User Interface Considerations for Algebraic Manipulation Systems

Gregg Foster

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

ABSTRACT

The background of mathematical manipulation systems is presented. Current user interface tools and issues are then surveyed, with a focus on the special problems of algebraic manipulation systems. A mathematical expression editor is used as an illustrative example.

1. Background

Mathematical manipulation systems have been around for a long time. The first mathematical display system was the "finger in the dust"‡ (FITD) system developed thousands of years ago. FITD was cheap, readily available (weather permitting), and the workspace could be arbitrarily large. But it suffered on several accounts: it had no long term memory, its computational backing was the human brain, and it provided few tools for expression manipulation.

Hundreds of years ago, "paper and pencil" (PAP) became firmly recognized as the dominant mathematical display technology. PAP retained most of the benefits of FITD and added a form of expression memory. Once on paper an expression could be preserved and referred to later. Computational backing was the human brain augmented by bits of paper with previous techniques and results preserved. PAP, now with more sophisticated backing, is still the dominant mathematical display technology.

In the 1960s, computer-based "algebraic manipulation systems" (AMS) were built for the first time. These systems of algorithms were inferior to PAP in many ways, particularly as display systems since they were usually limited to

‡ There is some controversy over whether the finger in the dust came before the the "stick in the sand" (SITS). SITS may have been the first formal display technology, but it is safe to assume that FITD was first used for mathematical display. There is suggestive evidence that FITD technology was used by pre-mathematical primates several millions of years ago.

small, fixed size character sets. Early AMSs also required users to adapt to the sequential nature of the programs. This "unnaturalness" made the systems difficult to learn and use. However, the AMSs *did* make the algorithmic power of computers accessible to mathematical manipulation. Some manipulations too large or complex for human beings to do in reasonable time (celestial mechanics calculations, for example) *could* be performed by an AMS *in reasonable time*. AMSs are powerful, but not pretty[1, 2, 3].

The algorithmic promise of AMSs *can* be beautiful and generally useful. Recently, many forms of display and manipulation technology have come together making displays of at least PAP quality possible and supporting a new class of natural manipulations.

2. Display Technology Dramatis Personae

We start with an introduction to the available hardware advances and software tools. It is always desirable to have a system design so strong and true that it does not depend on any particular technology. On the other hand, it is usually wise, when designing systems, to create systems that use available technology. Since it is our intention to build systems, the design considerations that follow necessarily focus on current technology.

2.1. Bitmapped Screens

A bitmapped screen is a raster display device where each picture element is under software control. This provides excellent display flexibility. Any character that can be represented by a pattern of bits can be placed on the screen. This certainly includes the traditional symbols of mathematics (integral signs, matrix braces, and the rest). This also encompasses any new symbols system designers (or users) may find useful. Bitmapped screens are typically capable of displaying many more characters than the usual 24 lines by 80 characters CRT.‡

‡ We are using Sun Workstations(tm) which can display about 45 lines of 115 "CRT-sized" characters at once.

2.2. Pointing Devices

The ability to point to things on the computer screen is the most powerful new tool in the mathematical manipulation armamentarium.

The usual pointing device is the *mouse*. A mouse is small box (about half the size of a pocket calculator) with 1 to 4 buttons on top. A mouse moves easily on a table top and has rollers, an optical position pad, or magnetic sensors to keep track of its motion. Moving the mouse gives asynchronous feedback of a screen cursor position. The mouse movement and buttons are used as input devices.

A pointing device (mouse, light pen, or finger) can be used to indicate any point or logical object visible on the screen. An object or point once indicated can be saved or manipulated subject only to software limitations.

2.3. Menus

A menu is a list of currently active possibilities. It is usually a small window containing a list of keywords or short command names. A menu can be used to assist memory or specify a command. If a user wants to do something with an object, he points at it and calls up a menu. The menu will usually contain only commands that make sense in the current working environment.

There are several different flavors of menus. *Constant* menus are on the screen all the time. They are predictable but take up valuable workspace. A *Pop-up* menu appears only when the user asks for one. It may cover existing information temporarily, but it is invisible most of the time. A compromise form is *Pull-down* menu. Pull-down menus have the name of the menu visible most of the time, but the whole menu is exposed only when the name is selected. The usual method of command selection from a menu is by mouse (or other pointing device). Once the menu has been made active (by moving into it or requesting it or pulling it down), commands can be selected by pointing. Feedback is provided by highlighting. A command is chosen by releasing the mouse button when the desired command is highlighted. If a menu is used simply as a memory aid, no command need be issued: no menu item need be selected.

There is a trade-off between short and long menus. Short menus are easy to scan and don't take up much screen space. But if many entries are active, some not visible on the short menu, it will be necessary to go through one or more levels of menu indirection to get to the needed command. Long menus

tend to have the needed command immediately visible, but a particular command may be difficult to find in a long list of commands. Moreover, long menus may cover up too much valuable display space.

An often-touted advantage of menus is that they help prevent command errors by making only reasonable commands available. It is true that menus do prevent the "syntax error" mistakes that command languages are prone to. However, menus promote a sneakier error: the user may mistakenly pick the wrong menu entry and do something reasonable, *but not what he intended.* Worse still, the user may not even notice that the wrong thing was done and blithely carry on.

2.4. Windows and Icons

Windows can be thought of as paper-thin computer screens scattered on a desk (the real screen). Windows allow many objects and processes to be visible and active on the screen at once. Windows may overlap (some may even be completely be hidden), moved about, exposed, hidden, stretched and shrunk.‡ In an AMS, each interesting sub-expressions could be given a separate manipulation window. There could be several, normally not displayed, windows for infrequently needed displays such as command audit trails† or system status.

One way to let a user know that a window or object is available but not currently displayed is to leave a reminder on the screen. *Icons* are windows replaced by a reminder, a small graphic symbol. For instance, the audit trail window, when not in use, might be displayed as a postage stamp sized picture of a ledger book placed near the edge of the screen. An AMS display is a busy place and icons will make noticeable targets.

2.5. Fonts

A variety of font disciplines and font faces are currently available. Variable sized (vector) fonts are flexible: they can be grown, shrunk, or stretched in virtually any way desired. Unfortunately, since they are necessarily generated on the fly (you don't get all that flexibility for nothing), they are slow. Static fonts

‡ There are also strong advocates of screens "tiled" with non-overlapping windows. There is a tradeoff between having a few completely visible windows and having many partially visible (or totally invisible) windows. Judicious use of icons, we feel, tips the scale in favor of overlapping windows.

† An *audit trail* is a list of the previous commands or checkpoints.

are less flexible, but much faster and, if bit-tuned, more attractive. We believe static fonts are best for most mathematical applications where only a few font sizes are required.

A range of font faces and sizes is a valuable asset to an AMS display. Traditional mathematical forms, such as large integral signs, special alphabets, and small sub- and super-scripts, can be faithfully rendered. Different font styles (or colors!) are useful in suggesting different mathematical "types" (for example, matrices can be shown in bold type and polynomials can be shown in script).

3. User Interface Considerations

Many user interface issues were hinted at above. Now we get more specific about the ideas driving AMS interfaces.

3.1. Expression Input

Current AMSs primarily use *FORTRAN* style input, such as:

```
x**2+2*x+1
```

A better approach might be to parse the input character by character, formatting and displaying the system's guess of what the expression is as it parses. Aesthetic advantages aside, this technique also avoids errors since the user gets immediate feedback of what the system thinks he typed in. Figure 1 shows a character by character example of system response to typing in the above expression. There are difficulties to overcome in this area, but work is being done[4].

Another possibility is to accept user drawn input using a stylus and sensitive pad. The computer then tries to make sense of the user's scribblings. This may become useful in the future, but, in spite of a history of attempts, no current system does an adequate job of processing human drawn mathematics[5].

3.2. Beginners and Experts

It is difficult for a system to satisfy both beginners and expert users. The experts want to be able to specify their commands easily; they know what they want to do and want to do it as quickly as possible. Beginners, on the other hand, want much more informative displays to help them find their way through the system; they are willing trade some speed for some assistance.

Input	Displayed expression
x	x
*	x ?
*	x ?
2	x^2
+	$x^2 + ?$
2	$x^2 + 2$
*	$x^2 + 2$?
x	$x^2 + 2x$
+	$x^2 + 2x + ?$
1	$x^2 + 2x + 1$

Feedback bitmapped input

Figure 1.

Unfortunately, the amount of help desired is not the only issue (if it were, novices and experts could both be satisfied with a "terse/noterse" flag). Experts prefer the faster naming conventions in command languages and beginners prefer the slower, but more helpful, menus[6]. More on this below.

In AMSs there is a second dimension to the the expertise problem: there are mathematical experts and mathematical novices. In fact, there may be an inverse correlation between computer-system expertise and math-applications sophistication.

3.3. Workspace Management

Screen "real estate" is a major concern in AMSs, where several large expressions (as well as other objects) may be displayed at the same time. The object of current interest should be displayed prominently. Other objects may need to be mnemonically accessable (perhaps as icons). The problem here is that users may focus their attention on many objects over a very small time period. One approach is to have many objects visible at once with, possibly, various levels of completeness. For example, in a window based display, there might be several windows visible at once: some as icons, some partially covered, and some completely visible. Each window would take up only a portion of the screen, allowing the user's eyes to scan the imperfect, but adequate,

information. Human beings have done reasonably well with messy desk tops for some time.

3.4. Menus vs. Command Languages

> "Factors that increase informativeness tend to decrease the amount of available workspace and system responsiveness."[6]

As suggested earlier: beginners want big informative displays to assist them in negotiating the complexities of a system; experts know what they want the system to do and how to tell the system to do it – they simply want it done with a minimum of fuss.

Both menu-based systems and command-language-based systems have their benefits. Menus tend to appeal more to novice users because they are easy to use and require little prior knowledge about the system. A user is encouraged to explore the system by trying out unknown commands he sees on the menu of possibilities. Menus help maintain integrity since only commands relevant to the current situation can be chosen. Command languages are generally attractive to expert users because they are fast and flexible. Command languages give users the freedom to type in anything they like to the system. Command languages also can be easily composed into programs, stored, and called up as modules for more ambitious programs.

Both have their drawbacks as well. Menus are slower, especially if the number of choices is large, which necessitates large menus or levels of indirection (in either case, display is slow and information tends to be obscured). Since all choices on a menu are valid at the moment the menu is invoked, a mistake in choosing a menu entry will do *something*, but perhaps not what the user wants. The main drawback for command languages is the large amount of knowledge required to use them. The user must learn the syntax and command names. Command languages also discourage exploration of the system since a user must know a command name to try it.

Ideally, an interface would use both menus and a command language and would allow free movement between them. Novices would be the primary users of the menus. Menus would be used less and less as a user became conversant with the command language. Alternatively, command-language-written modules would become part of the menus – giving users some control over the system abstraction level. An expert might use menus as an occasional memory aid or to explore an unfamiliar part of the system[7].

3.5. Man-Machine Interaction

Norman[8] refers to four stages of man-machine interaction: Intention, Choice, Specification, and Evaluation. For man-AMS interaction, there are apparently five different stages: Intention, Choice, Argument Selection, Command Setting, and Evaluation; where Argument Selection and Command Setting are a finer view of specification. If a user wants to change the subscript on a displayed variable, that is the *intention*. *Choice* is the act of settling on a sequence of activity to accomplish the intented change. Continuing out example, indicating the particular expression the user wants to replace the subscript with is the *argument selection*. Giving the command to replace the subscript and indicating where the offending subscript is located is *setting the command*. Looking at the result to see what happened is *evaluation*. Each stage has different goals, techniques, and problems.

3.5.1. Forming an Intention

The user somehow decides what she wants to do. This is the most amorphous stage. What a user wants to do is influenced by knowledge of the system being used and other analogous situations[9] she has been involved in. The user, in this stage, sets up goals.

3.5.2. Choosing an Action

The user decides what command or command sequence will achieve the intention. The user's knowledge of system limits what she thinks she can do and, therefore, what she really *can* do. The user may already know a good command sequence for the current goal, or she may figure one out, or she may have one suggested. The last possibility shows the most promise for interface assistance. Menus can serve to remind or suggest appropriate commands. A help command might serve the function in a command language milieu.

3.5.3. Picking the Arguments

The user indicates the (proposed) command arguments. Here the user has decided on a course of action and indicates the objects to be manipulated. She may indicate the objects by naming them or by pointing to them. Pointing versus naming was previously discussed (Beginners and Experts).

3.5.4. Setting the Command

The command is picked and executed. The user evaluates system feedback and something is actually done. An expression is moved, changed, evaluated, or otherwise manipulated. Again, the command may be pointed at, for instance by mouse and menu, or named by, perhaps, keyboard input. It is at this stage that system would probably want to "checkpoint"[‡] the session activity. The system can return to a checkpoint or repeat a checkpointed command.

3.5.5. Evaluating the result

Is this what the user wanted to happen? This is the stage where the user decides what to do next. Did the previous command do expected (or at least interesting) things? Was the user's goal satisfied? Or, has the goal been deflected? The procedure returns to the intention stage and a new goal is chosen.

3.6. Other considerations

A collection of small, but not necessarily minor, things.

3.6.1. Modes

A system should be as "modeless" as possible. Commands or mouse clicks should always have the same meanings regardless of the situation. This isn't always possible, but should striven for[10]. If more than one mode is necessary, the modes should be as obviously different as possible. Menus ameliorate some of the problems with modes by presenting only relevant current commands.

3.6.2. Undoing

It must be possible to undo things. Any system that does not let a user recover from unexpected results can hardly be considered friendly. It is also a good idea for un-undoable or hard-to-undo things to be more difficult to do. One way to make dangerous things more difficult to do is to require confirmation. Confirmation might be "really delete your filespace? (y or n)" from the keyboard, or might require more than one click of a mouse button. Another way is to put the items in a flashing red menu and start up some klaxons.

‡ By checkpointing, we mean that a new display/command is in effect and the previous one is remembered by the system.

3.6.3. WYSIWYG

What you see is what you get. By this we mean two things: the display must reflect the current state of the displayed expression and a picked sub-expression must exist as displayed. The first is obvious. An example of the second is: picking the x in $1/x$ should yield x, not x^{-1}. The user should not have to figure out what expression she's picked; it should be instantly apparent.

3.6.4. Moving between input devices

System design should minimize the need for the user to move between input devices (for instance, keyboard and mouse). This can't be completely avoided, but can be diminished.

4. An Illustrative Example: The "Safe" Editor

A visual (screen) editor for mathematics will be a direct result of a properly designed AM user interface. The user will see typeset quality mathematics and, since subexpressions are manipulable objects, be able to cut and paste existing expressions to form new expressions. Normally, pieces of existing expressions can be shipped off to the AMSs algorithms for simplification or evaluation. As an example, we explore a variation on Andersen's "Cosmetic Editor"[11]. The *Safe* editor allows only changes to an expression which do not change the semantics of the expression. It would be used for cosmetic operations like rearranging terms. This editor would allow a user to rearrange a displayed expression to suit himself (but only consistent with the original semantics).† It assumes everyday mathematical domains where operators like "+" and "*" are associative, and their operands commute.

Informally, the user picks arguments (subexpressions of the displayed expression) and puts them into a "bag". The bag attempts to make sense of its contents by traveling up the expression tree until it finds a common operator for the bag's contents (possibly the expression root). Once the arguments are chosen, a command is chosen from a (pull-down) menu, and a destination for the command is indicated. Finally, the user "doesit" and the new expression is checkpointed. The new expression is made the current display.

† Most AMSs display in canonical form. This is probably necessary as an internal form, but not particularly desirable for output. Since it's unreasonable to expect the system to always know the best form to display an expression to a particular user, the user must have the ability to organize his expressions (and his thoughts) as he wishes.

The *safe* editor edits a displayed expression, in place, in a window on a mouse driven bitmapped display. Pull-down menu names appear along the top of the window as appropriate. Any mouse button held down over a menu name will allow the user to pull the menu down; releasing the button over a command chooses the command.

Three classes of commands are given directly from the mouse by combinations of buttons being held down (figure 2).‡

Command	Button Pattern
Right	00●
Left	●00
Up	0●●
Down	●●0
Pick	0●0
Escape	●0●
Undo	●●●

Mouse Button Commands

Figure 2.

The *pick* command picks a destination or an argument to be put in the bag. The *doit* commands, *right* and *left*, are used to place the contents of the bag to the right or left of the destination. *Up* and *down* are the tree traversing commands. They allow the user to move up and down the expression tree from a picked sub-expression. *Escape* empties the bag and returns to the previous checkpoint. A user would do this to abort a partially completed operation. *Undo* returns to the checkpoint before the last. This is useful when the user has just done a *doit* (creating a new checkpoint) and doesn't like the result.

The patterns of mouse buttons are intended as graphic mnemonics for the commands. *Pick* and *escape* are roughly duals. *Left* and *right* are as expected. *Up* and *down* are more or less mnemonic (harmless experimentation would clear

‡ The required mouse button virtuosity in this editor is questionable. It has not been tested.

up any confusion between them). *Undo* seems to reflect what people do when frustrated with the mouse.

The basic editing loop is:

pick (bag)
{command}
{{destination}}
{doit}

That is, any number of picks followed by a command, an optional destination, and a doit.

Each iteration of the loop goes through the previously mentioned five stages of interaction (intention, choice, argument selection, command setting, and evaluation). In the example that follows the accent is on what happens in the last 3 stages (argument selection, setting the command, and evaluating what happened).

A simple example will make things clearer. The expression to be edited is $y/z + x$. We want the x to be the first term. We move the cursor over the expression (possibly traveling up or down the expression tree) until x is displayed in a reverse video box. We *pick* x as the argument (it goes into the bag) by pushing the center mouse button. The bag contents are highlighted in the expression by drawing open boxes around the sub-expressions that are in the bag. Next, we pull down the command menu and choose "move". Then we move the cursor over y/z (the destination) and *left doit*. The expression is now $x + y/z$ and we're done.

The *Safe* mathematical expression editor, even in the simple form presented, does many things right. It is designed to use the new graphics and display devices. The combination of menus and commands given directly from the mouse would probably be easy for beginner to learn, but somewhat faster than a pure menu driven system. The ability to point at objects is used effectively in this editor. Especially good are the two kinds of higlighting feedback (the open boxes drawn around "bagged" sub-expressions and reverse video highlighting on sub-expressions under the cursor). Since the *Safe* editor works in a single window with pull-down menus, display space is not a problem. It expresses the five stages of interaction in a reasonable fashion. The editor is largely modeless, supports various levels of "undoing", and requires no use of the keyboard (presumably, direct input was made outside the expression

editor).

The simple version of *Safe* presented above illustrates several principles and techniques, but it is inadequate as a full-fledged mathematical expression editor. The *Safe* editor was designed to be simple and safe so this criticism is unfair, but, since this is an illustrative example, we make it anyway to show how the editor might be extended. It would have to give the user the ability to save expression forms or use previously created expression objects. This implies a more complex window arrangement since the user may want to have several objects available on the display at once. A directly usable command language would be needed to give the user command access to the AMS underneath the expression editor.

5. Conclusion

The strengths of the above basic technologies (high-resolution screens, pointing devices, high-powered graphics software, and display space management) can be used to give the user of an AMS a powerful but natural interface. The high-resolution bitmapped screens can display more and and can do it better than fixed-font CRTs. The most useful new tools for users of a mathematical manipulation interface are the pointing devices. Graphics software (including windows, icons, menus, and fonts) give the user display quality and clarity previously unavailable in computer-based systems.‡ Software that supports both menus and a command language encourages system exploration while allowing speedy manipulations. Space management techniques give the user enough, but not too much, information to work with.

An AMS with a powerful and natural interface would be useful enough to replace pencil and paper as the primary mathematical manipulation tool. Given additional progress in the other areas of knowledge and algebraic algorithms, the next step for is a more intense consideration of man-machine interaction.

‡ Color displays are available, but at a higher cost and usually lower spatial resolution. It will be interesting to see what use can be made of them.

References

1. John K. Foderaro, *Typesetting Macsyma Equations*, December 1978. MS report (UC Berkeley).
2. William A. Martin, *Symbolic Mathematical Laboratory*, January 1967. PhD Thesis (MIT).
3. J. K. Millen, *CHARYBDIS: A Lisp Program to Display Mathematical Expressions on Typewriter-like Devices*, August 1967. ACM Symposium on Interactive Systems for Experimental Applied Mathematics.
4. John K. Foderaro. Personal Communication.
5. William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill Book Company, 1979.
6. Donald A. Norman, *Design Principles for Human-Computer Interfaces*, December 1983. CHI 1983 Conference on Human Factors in Computer Systems.
7. Tony DeRose and John Gross, *Ned/Narc Users' Manual*, November 1983. Berkeley Computer Graphics Laboratory.
8. Donald A. Norman, *Four Stages of User Activities*, October 1983. Workshop on Intelligent User Interfaces.
9. Frank Halasz and Thomas P. Moran, *Analogy Considered Harmful*, March 1982. Conference on Human Factors in Computer Systems.
10. Larry Tesler, "The Smalltalk Environment," *BYTE*, pp. 90-162, August 1981.
11. Carl Andersen, *A Cosmetic Expression Editor*, 1983. Unpublished Manuscript.