

**cosc460 Honours Project 1986.**  
**Computer Science Department,**  
**University of Canterbury.**

# Learning To Read With Computers

**Alastair Kenworthy**

## Contents

	Introduction	1
Chapter	1. Learning to Read Without Computers	2
	2. Potential and Limitations of CAL	5
	3. Present Forms of CAL	8
	4. Courseware Authoring	12
	5. An Experimental Authoring System	14
	6. Future Work	17
	Conclusions	18
	References	19
Appendix	A. Experiments	
	B. Teachers' Manual	
	C. Program Listing	

## **Introduction**

Microcomputers are widespread in schools and are becoming so in centres for special education - the education of people with learning disabilities. Across the curriculum, Computer-Assisted Learning (CAL) has augmented conventional learning in some of these places.

The object of this project has been to find the extent to which CAL can help intellectually handicapped (IH) adults learn to read.

The work has involved researching educational techniques, observing existing special education CAL programmes, experimenting with CAL techniques, and developing advanced educational software. Most of the practical work has been carried out in conjunction with the Canterbury Sheltered Workshop, where IH adults are given conventional reading instruction.

Chapter 1 is an exposition of the teaching techniques used for reading, irrespective of the media used to deliver them. Special requirements for teaching IH adults are examined.

Chapter 2 identifies CAL's niche in a reading programme for adults, and looks at computer techniques that could be used in this niche.

Chapter 3 begins with a summary of CAL reading software in operation and its deficiencies. The chapter closes with a characterisation of the problems involved in producing the necessary software.

Chapter 4 continues with the software production problem - describing as yet underdeveloped productivity tools for its resolution.

Chapter 5 documents the author's own attempt at such a system from design through to field trials.

Chapter 6 is an outline of further work that could be done to follow up work in this project and to tackle work outside this project's scope.

## **1. Learning to Read Without Computers**

Unlike children entering school, adults enrolled in reading programmes have almost always chosen to learn to read. They are aware of the benefits that reading can give them; such as the enjoyment of literature, or the ability to understand instructive pieces of writing such as bus timetables or advertisements. The most important aspect of helping IH adults to read - above all teaching techniques - is maintaining their initial interest. Without motivation, nothing will be learnt [1].

### **Components of a Reading Programme**

Reading programmes for IH adults consist mainly of: independent reading (reading by themselves), and supported reading, which may be anything from reading in a pair with a teacher, to drama. All learners participate in these activities.

Minor activities in reading programmes, that would supplement the above activities for some learners, are: skills activities (eg letter study, basic words, clozing - supplying missing letters from a word) and games, not necessarily in the reading domain. Skills activities appear in textbooks or on sets of cards prepared commercially or by the teacher. Application of supplementary activities is based on individuals' weaknesses and not on general principles; making diagnosis of reading failure an important function of the teacher [2].

Each skills activity practises a particular reading skill. For example, all readers have a group of words they can recognise by sight. Words outside this vocabulary, when encountered, can be deciphered via the *Central Method of Word Attack*:: recognition of the first part of a word, plus the word's context enable the reader to predict the remainder of the word's sound and confirm it in their spoken vocabulary. (It is not usual to teach reading words outside of a learner's spoken vocabulary.) Cloze exercises help this prediction skill.

### **Opposing Teaching Philosophies**

The traditional view is that learning to read is the acquisition of a set of such skills, which are assumed to be identifiable and teachable [3]. Reading programmes adhering to this philosophy incrementally introduce skills to a learner through exercises, while showing the learner how to use these skills in functional reading.

A more modern view is that reading is a problem solving process which learners discover for themselves. New passages of writing are approached with expectations about print (ordering of letters, words) and meaning (according to background knowledge). On the basis of various cues found in the passage (language, visual arrangement and meaning), the learner adjusts his or her

mental schema of reading knowledge [4]. This view does not suggest that reading programmes should consist entirely of independent and supported reading, but that supplementary activities should be looser; directed more toward acquiring a general critical ability than precise skills.

An example would be giving a student a newspaper and asking specific questions about it. The student would hunt columns, look at page numbers and photographs, take most notice of large print, and so on, in searching for answers. As well as being a problem solving exercise, this example is simulation. In learning to read, the student is shown an everyday application of his or her reading [5].

Corresponding to these opposing teaching philosophies are two methods for introducing new concepts:

(a) Socratic (traditional) - The student is guided through the logic of new work with a series of questions aimed at discovering, and then altering, misconceptions.

(b) Coaching - Learning is encouraged indirectly by presenting problems to the student. Students explore a variety of problem solving methods to find the most applicable [6].

Teachers apply these methods to suit individuals.

### **Teaching Machines**

The most rigid approach to the straight teaching of reading skills is the so called *teaching machine*. Teaching machines are typically large, graded collections of cards containing comprehension and word skills exercises. A common one is the SRA (Scientific Reading Association of America) series.

While most teachers these days regard frequent testing of students as bad, SRA relies on it. After completing exercise cards, learners mark their own work against answer cards and progress to the next hardest level on a good score. Teachers assess performance much less directly, but it is claimed SRA ensures instruction is presented at a challenging level and sequential instruction is provided - both desirable features [6].

The pace of courses should be suited to individuals. IH people often need to be paced very slowly. SRA's internal pacing works fairly well except when learners compete, but studies have found moderate amounts of external pacing (eg promotion up a few levels by the teacher) to be superior [7].

SRA is used heavily by the Sheltered Workshop and other similar places. In general, New Zealand educators of the intellectually handicapped favour the traditional, strictly skills method. Overseas research doesn't support this. Possibly, weak funding forces centres to persist with old teaching aids. Short staffing may make teaching machines attractive because teachers don't have the time to chart individual courses for all their students.

### **Providing Suitable Material for Adults**

When confronted with a piece of writing, someone learning to read may be repelled for two reasons: it may be too difficult or easy, or it may be unsuited to the learner's interests or culture. These problems are particularly manifest in teaching adults to read because their reading ability will be that of children but they will have adult interests and needs. An example of the former problem would be giving an adult learner a normal adult text - if read aloud to the learner, it may be found interesting but its vocabulary would be too large and its sentences too complex for the learner to read unsupported. Alternatively, a new entrant's book may be the required difficulty but not at all interesting.

There is a huge volume of reading material for children with children's interests and reading ability, and similarly for adults. Very little material is available for adults with the reading ability of children.

Adult Literacy sections in public libraries provide simplified versions of adult texts which suffice for the independent and supported reading components of a full course. Unfortunately, there are hardly any locally made, adult-oriented, minor exercises.

Overseas material is culturally unsuitable to various degrees. Most obviously, Americans spell some words differently; all countries' languages are idiomatically different. Teachers often employ imported material (eg SRA) in their programmes because the alternative, producing their own, is too time consuming.

Ideally, all exercises presented to a learner should have immediate relevance (eg if, for example, street names are used, they should be familiar street names) and novel situations should always be available for practice [8].

### **Reinforcement and Feedback**

Continually new material is required for learners so that recently learnt words and concepts are reinforced through repetition. Reinforcement is the process which holds new ideas in a learner's mind. Some other types of reinforcement are: aural - associating sounds with their written representations, and manual - learning the shapes of letters through drawing or tracing them.

Sometimes a reward other than that of understanding a piece of writing is needed to motivate a learner. Feedback such as giving rewards for good work and discouraging bad work is also reinforcement.

The intellectually handicapped require a lot of practice and systematic reinforcement.

## **2. Potential and Limitations of CAL**

What roles is a computer capable of in a full reading programme?

### **The Computer's Niche**

At first appearance, CAL seems as though it could replace conventional minor activities in a reading programme. It is easy to imagine computer-mediated skills exercises or games. Equally apparent is that the computer has no place in independent reading. Independent reading gives the reader real-life practice, which will almost always involve reading printed pieces of paper.

Supported reading stresses human communication: spoken language, bodily gestures and facial expressions, in relation to pieces of writing. Here, computers could possibly be used because of their provocativeness. Some educators have seen them as tools for focussing group attention and discussion rather than as isolated helpers of individuals [9].

Individual work with the computer would be inappropriate in some cases. For example, reading failure may be due to lack of spoken ability [1]. However a computer is used, no physically able student should sit in front of a screen for more than a couple of hours per day [10].

### **Learner-Computer Interaction**

The computer can communicate with learners through other than the usual pairing of VDU and keyboard. Audio-visual and physical aids such as sound and speech synthesis, video cameras and displays, pointing devices and voice recognition are all adaptable to educational use.

A main function of the computer in special education may be to substitute for individual help by the teacher. A session with a learner would then have to be fairly self-sufficient - requiring minimal teacher supervision. Such sessions would have to involve a wide range of communication aids to cater for different reading abilities and associated disabilities.

An experiment was conducted at the Sheltered Workshop to discover trainees' reactions to a computer and the level of interaction they could maintain with it. The experiment (Experiment 1, described in detail in Appendix A) involved informal testing of trainees' manual skill, conceptual ability, concentration, and ability to understand synthesised speech. A Macintosh was used for the experiment because of its ease of use with the mouse, and the availability of appealing introductory software (MacPaint, action games). The computer was very well received by both trainees and staff.

One finding of Experiment 1 was that speech synthesis is not yet advanced enough for educational use. Very poor readers will be unable to follow even simple written directions on the screen. Fortunately, the trainees were found capable of taking cues from other sources, such as

icons.

Some adults are embarrassed about their deficient reading and do not respond well in class or alone with the teacher. A computer would allow people like this to experiment with words and sentences unselfconsciously - a step into the unknown is necessary for all learning.

### **Skills Based Programmes**

The teaching machine approach could easily be translated to the computer. The computer is a good medium for the techniques of repetition and incremental learning of skills. Advantages in feedback over manual teaching machines would be: learners could be notified of their mistakes immediately and given several chances to rectify them; exciting rewards could be given; and a more complex promotion or demotion path could be decided upon diagnosis of mistakes [11].

Participants in a skills programme keep notebooks of the letters, letter pairs and words they can recognise by sight. A computer could manage electronic versions of these notebooks and ensure that all entries were regularly reinforced for a particular participant.

The use of the computer as a smart workbook might suffer, as do teaching machines, from being tailored to monitoring ahead of value to the student. Management and diagnosis form a very large, attractive area in CAL. However, they should be subservient to the reading exercises themselves.

### **Local Exercise Production**

Locally written exercises could possibly be developed faster for a computerised system than for a conventional one. Photocopied pieces of paper quickly get ragged and this might discourage teachers from making their own exercises. Once written, it is easier to revitalise with new material CAL exercises than paper based ones. Computer-stored dictionaries of graded words might help teachers in filling out exercise content. These advantages are basically concerned with wordprocessing, but that is irrelevant if it contributes to more immediate and interesting material for the learners.

The ideal situation would be the one in which a teacher could afford to design exercises with particular individuals in mind.

### **Advanced Programmes**

The computer's greatest attribute in relation to reading is responsiveness. Conventional pages are limited; the computer might manipulate text in ways to improve comprehension. Text mediation could be controlled by the computer upon sensing a learner's difficulty, or by the reader. The



## Potential and Limitations of CAL

reader may request illustrations at any point in a passage, paraphrasing, or definitions of unfamiliar words. By examining the aids a reader seeks, the teacher could diagnose the cause of a reader's trouble. The claim has been made that a low ability reader using computer mediated text achieves the comprehension of a high ability reader [12].

Problem solving exercises require responsive environments. Video games with complex, intertwined rules have shown that the computer is an excellent medium for problem solving exercises. Players learn the rules of video game universes through experimentation. Successful strategies can be rewarding colour, sound and harder situations - driven by additional rules. Readers could benefit similarly if the embodied rules were of a reading nature [13].

Experiment 1 showed that IH trainees are capable of the high level, sustained interaction necessary with problem solving exercises.

Simulation is problem solving in an imitation of a real-world situation. Trainees at the Sheltered Workshop might benefit from CAL simulation especially, because of their need to be self-sufficient in the city. For example, a simulation of a job interview would acquaint them with the language of job applications and awards.

Unlike skills exercises, problem solving exercises have no "pen and paper" equivalent. Their use in CAL is potentially very great.

### **3. Present Forms of CAL**

CAL is a sure bet in regard to the simple skills teaching strategy. It promises to be even more effective for more modern strategies. What follows is a review of current educational reading software - courseware, and how well it meets its potential.

#### **Local Centres for Special Education**

Around Christchurch there are several centres using computers to teach reading. All these centres cater for slow learning or intellectually handicapped, primary or secondary school students. A variety of microcomputers (Apple ][, BBC micro, Commodore 64, Poly) and courseware is used, with a large proportion of the courseware being locally written. The author visited these places and assessed the methods used in their work.

It soon became apparent that no computer had a clear edge regarding courseware. For this reason, it was decided to persist with the Macintosh for studies at the Sheltered Workshop. Another experiment was held there to determine the usefulness of Macintosh courseware (Experiment 2, Appendix A).

Local observations and the results of Experiment 2 appear in the following discussion.

#### **Teaching Methods Used**

Whereas both Socratic and Coaching methods are used in normal classroom teaching to introduce new work, courseware was found to be almost entirely Socratic. Considerable repetitive reinforcement, called *drill and practice*, is used in support of this method.

Socratic courseware is most often tutorial style exercises. A sequence of pages of information is presented; usually with prerequisites for mastery having to be satisfied before progression to the next page. The sequence can be linear or, if divergent routes are taken on right and wrong answers, branched. Reading exercises of this nature often consist of several passages of writing with comprehension or language questions asked after each. The passages would be structured to highlight particular skills. Pictures and opportunities to skip back a page are simple forms of text mediation in tutorial exercises.

To be useful, tutorial exercises must allow the teacher to replace the material in them.

A variation on the tutorial theme was the Christchurch Teachers College (CTC) Twist-A-Plot, a modifiable adventure game. Students enjoyed making decisions on which paths to take and eagerly read the teacher-inserted text accompanying its pictures. It was interesting to see that the usually well-defined role of the teacher as supplier of material for exercises was broken down. Students could devise their own branching stories, choose illustrations from a picture library, and dictate the

text to the teacher - a very good way of obtaining appropriate material. The teacher would then do the tedious work of entering it all. Twist-A-Plot is well intentioned but entering stories takes a prohibitively long time for anything but a small class. Also, the picture library was quite limited.

Drill and practice courseware was distinguished by attractive feedback covering basically dull exercises. For example, Plato (CTC) uses animated, colour pictures to reward correct answers to cloze exercises. Less soundly, it greeted incorrect answers with interesting noises; unintentionally encouraging learners to make deliberate mistakes. In contrast, and much worse, the Macintosh's drill and practice program, Flashcard, barely differed in its responses to right and wrong answers.

Flashcard is general-purpose, public domain software while Plato is expensive and tailored for special education. General educational software does not emphasise reinforcement enough, and is too intolerantly paced, for use in special education.

The Milliken (CTC) courseware package for teaching maths exhibited computer managed instruction (CMI). Students' progress was reported to the teacher quite elaborately through graphs and ratios of right to wrong answers. The teacher could set courses for individuals using a student database manager. The exercises supported were only simple drill and practice.

Presenting a learner with an exercise should be the primary role of any CAL system. Managing which exercises to present, analysing answers and diagnosing faults in them is not as immediately important [14].

No simulation courseware is used anywhere in Christchurch despite its potential usefulness. The closest approach to any problem solving courseware was some skills exercises arranged as a game (Commodore 64 with a light pen). The reward came from playing the game itself, which is preferable to the explicit bribes given in straight skills exercises.

### **Unfulfilled Potential**

The best of the commercial exercises available do satisfy a "concern for individualised learning, self pacing, immediate knowledge of the correctness of response, and reinforcement" [15] but are guilty of "the tendency to focus computer teaching on limited, narrowly conceived functional skills" [16].

Clearly, today's courseware is nowhere near to meeting its potential. The editor of Educational Technology has asked, "Where is the imaginative, unstructured educational software?" and then gone on to state that software publishers produce solely drill and practice courseware because it is guaranteed to sell. He observes that most classrooms were entrenched in old fashioned teaching methods before they took on computers, and now resist courseware they don't understand [17].

### **Problems of Producing Courseware**

## Present Forms of CAL

Several local teachers expressed their frustration at the scarcity of good courseware to present to their students. Most commercial courseware is educationally useless and, of the high quality material, it is very costly assembling comprehensive programmes.

One teacher gives a third form slow learners class at Hornby Highschool one hour's reading work per week on BBC micros. She would like to extend this time but the courseware available barely justifies even this short exposure.

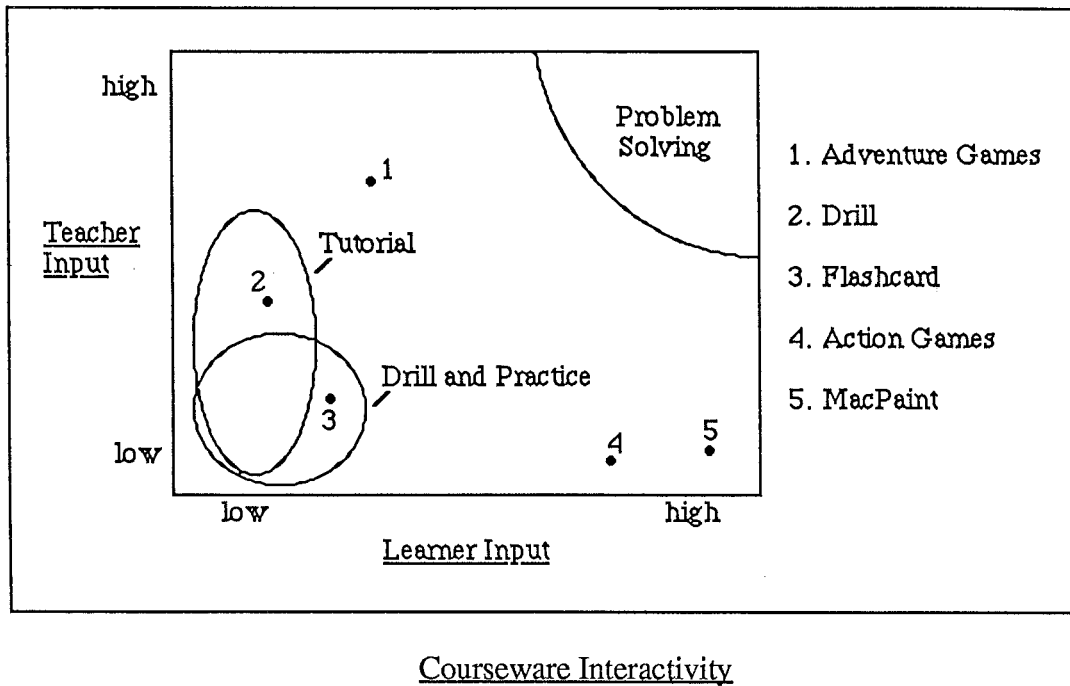
The courseware was written at the school to provide drill and practice in such areas as comprehension and proofreading. Students select exercises from a menu. The package follows established rules: give learners several chances to fix mistakes, use sound and colour, provide for frequent replenishment of written material. However, this package is unsatisfactory for three reasons that typify the genre:-

### (1) Dull Interaction

Experiment 1 at the Sheltered Workshop found that typing isolated the learner from the action on the screen. Other educators regard keyboards similarly and have de-emphasised their use with courseware. The Hornby Highschool package, to reduce the keyboard's role, resorts to multiple choice. The students under observation quickly tired of these. IH students working with multiple choice during Experiment 2 at the Sheltered Workshop became bored also. The most favourably accepted exercise at the highschool was a proofreading one in which navigational cursor key movements were needed. From these observations and the success of MacPaint in Experiment 1, learners seem to respond best when they can engage in complex interaction with the computer. A fact most courseware doesn't recognise.

Courseware is needed that allows a high level of both teacher and learner input so that learners can be involved and creative while the teacher's input, entered previously, flexibly directs the course of exercises. The figure below plots the amount of teacher and learner input supported by present courseware. It is obvious that much more interactive courseware should be attempted.

## Present Forms of CAL



### (2) Static Exercises

Although it was quite easy for the teacher to type new passages into, for example, the comprehension section of the Hornby package, exercise structure could not be altered. There were none of the surprises necessary to maintain interest in skills exercises; rewards were always the same, exercises were entirely predictable after a while. Several students complained about having to wade through all the simple exercises to get to the challenging ones. For months of development work, a handful of static exercise formats that are soon boring is a very low return.

In general, courseware must be extensive so that each student in a programme is continually provoked by new situations, and the entire range of interactive and reading abilities is covered.

### (3) Poor Co-ordination of Teaching and Programming

The Hornby package was written in Basic by a teacher with no formal programming training. Low programming expertise probably accounts for the lack of graphics. However, the teacher was not an English teacher, which resulted in some dubious presentation. For example, a speedreading exercise advanced the leading edge of a passage ahead one letter at a time - speedreading is meant to encourage reading a whole phrase at a time.

In the past, the resolution of these related problems has been to form closely bound development teams of specialist teachers and programmers - teachers specifying and evaluating exercises, and programmers coding them [18].

Overseas, the expectation that courseware should be developed by teachers for their own classes

## Present Forms of CAL

has proven to be unfulfilled. There have been exceptions, but in the main it has been found that instructors don't have the time, skills and interest required to design, write, evaluate and integrate courseware [19].

#### **4. Courseware Authoring**

The infusion of CAL into all areas of special education has been hindered most by the insufficient supply of high quality software, and the adoption of only the most primitive teaching techniques for use in courseware [20].

Ambitious teachers have confronted these problems by attempting to write their own courses. Flawed courseware has often resulted, which teachers have not had the luxury of discarding because of the time spent on it. Production has been slow even using qualified programmers. Normal production techniques yield typically one hour of exercise time from 50 to 500 hours of programming time [18]. The ideal solution in which the teacher can tailor exercises to individuals has certainly not been approached.

The solution to the production problem lies in courseware authoring: the use by teachers of special purpose production systems designed to make course writing easier. Difficult programming details like screen display and accepting input from a mouse are hidden. Courseware authoring's primary aim is to speed production. A consequence of faster production may be that teachers will be able to afford to be more experimental in designing exercises; simulation and problem solving courses might emerge.

##### **Current Authoring Systems**

There are several authoring systems on the market. Disappointingly, as drill and practice, and tutorial styles dominate courseware, authoring systems limited to these styles dominate authoring. Exercises under these systems are based on text scripts containing passages of writing interspersed with formatting, control and learner-questioning commands.

Most effort has been spent on matching learners' responses with acceptable responses in the mistaken belief that "the heart of any authoring system is its answer judging capabilities" [21]. Monitoring learners' understanding is important but it doesn't help them learn anything in the first place.

Discussions of system's capabilities centre around whether exercise control is *linear* or *branched* - oldfashioned ideas. The Macintosh program, Drill, is a linear authoring system in this sense. Experiment 2 with Drill highlighted some common faults of authoring systems: exercises were interpreted making them too slow - feedback to learners is meant to be instantaneous, the graphics were non-interactive (Another system, Coursewriter, has no graphics at all [22].); no sound.

Pilot, for the Apple II, does have sound but it is not possible for the teacher to precisely format the screen. Both Flashcard and Drill showed that very poor readers are distracted by screen clutter; the teacher must be able to decide exactly what is visually presented to these readers.

An exception to script-based systems is RuleTutor [23]. Its users can modify flow diagrams on the screen to build exercises. Although easy to use, this is the same, simple control strategy as other systems and allows little variation in exercises. RuleTutor is a generative system; it was designed to overcome the requirement that the author specify in advance all instructional material to be presented to the learner, all likely answers to all questions, and all feedback that is to occur as a result of a learner's response. A subject matter specialist enters a database of course material which the system substitutes in appropriate exercises. A Christchurch teacher uses the same idea very successfully in conjunction with exercises written in Basic.

None of the authoring systems surveyed provides the facilities for a usefully substantial body of courseware to be developed for a reading programme for the intellectually handicapped. Although teachers would be able to write exercise scripts, the resulting exercises would be deficient interactively, graphically, in speed and in teaching methodology.

### A Simulation Authoring Model

*Educational Technology* acknowledges that "even the present, advanced authoring systems are built on a tutorial model. The future systems will be simulation authoring environments" [21].

Current fourth generation language research suggests a declarative programming style is easier for non-programmers to handle than a procedural style. The implicit *branching* that accompanies a declarative style is suited to simulation and problem solving exercises. Statements are executed on given conditions instead of in a programmer-defined sequence. If such an approach were adopted for an authoring system, teachers should be able to write advanced courses easily; and learners, when presented with an exercise, would then not be tied to any teacher-anticipated order of doing things.

The developers of Coursewriter claim that simulation requires so much interactivity that nothing less than a high level language will suffice for programming it. However, the Macintosh environment functions very responsively with a small set of mouse operations (clicking, moving, dragging) - there is no reason why an authoring system could not incorporate these.



## **5. An Experimental Authoring System**

The development of an authoring system for the previously ignored area of problem solving and simulation seemed possible. Because of this area's importance to the future of CAL in reading, an experimental version was developed.

### **Design Criteria**

The shortcomings of conventional CAL, observations of trainees' reactions to the Macintosh, and CAL studies in the literature suggested the following criteria for a courseware authoring system:-

#### **(1) Support Modern Teaching Philosophy**

- Allow the building of problem solving and simulation exercises as well as drill and practice exercises.

#### **(2) Make Learner Interaction Easy and Expressive**

- Incorporate the mouse. Keyboards distract attention from the screen and are difficult for some trainees to use.
- Let exercises be very responsive to the learner. Immediate feedback is necessary for reinforcement and to hold interest. Encourage experimentation.
- Allow exercises to move at the learner's pace.
- Cater for the whole spectrum of conceptual, manual and reading abilities. Early or poor readers will need to communicate through symbols and pictures rather than words.

#### **(3) Promote Local Exercise Production**

- Help teachers build their own exercises easily and quickly.
- Let the teacher's specific intention drive an exercise, including the precise presentation of stimuli to learners.

### **Development**

The system described above was developed as a Macintosh application using Pascal. (See the users' manual in Appendix B.) The program consists of three modules:-

- (a) A recursive descent compiler. The teacher's exercise script is parsed and a table of actions to take on various events is built up.
- (b) An interface to MacPaint so that MacPaint pictures can be used in exercises.
- (c) An interactive run-time module to present the exercise, accept learners' actions and, with reference to the table built by the first module, perform the appropriate teacher-defined actions.

Further technical documentation appears with the program code in Appendix C.

### **Field Trials**

Upon completion of the system, a selection of widely different reading exercises were prepared by the author. The selection, part of which appears with the user manual, shows the system's versatility in producing problem solving and simulation exercises, its easy-to-write scripts and the expressiveness of its interaction with the learner.

When shown to a non-intellectually handicapped audience, some exercises were instantly very popular while others were regarded with puzzlement. All the exercises accurately embodied ideas of the author. That some weren't very good wasn't the fault of the authoring system - the unpopular exercises were based on unpopular ideas. No other authoring systems are as transparent to the teacher's intention.

Unpopular exercises can be discarded in favour of quickly written replacements. Evaluation is often ignored after development of courseware with other authoring systems because the faults in the courseware are most likely to be inherent to the system (eg slow feedback in Drill). Evaluation of purchased courseware or courseware programmed in a high level language is usually pointless because of the prohibitive cost of replacement in time or money.

A local reading teacher familiar with CAL was given a demonstration of the system. She uses a mixture of public domain and homemade CAL exercises to teach reading to dyslexic children. The homemade exercises are invented by her and programmed in Basic by a friend. Translating her ideas into programs in this way is very slow and most of her ideas are never implemented. The authoring system should suit her ideally.

Her reaction to the samples was very positive - she praised the graphics and feedback, and suggested some changes that could be made to one of the exercises to make it easier for her children. After being shown how to use MacPaint and the text editor, she successfully made these changes, ran the exercise again and was satisfied.

Over a period of about one and a half hours she made three completely new exercises (eg

Sample D in Appendix B). She remarked that her ideas translated naturally into the system's condition-oriented notation, and was in no doubt that she had created educationally useful exercises. Each exercise, she estimated, would have required at least twenty hours programming in Basic - for an inferior result in presentation, reliability and modifiability. The exercises would probably occupy a learner for about ten minutes each; giving a preparation time to learner time ratio of 3:1. This ratio is many times better than for conventional programming techniques.

Despite this improvement, she felt that the MacPaint picture preparation process took a disproportionate amount of time compared with the supposedly more complicated task of writing the exercise script. She also criticised the lack of a special feature for refreshing the material within an exercise. Both these problems could be solved if a more efficient interface to MacPaint were to be developed.

### Appraisal and Improvements

The developed system is consistent with its design criteria. The learner's environment lacks two important stimuli: sound and colour. An interface to synthesiser software would provide sound for the system. The Macintosh doesn't have colour. A more suitable host for the system would be the new Apple ][ GS, which has the Macintosh advantages of a mouse and bitmapped screen, plus colour.

Judging by the initial interest shown, teachers will be motivated to write their own courseware using the system. With an improved interface to MacPaint and possibly interactive exercise definition, teachers should be very productive. Of course, it will never be a trivial task thinking up good exercises.

## **6. Future Work**

### **The Effect of Advanced Courseware**

Studies have shown that reading students respond better to a mix of problem solving and conventional skills exercises than to skills exercises alone. In special education, teaching has been heavily biased towards the skills approach. The authoring system developed during this project makes it possible for a large body of problem solving courseware to be written. A further study could use this courseware, in conjunction with drill and practice courseware, to gauge the effect of a combined strategy in teaching the intellectually handicapped.

### **Computer Managed Instruction**

The practical work of this project has been concerned exclusively with the computer as a reading exercise mediator. For a usable system, this function has to be integrated with management functions (CMI) such as presenting exercises in suitable order and reporting student progress to the teacher. Management is particularly important in situations where students work alone with the computer, which would probably be the case at the Sheltered Workshop.

It is not clear to what degree CMI should be used in teaching the intellectually handicapped. Learner controlled menu selection among exercises is probably the simplest possible CMI, while AI systems could take over all identifiable diagnosing and course-plotting roles of the teacher. Whatever way CMI is used, it should be unobtrusive.

## **Conclusions**

The techniques used to teach IH adults to read were found to be basically the same as for teaching children or normal adults, but with additional emphasis in some areas. Some elements of teaching reading seemed adaptable to CAL. As well as enhancing traditional methods for helping the intellectually handicapped, CAL promised to support a more imaginative, effective teaching style.

Some initial experiments using a Macintosh showed Sheltered Workshop trainees to be enthusiastic and their teachers interested.

Observations of local attempts at infusing CAL into reading programmes identified the difficulty of producing good courseware as the biggest impediment to growth. The attempted solution to this problem, authoring systems to help teachers write their own courses, has failed in the past. However, a system developed by the author, based on an alternative teaching strategy has performed very well in early trials.

Work remains to be done on the authoring system, and in related areas of CAL.

## References

### References

- [1] "Computers, Education and Special Needs", E.Goldenberg, S.Russell, C.Carter, 1984.
- [2] "Suggestions for Teaching Children with Reading Difficulties in Primary and Secondary Schools", Department of Education, 1978.
- [3] "Helping Adults to Read", Tom McFarlane, Adult Literacy Guides, 1978.
- [4] "Investigating the Impact of Computer Instruction on Elementary Students' Reading Achievement", P.Norton, V.Resta, *Educational Technology*, March 1986.
- [5] "Releasing the Remedial Reader's Creative Power", L. Mountain, *Journal of Learning Disabilities*, January 1986.
- [6] "Artificial Intelligence: Applications in Education", R. Thorkildsen, "Computers in the Classroom" issue, *Educational Research Quarterly*, Vol. 10, No. 1, 1986.
- [7] "The Effect of the Locus of CAI Control Strategies on the Learning of Mathematics Rules", L.Goetzfried, M.Hannafin, *American Educational Research Journal*, Vol. 22, No. 2.
- [8] "Early Reading In-Service Course", Department of Education, 1984.
- [8] "Computer Assisted Co-operative Learning", D. Johnson, R. Johnson, *Educational Technology*, January 1986.
- [10] "Toward the Advancement of Microcomputer Technology in Special Education: An Overview of Intelligent CAI Systems", F.Roberts, *Peabody Journal of Education*, Vol. 62, No. 1, 1984.
- [11] "Teaching Basic Skills Through Microcomputer Assisted Instruction", G.Bass, R. Ries, W.Sharpe, *Journal of Educational Computing Research*, Vol. 2, No. 2, 1986.
- [12] "The Effects of Computer-Mediated Text on Measures of Reading Comprehension and Reading Behaviour", D.Reinking, R.Schreiner, *Reading Research Quarterly*, Vol. 20, No. 5.
- [13] "Classroom Use of Video Games", S. Silvern, *Educational Research Quarterly*, "Computers in the Classroom" issue, Vol. 10, No. 1, 1986.

## References

- [14] "The Computer in Education: A Critical Perspective", edited by Douglas Stern, New York Teachers College Press, 1985.
- [15] "Where is the Imaginative, Unstructured Educational Software?", Editorial, *Educational Technology*, May 1986.
- [16] "Computers and Education: the Software Production Problem", R.Nicolson, P.Scott, *British Journal of Educational Technology*, January 1986.
- [17] "Intelligent Course Authoring Systems", Computer Comments, *Educational Technology*, May 1986.
- [18] "Software Infusion: Using Computers to Enhance Instruction", S. Schiffman, *Educational Technology*, January 1986.
- [19] "Criteria for Evaluating Microcomputer Software for Reading Development: Observations Based on Three British Case Studies", C.Harrison, *Journal of Educational Computing Research*, Vol. 1, No. 2, 1985.
- [20] "Modern Technology in Education: From Teaching Machine to Microcomputers and Student Response Systems", G. McMeen, *Educational Technology*, August 1986.
- [21] "Present and Future Computer Courseware Authoring Systems", R.Kozma, *Educational Technology*, June 1986.
- [22] "ESTOP: Software Tools for Computer-Based Education", G.E.Quick, *Research in Education*, May 1985.
- [23] "A Generative Authoring System Based on Cognitive Procedures for Diagnosis and Instruction", Research Notes edited by P.A.Butler, *Journal of Educational Research*, Vol. 2, No. 2, 1986.

## **Appendix A: Experiments**



## **Experiments**

Two experiments were conducted at the Sheltered Workshop: the first to introduce the Workshop trainees to a computer, and the second to test the usefulness of existing Macintosh reading software.

Four trainees participated in the experiments; deliberately chosen to cover a wide range of manual and reading abilities.

### **Experiment 1: Learners' Reactions to a Computer**

The following areas of competency with the Macintosh were informally tested:-

#### **(a) Physical Skills**

It was expected that the trainees would be quite manually adept as their activities at the Workshop include such things as weaving and woodwork.

The trainees were shown how to use the mouse and then asked to draw some simple shapes using the MacPaint pen. All four trainees quickly mastered these techniques apart from mistakes, also made by their teachers, such as holding the mouse with both hands and getting the mouse trapped at the edge of the desk. Only one of the four had any difficulty drawing a circle.

The trainees were already familiar with keyboards. They did have trouble with rapidly repeating keys, which was rectified by altering the keyboard settings. Their greatest difficulty was in associating the keys pressed with the altered image on the screen - they were losing their place on the screen through having to look down at the keyboard while typing.

#### **(b) Conceptual Ability**

Complex, graphical computer environments, like the Macintosh's, could easily be very confusing for IH people. The trainees' toleration and understanding of the following items was examined: two and three dimensional, abstract and realistic pictures, screen clutter, and different text sizes and fonts. Overall, they managed very well, even when asked to identify quite indistinct digitised photographs (created with the Thunderscanner digitiser).

MacPaint's screen was too cluttered for one trainee to be able to find and select the drawing implement he wanted. When the desired implement was pointed out to him, he was able to repeatedly re-identify it. In general, the trainees responded very well to visual prompts.

The New York font was the most suitable. 12 point type was too small for two of the trainees to read, but all could manage 24 point type.

### (c) Concentration

During their initial exposure, the trainees had a more relaxed attitude toward the computer than their teachers did. They were not put off by their mistakes; they seemed to realise they could experiment with the computer. When confronted with problems they couldn't solve for themselves, they listened unselfconsciously to advice.

Prior to seeing the trainees on the computer, the teachers had remarked that some of the trainees had very short attention spans and would probably tire of the computer quickly. Surprisingly then, each trainee was engrossed for over an hour. The computer kept their attention better than a book or television because of its reactivity.

### (d) Ability to Understand the Speech Synthesiser

The Macintosh speech synthesiser is only barely understandable when accompanied by a transcript of its speech. Unsurprisingly, the trainees did not understand it at all. It was possible to alter its pronunciation but this had little effect. Current research has abandoned true synthesis in favour of digital recording. Usable software will probably become available in a couple of years.

## Discussion

The trainees preferred MacPaint to some action games shown to them. The difference seemed to be in pacing; the games imposed their timing on the trainees while MacPaint is entirely responsive and moved at the trainees' speeds.

Interestingly, manual ability and intelligence were independent of reading ability. The discrepancy between how well a trainee could interact with a computer and how well he or she reads, means software presenting more difficult reading passages should not necessarily be more interactively complex.

## Experiment 2: Macintosh Courseware

### Flashcard: Drill and Practice Courseware

Several of the skills exercises at the Sheltered Workshop involve multiple choice, such as choosing the correct substitution in a cloze exercise. Flashcard takes several question-answer pairs from the teacher and presents the learner with the questions, one at a time, with a multiple choice

## Appendix A: Experiments

selection of answers. Flashcard was not designed specifically for teaching reading or for the intellectually handicapped.

The trainees understood Flashcard as it closely imitated Workshop pen and paper exercises. Selecting correct answers was quite easy with the mouse, but they tired of the unvarying multiple choice format.

They found extraneous words on the screen, such as the title, distracting. The only reward given for a correct answer, "Good!", was insufficient.

### Drill: Tutorial Courseware

Drill is a linear tutorial building system that can be used by reading teachers to present passages with embedded questions to the learner. The common exercise of presenting a learner with a passage and then asking him or her to find a synonym in the passage for a given word was modelled in Drill. Drill requires learners to manipulate a window containing the passage and answer using the keyboard.

Drill rejected several one word answers because although the trainees had correctly identified the word referred to in the passage, they spelt it incorrectly. Learning to spell is not a part of learning to read, so this was confusing to the trainees - especially when the spelling error was caused by mistyping. It would be preferable for the learner to indicate the correct word using the mouse.

As a very simple text mediation technique, Drill displays MacPaint pictures at various points in an exercise. The trainees were distracted by the length of time it took to display these pictures, and the sometimes jerky movements of the text in the text window.

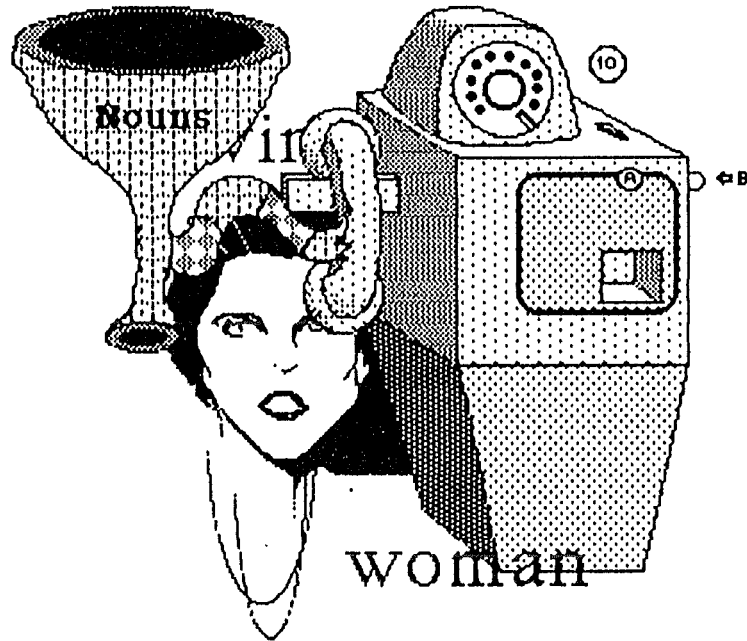
### Discussion

A CAL reading programme could not be built on Drill and Flashcard. Neither program was written for special education. They fail to use the Macintosh in the imaginative way that Experiment 1 showed trainees respond to so well.

## **Appendix B: Teachers' Manual**

# Ice Cream Cake

A Simulation and Problem Solving  
Authoring System



# Ice Cream Cake

Ice Cream Cake is a system for inventing computer-assisted learning (CAL) reading exercises suited to slow learners and the intellectually handicapped. It gives you, the teacher, the means to create very interactive, illustrated exercises for a computer in a short time. These exercises may allow the learner to practise skills, simulate an activity useful to the learner, or give the learner the chance to be creative in solving a problem.

It is assumed that the reader is familiar with the Macintosh and can use MacPaint and a text editor. Descriptions of these can be found in the appropriate manuals.

Two major premisses behind Ice Cream Cake's design are: exercises should be locally produced so that they 'fit' the learners culturally and in their level of difficulty; and there should be a large turnover of exercises because old material becomes boring, concepts need continual reinforcement in new material, and new concepts need to be introduced.

The following sections show you how to use Ice Cream Cake:

(1) <u>What an Exercise Looks Like</u>	page 2
(2) <u>Building an Exercise</u>	3
(3) <u>Sample Exercises</u>	8

## **(1) What an Exercise Looks Like**

Ice Cream Cake exercises are "objects worlds". An object may be either a piece of text or a picture, and will appear on the computer's screen. Objects are described by you, and experimented with by the learner. Experimenting may involve touching objects, moving them or placing them on top of one another. All such actions by the learner can be accomplished using the mouse. Various things can be described by you to happen after anything done by the learner. Ice Cream Cake supports a high degree of input from both the learner and the teacher, so that the learner is stimulated while your instructions flexibly direct the exercise.

Your object descriptions and instructions for an exercise are typed into a normal text file, which is called the script for that exercise.

### **Example**

An exercise could be built to help a learner associate words with pictures, eg Associating the word "woman" with the picture of a woman's face. In this case you might define two objects, "woman" and its corresponding picture, and instruct that some reward should be given to the learner after he or she places the word on the picture. (See Sample Exercise A.)

## (2) Building an Exercise

The script of an exercise is divided into two sections: the Objects section and the Instructions section. The Objects section contains a list of all the objects you want to present on the screen to the learner. The Instructions section specifies what you wish to happen whenever the learner does something important (such as placing a word on its corresponding picture in the previous example).

### Describing Objects

As mentioned earlier, there are two types of objects: text objects and picture objects. Both types can be handled in exactly the same manner by the learner, but their description by the teacher is different.

The text of a text object is given in the exercise script while the picture of a picture object comes from MacPaint. This means any MacPaint picture, whether drawn by you, taken from a catalogue or a digitised photograph, can be used as an object. Exactly how to use MacPaint for object making is described later.

All objects have names that you give them. To describe a text object you first give its name and then the text for it between double quotes.

eg                *Sentence, "Who is her boyfriend?".*  
                    *A, "A".*

describes two objects: *Sentence* and *A*.

Names may be any unbroken sequence of characters.

Declaring a picture object is even easier - you only have to give the name of the MacPaint file containing its picture. It is best to give these files names that represent their contents.

eg    In the script declare the object *10-Speed*, and prepare a picture of a 10-Speed in a MacPaint file called "10-Speed".

### Attributes

Objects have attributes which may be used to make them more interesting and more lifelike. An object may be **movable** or **immovable**, **visible** or **invisible**, and **transparent** or **opaque**. Attributes affect the ways in which the learner can interact with an object. For example, any **visible** object may be touched by the learner; if it is additionally **movable** it may be dragged around the screen. **Transparent** objects allow objects overlapped by them to still be seen.

Text objects may also be **big** or **small** depending on the size you want their letters.



When you declare an object, it is assumed to be **immovable**, **opaque** and **visible** (and **small** if applicable). You may change these assumed attributes by listing the attributes you want an object to have after its name.

eg                    *10-Speed, movable.*

makes the object, 10-Speed, **movable** as well as **visible** and **opaque**.

*Sentence, "Who is her boyfriend?", movable, big.*

makes Sentence **movable** and **big** as well as **visible** and **opaque**.

Attributes may be changed while the exercise is in progress and that process is detailed in Giving Instructions.

### Positioning Objects

Another attribute objects have is position, which cannot be ascribed in the same way as the other attributes.

The size of the exercise screen is the same as the MacPaint screen, and an object's position in MacPaint determines its initial position on the exercise screen. This is very useful for positioning objects with respect to one another, even though each object must have its own, separate MacPaint file.

To position all of the objects in an exercise, collect and position them all on the MacPaint screen and save this. Then, using this original, for each object, rubout all the surrounding objects and save the result as the MacPaint image of that object.

Positioning text objects is done with reference to picture objects. The method for this appears in the next section.

### Giving Instructions

While an exercise is running and the learner is manipulating the objects you have supplied, you may direct his or her learning attention by triggering actions upon certain events. For example, you may want something to happen after the learner has touched a particular object; or at the very start of the exercise. For this purpose there are instructions - given in the Instructions section.

All instructions begin with an indication of when you want them to occur.

eg                    *After touching Apple.*

Then follows a list of actions to be taken by the computer, perhaps *Show ToothMarks*, *hide Pear*.

The possible actions are:-

**Make** <an object> <a list of attributes>

*Make car visible, movable.* for example.

This is how an object's attributes may be changed while an exercise is in progress.

**Place** <an object> **on** <another object>

eg *Place food on table.*

Placement like this causes the first object to be moved abruptly so that its top lefthand corner coincides with that of the second object.

**Hide** <an object>

Shorthand for making an object invisible.

**Show** <an object>

Shorthand for making an object visible.

**Animate** <a sequence of objects>

eg *Animate SadMouth, SmallSmile, BigGrin.*

When a sequence of objects is animated, each object is treated like a frame in a film - shown for an instant. Animating a sequence is a good reward. If an object is only to be used in an animated reward, it should be declared **invisible**.

The times when you can direct something to happen are:-

**To start with . . .**

**To finish with . . .**

**Before moving** <an object>

**After touching** <an object>

**After moving** <an object>

**After placing** <one object> **on** <another object>

Starting actions given after **To start with** are all performed by the computer before the learner has a chance to do anything. The most common use of this clause is to position text objects. Unlike picture objects, text objects initially have no position. Text objects are positioned by placing them on picture objects.

eg

*To start with:*

*Place SentenceA on PositionA.*

*Place SentenceB on PositionB.*

where *PositionA* and *PositionB* are picture objects.

Usually picture objects used only as places to put text objects would be declared **invisible**.

Finishing actions given in the **To finish with** clause might be to show a farewell message to the learner or to show a correct answer. An exercise finishes when the learner touches the keyboard.

eg

*To finish with: Show GoodBye.*

Don't confuse the **After placing** clause, which refers to learner placement, with the computer-initiated placement (**Place . . . on . . .**).

Instructions may be given in any order - write them down as they occur to you. You do not need to anticipate the order in which a learner will try things out in your exercise.

A complete exercise script must contain two headings, **OBJECTS** and **INSTRUCTIONS** to introduce their respective sections. The word **END** should also be given to mark the end of the script.

ie

## **OBJECTS**

All your object descriptions.

## **INSTRUCTIONS**

Your instructions.

**END.**

### **Putting Comments in Scripts**

It can be useful to make comments about an exercise in its script. Comments are for your reference and are ignored by the computer. A comment could possibly state for which learners an exercise is suitable and give a brief description of it. To make a comment you type an asterisk at the left margin of the line and use the rest of the line for your comment.

eg

\* Practice with sight words for Warren.

\* Written 6th September '86.

## Style

The textual format of exercise scripts is up to you. Use punctuation to make your script more understandable to you, and split instructions over several lines if they look better that way. Use appropriate object names. To avoid confusing the computer, none of the words given in bold type above can be used as names for objects.

Once a learner has discovered the interactive vocabulary of the mouse - touching objects, moving them and placing them on top of one another - there should be no need for written instructions on the screen or constant teacher supervision.

To make a change to an exercise, just change its script.

### **(3) Sample Exercises**

The following sample exercises have been chosen to demonstrate all of Ice Cream Cake's features as well as showing how they can be used to create exercises of widely different natures.

- A: Associating Words with Pictures
- B: Letter Shapes
- C: Ordering Words in a Sentence
- D: Navigating a Page of Writing
- E: Using a Public Telephone
- F: A Memory Game

\* Sample Exercise A: Associating Words with Pictures.

\* Quite a simple exercise: the learner must move the words around  
\* and place them on the appropriate picture. A reward is given  
\* after each successful placement. This sort of exercise could help \*  
the learner identify nouns through reinforcement with pictures.

#### OBJECTS

\* Positions on the screen where the words will appear at the start  
\* of the exercise.

Line1 is invisible. Line2 is invisible.  
Line3 is invisible. Line4 is invisible.

\* Pictures to match the words and rewards.  
\* The reward given is a small animation of the identified object.  
\* The frames of the animation must be hidden until needed.

UglyLittleChair.  
CrackedLeg is invisible. FallingLeg is invisible.  
DeadLeg is invisible.

PrettyModel.  
HaveALook is invisible. GetLipsReady is invisible.  
Kiss is invisible.

DoughyApple.  
WormHole is invisible. Worm is invisible.  
WormTongue is invisible.

NewBike.  
Wheel1 is invisible. Wheel2 is invisible. Wheel3 is invisible.

\* The words.

Chair is "chair", movable, transparent, big.  
Woman is "woman", movable, transparent, big.  
Apple is "apple", movable, transparent, big.  
10-Speed is "10-Speed", movable, transparent, big.

Goodbye is invisible, transparent.

#### INSTRUCTIONS

To finish with: Show Goodbye.

To start with:

Place Apple on Line1.  
Place Woman on Line2.  
Place 10-Speed on Line3.  
Place Chair on Line4.

After placing Chair on UglyLittleChair:  
Animate CrackedLeg, FallingLeg, DeadLeg.

After placing Apple on DoughyApple:

    Animate WormHole, Worm, WormTongue, Worm, WormTongue, Worm.  
    Show WormHole.

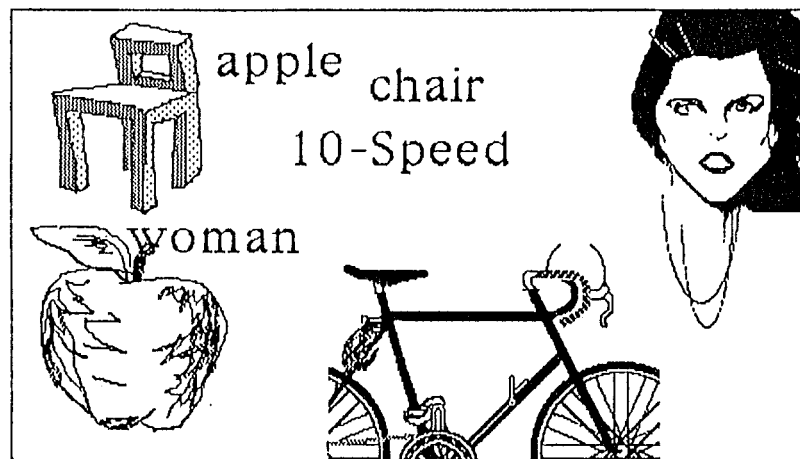
After placing 10-Speed on NewBike:

    Animate Wheel1, Wheel2, Wheel3, Wheel1, Wheel2, Wheel3.

END

## Sample A: Associating Words with Pictures

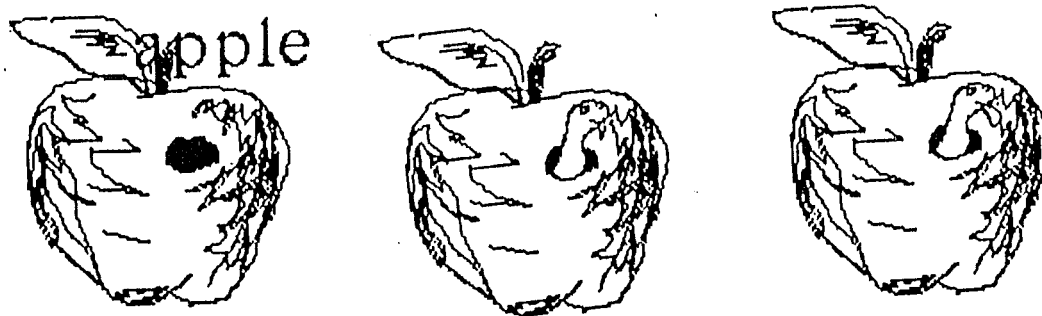
The exercise screen soon after the start of the exercise:



You would have collected all these pictures on the MacPaint screen, exactly as appears here, before cutting each into its own, separate file.

The learner has tried a few things and found that the words can be moved around. Notice, in the script, that words have been declared **movable** while the picture objects have all been assumed to be **immovable**. The words are also big (New York font, 24 point) and **transparent** so that the pictures can be seen beneath them.

The learner moves the word "apple" onto the picture of the apple (called DoughyApple in the script), giving an animated reward:



You would have created the animated worm by drawing it on top of the apple in MacPaint, erasing the surrounding apple and saving the worm separately. The pieces of animation are WormHole, Worm, WormTongue.

The learner may repeat a correct placement and still get the appropriate reward.



\* Sample Exercise C: Ordering Words in a Sentence.

OBJECTS

Reward.

\* The source of the words.

WordWell invisible.

Slice1. Slice2. Slice3. Slice4. Slice5. Slice6.

\* Correct places to put the words.

[We] [went] [to\_the] [pictures] [on] [Saturday] [night].

\* The words.

We "We", movable, big, transparent.

went "went", movable, big, transparent.

to\_the "to the", movable, big, transparent.

pictures "pictures", movable, big, transparent.

on\* "on", movable, big, transparent.

Saturday "Saturday", movable, big, transparent.

night "night.", movable, big, transparent.

Marker1 invisible.

Marker2 invisible.

OtherSentence1 "The bus was crowded on the", big.

OtherSentence2 "way home.", big.

INSTRUCTIONS

After placing We on [We]:

Place We on [We]. Make We immovable.

Hide Slice1. Hide [We].

After placing went on [went]:

Place went on [went]. Make went immovable.

Hide Slice2. Hide [went].

After placing pictures on [pictures]:

Place pictures on [pictures]. Make pictures immovable.

Hide Slice3. Hide [pictures].

After placing on\* on [on]:

Place on\* on [on]. Make on\* immovable.

Hide Slice4. Hide [on].

After placing Saturday on [Saturday]:

Place Saturday on [Saturday]. Make Saturday immovable.

Hide Slice5. Hide [Saturday].

After placing night on [night]:

Place night on [night]. Make night immovable.

Hide Slice6. Hide [night].

After touching Slice3: Place pictures on WordWell.  
After touching Slice4: Place on\* on WordWell.  
After touching Slice5: Place Saturday on WordWell.  
After touching Slice6: Place night on WordWell.

To start with:

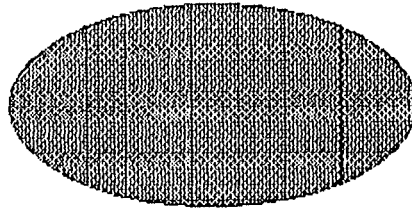
Place to\_the on [to\_the]. Hide [to\_the].  
Make to\_the immovable, opaque.

Place OtherSentence1 on Marker1.  
Place OtherSentence2 on Marker2.

END

### Sample C: Ordering Words in a Sentence

To begin with, two related sentences are shown with several words missing from the first:

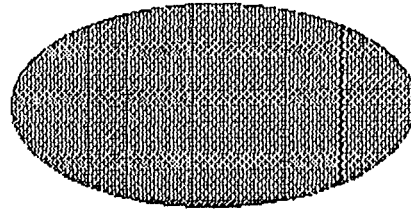


to the

The bus was crowded on the  
way home.

After some experimentation, the learner finds that the large oval yields words when touched:

went

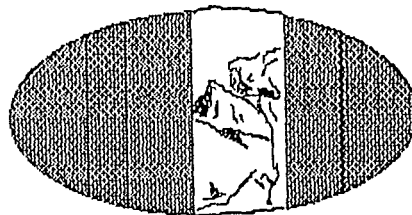


to the

The bus was crowded on the  
way home.

Successful placement of the word removes part of the oval to reveal an interesting picture:

We  
Saturday



went to the

The bus was crowded on the  
way home.

- \* Sample Exercise D: Navigating a Page of Writing.
- \* Written by a local teacher.
- \* Some people have difficulty in following words across a line,
- \* and lines down a page. This exercise forces the correct motion.
- \* This script may look clumsily repetitive but it is very easy to
- \* prepare using a cut and paste editor.

## OBJECTS

- \* Places to put the pieces of sentences.

Place1.	Place2.	Place3 invisible.
Place4 invisible.	Place5 invisible.	Place6 invisible.
Place7 invisible.	Place8 invisible.	Place9 invisible.
Place10 invisible.	Place11 invisible.	Place12 invisible.
Place13 invisible.	Place14 invisible.	Place15 invisible.
Place16 invisible.	Place17 invisible.	Place18 invisible.

- \* The pieces of sentences.

Piece1 "Bond's mouth was dry."  
 Piece2 "He wet his lips"  
 Piece3 "with his tongue"  
 Piece4 "and searched the lake"  
 Piece5 "with his telescope."  
 Piece6 "There was"  
 Piece7 "something pink deep down."  
 Piece8 "It rose slowly as Vickers'"  
 Piece9 "body came to the surface."  
 Piece10 "It lay on the water"  
 Piece11 "with its "  
 Piece12 "head down."  
 Piece13 "A short length of steel"  
 Piece14 "was sticking out of"  
 Piece15 "the body"  
 Piece16 "below the left shoulder."  
 Piece17 "The sun shone on the"  
 Piece18 "metal feathers."


## INSTRUCTIONS


To start with: Place Piece1 on Place1.


After touching Place2: Place Piece2 on Place2. Show Place3.  
 After touching Place3: Place Piece3 on Place3. Show Place4.  
 After touching Place4: Place Piece4 on Place4. Show Place5.  
 After touching Place5: Place Piece5 on Place5. Show Place6.  
 After touching Place6: Place Piece6 on Place6. Show Place7.  
 After touching Place7: Place Piece7 on Place7. Show Place8.  
 After touching Place8: Place Piece8 on Place8. Show Place9.  
 After touching Place9: Place Piece9 on Place9. Show Place10.  
 After touching Place10: Place Piece10 on Place10. Show Place11.  
 After touching Place11: Place Piece11 on Place11. Show Place12.  
 After touching Place12: Place Piece12 on Place12. Show Place13.

After touching Place15: Place Piece15 on Place15. Show Place16.  
After touching Place16: Place Piece16 on Place16. Show Place17.  
After touching Place17: Place Piece17 on Place17. Show Place18.  
After touching Place18: Place Piece18 on Place18.

END

1. Bond's mouth was dry. 

2. Bond's mouth was dry. He wet his lips 

3. Bond's mouth was dry. He wet his lips with his tongue  


\* Sample Exercise E: Using a Public Telephone.

OBJECTS

\* Places to put directions.

TheHeading, invisible.  
TheLine, invisible.

\* Parts of the telephone.

BodyOfPhone.  
CoinSlot, transparent.  
DialCircle, transparent, invisible.  
RefundBox.  
ButtonA, invisible.  
ButtonB, invisible.  
Receiver, movable.  
Coin, transparent.

\* Frames to be animated.

A invisible. B invisible.

Disappear1 invisible, transparent.  
Disappear2 invisible, transparent.  
Disappear3 invisible, transparent.

FlashSlot invisible.

Dialling1 invisible.  
Dialling2 invisible.  
Dialling3 invisible.

\* Vocabulary.

>PublicTelephone "Public Telephone", big.

>PickUp "Pick up the receiver.", invisible.  
>InsertCoin "Put the coin in the slot.", invisible.  
>DialNumber "Dial the number.", invisible.  
>AorB "Press button A or B.", invisible.  
>GotRefund "You got a refund.", invisible.  
>Successful "Now start speaking.", invisible.  
>Good! "Good!", invisible.

INSTRUCTIONS

To start with:

Place >PickUp on TheLine. Place >InsertCoin on TheLine.  
Place >DialNumber on TheLine. Place >AorB on TheLine.  
Place >GotRefund on TheLine. Place >Successful on TheLine.  
Place >Good! on TheLine.

Place >PublicTelephone on TheHeading.

After placing Coin on CoinSlot:

    Animate Disappear1, Disappear2, Disappear3.

    Hide Coin. Hide >InsertCoin.

    Show DialCircle. Show >DialNumber.

Before moving Receiver:

    Hide >PickUp. Show >Good!.

After moving Receiver:

    Make Receiver immovable.

    Make Coin movable.

    Animate FlashSlot, CoinSlot, FlashSlot, CoinSlot, FlashSlot.

    Hide >Good!. Show >InsertCoin.

After touching ButtonB:

    Hide ButtonB. Hide ButtonA. Hide >AorB.

    Show >GotRefund.

    Show Coin. Place Coin on RefundBox.

After touching ButtonA:

    Hide ButtonA. Hide ButtonB. Hide >AorB.

    Show >Successful.

After touching DialCircle:

    Hide >DialNumber. Hide >GotRefund. Hide DialCircle.

    Show ButtonA. Show ButtonB. Show >AorB.

    Animate A, B, A, B, A, B, A, B, A, B.

    Animate Dialling1, Dialling2, Dialling3.

END



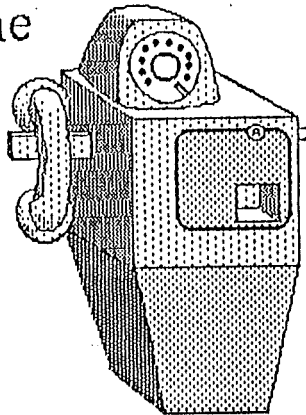
## Sample E: Using a Public Telephone

The receiver is the only movable object initially.

### Public Telephone

⑩

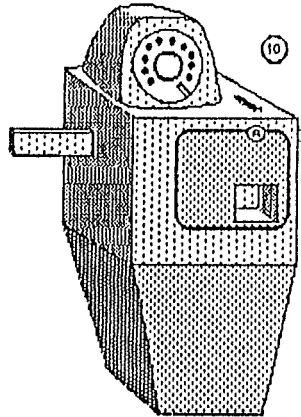
Pick up the receiver.



As soon as the receiver is moved, the instruction at the bottom of the screen changes:



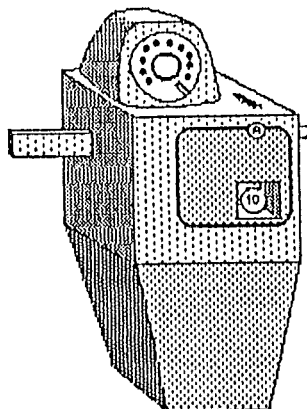
Put the coin in the slot.



The coin is shown disappearing down the slot. It can be retrieved by pressing button B:



You got a refund.



\* Sample Exercise F: A Memory Game.

\* Not obviously a reading exercise - it is intended to help readers

\* with poor concentration.

## OBJECTS

Background.

1 transparent. 2 transparent. 3 transparent. 4 transparent.  
5 transparent. 6 transparent. 7 transparent. 8 transparent.  
9 transparent. 10 transparent. 11 transparent. 12 transparent.  
13 transparent. 14 transparent. 15 transparent. 16 transparent.  
17 transparent. 18 transparent. 19 transparent. 20 transparent.  
21 transparent. 22 transparent. 23 transparent. 24 transparent.  
25 transparent.

## INSTRUCTIONS

After touching 1: Hide 1. Hide 9. Hide 14. Hide 19.  
Show 6. Show 7. Show 8.

After touching 2: Hide 2. Hide 12. Hide 23. Hide 25.  
Show 16. Show 17. Show 21. Show 22.

After touching 3: Hide 3. Hide 4. Hide 5. Hide 9. Hide 14.  
Show 7. Show 8. Show 14. Show 19. Show 17.  
Show 18. Show 12.

After touching 4: Hide 4. Show 1. Show 7. Show 13. Show 19.  
Show 25.

After touching 5: Hide 5. Hide 6. Show 21. Show 1. Show 25.

After touching 6: Hide 6. Hide 23. Hide 17. Hide 19. Hide 13.  
Show 11.

After touching 7: Hide 7. Hide 2. Hide 8. Hide 6. Hide 12.  
Show 1. Show 3. Show 11. Show 13.

After touching 8: Hide 8. Hide 24. Hide 23. Hide 22. Hide 6.  
Hide 11. Hide 16. Hide 2. Hide 3. Hide 4.  
Hide 10. Hide 15. Hide 20. Show 22.

After touching 9: Hide 9. Hide 13. Show 1. Show 3. Show 5.  
Show 7.

After touching 10: Hide 10. Hide 1. Hide 7. Hide 21. Hide 17.  
Show 15.

After touching 11: Hide 11. Show 1. Show 21. Show 15.

After touching 12: Hide 12. Hide 7. Hide 11. Hide 6. Show 9.  
Show 10. Show 14. Show 15.

After touching 13: Hide 13. Hide 8. Hide 20. Hide 24. Show 4.

After touching 14: Hide 14. Hide 13. Show 25.

After touching 15: Hide 15. Hide 11. Hide 12. Hide 13. Hide 17.  
Hide 18. Hide 19.

After touching 16: Hide 16. Hide 2. Hide 7. Hide 12. Hide 17.  
Hide 22. Show 10. Show 14. Show 18. Show 22.

After touching 17: Hide 17. Show 4. Show 15.

After touching 18: Hide 18. Show 8.

After touching 19: Hide 19. Hide 9. Hide 14. Hide 19. Show 13.  
Show 1. Show 11. Show 6.

After touching 20: Hide 20. Hide 14. Hide 15. Hide 19. Hide 24.  
Hide 25. Show 1. Show 17. Show 10.

After touching 21: Hide 21. Hide 22. Hide 14. Hide 18. Show 10.  
Show 4.

After touching 22: Hide 23. Hide 22. Hide 4. Hide 5. Hide 10.  
Hide 16. Hide 21.

After touching 23: Hide 23. Hide 9. Hide 14. Hide 19. Show 11.  
Show 12. Show 9. Show 10.

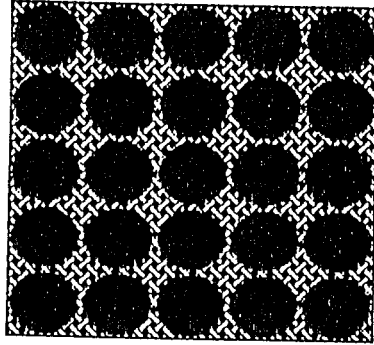
After touching 24: Hide 24. Hide 16. Show 2.

After touching 25: Hide 25. Hide 19. Show 14. Show 20. Show 24.  
Show 18.

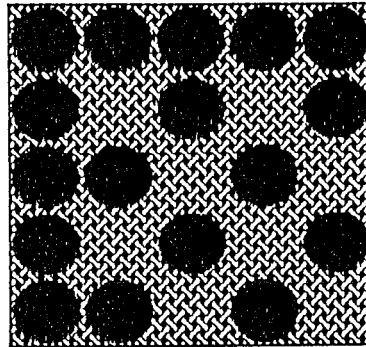
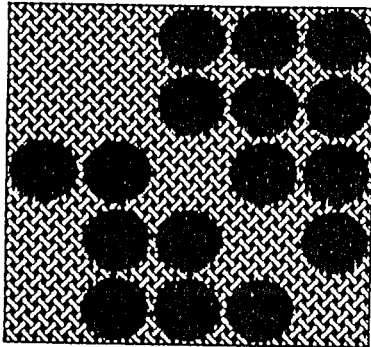
END

## Sample F: A Memory Game

The game starts with a square grid of discs:



Touching a disc causes some other discs to disappear and some to reappear:



The object is to remove all the discs.

## **Appendix C: Program Listing**

**Module 1 - Compiler**

**Module 2 - MacPaint Access**

**Module 3 - Run System**

## **Module 1 - Compiler**

(

## LANGUAGE DESCRIPTION

... = at least one, [] = optional.

<exercise> ::= <object section><instruction section><end sy>

<object section> ::= <objects sy><object list>

<object list> ::= <object>...

<object> ::= <object name>[<sentence>][<attribute list>]

<attribute list> ::= <attribute>...

<attribute> ::= <movable sy>|<immovable sy>|<transparent sy>|<opaque sy>|  
<visible sy>|<invisible sy>|<big sy>|<small sy>

<instruction section> ::= <instructions sy><instruction list>

<instruction list> ::= <instruction>...

<instruction> ::= <normal instruction>|<start instruction>|<finish instruction>

<start instruction> ::= <to start with sy><action list>

<finish instruction> ::= <to finish with sy><action list>

<normal instruction> ::= <when preposition><event><action list>

<when preposition> ::= <before sy>|<after sy>

<event> ::= <moving event>|<touching event>|<placing event>

<moving event> ::= <moving sy><object name>

<touching event> ::= <touching sy><object name>

<placing event> ::= <placing sy><object name><object name>

<action list> ::= <action>...

<action> ::= <place action>|<make action>|<show action>|<hide action>|  
<animate action>

<place action> ::= <place sy><object name><object name>

<make action> ::= <make sy><object name><attribute list>

<show action> ::= <show sy><object name>

<hide action> ::= <hide sy><object name>

<animate action> ::= <animate sy><animate list>

<animate list> ::= <object name>...

Separators can be spaces, commas, colons, semi-colons, or fullstops. An object name is any sequence not containing separators, and a sentence any sequence between double quotes, on a single line.

Some redundant words can be used: the, in, on, is, to, with, and.

A line can be used for commenting by putting an '\*' in the first position.  
}



```

program Beck;

($i MemTypes.ipas  )
($i QuickDraw.ipas )
($i OSIntf.ipas    )
($i ToolIntf.ipas  )

($i Decls.ipas      )
var ticks: longint;
    tRect: rect;

($i InitScreen.ipas )
($i Compile.ipas    )
($i GetPictures.ipas)
($i Run.ipas        )

($L Beck/Rsrc       )

begin
MoreMasters; {Create extra heap pointer blocks}
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;
MoreMasters;

InitScreen;
CompileExercise;
GetPictures;
RunExercise
end.

```

{Declarations for the compiler and the run system.}

```
const maxObjects = 100;
    maxIgnore = 7;

    space      = ' ';
    quote      = '"';
    commentIndicator = '*';

    successful = 0;      {for 10 operations}

    dontCare = 100;      {any object}

    lumpSize = 30000;    {not 50K with RAMdisk}

    dragDelay = 6; {1/60 sec}
    animateDelay = 15;
    finishDelay = 180;

    bigAdjust = 5;      {the tails of letters}
    smallAdjust = 3;

    {-----}

type {SYMBOLS}
    symbols = (firstSy,
                endSy,
                objectsSy,
                movableSy, immovableSy, transparentSy, opaqueSy, bigSy, smallSy,
                visibleSy, invisibleSy,
                instructionsSy,
                toStartWithSy, toFinishWithSy,
                beforeSy, afterSy,
                placingSy, movingSy, touchingSy,
                placeSy, makeSy, showSy, hideSy, animateSy,
                identSy);

    actions = (placeAc, makeAc, showAc, hideAc, animateAc);

    whenPreps = (beforePr, afterPr);

    attributes = (transparentAt, visibleAt, bigAt, movableAt, {positive}
                  invisibleAt, immovableAt, smallAt, opaqueAt); {negative}

    verbs = (placingVb, movingVb, touchingVb);

    symbolSets = set of symbols;

    charSets = set of char;

    {ATTRIBUTES}
    attributeBunches = set of attributes;

    {OBJECTS}
```

```

objectNos = 1..maxObjects;

objects = record
    name:      str255;
    bunch:     attributeBunches;
    detail:    bitMap;
    hOffset:   integer; {within detail}
    outline:   rgnHandle;
    position:  rect;
    sentence:  str255
end;

{EVENTS}
events = record
    verb:      verbs;
    object2:   objectNos
end;

{ANIMATION}
animatePtrs = ^animateItems;
animateItems = record
    object:   objectNos;
    next:     animatePtrs
end;

{ACTIONS}
actionPtrs = ^actionItems;
actionItems = record
    operator:   actions;
    object1,
    object2:   objectNos;
    addBunch:   attributeBunches;
    animateList: animatePtrs;
    next:       actionPtrs
end;

{TABLE}
tablePtrs = ^tableItems;
tableItems = record
    when:           whenPreps;
    observedEvent:  events;
    actionsToTake:  actionPtrs;
    next:           tablePtrs
end;

{-----}

var {MOUSE}
    theEvent:   eventRecord;
    oldPt,newPt: point;

    onlyStarting,
    dragging:    boolean;

{-----}

```

```

{WINDOWS}
wRecord:  windowRecord;
theWindow: windowPtr;

{-----}

{PICTURES}
picName: str255;
picFile: packed file of byte; {the cutout section of MacPaint}
picSize: integer;

{space for pictures is allocated in here}
largeLump: packed array[0..lumpSize] of byte;
lumpStart: integer; {position in largeLump for the next picture}

{-----}

{SYMBOLS}
sy: symbols; {current symbol}

word: str255; {current word}

ignoreList: array[1..maxIgnore] of str255;

separators,
uppercase:      charSets;

pictureSet,
verbSet,
attributeSet,
actionSet,
instructionSet: symbolSets;

{EVENT TABLE - the run system refers to this after each event to see
 what actions to take. Indexed by the main object in the event.}

eventTable: array[objectNos] of tablePtrs;

startActions,
finishActions: actionPtrs;

{TEMPORARY stores for the latest occurrences}
tBunch: attributeBunches;

tActionPtr,
tActionList:  actionPtrs;

tAnimatePtr,
tAnimateList: animatePtrs;

tWhenPrep:    whenPreps;

emptyEvent,
tEvent:       events;

tTablePtr:    tablePtrs;

```

```
tObjectName: str255;

tMainObject, {of the instruction}
tObjectNo,
nObjects: objectNos;

tSentence: str255;

{SOURCE}
lineNo: integer;

aSentence: boolean; {whether a sentence discovered}

srcFile: text;

errLine,
srcLine,
srcName: str255;

{OBJECTS}
objectTable: array[objectNos] of objects;

{DIAGNOSTICS}
errFile, {contains errLine and an explanation}
diag: text;

diagRect: rect;

{DRAGGING}
curRgn,prevRgn,exposedRgn: rgnHandle;
dummyRect: rect;
```

```

{
  A recursive descent compiler to generate a table of actions to take
  when various events occur.
}

{-----}

procedure ScreenDiag(mes: str255);
begin
  EraseRect(diagRect);
  MoveTo(10, 10);
  DrawString(word);
  MoveTo(150, 10);
  DrawString(mes)
end;

{-----}

procedure CloseFiles;
begin
  close(srcFile);
  close(diag);
  close(errFile)
end;

{-----}

procedure OpenFiles;
begin
  open(srcFile, 'Source');
  open(errFile, 'Source.Err');
  open(diag, 'D1');
  reset(srcFile);
  rewrite(errFile);
  rewrite(diag)
end;

{-----}

procedure InitSymbolSets;
begin
  attributeSet := [movableSy, immovableSy, transparentSy, opaqueSy,
                  visibleSy, invisibleSy, bigSy, smallSy];
  instructionSet := [toStartWithSy, toFinishWithSy, afterSy, beforeSy];
  actionSet := [placeSy, makeSy, showSy, hideSy, animateSy];
  verbSet := [touchingSy, placingSy, movingSy];
end;

{-----}

procedure InitIgnoreList;
begin
  ignoreList[ 1] := 'the';
  ignoreList[ 2] := 'in';
  ignoreList[ 3] := 'on';
  ignoreList[ 4] := 'is';
  ignoreList[ 5] := 'to';

```

```

ignoreList[ 6] := 'with';
ignoreList[ 7] := 'and';
end;

{-----}

procedure InitEventTable;

var i: integer;

begin
for i := 1 to maxObjects do eventTable[i] := nil;
end;

{-----}

procedure Init;
begin
InitSymbolSets;
InitIgnoreList;
InitEventTable;

startActions := nil;
finishActions := nil;

uppercase := ['A'..'Z'];
separators:= [' ', ',', ';', '.', ':', chr(9) {tab}];

lineNo := 0;
nObjects := 0;

SetRect(dummyRect,0,0,0,0);

srcLine := ' ';

emptyEvent.object2 := dontCare;
end;

{-----}

procedure Stop(message: str255);
begin
writeln(errFile,'Line ',lineNo,' : ',errLine); writeln(errFile);
writeln(errFile,'"',word,'" - ',message);
CloseFiles;
halt
end;

{-----}

function TextObject(object: objectNos): boolean;
begin
TextObject := (length(objectTable[object].sentence) > 0)
end;

{-----}

```

```

procedure InsertTail;
var t1,t2: animatePtrs;

begin
  if tAnimateList = nil then
    tAnimateList := tAnimatePtr
  else
    begin
      t1 := tAnimateList;
      repeat
        t2 := t1;
        t1 := t1^.next
      until t1 = nil;
      t2^.next := tAnimatePtr
    end
end;

{-----}

function Redundant(word: str255): boolean;

var i: integer;
    found: boolean;

begin
  i := 0;
  found := false;
  while (i < maxignore) and (not found) do
    begin
      i := i+1;
      found := word = ignoreList[i]
    end;

  Redundant := found
end;

{-----}

procedure Lower(var s: str255);

var i: integer;

begin
  for i := 1 to length(s) do
    if s[i] in uppercase then
      s[i] := chr(ord(s[i])+32)
end;

{-----}

function Blank(s: str255): boolean;

var i: integer;

begin
  Blank := true;

```



```

if length(s) > 0 then
  if s[1] <> commentIndicator then
    for i := 1 to length(s) do
      if not (s[i] in separators) then Blank := false
    end;
  {-----}

procedure FindEnds(s: str255; var start, finish: integer);

var i, len: integer;

begin
  len := length(s);

  i := 0;
  repeat i := i+1 until not (s[i] in separators);
  start := i;

  if start = len then
    finish := start
  else
    repeat i := i+1 until (s[i] in separators) or (i = len);

  if s[i] in separators then finish := i-1 else finish := i
end;

{-----}

function GetSentence(start: integer): boolean;

var len, i: integer;

begin
  if srcLine[start] = quote then
    begin
      GetSentence := true;
      len := length(srcLine);
      i := start+1;
      while (i < len) and (srcLine[i] <> quote) do i := i+1;

      if srcLine[i] = quote then {closed properly}
        begin
          tSentence := copy(srcLine, start+1, i-start-1);
          writeln(diag, 'Sentence: ', tSentence, '');
          delete(srcLine, start, i)
        end
      else
        Stop('Unclosed quote at the end of the sentence.')
      end
    end
  else
    GetSentence := false
end;

{-----}

```

```

procedure NextWord;
var start, finish: integer;
begin
repeat
    while Blank(srcLine) and (not eof(srcFile)) do
    begin
        readln(srcFile, srcLine);
        errLine := srcLine;
        lineNo := lineNo+1
    end;

    FindEnds(srcLine, start, finish);
    aSentence := GetSentence(start);

    if not aSentence then
    begin
        word := copy(srcLine, start, finish-start+1);
        Lower(word);
        delete(srcLine, 1, finish)
    end;

    until (not (Redundant(word) or aSentence)) or eof(srcFile)
end {NextWord};

{-----}

procedure NextSy(expectSet: symbolSets; message: str255);
begin
NextWord;
writeln(diag, ' ', word, ' ');

sy := identSy;

if word = 'end'           then sy := endSy;
if word = 'objects'       then sy := objectsSy;
if word = 'movable'       then sy := movableSy;
if word = 'immovable'     then sy := immovableSy;
if word = 'visible'       then sy := visibleSy;
if word = 'invisible'     then sy := invisibleSy;
if word = 'transparent'   then sy := transparentSy;
if word = 'opaque'        then sy := opaqueSy;
if word = 'instructions'  then sy := instructionsSy;
if word = 'start'         then sy := toStartWithSy;
if word = 'finish'        then sy := toFinishWithSy;
if word = 'before'        then sy := beforeSy;
if word = 'after'         then sy := afterSy;
if word = 'moving'        then sy := movingSy;
if word = 'touching'      then sy := touchingSy;
if word = 'placing'       then sy := placingSy;
if word = 'show'          then sy := showSy;
if word = 'hide'          then sy := hideSy;
if word = 'make'          then sy := makeSy;

```

Compiler Bugs:-

- (1) Couldn't step through an enumerated type.
- (2) Didn't like nested if-then-else.

```

if word = 'place'      then sy := placeSy;
if word = 'small'     then sy := smallSy;
if word = 'big'       then sy := bigSy;
if word = 'animate'   then sy := animateSy;

```

```

if not (sy in expectSet) then Stop(message)
end;

```

```

{-----}

```

```

procedure ObjectName(var theName: str255);
begin
writeln(diag, 'ObjectName'); ScreenDiag('ObjectName');

theName := word
end;

```

```

{-----}

```

```

function Declared: boolean;

```

```

var found: boolean;
    i:      integer;

```

```

begin
found := false;
i := 0;
repeat
    i := i+1;
    found := tObjectName = objectTable[i].name
until (i = nObjects) or found;

```

```

tObjectNo := i;
Declared := found
end;

```

```

{-----}

```

```

procedure DeclaredObject(var objectNo: objectNos);
begin
writeln(diag, 'DeclaredObject'); ScreenDiag('DeclaredObject');

NextSy([identSy], 'Object name expected. ');
ObjectName(tObjectName);
if not Declared then Stop('This object has not been declared. ');
objectNo := tObjectNo
end;

```

```

{-----}

```

{An object may be transparent, visible, big and movable, or the opposites of these. The opposites are recorded as the absence of these positive attributes in the bunch.}

```

procedure Attribute;
begin
writeln(diag, 'Attribute'); ScreenDiag('Attribute');

```

```

case sy of
  movableSy:    tBunch := tBunch + [movableAt];
  immovableSy:  tBunch := tBunch - [movableAt];
  transparentSy: tBunch := tBunch + [transparentAt];
  opaqueSy:     tBunch := tBunch - [transparentAt];
  visibleSy:    tBunch := tBunch + [visibleAt];
  invisibleSy:  tBunch := tBunch - [visibleAt];
  bigSy:        tBunch := tBunch + [bigAt];
  smallSy:      tBunch := tBunch - [bigAt];
end;

NextSy(attributeSet+[instructionsSy,identSy],
        'Attribute, object name or "INSTRUCTIONS" expected.')
```

end;

{-----}

```

procedure MakeItem;
begin
  writeln(diag,'MakeItem'); ScreenDiag('MakeItem');
```

```

case sy of
  movableSy:    tBunch := tBunch + [movableAt];
  immovableSy:  tBunch := tBunch + [immovableAt];
  transparentSy: tBunch := tBunch + [transparentAt];
  opaqueSy:     tBunch := tBunch + [opaqueAt];
  visibleSy:    tBunch := tBunch + [visibleAt];
  invisibleSy:  tBunch := tBunch + [invisibleAt];
  bigSy,
  smallSy:      (makes no sense to change its size)
end;
```

```

NextSy(attributeSet+instructionSet+actionSet+[endSy],
        'Attribute, instruction, action or "END" expected.')
```

end;

{-----}

```

procedure MakeList;
begin
  writeln(diag,'MakeList'); ScreenDiag('MakeList');
```

```

repeat MakeItem until not (sy in attributeSet)
end;
```

{-----}

```

procedure PlaceAction;
begin
  writeln(diag,'PlaceAction'); ScreenDiag('PlaceAction');
```

```

  tActionPtr^.operator := placeAc;
  DeclaredObject(tActionPtr^.object1);
  DeclaredObject(tActionPtr^.object2);
  NextSy(actionSet+instructionSet+[endSy],'Action or "END" expected.')
```

end;

```

{-----}

procedure MakeAction;
begin
writeln(diag, 'MakeAction'); ScreenDiag('MakeAction');

tActionPtr^.operator := makeAc;
tBunch := [];

DeclaredObject(tActionPtr^.object1);
NextSy(attributeSet, 'A list of attributes was expected. ');
MakeList;

tActionPtr^.addBunch := tBunch
end;

{-----}

procedure ShowAction;
begin
writeln(diag, 'ShowAction'); ScreenDiag('ShowAction');

tActionPtr^.operator := showAc;
DeclaredObject(tActionPtr^.object1);
NextSy(actionSet+instructionSet+[endSy], 'Action or "END" expected. ')
end;

{-----}

procedure HideAction;
begin
writeln(diag, 'HideAction'); ScreenDiag('HideAction');

tActionPtr^.operator := hideAc;
DeclaredObject(tActionPtr^.object1);
NextSy(actionSet+instructionSet+[endSy], 'Action or "END" expected. ')
end;

{-----}

procedure AnimateObject;

var t1,t2: animatePtrs;

begin
writeln(diag, 'AnimateObject'); ScreenDiag('AnimateObject');

new(tAnimatePtr);

with tAnimatePtr^ do
begin
ObjectName(tObjectName);
if Declared then
object := tObjectNo
else
Stop('This object has not been declared. ');

```

```

    next := nil
end;

InsertTail;

NextSy([IdentSy,endSy]+instructionSet+actionSet,
      'Animation object, action, "END" or instruction expected.')
end;

{-----}

procedure AnimateList;
begin
writeln(diag,'AnimateList'); ScreenDiag('AnimateList');

repeat AnimateObject until not (sy = identSy)
end;

{-----}

procedure AnimateAction;
begin
writeln(diag,'AnimateAction'); ScreenDiag('AnimateAction');
tActionPtr^.operator := animateAc;

tAnimateList := nil;

NextSy([IdentSy],'A list of objects to animate was expected. ');
AnimateList;

tActionPtr^.animateList := tAnimateList
end;

{-----}

procedure Action;
begin
writeln(diag,'Action'); ScreenDiag('Action');

new(tActionPtr);

case sy of
    makeSy:    MakeAction;
    placeSy:   PlaceAction;
    showSy:    ShowAction;
    hideSy:    HideAction;
    animateSy: AnimateAction
end;

tActionPtr^.next := tActionList;
tActionList := tActionPtr
end;

{-----}

procedure ActionList;
begin

```

```

writeln(diag, 'ActionList'); ScreenDiag('ActionList');

tActionList := nil;
repeat Action until not (sy in actionSet)
end;

{-----}

procedure TouchingEvent;
begin
writeln(diag, 'TouchingEvent'); ScreenDiag('TouchingEvent');

tEvent.verb := touchingVb;
DeclaredObject(tMainObject)
end;

{-----}

procedure MovingEvent;
begin
writeln(diag, 'MovingEvent'); ScreenDiag('MovingEvent');

tEvent.verb := movingVb;
DeclaredObject(tMainObject);
end;

{-----}

procedure PlacingEvent;
begin
writeln(diag, 'PlacingEvent'); ScreenDiag('PlacingEvent');

tEvent.verb := placingVb;
DeclaredObject(tMainObject);
DeclaredObject(tEvent.object2)
end;

{-----}

procedure Event;
begin
writeln(diag, 'Event'); ScreenDiag('Event');

tEvent := emptyEvent;

case sy of
  movingSy:   MovingEvent;
  touchingSy: TouchingEvent;
  placingSy:  PlacingEvent;
end;

NextSy(actionSet, 'A list of actions was expected.')
end;

{-----}

procedure WhenPreposition;

```

```

begin
writeln(diag,'WhenPreposition'); ScreenDiag('WhenPreposition');

case sy of
  beforeSy: tWhenPrep := beforePr;
  afterSy:  tWhenPrep := afterPr;
end;

NextSy(verbSet,'Verb expected.')
end;

{-----}

procedure NormalInstruction;
begin
writeln(diag,'NormalInstruction'); ScreenDiag('NormalInstruction');

WhenPreposition;
Event;
ActionList;

tTablePtr := eventTable[MainObject];
new(eventTable[MainObject]);

with eventTable[MainObject] do
begin
  when      := tWhenPrep;
  observedEvent := tEvent;
  actionsToTake := tActionList;
  next      := tTablePtr;
end

end;

{-----}

procedure FinishInstruction;
begin
writeln(diag,'FinishInstruction'); ScreenDiag('FinishInstruction');

NextSy(actionSet,'A list of actions to take was expected.');
ActionList;
finishActions := tActionList;
end;

{-----}

procedure StartInstruction;
begin
writeln(diag,'StartInstruction'); ScreenDiag('StartInstruction');

NextSy(actionSet,'A list of actions to take was expected.');
ActionList;
startActions := tActionList;
end;

{-----}

```



```

procedure Instruction;
begin
writeln(diag,'Instruction'); ScreenDiag('Instruction');

case sy of
  toStartWithSy: StartInstruction;
  toFinishWithSy: FinishInstruction;
  afterSy,
  beforeSy:      NormalInstruction;
end
end;

{-----}

procedure InstructionList;
begin
writeln(diag,'InstructionList'); ScreenDiag('InstructionList');

repeat Instruction until not (sy in instructionSet)
end;

{-----}

procedure InstructionSection;
begin
writeln(diag,'InstructionSection'); ScreenDiag('InstructionSection');

NextSy(instructionSet,'List of instructions expected. ');
InstructionList
end;

{-----}

procedure AttributeList;
begin
writeln(diag,'AttributeList'); ScreenDiag('AttributeList');

repeat Attribute until not (sy in attributeSet)
end;

{-----}

procedure Object;
begin
writeln(diag,'Object'); ScreenDiag('Object');

nObjects := nObjects + 1;
ObjectName(tObjectName);

tBunch := [visibleAt];
delete(tSentence,1,length(tSentence));

NextSy(attributeSet+[identSy,instructionsSy],
  'Attribute list, object name or "Instructions" expected. ');

if sy in attributeSet then AttributeList;

```

```

with objectTableInObjects do
  begin
    name      := tObjectName;
    bunch     := tBunch;
    sentence  := tSentence
  end

end;

{-----}

procedure ObjectList;
begin
  writeln(diag, 'ObjectList'); ScreenDiag('ObjectList');

  repeat Object until sy = instructionsSy
end;

{-----}

procedure ObjectSection;
begin
  writeln(diag, 'ObjectSection'); ScreenDiag('ObjectSection');

  NextSy([identSy], 'An object name was expected. ');
  ObjectList
end;

{-----}

procedure Exercise;
begin
  writeln(diag, 'Exercise'); ScreenDiag('Exercise');

  ObjectSection;
  InstructionSection;
  writeln(errFile, 'No mistakes. ')
end;

{-----}

procedure CompileExercise;
begin
  Init;
  OpenFiles;

  NextSy([objectsSy], '"OBJECTS" expected. ');
  Exercise;

  CloseFiles
end;

```

## **Module 2 - MacPaint Access**

{Takes a MacPaint file and cuts the picture of the object out of it.  
Outline and detail files are produced.}

```
program PreparePicture(input,output);

($i MemTypes.ipas  )
($i QuickDraw.ipas )
($i OSIntf.ipas    )
($i ToolIntf.ipas  )

const pageDepth = 239;    {MacPaint screen dimensions}
      pageWidth  = 415;

      successful = 0;      {for 10 operations}

type  discblocks = packed array[1..512] of byte;
      discfiles  = file of discblocks;

      byteLines  = packed array[0..71] of byte;
      byteLinePtrs = ^byteLines;

var   picName: str255;
      picFile: packed file of byte; {the cutout section of MacPaint}
      picSize: integer;

      MacPaintFile: discfiles;
      paintingBits: bitMap;
      painting:     array[0..pageDepth] of byteLines;

      lineBits: array[0..575] of 0..1;

      leftByte,rightByte,
      i,j,t           : integer;

($i PaintAccess.ipas)

function Another: boolean;
var ch: char;
begin
  writeln; write('Another? '); readln(ch); writeln; Another := ch = 'y'
end;

begin
  repeat
    write('Object: '); readln(picName);
    open(MacPaintFile,picName);

    with paintingBits do
      begin
        write('Reading MacPaint: ',picName,'...');
        baseAddr := @painting;
```

```

ReadPainting(baseAddr);
writeln('finished.');
```

with bounds do

```

  begin
    write('Cutting out picture...');
    GetPaintingRgn(left,top,right,bottom);
    writeln('finished.');
```

writeln('Bounds: ',left,', ',top,', ',right,', ',bottom,', ');

```

    leftByte := left div 8;    {rowBytes must be even}
    rightByte := right div 8;
    if odd(rightByte-leftByte+1) then rightByte := rightByte+1;

    insert('.Pic',picName,length(picName)+1);
    open(picFile,picName);
    rewrite(picFile);

    write('Writing picture: ',picName,'...');
    write(picFile,left div 256,left mod 256,
              top div 256,top mod 256,
              right div 256,right mod 256,
              bottom div 256,bottom mod 256);

    for i := top to bottom do
      for j := leftByte to rightByte do
        begin
          t := painting[i][j];
          write(picFile,t)
        end;

    writeln('finished.')
```

end;

```

close(picFile)
end

until not Another
end.
```

```

{Reads the MacPaint files to get the pictures for each object.}

{-----}

procedure ReadData(var buf: discblocks);
begin
  if not eof(MacPaintFile) then read(MacPaintFile,buf)
end;

{-----}

{Reads a MacPaint file.}

procedure ReadPainting(dstPtr: Ptr);
var srcBuf: array[1..2] of discblocks;
    srcPtr: ptr;
    i: integer;

begin
  reset(MacPaintFile);

  ReadData(srcBuf[1]); {skip the header}

  ReadData(srcBuf[1]); {prime the buffer}
  ReadData(srcBuf[2]);

  srcPtr := @srcBuf;

  for i := 0 to pageDepth {pageDepth} do
    begin
      UnPackBits(srcPtr,dstPtr,72);
      if ord(srcPtr) > ord(@srcBuf) + 512 then
        begin
          srcBuf[1] := srcBuf[2];
          ReadData(srcBuf[2]);
          srcPtr := pointer(ord(srcPtr) - 512)
        end
      end;

  close(MacPaintFile)
end {ReadPainting};

{-----}

function LeftBit(b: byte): integer;
begin
  LeftBit := 7;
  if b >= 2 then LeftBit := 6;
  if b >= 4 then LeftBit := 5;
  if b >= 8 then LeftBit := 4;
  if b >= 16 then LeftBit := 3;
  if b >= 32 then LeftBit := 2;
  if b >= 64 then LeftBit := 1;
  if b >= 128 then LeftBit := 0;
end;

```

{-----}

```
function RightBit(b: byte): integer;
begin
  if b div 128 = 1 then RightBit := 0;
  if (b mod 128) div 64 = 1 then RightBit := 1;
  if (b mod 64) div 32 = 1 then RightBit := 2;
  if (b mod 32) div 16 = 1 then RightBit := 3;
  if (b mod 16) div 8 = 1 then RightBit := 4;
  if (b mod 8) div 4 = 1 then RightBit := 5;
  if (b mod 4) div 2 = 1 then RightBit := 6;
  if b mod 2 = 1 then RightBit := 7;
end;
```

{-----}

{Calculates the region occupied by a picture of an object, and gives its position on the screen.}

```
procedure GetPaintingRgn(var leftmost, topmost,
                        rightmost, bottommost: integer);
```

```
var pageUsed,
    lineUsed:   boolean;

    left, right,
    i, lineNo:  integer;

    byteLine:   byteLines;
```

```
begin
  leftmost := pageWidth;
  rightmost := 0;
  pageUsed := false;

  for lineNo := 0 to pageDepth do
    begin
      byteLine := painting[lineNo];
      lineUsed := false;
      i := 0;
      while (i <= 71) and (not lineUsed) do
        if byteLine[i] > 0 then
          begin
            lineUsed := true;
            left := 8*i + LeftBit(byteLine[i]);
          end
        else
          i := i+1;

      if lineUsed then
        begin
          bottommost := lineNo;
          if not pageUsed then
            begin
              pageUsed := true;
              topmost := lineNo;
            end;
```

```

    i := 71;
    while byteLine[i] = 0 do i := i-1;
    right := 8*i + RightBit(byteLine[i]);

    if left < leftmost then leftmost := left;
    if right > rightmost then rightmost := right
    end;

end

end {GetPaintingRgn};

```



### **Module 3 - Run System**

(Sets up the screen, with as few windows and things as possible.)

```
procedure InitScreen;
begin
  InitGraf(@thePort);
  InitFonts;
  FlushEvents(everyEvent,0);
  InitWindows;
  InitDialogs(NIL);
  SetCursor(arrow);
  InitCursor;
  theWindow := GetNewWindow(256,@wRecord,pointer(-1));
  SetPort(theWindow);
  theWindow^.txFont := 2;
  DrawControls(theWindow);
  SetOrigin(0,0);

  SetRect(diagRect,0,0,350,15)
end;
```

```

function CopyMode(bunch: attributeBunches): integer;
begin
  if transparentAt in bunch then
    CopyMode := srcOr
  else
    CopyMode := srcCopy
  end;

  {-----}

procedure DrawObject(object: objectNos; maskRgn: rgnHandle);

var tRect: rect;

begin
  with objectTable[object],theWindow^.position do
    if TextObject(object) then
      begin
        if not ((transparentAt in bunch) or dragging) then
          EraseRect(position);
        if bigAt in bunch then
          begin
            txSize := 24;
            MoveTo(left,bottom-bigAdjust)
          end
        else
          begin
            txSize := 12;
            MoveTo(left,bottom-smallAdjust)
          end;
        DrawString(sentence)
      end
    else
      begin
        tRect := detail.bounds;
        OffsetRect(tRect,hOffset,0);
        CopyBits(detail,theWindow^.portBits,
                  tRect,position,
                  CopyMode(bunch),maskRgn)
      end
  end;

  {-----}

procedure DrawAllObjects;

var i: integer;

begin
  RectRgn(exposedRgn,theWindow^.portBits.bounds);
  for i := 1 to nObjects do
    with objectTable[i] do
      if visibleAt in bunch then DrawObject(i,exposedRgn)
  end;

  {-----}

```

```
procedure DrawObjects(alteredObject: objectNos; oldPosition: rect);
```

```
var i: integer;
```

```
begin
```

```
EraseRect(oldPosition);
```

```
for i := 1 to nObjects do
```

```
  with objectTable[i] do
```

```
    if visibleAt in bunch then
```

```
      if i = alteredObject then
```

```
        DrawObject(i,nil)
```

```
      else if RectInRgn(position,exposedRgn) then
```

```
        DrawObject(i,exposedRgn)
```

```
end;
```

```
{-----}
```

```
  ACTIONS: Place    - place one object on another object.
```

```
        Make      - change some of an object's attributes.
```

```
        Show     - make an object visible.
```

```
        Hide     - make an object invisible.
```

```
        Animate  - show several objects quickly in sequence.
```

```
-----}
```

```
procedure Animate(t: animatePtrs);
```

```
var ticks: longint;
```

```
begin
```

```
while t <> nil do
```

```
  with t^,objectTable[object] do
```

```
    begin
```

```
      if visibleAt in bunch then Delay(animateDelay,ticks)
```

```
    else
```

```
      begin
```

```
        bunch := bunch + [visibleAt];    {make temporarily visible}
```

```
        RectRgn(exposedRgn,position);
```

```
        DrawObjects(object,position);
```

```
        Delay(animateDelay,ticks);
```

```
        bunch := bunch - [visibleAt];
```

```
        DrawObjects(object,position)
```

```
      end;
```

```
    t := next
```

```
  end
```

```
end;
```

```
{-----}
```

```
procedure Place(object1,object2: objectNos);
```

```
var oldPosition: rect;
```

```
  left,top,
```

```
  dh,dv:      integer;
```

```

begin
with objectTable[object2] do
begin
left := position.left;
top := position.top;
end;

with objectTable[object1] do
begin
oldPosition := position;
dh := position.right - position.left;
dv := position.bottom - position.top;
SetRect(position, left, top, left+dh, top+dv);
RectRgn(prevRgn, oldPosition);
RectRgn(curRgn, position);
UnionRgn(prevRgn, curRgn, exposedRgn);
if not onlyStarting then DrawObjects(object1, oldPosition)
end
end;
end;

{-----}

procedure Make(object: objectNos; addBunch: attributeBunches);

var at:      attributes;
    oldPosition: rect;

begin
with objectTable[object] do
for at := transparentAt to opaqueAt do
if at in addBunch then
case at of
transparentAt: if not (transparentAt in bunch) then
begin
bunch := bunch + [transparentAt];
if visibleAt in bunch then
begin
RectRgn(exposedRgn, position);
DrawObjects(object, position)
end
end;

opaqueAt: if transparentAt in bunch then
begin
bunch := bunch - [transparentAt];
if visibleAt in bunch then
begin
RectRgn(exposedRgn, position);
DrawObjects(object, position)
end
end;

movableAt: bunch := bunch + [movableAt];
immovableAt: bunch := bunch - [movableAt];

visibleAt: if not (visibleAt in bunch) then

```

```

begin
    bunch := bunch + [visibleAt];
    RectRgn(exposedRgn, position);
    DrawObjects(object, position)
end;

invisibleAt:  if visibleAt in bunch then
begin
    bunch := bunch - [visibleAt];
    RectRgn(exposedRgn, position);
    DrawObjects(object, position)
end
end;

end;

{-----}

procedure Show(object: objectNos);
begin
with objectTable[object] do
    if not (visibleAt in bunch) then
        begin
            bunch := bunch + [visibleAt];
            RectRgn(exposedRgn, position);
            DrawObjects(object, position)
        end
end;

end;

{-----}

procedure Hide(object: objectNos);
begin
with objectTable[object] do
    if visibleAt in bunch then
        begin
            bunch := bunch - [visibleAt];
            RectRgn(exposedRgn, position);
            DrawObjects(object, position)
        end
end;

end;

{-----}

procedure PerformAction(actionItem: actionItems);
begin
with actionItem do
    case operator of
        placeAc:   Place(object1, object2);
        makeAc:    Make(object1, addBunch);
        showAc:    Show(object1);
        hideAc:    Hide(object1);
        animateAc: Animate(animateList)
    end
end;

end;

{-----}

```

```

procedure LookAtTable(object1: objectNos; when: whenPreps;
                      verb: verbs; object2: objectNos);

var t1: tablePtrs;
    t2: actionPtrs;

begin
  t1 := eventTable[object1];
  while t1 <> nil do
    begin
      if (t1^.when = when) and (t1^.observedEvent.verb = verb)
        and (t1^.observedEvent.object2 = object2) then
        begin
          t2 := t1^.actionsToTake;
          while t2 <> nil do
            begin
              PerformAction(t2^);
              t2 := t2^.next
            end
          end;

          t1 := t1^.next
        end
      end
    end

end {LookAtTable};

{-----}

procedure DoStartActions;

var t: actionPtrs;

begin
  onlyStarting := true;

  t := startActions;
  while t <> nil do
    begin
      PerformAction(t^);
      t := t^.next
    end;

  onlyStarting := false
end;

{-----}

procedure DoFinishActions;

var t:    actionPtrs;
    ticks: longint;

begin
  t := finishActions;
  if t <> nil then
    begin
      repeat

```

```

        PerformAction(t^);
        t := t^.next
    until t = nil;
    Delay(finishDelay,ticks)
end
end;

```

{-----}

```

function OtherObject(object1: objectNos; var object2: objectNos): boolean;

```

```

var found: boolean;
    i: integer;
    thePt: point;

```

```

begin
    found := false;
    GetMouse(thePt);

    i := nObjects;
    while (i > 0) and (not found) do
        with objectTable[i] do
            begin
                if (i <> object1) and (visibleAt in bunch) and
                    (PtInRect(thePt,position)) then
                    begin
                        found := true;
                        object2 := i
                    end;
                i := i-1
            end;
        end;
    end;

```

```

OtherObject := found
end;

```

{-----}

```

function ObjectTouched(var object: objectNos): boolean;

```

```

var found: boolean;
    i: integer;

```

```

begin
    found := false;
    GetMouse(newPt);

    i := nObjects;
    while (i > 0) and (not found) do
        with objectTable[i] do
            begin
                if PtInRect(newPt,position) and (visibleAt in bunch) then
                    begin
                        found := true;
                        object := i
                    end;
                i := i-1
            end;
        end;
    end;

```



```

ObjectTouched := found
end;

{-----}

procedure DragObject(object: objectNos; var moved: boolean);

var ticks:      longint;
    oldPosition: rect;

begin
dragging := true;
with objectTable[object] do
begin
oldPt := newPt;
GetMouse(newPt);
Delay(dragDelay,ticks);

if not equalPt(oldPt,newPt) then {some movement}
begin
moved := true;
oldPosition := position;
OffsetRect(position,newPt.h-oldPt.h,newPt.v-oldPt.v);

RectRgn(prevRgn,oldPosition);
RectRgn(curRgn,position);
if transparentAt in bunch then
UnionRgn(prevRgn,curRgn,exposedRgn)
else
DiffRgn(prevRgn,curRgn,exposedRgn);

DrawObjects(object,oldPosition)
end
end;

dragging := false
end;

{-----}

procedure SetRgns;
begin
curRgn      := NewRgn;
prevRgn     := NewRgn;
exposedRgn := NewRgn
end;

{-----}

procedure RunExercise;

var t,moved:      boolean;

    object1,
    object2:      objectNos;

```

```

begin
EraseRect(theWindow^.portRect);
SetRgns;
DoStartActions;
DrawAllObjects;

repeat
    t := GetNextEvent(everyEvent, theEvent);

    if theEvent.what = mouseDown then
        begin
            if ObjectTouched(object1) then
                with objectTable[object1] do
                    begin
                        LookAtTable(object1, afterPr, touchingVb, dontCare);

                        if movableRt in bunch then
                            LookAtTable(object1, beforePr, movingVb, dontCare);

                        moved := false;
                        if movableRt in bunch then
                            while StillDown do DragObject(object1, moved);

                        if moved then
                            begin
                                UnionRgn(prevRgn, curRgn, exposedRgn);
                                DrawObjects(object1, position);

                                if OtherObject(object1, object2) then
                                    LookAtTable(object1, afterPr, placingVb, object2);

                                LookAtTable(object1, afterPr, movingVb, dontCare)
                            end
                        end
                    end
                end
            end

        until theEvent.what = keyDown;

DoFinishActions
end {RunExercise};

```

```

procedure messy(a: ptr; b: integer);
begin
  a^ := b
end;

{-----}

procedure CheckSpace;
begin
  if (lumpStart+picSize-1) > lumpSize then
    begin
      writeln(diag, 'No space left for this picture. ');
      halt
    end
  end;
end;

{-----}

procedure StuffLump;

var i, t: integer;

begin
  for i := lumpStart to (lumpStart+picSize-1) do
    begin
      read(picFile, t);
      messy(@largeLump[i], t)
    end;
  lumpStart := lumpStart+picSize
end;

{-----}

{Gets the picture of the object from the ".Pic" file.}

procedure PictureDetail(object: objectNos);

var leftByte, rightByte,
    j: integer;
    buf: array[0..7] of byte;

begin
  EraseRect(diagRect);
  MoveTo(10, 10);

  with objectTable[object], position, detail do
    begin
      DrawString(name);
      writeln(diag, 'Object: ', name);
      picName := name;
      insert('.Pic', picName, length(picName)+1);
      open(picFile, picName);
      reset(picFile);

      for j := 0 to 7 do read(picFile, buf[j]);
      left := 256*buf[0] + buf[1];
      top := 256*buf[2] + buf[3];
    end
  end;
end;

```

```

right := 256*buf[4] + buf[5];
bottom := 256*buf[6] + buf[7];

right := right + 1;
bottom := bottom + 1;

leftByte := left div 8;
rightByte := right div 8;

if odd(rightByte-leftByte+1) then rightByte := rightByte+1;
rowBytes := rightByte-leftByte+1;
baseAddr := @largeLump[lumpStart];
picSize := (bottom-top+1) * rowBytes;
SetRect(bounds,0,0,right-left,bottom-top);
hOffset := left mod 8;

writeln(diag,'    position: ',left,',',top,',',right,',',bottom,',');
writeln(diag,'    h offset: ',hOffset,',');
writeln(diag,'    Size = ',picSize,' bytes. Start: ',lumpStart,',');
writeln(diag,'    rowBytes = ',rowBytes);
end;

CheckSpace;
StuffLump;

writeln(diag);
close(picFile)
end;

(-----)

procedure TextOutline(object: objectNos);
begin
with objectTable[object],position,theWindow^ do
begin
EraseRect(diagRect); MoveTo(10,10); DrawString(name);

left := 0;
top := -100; {to keep the sentence out of the way to start with}
if bigRt in bunch then
begin
right := 2*StringWidth(sentence);
bottom := top+24
end
else
begin
right := StringWidth(sentence);
bottom := top+12
end
end
end;

(-----)

procedure GetPictures;

var i: integer;

```

```
begin
open(diag,'D2');
rewrite(diag);

lumpStart := 0;

for i := 1 to nObjects do
  if TextObject(i) then TextOutline(i) else PictureDetail(i);

close(diag)
end;
```