User Friendly Authoring Languages: An Alternative Approach

Charles R. Myers, Jr
Roger A. Schaefer

Grumman Aerospace Corporation

ABSTRACT

Two general types of programming languages have been developed for computer based educational systems. The first type of language, which is patterned after such high order languages as PASCAL or FORTRAN, provides great programming flexibility. However, its structured, syntactical constructs require either that an experienced programmer be involved in lesson generation or that instructional personnel become skilled in sound programming techniques. Its use frequently results in other problems as well, such as communication difficulties between instructional and programming personnel in the implementation of the lesson design and development process. To avoid these problems, the second type of language was developed. It allows instructional personnel to generate on-line instructional materials without acquiring sophisticated programming skills. This second category of languages is often thought of as being "user friendly." Such languages usually take an algorithmic approach to instruction and rely heavily on prompting as the means of lesson program entry. They serve very well for many applications, but their use has not been without problems. Not the least of these problems has been a lack of flexibility in the presentation and creation formats. This paper describes the OMEGA authoring system, a lesson authoring approach that provides instructional personnel with the positive features of both of the above language types. The approach has been implemented on an educational system that includes capabilities for the integrated use of interactive videodisc and three dimensional simulation. The paper first relates some basic facts about computer systems in general, and then discusses the various aspects of user friendliness in the context of educational programming. It then describes and evaluates both the traditional and OMEGA approaches to user friendly authoring.

Users of educationally oriented computer systems have a limited number of authoring language options available to them. Most systems are equipped with two types of authoring languages. The first type offers the author a great deal of flexibility in instructional design, but requires a very sophisticated knowledge of computer programming to be used effectively. The type, which is advertised by manufacturers as "user friendly," requires little or no computer background, but limits the author to a relatively rigid instructional presentation format.

Today's instructional environment requires a language option that is more adaptable than either of the two discussed above. An author's computer background can range from nonexistent to extensive, and instructional applications vary from simple to extremely complex.

This paper describes the OMEGA authoring system developed by the Training Systems Department of the Grumman Aerospace Corporation. OMEGA was designed to bridge the gap between the language option types discussed above. It combines adaptability to any level of computing ability with the flexibility and power required for sophisticated instructional applications.

The paper first discusses a few basics of computers to define terms used in the discussion that follows. That discussion begins with a description of the various aspects of user friendliness. It then describes and evaluates the approaches that have traditionally been taken toward making educationally oriented computer systems user friendly. Finally, the OMEGA approach to lesson authoring will be described and evaluated.

COMPUTER ESSENTIALS

A computer is basically a piece of machinery that uses electronic means to manipulate data. The machinery -- called "hardware" in computer jargon -- consists of a processor and various support items known as "peripherals." These peripherals allow the processor to communicate and interface with the outside world. Some peripherals (e.g., keyboards and card readers) are called "input" devices; they allow the user to *give information to the processor.* Other peripherals (e.g., printers and terminals) are known as "output" devices; they allow the processor to display information for the user. Mass memory is one of the most important peripherals in a computer system. It stores data that is not currently being used by the processor, but which will be required at some future time. This memory is both an input and an output device, since the processor "reads" from it or "writes" to it.

Computer hardware is controlled by programs, or "software." These programs are nothing more than structured sets of instructions that can be understood and executed by the computer's processor. Programs control the sequence and nature of the processor's interactions with its peripherals and the way it manipulates and formats data.

Programs can be written in various computer "languages," which, like human languages, consist of a vocabulary and syntax. The "vocabulary" of a computer language is the set of instructions (also called commands) it contains. Its "syntax" is the set of rules governing the way in which the instructions must be structured. Many different languages have been developed for computers over

the years, and several language options are usually available for any given computer.

The most fundamental language for a computer is its "machine language." Each instruction in machine language is a series of binary states, commonly represented by 1's and 0's. Machine language is considered fundamental because all programs, no matter what language they are written in, eventually become machine language. It is the only language a computer "understands," and is unique to a particular processor.

Programs can be written directly in machine language, but it is very difficult to do so. To make the programming task easier, a second type of language, known as "assembly," was developed. There is a one to one correspondence between assembly language instructions and machine language instructions; but the instructions in assembly language are easily recognizable mnemonics, and are therefore much easier to work with than machine language instructions. Programs written in assembly language ("source code") are mechanically converted into machine language programs ("object code") by a utility program known as an "assembler." Like machine languages, assembly languages are unique to a given processor.

Although programming in assembly language is much easier than programming in machine language, it is still a tedious and time-consuming task. For this reason, various "high order" languages (HOLs) have been developed. Source code written in these languages is translated into object code using utility programs known as compilers or interpreters. HOLs differ from assembly language in that a single command normally corresponds to a structured group of machine language instructions.

High order languages usually have been developed with specific applications in mind, e.g., arithmetic computation, large data base management, string manipulation, or instructional delivery. Their specific design orientations make it relatively easy for programmers to write programs for their intended applications, but programming for applications outside of those for which the language was designed can be very difficult or inefficient.

Source code for a program written in either assembly language or an HOL is usually composed at a computer terminal using some sort of editing program. The code thus produced is put into a logical entity, known as a "file," in computer memory. This file is used as input to the assembler, compiler, or interpreter, as appropriate, to produce object code. This object code is put into another file. When executed by the computer, the object code causes the computer to do the things desired by the programmer.

A note of clarification is perhaps in order here. The user friendly language options provided for educational computing systems sometimes make it appear that lesson authoring is something other than programming. Despite appearances, however, it is important to remember that the author is actually writing a computer program. It doesn't matter what the author's instruction set looks like or how the instructions are specified. When the data that is entered becomes machine language, the computer treats it the same as any other data base. Ultimately, programs must be written in some computer language, and the instructions in that program must be translated into the machine language of the computer being used.

ASPECTS OF USER FRIENDLINESS

A system that is user friendly is one that is easy to learn and use. There are two approaches to making a computer authoring system user friendly. One deals with the language to be used, while the other deals with the means of producing source code in that language, or editing.

Language

It is obvious that a language with complicated syntax is less user friendly than one with a simple and straightforward syntax. In addition, an instruction set consisting of mnemonics that are natural and meaningful to the author will be easier to use than one whose instruction set is unnatural.

These are the aspects of language that are most often thought of relative to user friendliness, but there are others as well. These include the following:

Capability: It doesn't matter how simple a language is if it doesn't provide the author with the tools necessary to do what is required. An instruction set must be complete enough to provide flexibility.

Adaptability: Closely tied to capability, adaptability implies that a language can be used for diverse applications. For example, a language developed only for interactive videodisc applications could not be used by an author in applications requiring interaction with a simulator.

Tolerance for Faults/Diagnostics: Anyone who has done any programming knows that programs rarely run correctly the first time. A useful feature of any language is toleration for such faults as typographical errors. Where errors exceed the limits of reason, diagnostics indicating what sort of error has been made (and where) are a great help in identifying what needs to be changed to make the program run correctly.

Editing

As was the case with language considerations, simplicity is the most obvious characteristic that will make an editor user friendly. The simpler the editor is in function, the easier it will be to learn and use it. However, there are some additional, and perhaps less obvious, features that can be useful:

Internal Helps: Closely associated with simplicity, internal helps provide the user with assistance when he doesn't know what to do next.

Speed: How fast can source code be entered on a sustained basis?

Documentation: To what degree can comments be added to the source code to provide information on program logic? This is a very important feature when the author must go back into the

program to correct errors or change it for some reason.

JUDGMENT CRITERIA

From the foregoing discussion, it is obvious that there are at least seven factors that can be used to judge the user friendliness of a given system:

- o Simplicity
- o Capability
- o Adaptability
- o Tolerance for faults/diagnostics
- o Internal helps when editing
- o Editing speed
- o Documentation.

These criteria provide the basis for the discussion that follows.

HISTORICAL SYSTEM DESIGN

The most common system design for a user friendly authoring language actually obscures the language being used. The author deals exclusively with an editor that provides menus and templates to extract the required program information. As the author responds, the editor composes and formats the requisite commands, and inserts them at the required places in the program. In this way, the authoring language is totally transparent to the author. This approach can be evaluated as follows:

Simplicity

A template system is very simple to learn. Typically, a new author can be on the system and working within a few hours. Authoring vocabulary and syntax are not problems, since the author is isolated from the actual language base.

As simple as this approach seems, however, there are problems with it. While it is true that the new author can begin to work quickly, total proficiency comes much more slowly. A flexible language, with extensive capabilities, requires a great many menus and templates, and the sheer numbers may be overwhelming at first.

As proficiency is gained, a new problem develops. With experience, the author outgrows the need for many of the prompts the system gives. The speed with which the author can create reaches a terminal point, and frustration with the system sets in.

Capability

As discussed above, the capabilities of this type of system may be quite extensive. The nature of the approach, however, limits the ability to extend those capabilities. Each new feature requires not only that new commands and programming routines be prepared, but also that new code be written for the editor that allows authors to use the feature. This is a time consuming and expensive proposition, since the updated editor must be thoroughly tested to ensure that the new features do not interfere with previously existing features.

Adaptability

This approach makes it very difficult to adapt to new applications for the same reasons listed for increasing capability. The problems are of the same nature, but are more extensive.

Tolerance for Faults/Diagnostics

This approach scores very high for this criterion for one potential error situation, since the system does not allow the author to enter an unrecognizable command. Problems can result, however, if an entry is made that is syntactically correct, but logically incorrect. The recognition and solution of this type of error will be extremely difficult and time consuming

Internal Helps

This is the strongest suit of the traditional system. Using an editor of this type provides excellent internal helps for the author. The next required entry is always displayed on the screen.

Speed

As noted earlier, the new author can begin to create very rapidly, and the time it takes will probably be faster than would be possible with other systems at first. However, as proficiency is gained, the prompts become a hindrance. Templates require that each prompt be addressed, whether the feature it represents is required or not. This creates the first bottleneck. A second problem is that it takes a finite amount of time to perceive the prompt and respond to it.

Documentation

The *ability to document the source code* varies from one system to another. Generally speaking, however, this approach does not provide for the extensive internal documentation required for a sophisticated program. This lack of documentation severely curtails the usefulness of the training system. Programs need to be changed from time to time, and changes are not always made by the same person who originally wrote the program. A new author may not be able to perceive the original author's logic if it is not well documented, and may be forced to recreate a lesson from scratch as a result.

THE OMEGA DESIGN

The OMEGA authoring system has been designed to incorporate as many of the strengths of the traditional system as possible, while overcoming its weaknesses. It consists of three elements: the OMEGA language itself, an editor used to create OMEGA source code files, and a set of procedures developed for using the editor effectively.

The OMEGA system is designed to support diverse instructional applications, including operation and maintenance of complex equipment. Unlike other authoring systems, OMEGA can control students as they acquire both cognitive skills using two dimensional (2D) media and psychomotor skills using three dimensional (3D) media. A typical hardware for these training applications could consist of:

- A microprocessor for system control
- Mass memory in the form of either floppy or hard disks
- Mass video memory in the form of laser videodiscs
- A color television monitor to display stored video and/or digital materials
- A touch sensitive bezel, mounted on the front of the monitor, to allow student inputs
- A voice recognition unit to allow for student inputs when both hands are occupied with performing a manual task
- A three dimensional (3D) simulator to allow the student to manipulate equipment as required by the training task.

Lesson design and authoring for 3D applications can be extremely complex. Experience has shown that it is sometimes necessary to anticipate 15 to 20 responses at one time while teaching the troubleshooting of a complex piece of electronic equipment. Sometimes, half of these are anticipated incorrect responses that require remediation. The other half in that situation are logically correct responses. Each correct response requires a separate branch path in the lesson that would accomodate the actions that follow.

The system can also be used for more traditional applications. There are times when the system is used to present standard interactive videodisc instruction, even when dealing with very sophisticated equipment. This instruction can consist of a basic series of instructional frames, or "pages," to be presented to the student. They are "turned" when the student indicates that he is ready for the next page, usually either by touching a designated spot on the screen or by giving a verbal command using the voice recognition unit. A student's understanding is tested from time to time by presenting questions on the screen. Incorrect responses are remediated, while correct responses take the student along the main path of the lesson.

The Language

The OMEGA language consists of approximately 50 commands, but only a dozen or so are required for most lesson authoring. These commands are divided into six categories, according to function:

- Lesson execution
- Instruction
- Student interaction
- Lesson variables
- Simulator communication
- Instructor communication.

Lesson Execution: The lesson execution commands serve two functions: they divide lessons into logical units called EVENTs, and they provide branching within a lesson that is not student-initiated. These commands are totally transparent to the student at lesson run time, but it is primarily these commands that give flexibility and power to the language.

Instruction: The commands that present instruction to the student at run time either display video images and/or audio messages from the videodisc, or they present computer generated graphics that can be overlaid on video graphics. As is the case with all computer based instructional systems, each instructional message is determined during lesson design. The OMEGA instructional delivery commands simply sequence the messages properly.

Student Interaction: Student interaction commands allow the student to communicate with the system. This communication can take the form of "go ahead" indications, responses to direct questions, or 3D manipulations, as discussed above. Student responses are judged as either correct, anticipated incorrect, or unanticipated. The system reacts differently to each category of response. For both correct and anticipated incorrect responses, it proceeds to an place in the program that has been specified in the command. At that place, the system either advances the student on through the lesson or provides remediation specific to the action taken.

Unanticipated actions are incorrect by definition. Since they have not been anticipated, it is not possible to provide specific remediation for them. In such situations, the OMEGA software provides a system generated message that tells the student what control was moved, what position it was moved to, and what position it was moved from. This allows the student to correct the error before attempting to continue through the lesson.

Lesson Variables: The author can define and manipulate variables using OMEGA commands. These variables can be used for a number of purposes. They can be used to keep track of student errors, including number and type. They can count the number of times a student exercises a procedure; this could be used in a situation where there are three different versions of the same help message. By keeping track of how many times the student has asked for help, the lesson can provide a different message each time. The variables can also be used as the basis for branching within a program.

Simulator Communication: When dealing with a 3D simulator, it is frequently necessary for the lesson programs to communicate with the simulator programs. Much of this communication is done automatically using the student interaction commands discussed above. There are times, however, when additional communication is required, either to manipulate variables in the simulator's programs or to determine the value of those variables. The simulator communication commands allow for this additional interaction between the lesson and the simulator.

Instructor Communication: The OMEGA system allows an instructor to monitor what the student is doing on a separate cathode ray tube (CRT). Alerts and/or informative messages can be incorporated into the OMEGA lessons to ensure that the instructor always knows what is happening.

All of the OMEGA commands are straightforward, regardless of the category they are in. Each command consists of a meaningful mnemonic followed by parameters. Most of the commands have three parameters or less, and the most commonly used typically have only one. The most common

values of the parameters are set as defaults to simplify their entry at edit time.

The syntax for organizing a lesson program is also quite simple. A lesson is organized as a series of EVENTs, which are logical groupings of OMEGA commands. The group of commands in an EVENT specify what can happen at one particular time in a lesson. Generally speaking, the elements of an EVENT include what the student will see and/ or hear, as well as acceptable student responses. Other information, such s instructor prompts, historical remarks, or documentation notations, can also be specified as required.

The Editor

The OMEGA system uses a sophisticated off-the-shelf word processing text editor, because of its power and its simplicity. It provides extensive edit time help in the form of menus, and the author can easily set (and reset) the amount of help desired while editing. The following capabilities are provided:

- o Inserting text (characters, complete files)
- o Deleting text (characters, words, lines, groups of lines)
- o Moving text (from single character to large blocks of text)
- o Global corrections
- o Saving text (blocks, complete files).

The Procedures

Each OMEGA lesson is constructed by putting a series of commands into a computer file using the editor. Four different methods can be used to enter this code, depending on the instructional requirements and the programming capabilities of the author:

Command: The author can enter individual commands simply by using the computer keyboard as a typewriter. This option provides maximum flexibility, as the author can group and sequence commands in any way desired.

Template: Here the OMEGA system begins to resemble the more traditional systems, but this resemblance is only superficial. Since certain groupings of commands tend to occur more often than others in any programming language, they can be grouped together to form a template. Specific details may vary from one specific grouping to another of the same type, but the command structure itself changes very little. For convenience in editing, common OMEGA command groupings have been put into individual files. Any of these files can be inserted into any source code file as an intact unit simply by using the "READ" option available in the editor.

The standard groupings can be considered as templates because they provide a predetermined structure to an individual EVENT. However, they are flexible templates since they can be tailored to individual situations by adding or deleting commands as necessary. The commands provided in the templates already have their most frequently used parameters filled in; parameters that change frequently are left blank to be completed at edit time.

Theoretically, any number of templates can be stored in the the system's mass memory, but we usually limit the number stored at any one time for the sake of simplicity. If the author sees that the same command structure will be required frequently, he can create a new template that can be saved, used, modified, and deleted without ever leaving the lesson file being edited.

Extended Template: As the name implies, an extended template is a special case of the template. Extended templates include more than one EVENT, and can include several hundred. They usually involve rather sophisticated logic, and so are normally prepared and saved by authors with extensive programming capabilities. These templates are used for instructional formats whose logic always remains constant, but whose details may change, e.g., interactive games used for drill and practice. Completing an extended template requires only a small amount of editing relative to the amount of lesson code generated.

Module: A module is distinguished from an extended template in that it is a totally intact unit. It requires no modification by the author. Neither the logic nor the details change in a module. It is always used for the same purpose, and it always presents the same instruction. Modules are normally used for hands on, interactive instruction, where the same set of actions is performed a number of times in different lessons. This feature represents a tremendous savings of time and labor, since the code for any module can be included in any lesson once it has been produced.

EVALUATING OMEGA

Simplicity

OMEGA is a rather simple system, since both the language and the editor are easy to learn and use. Though new authors totally unfamiliar with programming cannot begin productive work as fast as they could on a traditional "user friendly" system , they are able to do so within a period of about two weeks. Total proficiency can be gained as quickly as with the traditional approach.

Capability

OMEGA has a broad range of capabilities. Source code can be infinitely tailored to the requirements of any particular instructional circumstance.

Adaptability

The OMEGA system was designed from the beginning for the most sophisticated applications. These include not only traditional computer assisted instruction and interactive videodisc, but also interaction with a three dimensional simulator. For this reason, the system can be used to teach any cognitive or hands on objective.

Tolerance for Faults/Diagnostics

As is the case for any system, errors in command syntax (such as entering an alphanumeric character where a number is required) will cause problems with OMEGA. These errors are trapped when source code is compiled, however, and a

complete set of diagnostics helps the author identify and locate errors quickly. The extensive capabilities for internal lesson documentation also make lesson maintenance a relatively simple matter.

Internal Helps:

The editor provides selectable levels of help to the author at edit time. A new author can have help displayed constantly, and can gradually reduce the amount available as experience is gained.

Editing Speed

This is a major advantage of OMEGA. The use of adaptable templates, extended templates, and modules allows the author to enter large amounts of source code very quickly and efficiently. Experience with the system has shown that editing is considerably faster than with comparable menu-based systems. The most outstanding feature of this approach is that there is no loss of flexibility in using the templates since they are invoked and modified only as required.

Documentation

OMEGA source code can be thoroughly documented by inserting comments at any point deemed necessary by the author. Internal policy toward documentation is that virtually every command is commented, and that additional comments are inserted as required.

SUMMARY

The OMEGA approach to authoring lessons for computer based instructional systems is a significant improvement over traditional approaches. It is a simple system to learn and use, yet it has all the capabilities required for the most sophisticated applications. It can be used for a wide range of applications, including the teaching of both cognitive and hands on tasks. Editing proceeds quickly and efficiently, and various levels of help are available to any author at any time. The system provides for extensive internal lesson documentation, and extensive diagnostics are available to help the author identify, locate, and correct errors in lesson source code.

ABOUT THE AUTHORS

Mr. Charles R. Myers, Jr., is an Instructional Systems Engineer at Grumman Aerospace Corporation. He is currently the Principle Evaluator of the Instructional and User Qualities of Grumman Common Core CAI Software. He holds a B.S. degree from the United States Military Academy, and he has an M.S. degree and is a PhD candidate in Instructional Systems Technology at Indiana University.

Mr. Roger Schaefer is a systems project leader for the Grumman Aerospace Corporation. He is responsible for design and development of microprocessor based training systems. He holds a B.S. degree from Hofstra University and a patent for a microprocessor based interactive training aid.