software also changes physical-device input into logical-device input, and corresponds fairly well to a CORE-like input subsystem.

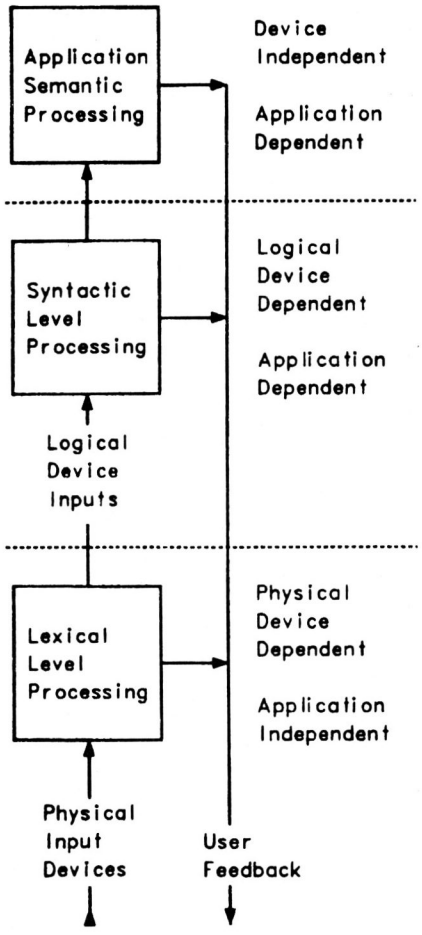

Fig. 1

At the second level of the model, the processing becomes application dependent. In most applications we studied, the processing was also device dependent because the application required certain (logical) devices to be used. Here the application-dependent syntax is checked to see if the series of user input forms a legal phrase in the user-input language of the application. Syntactic-level feedback is given at this level. This feedback may involve reasonable amounts of application-dependent processing, but this processing is not the primary processing of the application. If the input specification corresponds to a legal phrase in the user's input language, this processing often changes the input phrase (often a single parameter of a command) into a standard form for use by the rest of the application.

At the third (highest) level of our input model, the processing is application dependent and is usually logical-device independent. Changing user input phrases into some standard form in the middle level typically removes the device dependencies in the applications we studied. This third-level processing may gather up several such phrases (parameters) until a complete sentence (command) is available, and then perform the processing requested by the command. Command processing is the primary processing of the application.

III. SOFTWARE STRUCTURE

According to our model, software for graphical input should be able to isolate the device dependencies in an application to, at most, the middle level, and perhaps eliminate most device dependencies even from that level. Fig. 2 shows the basic software structure we use to try to accomplish this. At the bottom of Fig. 2, we have indicated a standard CORE-like input subsystem that accepts physical-device input and converts it into logical-device input. This subsystem corresponds to the lower level of the model and is application independent but device dependent. In the CORE proposal, the main application program would access this logical-device input directly through the event queue. However, we have added two modules at this point: the input handler and a parser. The input handler is part of the lower level (application independent) processing and the parser is at the middle level of the model. We feel that with proper grammar design the input handler and the lower level grammar rules in the parser can be used to remove the logical-device dependencies from the input.

The function of our input handler module is to continually scan the input-device event queue, changing input events into terminal symbols for the parser and making them available to the parser in a uniform manner. Our input handler module recognizes some special user-input actions that allow users to enable/disable the various input devices, thus giving them some

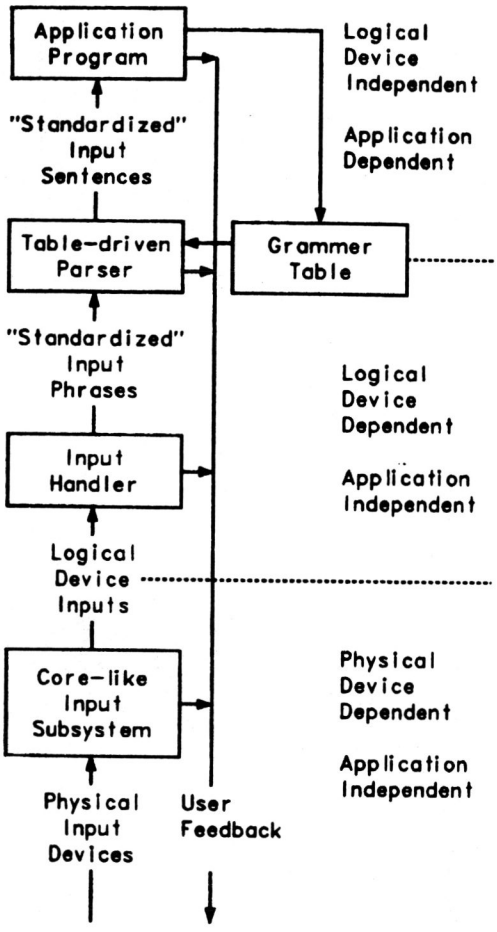

Fig. 2

measure of dynamic control over the choice of input device. It also removes most device dependencies by passing to the parser the type of input rather than the device from which the input came. For example, it makes selecting a menu item with a locator device indistinguishable from typing the name of the menu item on a keyboard or pushing a button associated with the menu item. It also makes a position indicated on a locator device indistinguishable from that position entered by typing the coordinates on a keyboard. Foley and Wallace have discussed simulating one class of input device by another class [3]. Generally, our input handler makes such simulation invisible to higher levels of software. Only the type of input is returned, not the device from which it came. The input handler may, by default, enable and arm a device that is normally used for the expected type of input, but this is not required.

Sometimes, however, the application programmer may wish to generate different syntactic-level feedback for input from different devices. Our software discourages this, but allows for it in order to handle special circumstances. For this reason an application can, by a request to the input handler, find out the logical device that produced each input terminal symbol. Use of this information, of course, introduces device dependencies at the next higher level. To isolate these dependencies as much as possible, our next level software is a table-driven parser whose lower level grammar rules are designed specifically for each application to remove the remaining logical-device dependencies. These lower level rules have the effect of transforming the input phrase into a standard form. The parser's semantic interface at this level provides the syntactic user feedback by calling application-dependent routines. These grammar rules can, with some effort, distinguish between different input devices. However, it is just as easy not to do so. We hope this will encourage device independence. With experience it might be possible to discover a set of often-used, lower level input grammar rules and build them into the input handler.

The higher level grammar rules used by the parser determine if user input forms legal statements in the user input language, and provide a semantic level interface to the application program, passing it user input commands that have been transformed into a standard form. This corresponds to the third (highest) level of our model.

IV. IMPLEMENTATION

This software structure has been implemented in Fortran on a DEC LSI-11 microcomputer. This places the first two levels of input processing on the microcomputer, isolating all device dependencies to the microcomputer so that the main application program running on a host computer is input- and output-device independent. The physical input devices include a keyboard, a data tablet, a joystick, a voice recognition unit, several knobs and switches, and a thumbwheel cursor, which is part of the Tektronix 4014 storage tube output device. Output devices include the storage tube and a high-resolution (768 x 1024) black-and-white video display.

An implementation of the proposed CORE input subsystem was not available, so our input handler scans the physical devices directly, funneling all input into a stream of terminal symbols to the parser. An application program can pass to the input handler the physical

layout and entries in a menu on either of the two locator devices. The input handler will then make indistinguishable to the parser keyboard entries of the menu items and locator hits on the menu items. Special keyboard keys are recognized by the input handler and allow the user to enable/disable the thumbwheel cursor and display of the menu on the storage tube screen. Although the application program can specify that it wants one device or the other enabled for the next input, the user is not required to use it. The only way the application programmer can absolutely require use of a certain device is to request the input handler to provide device information in the form of specific device-dependent terminal symbols and to specify these device-dependent symbols in the grammar used by the parser. Then, if the user uses anything but the required device, it will not parse and the application program's error routines will be invoked. Although this type of use of the system is possible, the system discourages this use by making it harder to require a specific device than it is to allow use of any device. This behavior is opposite to the way many existing systems work.

The parser used in this implementation is LANG-PAK [4], a table-driven parser in fairly wide use. It allows the application programmer to enter a grammar, along with semantic operations to be invoked upon matching the various grammar rules. Sample input sequences can then be interactively entered and checked by the parser, so that the application programmers can check their grammar. The semantic interface between LANG-PAK and the application program was changed in this implementation so that any Fortran statement or statements can be placed as semantic specifications anywhere in the grammar. These statements will be executed when the associated grammar rule, or partial rule, is matched in the user input string. These statements are typically CALL statements to the various application program subroutines that perform the actions associated with various user input.

V. USES

For interfacing to existing Laboratory applications, we have built on top of the graphical input software a small LSI-11 resident program that communicates with the main time-sharing network at the Laboratory. This system provides a user-tailorable front-end to other existing applications that run on the time-sharing system. User input from the various graphical and text input devices is mapped by the parser and associated semantic routines onto the input format required by existing Laboratory applications. Different grammar and semantic routines are used for different existing applications. The first application, MAPPER, is an existing Laboratory application for producing presentation slides [5]. MAPPER reads a file of commands that specify the slide, including x,y coordinates of graphics entities such as boxes, circles, and lines. The common mode of using MAPPER is to use a text editor to construct a command file and then execute MAPPER with this file as input. We observed that the most time-consuming aspect of this use of MAPPER is correctly entering the x,y coordinates of the various graphical entities. Iteration is necessary because we are forced to use a nongraphical keyboard to specify graphical objects. A grammar was written for the LSI-11 resident front-end program that accepted input from all devices and converted it to the MAPPER format. A menu was laid out providing an item for each MAPPER command. With this menu, users can trace existing sketches of slides or create new sketches on a data tablet. No modification to any program running on the time-sharing system was required. Using this front-end on several test slides, we found that the time required to generate a slide was reduced considerably because of the reduced number of tries needed to position the graphics objects correctly.

One problem with this usage is providing convenient syntactic-input language phrases and feedback on all devices without modifying the application on the host computer. We prefer to be able to specify graphical objects differently on different devices. For example, MAPPER requires a center and a radius to specify a circle. If we use the tablet for the center point, we must change to a valuator device or simulate a valuator with the tablet to give the radius. It would have been straightforward to use the tablet to enter a center point and a point on the circle, but the existing application was not written that way. Our solution was to perform syntactic processing in our microcomputer on two tablet points, a center and a point on the circle, to calculate a radius that was then passed to MAPPER. This modification of the user-input language worked successfully, but it introduced some device dependencies into the lower levels of the input-language grammar and introduced some device-dependent processing of the tablet input.

The second application to use this system was a small two-dimensional interactive drawing program called DRAWIT. DRAWIT allows the definition of sub-objects and instances of these sub-objects to be placed at various positions on the picture. It allows modification or deletion

of these sub-objects as entities and also allows drawing and deletion of individual lines and text in the picture. The user-input language was purposely kept quite simple to facilitate use on different logical-input devices. Each command consists of a logical button (to specify the command), optionally followed by a location. For example, the location following move or draw commands specifies where to move or draw to. The only exception is the command to place text in the picture, which is followed by a text string. This syntax was well suited to our Tektronix thumbwheel cursor, which allows us to couple a single keyboard character with the cursor location. Also, we could easily simulate this syntax using only the keyboard, using only the cursor, or using only the tablet by designating part of the tablet as a menu of commands. Our input handler alone was able to remove all device dependencies from higher levels of software in this case, which allowed the user to choose among all possible ways of using our three physical devices to specify two input items. Our parser in this case essentially performed the identity function.

With use of DRAWIT, we observed a user preference for the tablet device. We also observed that it was annoying to be forced to alternately move the tablet stylus between the locator area and menu area of the tablet, especially on the draw command, which was often repeated many times in succession. Therefore, we modified DRAWIT slightly to improve its use with the tablet by allowing the command to be omitted if it was the same as the previous command. Even though the impetus for this modification came from a particular device, DRAWIT is still input-device independent in the sense that it processes input from all devices in the same manner. Indeed, it does not know which device produced its input.

VI. CONCLUSIONS

This software structure has been used to successfully retrofit existing applications and remove device dependencies. This structure allows existing applications to make use of newly available input devices. The hardest problem has been providing good syntactic level phrases and feedback on all devices without modifying the existing applications.

With this system, we tend to continue using the current input device (to preserve tactile continuity) until we really need to switch to another one. This use is made possible by the user being able to select/deselect input devices without the application program's intervention. However, an application may require first an input from one device class and then an input from another device class. We can simulate one class of device with another. We took this approach in both applications described above. This system was successfully able to hide the simulation from the application, but it was not able to do so and still make optimal use of all the devices from a human engineering viewpoint. These difficulties were remedied in the two test applications, either by changing the application's user-input language or by introducing some device-dependent processing of some user input.

The table-driven parser has isolated the input language specifications and has made experimenting with user input languages much easier.

We conclude from our limited use of this software that it can successfully eliminate application dependence upon specific logical-input devices. However, the software can not guarantee successful human engineering for all devices.

REFERENCES

1. "Status Report of the Graphics Standards Planning Committee," Computer Graphics (13,3), August 1979.

2. Jan Van den Bos, "Definition and Use of Higher Level Graphics Input Tools," Computer Graphics (12,3), August 1978, pp. 38-42.

3. J. D. Foley, and V. L. Wallace, "The Art of Natural Graphic Man-Machine Conversation," Proceedings of the IEEE (62,4), April 1974, pp. 462-471.

4. L. E. Heindel, and Jerry Roberto, LANG-PAK - An Interactive Language Design System, American Elsevier, New York, 1975.

5. D. H. Dahl, "MAPPER User Manual," Los Alamos Program Library Write-up J5AJ (1979).