THREE MAJOR APPROACHES TO DEVELOPING COMPUTER-ASSISTED LANGUAGE LEARNING MATERIALS FOR MICROCOMPUTERS

By: David H. Wyatt

ABSTRACT

Whenever a new project to produce computer-assisted learning materials is begun, one of the most fundamental decisions concerns the selection of an approach to program development. There is a choice between at least three major approaches: using a general purpose programming language, an educational programming language, or an educational authoring system. There are considerable differences in factors such as programming expertise and development time required by the three approaches. Perhaps the most dramatic example of this is the fact that the third approach demands no programming expertise whatever. It is therefore important to be aware of the alternatives in order to make an informed choice of the most suitable approach for each materials development project.

Questions concerning the 'best' approach to developing courseware have often been the subject of intense debate (Holmes 1983; Wyatt 1983a). Some have argued that the current strong interest in producing one's own material is just a short-term phenomenon and that in the future CALL materials will largely be purchased in ready-to-use form. In this view, considerations regarding the optimum method of developing courseware will soon be relevant only to a small number of professionals. Others maintain that the role of language teachers (and other subject specialists) in software development should be limited to specifying the instructional content and its style of presentation on the screen; all questions of how to realize the material in programming terms are to be left to programmers (Bork, 1981).

Options

In spite of these views, the fact remains that an increasing number of language teachers and curriculum developers are becoming interested in producing their own courseware. This article is addressed to them in an attempt to provide a guide to the various options which should be considered. Three major vehicles for creating courseware will be examined and compared in some detail: general-purpose programming languages, educational programming languages, and educational authoring systems. The first two of these require the user to acquire programming skills, while the third type demands only the simplest level of computer literacy. Perhaps the most important point to bear in mind before we begin to focus on the different approaches is that all of them can be successfully used in different situations. They all have weaknesses as well as strengths, and the choice between the approaches depends very much on your individual circumstances and requirements.

To facilitate the comparison of development methods, we will consider CALL software as essentially being composed of three elements: the language content, the driver, and the management system. This is an oversimplification, and is not equally appropriate for all types of educational software, but it will serve to underline some fundamental points. The content may be separate from the program, residing in data files, or may be incorporated directly into the program itself. The driver is the computer program which takes the content and manipulates it so as to present examples and questions, accept answers, and provide appropriate responses. In general, a different type of CALL exercise or activity requires a different (and usually a separate) driver program. The management system is the element which determines how students can enter and use the courseware, what route they follow through the material, and what score and progress information is recorded. Management systems usually involve some separate programs, but management elements are generally also built into each driver program.
General-Purpose Programming Languages

The first main approach to the development of courseware involves the use of general-purpose programming languages such as BASIC and Pascal. In practice, BASIC has been the overwhelming choice of producers of commercial educational software for microcomputers. It has been estimated that 80 per cent of the software to date has been written in BASIC and the remainder in machine language with no significant contribution from any of the other programming languages.

There are excellent reasons for the popularity of BASIC in the creation of software for microcomputers. In theoretical terms, programmers working in general-purpose languages are in much more direct contact with the computer's microprocessor and memory than with the other two approaches. It is possible to exert much more control, and more flexible control, over each step of the operation of the computer. Programmers may design and create every step of the main drivers for their exercises. Similarly, they have complete control over the management system elements in their courseware. For example, the other two approaches frequently offer either rudimentary management systems or demand the presence of a second disk drive if score recording and other management capabilities are to be used. For these reasons, one project directed by the author (Wyatt 1983b) made use of BASIC to create a powerful management component which required only one disk drive for full utilization.

Practicality also dictates strong arguments in favor of general-purpose languages. One important factor is the lack of dependence of the educational programmer on other factors. In one well-known recent case, for instance, a new version of an educational programming language was released with the promise of a management component soon to follow. However, the management component was delayed and became available much later than expected, which must have caused difficulties for those to whom the management capabilities were important. A second practical point concerns the position when new microcomputers are released. In virtually every case, a version of BASIC is immediately available for new microcomputers, whereas a period of months or even years may pass before effective educational programming languages or authoring systems are produced for the computer. However, the reader has probably already perceived the roots of a problem during this discussion of general-purpose languages—control over all the individual steps of an exercise program, desirable as it may be, implies the need for time-consuming specification of all of those steps. In general, the time required to create a given student activity or exercise in a general-purpose language is significantly greater than with the other two approaches. To be more specific, it is the programming of the new driver (and, to a lesser extent, inclusion of the management system elements) which demands the greater investment of time. A second disadvantage is the considerable learning time required for the novice to become sufficiently expert in a general-purpose language in order to produce moderately sophisticated educational programs.

One of the main reasons for the time-consuming nature of producing educational programs in general-purpose languages is their lack of educational "power." Since general-purpose languages were intended for a wide range of different applications, these languages have few if any commands oriented towards the needs of the educator. After only a little experience with general-purpose languages, even novices in educational programming begin to recognize the lack of convenient commands. At a lower level, desirable features of a programming language would include simple commands to cause the program to wait a specified number of seconds and also accept student input in a fully controlled manner (filtering out any unwanted or troublesome key presses). At a higher level, we would look for some ability to perform natural language answer processing—to search students' answers for key morphemes, words, or phrases (Pusack 1983). Ideally, we would prefer intelligent answer processing, in which students' wrong answers are compared with the expected response, 'marked up' to indicate problem letters or words, and then turned back over to students in an 'edit' mode so that only the mistakes need be retyped at the second attempt. Even the lower-level educational commands described above are entirely absent from BASIC, although they can be duplicated by the programmer with varying degrees of difficulty and effort. These added capabilities are generally developed by the program in the form of subroutines which can be reused whenever they are needed in this or future driver programs. General-purpose language programmers tend to build up large libraries of such subroutines to facilitate their developmental work. Producing the higher-level answer processing features is a major undertaking, however, and BASIC is in any case not suitable for the final form of such routines because of technical reasons (for one thing, it runs too slowly).

Serious as the deficiencies of a general-purpose language may appear, there are effective remedies for many of them. In some cases, it is possible to buy a ready-made set of educational subroutines which will provide the low-level capabilities described above. As one example, the Minnesota Educational Computing Corporation has produced a well-documented collection of subroutines in BASIC for the Apple II (MECC 1980). These ready-made routines can immediately be used in creating courseware and provide the beginning programmer with a head start in general-purpose language work. In at least one instance, an enhancement to BASIC is available which will provide an excellent high-level...
answer processing capability (Tenczar et al 1983). When the problems are solved in this way, the advantages of intimate contact with the computer and complete control over program operation become very compelling.

**Educational Programming Languages**

The second major type of approach to developing courseware involves the use of programming languages specifically designed to meet the needs of educators. The best known of these educational programming languages in the context of microcomputers is PILOT (Burke 1983), although a recent development known as EnBASIC is also of considerable interest (Tenczar 1983). Because of their specific orientation, these languages incorporate a range of very convenient commands which provide both trivial and powerful educational facilities for the programmer.

We will illustrate the range of new commands by reference to the low- and higher-level facilities discussed in the previous section, using the SuperPILOT version of PILOT for the Apple II as our example. At the lower level, PILOT provides convenient commands to cause the exercise to wait a specified number of seconds before proceeding and to 'bombproof' the keyboard so that only desirable, meaningful key presses have any effect on the screen. In both these cases, simple one-line PILOT commands can permit the program to ignore unwanted key presses. At a higher level easy-to-use facilities also provide powerful answer processing. One example is the PILOT 'key search' capability. Using this, students may be permitted to enter their answer in a relatively free manner. Once entered, their input can be searched for the significant part of the answer, which might be an affix, single word, phrase, etc. Less important parts of their input can be disregarded, so that spelling mistakes can be tolerated where appropriate instead of causing an otherwise acceptable answer to be judged incorrect. However, PILOT stops short of providing highly intelligent answer processing. EnBASIC is the only current programming language for microcomputers in which this is available. Using EnBASIC, students' answers can be compared against the predicted responses, and then automatically 'marked up' with a simplified set of proofreader's symbols to indicate problem letters and words. Students are then put into a special 'second chance' mode in which they edit their original answer, changing only the incorrect portions. (EnBASIC also provides most of the facilities of PILOT mentioned above.)

SuperPILOT includes a number of other features in a convenient package to take advantage of specific capabilities of the Apple II microcomputer. These include utilities to permit creation of special characters and diacritics (such as those needed in foreign language courseware), simple music and sound effects, color graphics, and control over external videotape and videodisc players. At least one other version of PILOT for the Apple, PILOT Plus, provides very similar capabilities including provision for touch-sensitive screen input, and other versions of PILOT now exist for an increasing number of microcomputers. It should be stressed again that programmers in a general-purpose language can supply themselves with very similar capabilities. Features such as special character fonts, sound and musical effects, color graphics, and control of video devices can either be developed by the programmer as reusable subroutines or purchased commercially as 'utilities' in ready-to-use form. This piecemeal solution lacks the convenience of the package provided by an educational programming language, but this deficiency must be balanced against the greater control and flexibility involved in being able to select particular utilities and subroutines rather than being limited to a single packaged facility.

The powerful, convenient features described above make it potentially quicker and easier to program a new exercise driver in an educational programming language. A further saving of time is possible if a management system is also available as part of the package, as is the case with the PILOT Log component of SuperPILOT. The time-saving aspect of educational programming languages should not be overstated. Programming time will be less than in general-purpose languages, but will nevertheless be of the same order of magnitude. It is generally agreed that the time taken for a beginner to become proficient will also be significantly shorter in an educational programming language, but it is again important not to overestimate the gain. Claims of a few hours' learning time for educational programming languages refer only to use the capabilities of the language at a moderately sophisticated level, far more time is required. In its original form, PILOT was a greatly simplified language, but the extended versions produced for microcomputers have added more complex command and capabilities. Some have argued that these extended PILOT languages actually require the same learning time as general-purpose languages (Hardy and Elfen 1982). In reality, time-saving is probably no longer the most important advantage of educational programming languages. Perhaps their chief benefit lies in high-level capabilities such as natural language answer processing which enable programmers to develop courseware which is significantly more open-ended and sophisticated than would otherwise be possible.

In this generally positive situation, however, there is the potential for some serious problems. Educational programming languages represent an attempt by their designers to predict what types of commands and capabilities educators will need. Since computer memory capacities are limited, programming languages tend to represent
compromises—not all the desirable commands and features can be incorporated, and inclusion of a range of new educational commands may imply the loss of some of the facilities available in general-purpose languages. Indeed, with the original version of PILOT, the range of commands was deliberately kept to a minimum to permit rapid learning of its capabilities (Merrill 1982). In any case, the tendency to reduce the scope of commands usually available in general-purpose languages may lead to problems if the designer of the educational language has not fully anticipated the needs of the courseware developer.

We will illustrate the potential for difficulties with reference again to different versions of PILOT for microcomputers. In one early version, there was relatively little memory space allotted for the textual content of the exercise, so that repeated accessing of the diskette was necessary in order to load and display the necessary screen contents. This caused frequent short delays in the running of many language-oriented exercises. Apparently the designers of this PILOT version had not considered the delays to be significant, and some educators also found them acceptable. A sizable number of teachers, however, felt that the delays were irritating enough to students to seriously jeopardize the effectiveness of their courseware. This problem has apparently been much reduced or entirely eliminated in recent versions of PILOT for microcomputers.

A second specific instance of a problem not anticipated by the educational language designer concerns the character set editor available with PILOT. In general, this is a powerful utility which permits the creation of new symbols and letters needed for foreign language instruction. Unfortunately, however, in PILOT it is necessary to designate a specific key on the keyboard whose symbol will be replaced by the new character. Thus, the additional forms of the vowels in French (acute, grave, etc.) cannot be represented in any very logical fashion; ê, â, and ô must be represented by quite different keys. Using a general-purpose language, however, it can be arranged by the circumflex form to be generated simply by pressing the original vowel followed by a single ‘circumflex key’—as it happens, there is an appropriate symbol on the keyboard. Similarly, all acute vowels could be generated by pressing the base letter followed by the slant on the keyboard, whereas PILOT would require them to be represented by different symbols which could not be related to the base vowels. This is a complex but apt illustration of the difficulty of trying to predict, as the designer of an education programming language, what the requirements of the users are likely to be. Finally, a more general observation which has been made regarding educational programming languages such as PILOT is that they are structured so as to be well suited only to tutorial and drill-and-practice type of CALL, lacking the flexibility for use in developing more innovative and communicative activities.

Most of these drawbacks are theoretically avoidable, however, and as the relatively young field of microcomputer-assisted learning matures it can be seen that educational programming language designers are improving on their early efforts. Certainly one of the newer languages, EnBASIC, does not seem to suffer from any of these problems. It functions as an enhancement to Applesoft BASIC, so that virtually none of the capabilities of that general-purpose language are lost. At the same time, it supplies some very powerful commands and features for educational applications. The only major sacrifice is a sizable part of random access memory—approximately 16,000 bytes are occupied by the language itself.

Educational Authoring Systems

The third general method of developing courseware is through the use of an educational authoring system. Unfortunately, there is often considerable confusion in terminology between educational programming languages and educational authoring systems. The criterion used for the classification in this paper is whether the courseware developer is required to perform any type of programming while using the method; if so, it is at least partly composed of an educational programming language. If no actual programming is required, then the method is a pure authoring system.

This last point embodies the outstanding advantage of authoring systems: teachers do not need to learn any programming language in order to use them successfully. In fact, all that is absolutely required in terms of computer knowledge is familiarity with the keyboard, disk drives, and diskettes—in other words, an elementary level of computer literacy. This means that the time involved in learning to use an authoring system is essentially required for study of the features of the system itself rather than those of a computer or programming language. A dramatic reduction in learning time is thus possible. The simpler authoring systems can actually be mastered in a period of a few hours or less. More powerful authoring systems have features and techniques which may take considerably longer to master, but they still offer a tremendously shorter learning period than the approaches which employ programming languages.

Authoring systems can also greatly accelerate the speed of development of courseware. Once the language content of the lesson has been specified in the instructional design phase—required in any approach to CALL—the authoring system will dramatically reduce the time taken to translate the plans into the form of software. As an example, let us briefly examine the operation of a moderately sophisticated authoring system, the Assisted Instructional Development System, or AIDS (Wolfe 1982). AIDS is largely a menu-driven authoring system. When first activated by inserting the diskette and turning the computer on, the system offers a menu of actions from which to choose: creating a new exercise, editing existing exercises, transferring files from one student diskette to another, deleting unwanted
exercises, etc. If 'create a new exercise' is selected, the user is then prompted for the information required at virtually every step along the way. First, a student diskette must be inserted on which the new exercise is to be recorded. For each question of the exercise, the developer is asked to type in information such as the text to be read (if any), the question to be asked (if separate from the text), the correct answer or alternatives, comments to the student if correct, and to predicted wrong answers, with specific feedback messages on each of the individual problems involved. As each question is completed, the system automatically stores it on the student diskette previously inserted.

When all the questions for the exercise have been typed in, the system then asks if the developer wishes to review an edit the new exercise immediately. Because the process is guided and structured at every point by the authoring system, the creation of new exercises proceeds rapidly and smoothly. When the resulting material is used by students, the authoring system ensures that the screen presentation and question flow are handled very professionally. This is rather gratifying to the novice user, as even the first attempts appear quite polished when used by students. The system also makes it very difficult to create material which will 'crash' because of programming defects, so that the debugging process is greatly shortened and is essentially limited to correcting errors in the language content.

It can be seen that authoring systems supply all the necessary program drivers in ready-made form. Some systems also provide built-in management capabilities. With the AIDS system, for example, there is an extensive score recording and exercise routing capacity. The system also provides a simple keyboard answer processing feature.

Although AIDS is an excellent example of a medium-power authoring system, it is not specifically designed for foreign language applications, as is shown by the absence of ready-made foreign alphabet character sets. An example of an authoring system specifically for foreign language application is the DASHER (Pusack 1982) authoring system, which offers foreign characters and diacritics as well as a number of other interesting features. It incorporates an intelligent answer processing capability which is similar in power to that of EnBASIC. Mistakes in answers are indicated by simple 'mark-up' symbols, and students are then put into an editing mode which enables them to attempt to correct their answers by changing only the incorrect portions. The power and convenience of this edit mode makes this feature one of the best of its kind among current microcomputer-based systems and languages.

DASHER and AIDS are examples of medium-power authoring systems. At a higher level, a number of systems exist which will provide even more sophisticated capabilities. The PASS system from Bell and Howell, for example, will permit the control of videotape and videodisc players from within the courseware. At a lower level, there are a host of inexpensive authoring systems available. Many of these offer a range of relatively rigid exercise formats—typically including true/false, multiple choice, and matching—designed for general educational purposes. Recently, however, there has been a tendency to offer some language-oriented authoring systems which will generate a very limited but indefinitely reusable type of activity. For example, Clozemaster (Jones 1982) will generate a wide variety of different cloze exercises from any paragraphs which are typed into it.

The disadvantages of the authoring system approach are as obvious as its advantages: such systems are relatively quick and easy to use because they embody a built-in educational methodology and program logic. Almost invariably they are strongly instructional in nature, and it would be difficult or impossible to use them to create more open-ended or communicative activities of a collaborative type (Wyatt 1983c). In general they are not even suitable for instructional programs of the tutorial type, since their branching capabilities are very limited or non-existent. In the great majority of cases, they are suitable only for the creation of drill-and-practice exercises and quizzes. However, if it is precisely this type of courseware that is desired they offer a highly cost-effective option that deserves very serious consideration.

Summary

In summary, there is no single 'best' method of developing courseware. This paper has presented a wide spectrum of possible approaches to the creation of materials for computer-assisted language learning, all of which can be appropriate in different circumstances and for different purposes. It is important that prospective developers of computer-based curricula be aware of the alternatives open to them since different approaches may offer very significant variations in time and energy required for completion of a given project. It is to be expected that the situation will continue to change as new or improved versions of these approaches become available.

REFERENCES

Merrill, Paul F. 1982. The case against PILOT. Creative Computing July 82, 70-77.
1982 DASHER: A Natural Language Answer Processor. Iowa City, IA: Conduit.