strate one type of behaviour that we can all agree partakes of intelligence: it learns from its mistakes. So well does it do this that you can sit down at the keyboard and win a few games to demonstrate the rules to a visitor, turn the keyboard over to the visitor, and be sure that he will be beaten soundly in spite of his best efforts. Sound intriguing?

The problem

Whether called HEX, HEXPAWN, or HEXAPAWN, it's all the same game, and commercial programs are available. What's wrong with them, aside from uninspired graphics? There are three major criticisms: 1) You'd think that all cheating moves should be detected, but this is not the case; 2) The computer's series of responses is always the same; 3) The program informs the computer before the game of all possible board positions and all the legal moves from each position. Now you may have a version to which one of these criticisms does not apply, but I'll wager that no one has a version to which none apply. Therefore we will proceed to prevent all types of cheating, provide an option for random play (reproduceable play is desirable initially, when studying how the computer goes about learning), and we will force the computer to learn without knowing any moves ahead of time.

The solution

Crudis' Compendium of Games-Writing Standards consists of all the items in italics (unless the editor, who has a *thing* about italics, sneaks a few extra in). Stand by for the first one: You've heard it before and I'll say it again: *draw a block diagram*. It should describe the sequence of required operations and be independent of the computer language used. It should use generally-accepted symbols. It should be so simple that it can be contained on one page. Figure 1 is an example of what I mean. After the program is written, line numbers are added to the block diagram and will be of inestimable convenience to anyone studying the program in the future. Next, if not sooner, *draw a screen layout* showing the position of the various subsections of the display. Figure 2 is the kind of thing needed. For the case in hand it shows that there is a square 20 x 20 board display whose upper left corner (origin) is 18 cursor-right steps from home, a 7 x 14 scoreboard with origin at 3 cursor downs, and a 1 x 39 prompt message with origin at 22 cursor downs.

Thirdly, make up a preliminary *list of variables*. The names of all variables should be *mnemonic*; that is, they should give a clue to their usage. If your variable names, you have a right to feel cuckolded. All manufacturers should provide for two-letter and letter-digit names. Having performed these three steps, you will undoubtedly find that your mind is cleared of cobwebs and that you realize that you don't understand the problem. This is normal. Do not under any circumstances start to write the program at this point. Instead, iterate the three steps until satisfaction sets in.

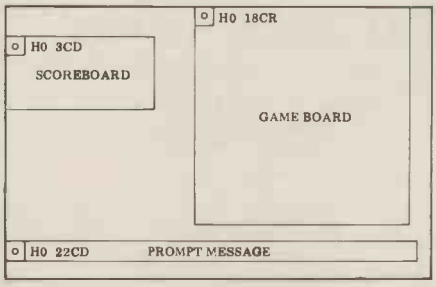

Fig. 2: Screen Layout for IMPHEX

Some sensible standards for variable names:

Reserve J, K, and L for counting variables in FOR-NEXT loops.

Reserve X, Y, and Z for coordinates.

Reserve D for delays, often useful in games to provide an impression of thoughtful response by the computer or just to give the human time to think.

There's nothing magic about the following list of variable names specific to IMPHEX, but you will find that, like a pair of new shoes, they will become comfortable after a few wearings and will seem to have been always right.

IMPHEX variables

Y,X: board coordinates.

HI, HF: human initial and final position, numerical keypad coordinates.

FNC(X): converts keyboard to X,Y coordinates.

XI, XF, YI, YF: X, Y initial and final positions.

W: relative move in X.

XT, YT, WT: trial ordinates.

XD, YD: display update scan ordinates.

BD(4,3): board array; stores current position.

BM(3, 3, 15): bad move array; stores up to 15 "bad move" positions.

SI, SF: temporary store values of BD during trial moves, initial and final; used to rewrite BD array if trial move not accepted.

P(3,3), M(3): position array and move array used to check that all positions and moves have been tried when in random scan mode.

N1, N2: position and move count when in random scan mode.

C: computer's turn

H: human's turn

TU: turn

CW: number of computer wins

HW: number of human wins

F1: illegal move count

F2: has "I needed that" been used?

F3: who won last game?

D$: down string; positions cursor on prompt line.

E$: erase string; erases prompt line.

BP$: board position string; positions cursor to write first piece on board.

BD$: board down string; moves cursor down on board row.

BL$: board left string; moves cursor left three board columns.

C$: computer piece

H$: human piece

BE$: board erase string; erases one piece

BU$: board up string; moves cursor up one board row.

Keypad to X,Y conversion

We are all agreed, aren't we, that it is desirable to use the familiar keypad format for inputting the human moves, but to convert each move to X, Y format to simplify computer analysis of their validity and legality and to make them compatible with the computer's moves and with storage in an X, Y array for analysis, yes? (If you are happy to input your moves at the keyboard in X, Y format at four digits each, initial X, initial Y, final X, final Y, then you can skip the following section.)

Fig. 3a

7	8	9
4	5	6
1	2	3

Keypad Orientation

Fig. 3b

31	32	33
21	22	23
11	12	13

X, Y Coordinates

Fig. 3: Naming the Cells of the IMPHEX Board

What is wanted is an algorithm to convert fig. 3A to fig. 3B. I'll give two examples, the second of which is a bit shorter. If anyone can come up with an even more compact solution, I'll be glad to have it. For those who may not be familiar with "DEF FNA(V)", this BASIC statement allows the user to define his own function which can subsequently be called up just as is any built-in function such as SIN, SQR, etc. V is a dummy variable. The function name is FN followed by any legal variable name.

The first time I saw a line like (X > 6)*(6—X) I was, to say the least, bemused, and attributed the ">" to the misprint gremlin who lives in all typewriters, linotypes, and similar inventions of the devil. However, as most of you no doubt know, (X > 6) has the value 0 if false and —1 if true, and (X > 6)*(6—X) is therefore a very convenient way of implementing IF X > 6 THEN FNA(X)=X—6 and compacting it within a more complex expression. Work out for yourself how line 10 below utilizes this technique.

Solution No. 1

```
10 DEF FNA(X)=(X > 6)*(6—X)+
(X<=6)*(3—X)—(X<=3)*3:XI=FNA
(HI):XF=FNA(HF)
20 DEF FNB(Y)=(Y<=3)—(Y>6)*3—
(Y<=6)*2:YI=FNB(HI):YF=FNB(HF)
```

Solution No. 2

```
30 DEF FNC(X)=X+10—(X>3)*7—
(X>6)*7:YI=INT(FNC(HI)/10)
40 XI=FNC(HI)—YI*10:YF=INT(FNC
HF)/10):XF=FNC(HF)—YF*10
```

Using an array to store the board

An array is a set of numbers arranged by coordinates. A one-dimensional array is no more than a list. A two-dimensional array is like a checkerboard, and arrays of any number of dimensions are theoretically possible. Some dialects of BASIC are sadly lacking in the number of arrays which can be set up simultaneously, the number of dimensions possible, and the size of the number which can be stored at each set of co-

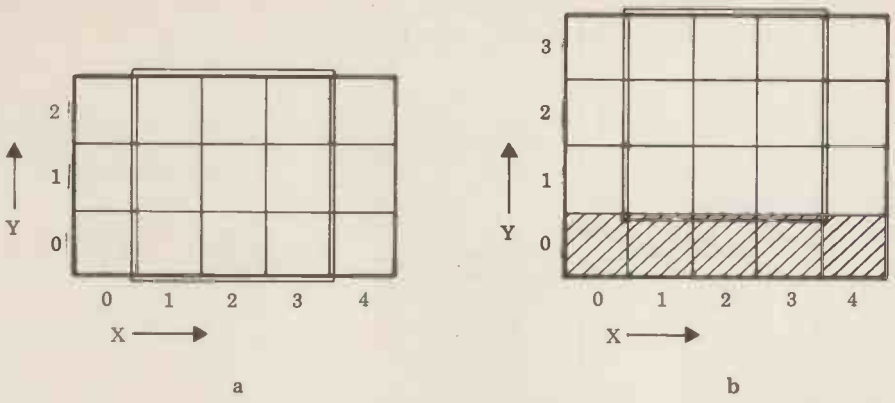

Fig. 4: The Board Array

ordinates. In PET's BASIC the number of arrays is limited only by the number of possible names, the number of dimensions is limited to about 34, and the number of elements to 255. Even this limitation is to be overcome in future ROMs. (Commodore's early literature refers to their arrays as matrices: I object to this usage as it implies that matrix arithmetic is provided, which it is not). To set up an array we program a line such as 50 DIM F(4,5) which defines an array F having two dimensions and coordinates 0 to 4 by 0 to 5. The contents of each coordinate point are automatically initialized at zero. Note that in PET the coordinate names start with 0: in some dialects they will start with 1. The array which we will implement to store the board position for IMPHEX is shown in fig. 4. The two extra columns shown at 4a simplify the computer's move-selection algorithm, as will be shown subsequently. We could define array BD by DIM BD(2,4), but then the rows would be numbered 0 to 2 instead of 1 to 3, so to avoid confusion we can throw away a bit of storage facility and define it as DIM BD(3,4) as shown at 4b which has the desired coordinates for the 3 x 3 board. The bottom row will not be used at all.

Similarly we are going to want another array to store bad moves, defined as the board pattern just before the human made his winning move. Pragmatic tests indicate that storage for 15 such patterns is more than enough, so this array can be defined as DIM BM(3,3,15). Here we don't need the extra columns, but have retained and wasted the zero X, Y, and Z ordinates so as to keep the numbering system straightforward. Array BM can be visualized as looking like fig. 5.

Detecting invalid and illegal moves

As part of the process of accepting human moves the computer must check them to determine if they are valid (correspond to agreed-upon nomenclature) and are legal (correspond to the agreed-upon rules). This is because humans tend, unlike computers, to misunderstand instructions, make mistakes, make mischief, or try to cheat. Fig. 6 blocks out the requirements, all of which can be written in two lines, thanks to the use of X, Y coordinates.

Any confusion on board nomenclature can be detected and responded to by:

420 IF HI<1 OR HI>9 OR HF<1 OR HF>9 THEN PRINT "NOT IN THIS GAME!" followed by another move query or a kindly and sympathetic offer to show the instructions again. If BD(XI, YI) is not equal to 1 (line 440) then the move is illegal because the initial cell is not occupied by a human piece (value +1). No human move can change the Y coordinate by other than +1, so if YF-YI is not equal to 1, (line 440) the move is illegal. A move can only be straight up or diagonal, so moves like 3, 4 are prohibited. These are tested for with IF ABS(XF−XI)>1 (line 440). When the human move is on the diagonal, it should be because he is capturing a computer piece (value −1).

Line 450 tests for this with IF ABS(XF−XI)=1 AND BD(XF,YF)<>-1 and similarly line 445 tests for the vertical move to an empty cell with IF XI=XF AND BD(XF,YF)<>0. No doubt we will always have with us the jokester who tries all illegal moves, so we can opt to cater for him by counting the number of illegal moves attempted and having the computer print an apt remark when the number reaches, say, 4. This printout should be positioned where it will not impede any other current display, so we will place it in the area reserved for the scoreboard, which is only seen at the end of each game (lines 510-540). Notice that *reversed text* should not be displayed nakedly, but *needs a surrounding border.* Later on I shall give you a "before and after" test to try out if you need any convincing on this point.

The computers move

Since we are not providing the computer with any strategy initially, its choice of piece to move can be made by one of two techniques, sequential or random. The sequential technique consists of looking at the board in an arbitrary fixed order, say, cells 7, 8, 9, 4, 5, 6. (No need to look at 1, 2, or 3 since the computer would have already won if it had occupied any of these cells.) The advantages of the sequential technique are that it is slightly easier to program for and that it makes it easier for the human who is studying the computer's responses to follow its learning processes. The advantages of the random technique come into play when the human player, after many sets of games, has learned to eke out his winning streak as long as possible by taking advantage of the computer's blind spots. Then converting from sequential to random selection makes for "a whole new ball game", as our American cousins say. I will append a discussion of random move generation as an option subsequently.

Having selected a trial piece, the computer must scan the possible moves for that piece until it finds a legal one. This scan also can be implemented either sequentially or randomly. If the move so selected is not a "bad move", it can be carried out and announced. A "bad move" is one which led to a win for the human in a previous game. In the implementations of HEX which I have seen to date, this process of identifying a bad move took place as follows: one array stored all possible board patterns (there are 33 of them). This entire array was scanned, board by board, line by line, cell by cell, until the pattern corresponding to the current game condition was found. Another array stored all possible computer moves for each possible pattern. As the end of each lost game occurred, the

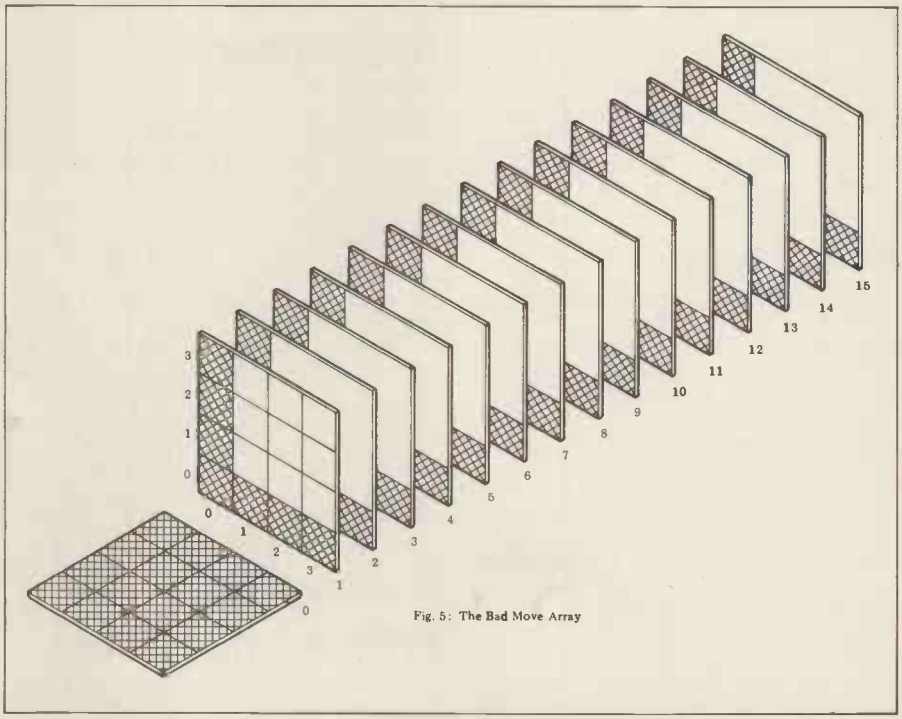

Fig. 5: The Bad Move Array

move leading to it was erased from the move array. If the computer determined that a board pattern it found itself in had no moves, it would resign, and that board pattern would be erased from the pattern array. Eventually only winning move series would exist in the two arrays. The best one can say of it is that this is a brute force technique!

The great improvement

To replace this technique, which on second thought I am willing to categorize as clumsy, inefficient, and lacking in charisma, I offer the following: The Bad Move array, which is tabula rasa or blank at the beginning of the game, will be used to record only the pattern reached in any game where the next move was a win for the human opponent. Then, before making any move, the computer will compare it with the Bad Move array and will reject it if it is found therein. In addition to the obvious appeal of its build-up from nothing, this technique is much more efficient in that it does not scan the whole Bad Move array but only the number of elements corresponding to the number of human wins to date. Also the scan of any board within the array can be cut off the moment the first cell being examined is found to be different from the corresponding cell in the contemplated move. And clearly the total memory requirement is much less.

Concept of trial move

I trust that you are all referring to fig. 1 as you read and will have noted that the test for a human win is economically combined with the computer's search for a move. Having found a legal move, the computer generates the position which would result from the move. This position is called a trial move since the computer is prepared to retreat from it if it is found to be a Bad Move.

To make a trial move, the computer stores the current position of the piece to be moved and the current condition of the space to be occupied as SI and SF respectively. Then, if the move is determined to be a bad move, it reverts to that position. Notice that only one pair of binary numbers need be temporarily remembered, not the entire board. If the Snark is not a Boojum, which is to say if the trial move is not a bad move, it is retained and the display updated accordingly.

A cute trick

Now the strange thing happens which is labelled on the block diagram, "See text for explanation"! The Acceptable Move is recorded in the Bad Move array!! This is so cute (in the sense of "clever, shrewd, ingenious") as to be unbelievable. Do you see the devilish beauty of it? If, on the next move, the human wins, it will have been the right thing to do. If, however, the human does not win, the computer will make another move and that move will be recorded in the same level of the array, overwriting the previous record. If the computer wins, the process continues to the next game and so on. The computer never scans this level of the array until after the human wins and the variable HW is incremented by one!

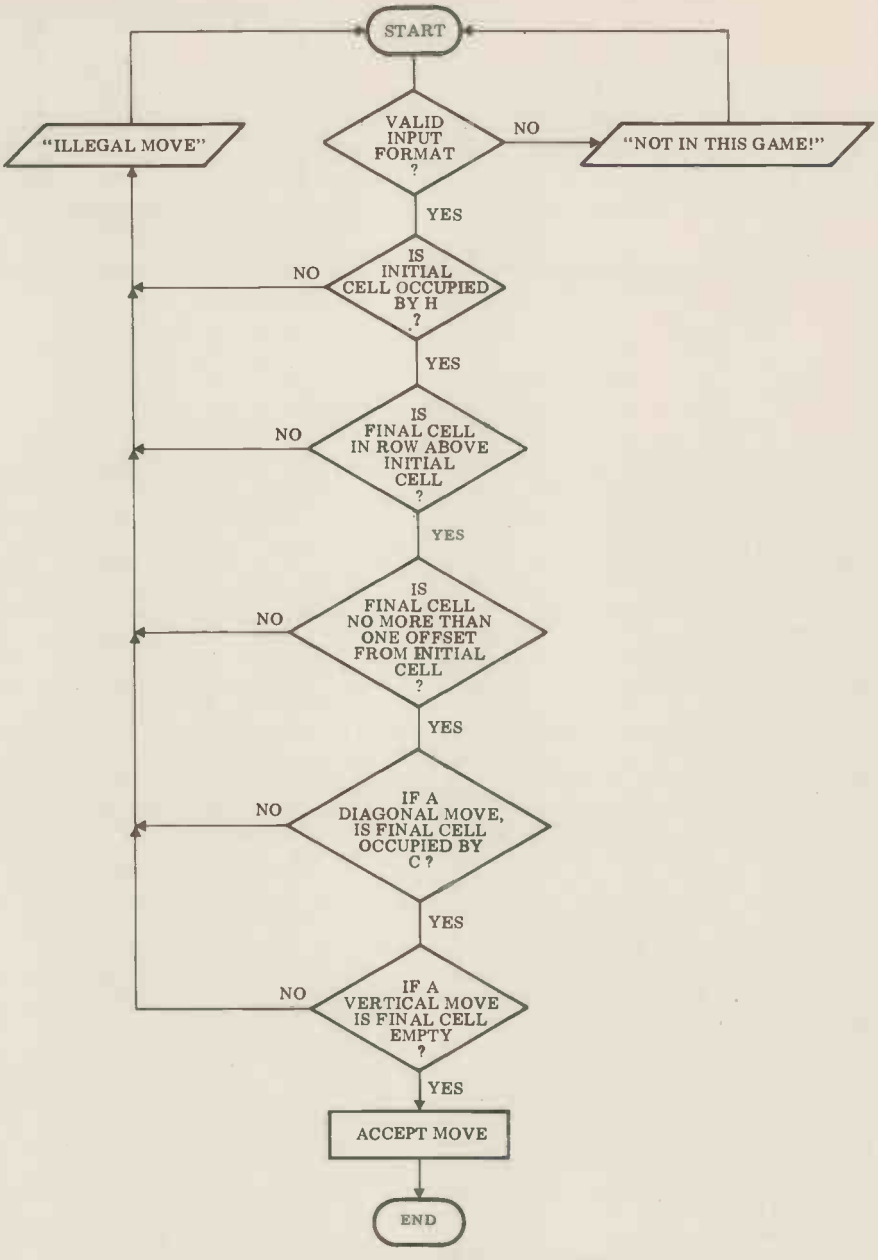

Fig. 6: Block Diagram for Checking Human's Moves in IMPHEX

Nested loops

Figure 7, which is a diagram of the nested loops which carry out all these inter-related decision-making processes, shows the various ways of falling out of or being pushed out of the nest. Fortunately for PET owners, PET can leave a FOR-NEXT loop before it is finished, with no ill effects. This is not the case for some other computers which shall be nameless because the editor needs the advertising fees. One more thing to look for on Brand X when you go shopping! (Don't ask the salesman: test it yourself. He may not know or he may lie. Remember that he got where he is by being unsuited for other positions. The PET 2001-16 with proper keyboard and all has a machine-language monitor resident in ROM: did my salesman know that? No, he had a very nice cassette which he insisted on loading to demonstrate the monitor!)

Frills and furbelows

If you can remember the difference between stalactites and stalagmites, you will have no difficulty in keeping straight the difference between frills and furbelows: frills are the unessential but attractive arrangements along the top edge of a dress, while furbelows perform a similar function on the bottom edge. In this category I offer a number of suggestions. (One of Crudis' Compendium of Games-Writing Standards — PCW Oct' 79 – you will recall, is that one might as well *use the entire memory* as not).

1. *End of Game:* As the whole point of IMPHEX is to see the computer demonstrating what it learned in the previous games, we must be prepared to cycle back to the all-games initialization point in the program on request (line 980). Likewise we can feel justified in allowing the computer a somewhat tart response if the human elects not to continue (line 995). Note that the first-game initialization point differs from the all-games initialization point in that the former defines functions, dimensions arrays, and sets win counts to zero. I recommend that all games, not just IMPHEX, should *include an ANOTHER GAME? option.* There is nothing more lacklustre than a program which retires into READY with a flash-

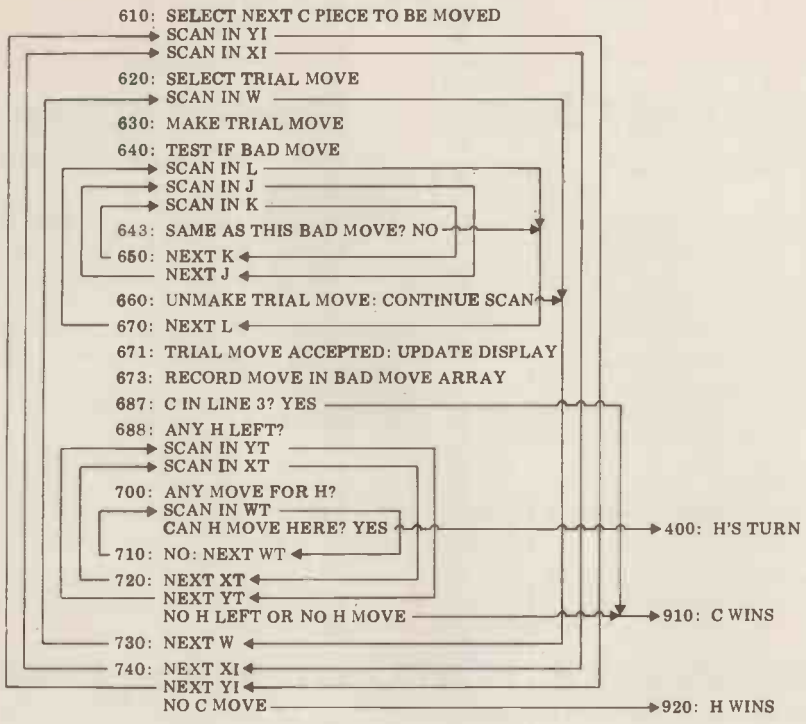

Fig. 7: Nested Loops in IMPHEX

ing cursor as soon as it has completed one tour of duty. Incidentally, note the positioning of "READY" in line 2 when the game sequence is finally ended. This avoids either an unattractive scroll-up with loss of part of the board display or two lines of readout with no space between them, the sign of a rank amateur (unless I do it for some special reason). See for yourself why the semicolon is required in line 995.

2. *Invalid and Illegal Move Response:* As it is easy to *differentiate invalid from illegal moves*, we can provide two different responses: "NOT IN THIS GAME!" for the former; "ILLEGAL MOVE" for the latter with, in both cases, erasure of the previous human input and a flicker to indicate a new message.

3. *Unsolicited Computer Responses:* One can go too far with this kind of thing, of course, but a little bit is like the right amount of salt in the soup. After each lost game, the "I HAVE LEARNED. . ." response (lines 1080-1095) is presented in varying form. If 6 games have passed before the computer wins its first one, we get "I NEEDED THAT!" (line 975). Children and clergymen are especially responsive to this sort of touch. In this category we also have the "HOW MANY TIMES DO I HAVE TO TELL YOU?" response, touched on previously, resulting from a selected number of illegal moves.

4. *Instructions:* Complete, concise *instructions should always be included* in the program, but *display should be optional* (lines 130, 140). Note that only two tests determine if I$ is "Y", "N", or neither one. *Don't mix GET and INPUT in the same program;* it invariably leads to the user bombing out by pressing READY at the wrong time. Note that PET permits a printout combined with an INPUT command as in INPUT "INSTRUCTIONS ('Y' OR 'N')"; I$. For some other computers you may have to rewrite this as PRINT "INSTRUCTIONS ('Y' OR 'N')";: INPUT I$. Always show the acceptable responses unless they are self-evident, and don't require the user to spell out "YES" and "NO". If you want to guard against someone writing "YES" when only "Y" is required, you can write IF LEFT$(I$,1)="Y" instead of IF I$="Y"

5. *Starting the Game after Reading the Instructions:* In line 1500 the command WAIT 59410,4,4 waits for the space key and no other to be depressed. This is the only exception to the rule that each input must be followed by RETURN; but the uninitiated user, I have found, sees the double-sized space key as a special function key related to and alternative to the double-sized return key, and learns to expect its use as (and restricted to) the start-of-game function. Therefore I recommend the *use of WAIT 59410,4,4 to start games* after reading instructions or *to turn pages of the instructions.* If you use it at all, you should standardize on it throughout your library.

6. *The Scoreboard:* A little attention to details here (lines 935-950) pays off in visual appeal. The use of black letters on a white background sets off the scoreboard and isolates it from other elements of the display. You will see that there is a little trick in line 950 which permits us to write the values of a variable in reverse mode without worrying about how many digits long it is or generating unwanted black spaces. Don't be so lazy as to write an isolated line of reverse characters. Since the characters all touch the top but not the bottom of the 8 x 8-dot pixel, such a technique gives a raw, unbalanced look. Instead, surround the reversed characters with a bit of a border. To see what I mean, compare these two printouts:

```
10 PRINT "[CS 2CD 16CR RE] BEFORE"
20 PRINT SPC(15)"[5CD RE SL]88888888"
30 PRINT SPC(15)"[RE SP]AFTER![SP]"
40 PRINT SPC(15)"[RE SL]""""""""""[SR]"
```

Note the minor subtlety that using reverse shifted "8" instead of shifted quotes gives the same number of scan lines above as below, making the lettering appear accurately centered.

7. *Ensuring that the Display is Up-to-Date:* Another sign of sloth, inebriety, or slovenliness is to leave no-longer-current displays on the screen. *Don't leave no-longer-current displays on the screen!* Go to a little trouble to generate suitable erasure strings such as E$, and use them to sweep off the offending prompt or no-longer-valid remark, just as you use similar strings such as EB$ to erase portions of the board.

Board graphics

Methods of designing and implementing graphics for board games were treated in a previous article, with HEX as an example. For any reader who doesn't have this issue available, I will explain that lines 1120-1260, which update the display, go to some trouble to erase a piece and then write it in the new position in rapid sequence so that the piece appears to move from one spot to the other, due to the persistence of human vision.

Choosing the next move randomly

As previously explained, the options open to the computer are to look for its next move along a programmed scan sequence or to choose randomly which piece shall be moved next. Both approaches have advantages and it is instructive to see both in operation. For my own use I have incorporated both methods into the program and you may choose to do the same.

It is not enough to call for Y=INT (2*RND(1)+2):X=INT(3*RND(1)+1). That would indeed generate Y= 3 or 2 and X= 3, 2, or 1 but it might generate the same set of ordinates twice or three times or indefinitely. What is needed is an algorithm which sets up a random sequence of the cell ordinates but allows each position to be generated only once and for each position sets up a random sequence of trial moves but allows each trial move to be generated only once. The problem is congruent to shuffling a deck of cards, and the simplest implementation I have been able to generate is as follows:

Set up an array for piece selection DIM P(3,3) and one for move selection DIM M(3). These will be used simply to record that each position and move has been tried.

750: zero the piece selection array.

755: select Y randomly from the values 2 or 3: select X randomly from the values 1, 2 or 3.

760: if this pair has been used before on this turn, try again.

765: if there is no computer piece at this point, go to 855.

770: if there is a computer piece at this point, zero the move array.

775: select a move randomly from the values 1, 2, or 3.

780: if this move has been used before, try again.

785: if this move is not legal, go to 850.

850: put a 1 in the move array to show that this move has been tried. Increment the move count, N2. If all three moves have not been generated, try again.

855: put a 1 in the piece array to show that this cell has been tried: increment the position count, N1: try again until

Continued in Programs — P.113

Cont. from page 77

all six positions have been tested.

I would be indebted to any reader who, understanding the problem, can implement the solution more efficiently.

abbreviations used in text.

[ ]: any text in brackets is to be interpreted as instructions to print cursor, clear, home, space, or reverse symbols.
CU: cursor up.
CD: cursor down.
CL: cursor left.
CR: cursor right.
CS: clear screen.
HO: home.
RE: reverse.
RO: reverse off.
SH: shift (hold shift down for next symbol outside of brackets.
SL: shift lock (hold shift down for all symbols outside of brackets until advised otherwise or end of line.
SR: shift release (cancels SL).
SP: space.
Example: "[5CU 3CR]" : print 5 cursor up symbols followed by 3 cursor rights. Note that it is not necessary to specify that these require the use of the shift key.

The program listing

If you like everything that has been offered, you are at liberty to transcribe the program exactly as follows, for non-commercial use. If you don't like it, feel free to chop and change as you see fit. Lines 750-860 contain the random scan option and there is some duplication with lines 610-740, which contain the sequential scan. They could have been interwoven but I have kept them separate to make the program easier to read and to delete the random scan if desired. If transcribing all that data is too much trouble, you can always wait until it comes on the market.

```
100 PRINT"J"SPC(16)"IMPHEX ":PRINTSPC(15)"────────
105 PRINTSPC(8)"▒A NEW VERSION OF HEXAPAWN"
110 REM:BY T.CRUDIS 5/79.
130 INPUT"▒▒INSTRUCTIONS ('Y' OR 'N')";I$:IF I$="Y" GOTO 1310
140 IF I$<>"N" GOTO 120
145 PRINT"▒
150 PRINT"▒I CAN BE PROGRAMMED TO SELECT MY MOVES
155 PRINT"▒IN A RANDOM OR A SEQUENTIAL MANNER. IF
160 PRINT"▒YOU CHOOSE 'SEQUENTIAL' MY RESPONSE TO A
165 PRINT"GIVEN SITUATION WILL ALWAYS BE THE SAME.
170 PRINT"WHICH DO YOU WISH TO HAVE ME USE:
175 INPUT"▒('R' OR 'S')";M$:IF M$<>"R" AND M$<>"S" GOTO 175
210 C=-1:H=1:CW=0:HW=0:DIMBD(4,3),BM(3,3,15),P(3,3)
220 D$="▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":E$="                              "
230 BP$="▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":BD$="▒▒▒▒▒▒▒":BL$="▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒
240 C$="  ▒ ▒▒▒▒▒  ▒ ▒▒▒▒▒▒ ▒▒▒▒▒▒▒ ▒▒▒▒▒▒":H$="▒"+C$+"▒
330 :FOR Y=1 TO 3:FOR X=1 TO 3:BD(X,1)=H:BD(X,2)=0:BD(X,3)=C:NEXTX,Y
335 TU=H:F1=0:F2=0
340 GOSUB 1010
380 PRINT BP$C$C$C$BD$BD$BL$H$H$H$
400 PRINTD$E$D$;
410 PRINT"YOUR MOVE";:INPUT HI,HF
420 IF HI<1 OR HI>9 OR HF<1 OR HF>9 THENPRINTD$"NOT IN THIS GAME!   ";:GOTO 410
425 DEF FNC(X)=X+10-(X>3)*7-(X>6)*7
430 YI=INT(FNC(HI)/10):XI=FNC(HI)-YI*10:YF=INT(FNC(HF)/10):XF=FNC(HF)-YF*10
440 IF BD(XI,YI)<>1 OR ABS(XF-XI)>1 OR YF-YI<>1 GOTO 500
450 IF (XI=XF AND BD(XF,YF)<>0) OR (ABS(XF-XI)=1 AND BD(XF,YF)<>-1) GOTO 500
455 PRINT"▒▒▒▒":FOR J=0 TO 7:PRINT"                              ":NEXT J
470 GOSUB 1120.
480 IF YF=3 GOTO 920
485 BD(XI,YI)=0:BD(XF,YF)=1
490 GOTO 600
500 PRINT D$E$D$"ILLEGAL MOVE!   ";:F1=F1+1:IF F1<>4 THEN 410
510 PRINT"▒▒▒▒▒HOW MANY TIMES":PRINT"▒DO I HAVE TO ":PRINT"▒TELL YOU?
515 PRINT" ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒ ":PRINT"▒ILLEGAL MOVE!▒"
520 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":GOTO 400
600 PRINT D$E$:TU=C:FOR D=1 TO 500:NEXT D
605 IF M$="R" GOTO 750
610 :FOR YI=3 TO 2 STEP-1:FOR XI=1 TO 3:IF BD(XI,YI)<>-1 THEN 740
620 ::FOR W=1 TO 3:IF BD(XI+2-W,YI-1)<>ABS(2-W) THEN 730
630 ::SI=BD(XI,YI):SF=BD(XI+2-W,YI-1):BD(XI,YI)=0:BD(XI+2-W,YI-1)=-1
640 :::FOR L=0 TO HW:FOR J=1 TO 3:FOR K=1 TO 3
643 :::::IF BD(J,K)<>BM(J,K,L) THEN 670
650 :::::NEXT K,J
660 :::BD(XI,YI)=SI:BD(XI+2-W,YI-1)=SF:GOTO 730
670 :::NEXT L
671 ::GOSUB 1120
673 :::FOR Y=1 TO 3:FOR X=1 TO 3:BM(X,Y,HW)=BD(X,Y):NEXT X,Y
687 ::IF YI=2 THEN 910
688 :::FOR YT=1 TO 3:FOR XT=1 TO 3:IF BD(XT,YT)<>1 THEN 720
700 :::::FOR WT=-1 TO 1:IF BD(XT+WT,YT+1)=-ABS(WT) THEN TU=H:GOTO 400
710 :::::: NEXT WT
720 ::::NEXT XT,YT:GOTO910
730 ::NEXT W
740 :NEXT XI,YI:GOTO 920
745 END
750 FOR Y=2 TO 3:FOR X=1 TO 3:P(X,Y)=0:NEXT X,Y:N1=0
755 N2=0:YI=INT(2*RND(1)+2):XI=INT(3*RND(1)+1)
760 IF P(XI,YI)=1 GOTO 755
765 IF BD(XI,YI)<>-1 GOTO 855
770 FOR W=1 TO 3:M(W)=0:NEXT W
775 W=INT(3*RND(1)+1)
780 IF M(W)=1 GOTO 775
785 IF BD(XI+2-W,YI-1)<>ABS(2-W) GOTO 850
790 SI=BD(XI,YI):SF=BD(XI+2-W,YI-1):BD(XI,YI)=0:BD(XI+2-W,YI-1)=-1
795 :FOR L=0 TO HW:FOR J=1 TO 3:FOR K=1 TO 3
800 :::IF BD(J,K)<>BM(J,K,L) THEN 815
805 ::NEXT K,J
810 :BD(XI,YI)=SI:BD(XI+2-W,YI-1)=SF:GOTO 850
815 :NEXT L:GOSUB 1120
820 FOR Y=1 TO 3:FOR X=1 TO 3:BM(X,Y,HW)=BD(X,Y):NEXT X,Y
825 IF YI=2 THEN 910
830 :FOR YT=1 TO 3:FOR XT=1 TO 3:IF BD(XT,YT)<>1 THEN 840
835 :::FOR WT=-1 TO 1:IF BD(XT+WT,YT+1)=-ABS(WT) THEN TU=H:GOTO 400
836 ::: NEXT WT
840 ::NEXT XT,YT:GOTO 910
850 M(W)=1:N2=N2+1:IF N2<3 GOTO 775
855 P(XI,YI)=1:N1=N1+1:IF N1<6 GOTO755
860 GOTO 920
910 PRINTD$"I WIN!          ":CW=CW+1:F3=1:GOTO 930
920 PRINT D$+"YOU WIN!          ":HW=HW+1:F3=0
930 FOR D=1 TO 1000:NEXT
935 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":PRINT"▒    SCORE      ":PRINT"▒──────────────
940 PRINT"▒ COMPUTER:      ":PRINT"▒                    ":PRINT"▒    HUMAN:      "
950 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒":PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"CW:PRINT"▒▒▒▒▒▒▒▒▒▒▒▒"HW
```

```
970 IF CW>HW THEN F2=1
975 IFF2=0AND CW=1 AND HW>5 THEN FOR D=1TO500 NEXT PRINT"I NEEDED THAT!" F2=1
980 PRINT D$"ANOTHER GAME ('Y' OR 'N')"; INPUT G$ IF G$="Y"GOTO 240
990 IF G$<>"N"GOTO 980
995 PRINT D$E$D$"SOUR GRAPES!   SO TURN ME OFF!"; END
1010 PRINT"J"SPC(17)"
1020 FOR J=1 TO 3 PRINT SPC(17)"
1030 FOR K=1 TO 4 PRINT SPC(17)" |     ||     ||     |" NEXT K
1040 PRINT SPC(17)"" NEXT J
1050 PRINT SPC(17)"
1080 IF F3=1 GOTO 1095
1090 IF HW>0 THEN PRINT"I HAVE LEARNED
1091 IF HW=1 THEN PRINT"A BAD MOVE.
1092 IF HW=2 THEN PRINT "ANOTHER ONE!
1093 IF HW>2 THEN PRINT""HW"BAD MOVES.
1094 IF HW>5 THEN PRINT"THE HARD WAY!
1095 RETURN
1120 BE$="
1130 BU$="TTTTT" H$=""+C$+"
1140 IF TU=H THEN W=XI-XF+2
1150 IF TU=C THEN BU$=BD$ H$=C$
1160 PRINT BP$; FOR YD=3 TO 1 STEP-1 FOR XD=1 TO 3
1165   IF XD<>XI OR YD<>YI THEN PRINT""; GOTO 1250
1170   PRINT BE$BU$MID$(BL$,W,W↑2+W)H$ GOTO 1260
1250   NEXT XD PRINTBD$BL$; NEXTYD
1260 RETURN
1310 PRINT"HEXAPAWN IS PLAYED WITH CHESS PAWNS ON
1320 PRINT"A 3 X 3 BOARD. YOUR PAWNS ARE WHITE.
1330 PRINT"TO ENTER A MOVE, TYPE THE NUMBER OF THE
1340 PRINT"SQUARE YOU WISH TO MOVE FROM, A COMMA,
1350 PRINT"AND THE NUMBER OF THE SQUARE YOU WISH TO
1360 PRINT"MOVE TO, FOR EXAMPLE '2,5'. THE SQUARES
1370 PRINT"ARE NUMBERED LIKE THE KEYBOARD " PRINTSPC(16)"      "
1375 PRINTSPC(16)"7 8 9" PRINTSPC(16)"
1380 PRINTSPC(16)"4 5 6" PRINTSPC(16)"
1385 PRINTSPC(16)"1 2 3" PRINTSPC(16)"
1390 PRINTSPC(7)"PRESS 'SPACE' TO TURN PAGE." WAIT 59410,4,4
1400 PRINT"TO WIN, MOVE ONE OF YOUR PIECES TO THE
1410 PRINT"OPPONENT'S SIDE OF THE BOARD FIRST,
1420 PRINT"CAPTURE ALL OF THE OPPONENT'S PIECES, OR
1430 PRINT"BLOCK HIM FROM MAKING ANY MOVE.
1440 PRINT"THE COMPUTER STARTS A SERIES OF GAMES
1450 PRINT"KNOWING ONLY HOW TO MOVE AND HOW TO
1460 PRINT"RECOGNIZE A WIN. IT HAS NO INITIAL
1470 PRINT"STRATEGY, BUT IT KEEPS A RECORD OF ALL
1480 PRINT"BAD MOVES AND IMPROVES ITS STRATEGY
1490 PRINT"UNTIL IT BECOMES INVINCIBLE.
1500 PRINTSPC(7)"PRESS SPACE TO START GAME." WAIT59410,4,4 PRINT"J" GOTO150
```

Pascal, cont. from p. 96

```
    REPEAT
        READLN(INFILE) ;
        I := 1 ;
        REPEAT
            READ(INFILE, BUFFER[I]) ;
            I := I + 1
        UNTIL EOLN(INFILE)
    UNTIL EOF(INFILE) OR (LENGTH(BUFFER)>0) ;
    FOR J := I TO MAXLINE DO BUFFER[J] := ' '
END ; (*FREADLN*)

PROCEDURE FWRITELN(VAR OUTFILE : TEXT ; BUFFER : STRING) ;
(*WRITELN BUFFER TO OUTFILE*)
VAR I,J : 1..MAXLINE ;
BEGIN
    FOR I := 1 TO LENGTH(BUFFER) DO WRITE(OUTFILE,BUFFER[I]) ;
    WRITELN(OUTFILE)
END ; (*FWRITELN *)

PROCEDURE FLUSH (VAR BUFFER : STRING) ;
(*OUTPUTS SPACING BLANK LINES, FLUSHES BUFFER AND THEN FILLS WITH SPACES*)
VAR I : INTEGER ;
PROCEDURE LINESPACING ;
VAR I : 2 .. 10 ;
BEGIN
    IF USERFORMAT.SPACING>1
    THEN
        FOR I := 2 TO USERFORMAT.SPACING DO
        BEGIN
            WRITELN (OUTFILE) ;
            CURRENTLINE := CURRENTLINE + 1
        END
END ; (*LINESPACING*)
BEGIN
    IF NOT EMPTYOUTBUFFER
    THEN
    BEGIN
        FWRITELN (OUTFILE,BUFFER) ;
        CURRENTLINE := CURRENTLINE + 1 ;
        LINESPACING
    END ;
    FOR I := 1 TO MAXLINE DO BUFFER [I] := ' ' ;
    EMPTYOUTBUFFER := TRUE
END ; (* FLUSH*)

FUNCTION COMMAND : BOOLEAN ;
(* TRUE IF IN BUFFER IS A COMMAND AND FALSE OTHERWISE *)
BEGIN
    COMMAND := INBUFFER[1] = '.'
END ; (* COMMAND *)

PROCEDURE OBEY (VAR INBUFFER,OUTBUFFER : STRING ; VAR EMPTYINBUFFER,
                    EMPTYOUTBUFFER : BOOLEAN ; VAR FORM : FORMAT ) ;
(* CARRIES OUT FORMATTING COMMANDS   *)

VAR VALID : SET OF CHAR ;
    I : INTEGER ;

FUNCTION PARAMETERS (BUFFER : STRING ) : BOOLEAN ;
BEGIN
    PARAMETERS := LENGTH(BUFFER) > 2
END ; (* PARAMETERS *)

FUNCTION NEXTPARAMETER (START : INTEGER) : INTEGER ;
VAR I,NUM : INTEGER ;
BEGIN
    I := START ;
    WHILE INBUFFER[I] = ' ' DO I := I+1 ;
```