FANTASY GAMER

3 BIG PROGRAMS FOR THE ADAM*

2 ADVENTURES WITH GRAPHICS
PLUS
ADVENTURE CREATOR

Design Your Own Adventure Games!

CONTENTS

INSERT & PULL RESET

Insert + Pull Reset

INTRODUCTION TO ADVENTURING

Adventure games are like stories in which the player is the hero, and the outcome of the story depends on the player's wisdom and actions. Adventure games can be grouped into two general classes--puzzle adventures and fantasy adventures (or interactive fiction). The game Bomb Squad is a puzzle adventure; there is one set of solutions, and your success at finding the solution will depend on your logic and deductive reasoning. The Visitor is more of a story with variable outcomes and more description and character development. Your success in a fantasy adventure may depend more on your understanding of an opponent's personality than on deductive reasoning.

To play the game, you read information about your position and status on the screen, and sometimes you must study pictures presented graphically for clues. The game will ask you what you want to do, and you will enter two-word commands (a verb and a noun) like "kill dragon" or "charm princess" or "go south".

<u>Mapping the game</u>.

It is essential to keep track of the world you are moving around it by drawing a map as you go, noting objects in various locations, in case you should need them later. Adventure games consist of various specific locations in which action takes place, so start your map with a rectangle representing your present location. Add rectangles ("rooms") as you go.

Develop your own shorthand to keep track of how you got from place to place, what objects are in each location, and which directions are not passable in each room.

<u>Inventory management</u>.

In most games (including Bomb Squad and The Visitor) you can carry only a limited number of things--partly depending on your physical condition. You never know what you might need for any one situation, so at times you will have to drop things in order to pick up others. Keep track of where these things are, in case you need them again.

Using FANTASY GAMER

This manual and tape cover three separate programs. It would probably be a good idea to play the two games a few times before you study the material called Adventure Creator, because once you start analyzing the program in detail, most of the solutions to the games will become obvious and spoil all your fun. Also, if your goal is to write your own adventure games, you should have some feel for how the player perceives these games without having all the clues ahead of time.

Adventure Creator consists of a "framework program", which is not a game in its own right. It is designed to let you fill in the details of your own game. The written instructions for Adventure Creator are probably just as important as the program on the tape, since just having the program won't do you much good, without the tutorial material that explains it.

ADVENTURE CREATOR

Getting started

When you write an adventure game, you will be creating a fantasy world in which you make up all the events and rules. The world you create can be as fantastic or as realistic as you want to make it. Many adventure games use magic, and will be up to you decide how much magic is permitted.

First, decide on a theme for your story. You might choose a particular time in history, or even prehistory--and have your player try to prevent the extinction of the dinosaur. Haunted mansions are big favorites, and some adventure games teach a little history by being factually accurate in details, while the player tries to do something like help Julius Caesar avoid assasination. We're not sure what the implications would be if the player succeeds.

Once you have a general theme, work out the actual story, which should be built around an objective--defuse the bombs, find the alien's mother ship, retrieve the Ring from the Lord of Darkness, or whatever. The course of your story will always be built around this ultimate objective.

Then sketch out your world in a rough map. This stage will take some imagination and many false starts, as you think about your story line in relation to the place where it occurs.

Drawing your "world"

The first step in making your story into an adventure game is to transfer your sketched in "world" to a square grid, like the one in the next figure. Of course you can make your world any size, but limitations of the computer's memory put some limits on you, especially if you want to use some memory for graphics and have lots of options in the possible actions in the story.

The grid used in this example is 6x6, so we can have 36 locations or "rooms" in our game. Make your grid as large as possible, because you will want to write in lots of notes, treasures and object names. You will refer to this diagram many times as you plan story action and keep track of which objects are in which locations.

Number each location, starting in the top left corner, as in the sample grid (which, obviously, is from Bomb Squad). We find it clearest to start numbering at 1, rather than 0 (as some games do). Use a pencil to lightly mark in brief room names for each location and the exits possible from each room. As your grid drawing develops, you will be able to darken in lines for exterior walls and to set off such things as cellars and attics.

As you mark the entrances and exits in each room, use the four points of the compass as direction markers. You can use little arrows, as in the example grid. If you add the interest of going up ladders and down stairs, etc., plan routes carefully. Obviously, even going "up" is going to require the player to go north, south, west, or east. It is a good idea to get used to always naming the directions in this order, since we will be using numbers to indicate directions in the game (north=1 south=2 west=3 east=4). Some of your routes may be one-way (the door locks behind you; a tunnel collapses after you go through it etc.).

Make a list of your rooms with all legal exits from the room, like this:

1. wine cellar	S
2. TV room	SE
3. patio	WE

etc. for later use.

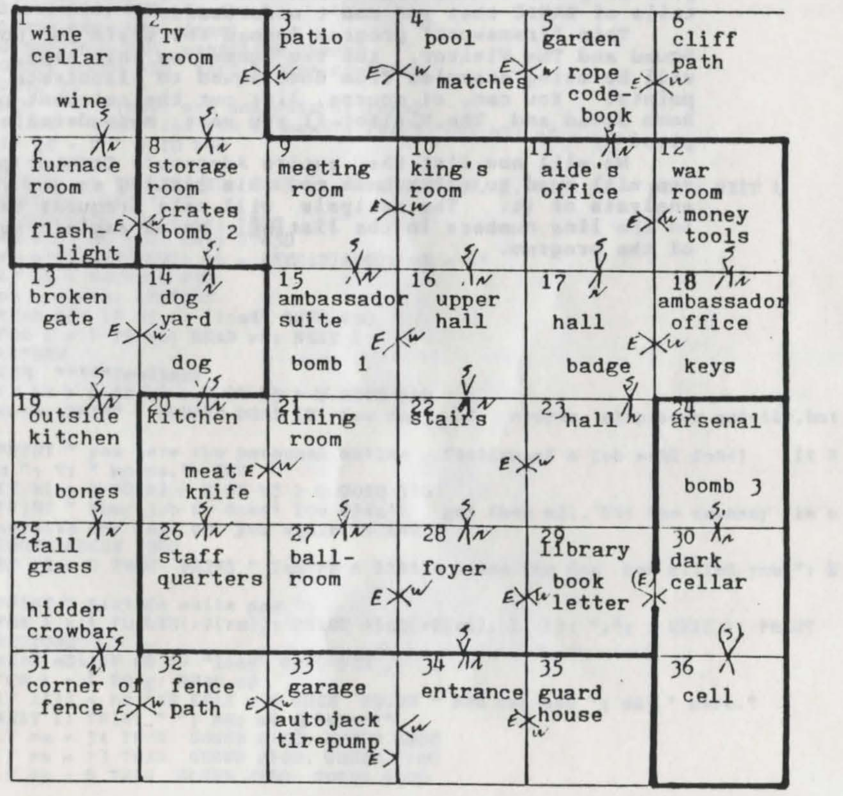
Building the story

Now that your adventure world is mapped out, it's time to get serious about our story. You need to plan what the player can do in each room, and what objects will be needed to do it. In the process of doing this, you will be building up a list of verbs to cover the actions needed and a list of "gettable" objects (things the player can pick up) and "non-gettable" objects that will be in the room to stay. Keep a separate list of verbs and objects, and list the gettable objects ahead of the non-gettable objects in your object list. Eventually, you will want an object list that looks like this:

Object number	Object	Location
1.	keys	21
2.	amulet	31
3.	scrolls	1
etc.		

As you place (and perhaps hide) your props around the environment, you will be thinking about what the player will do with them in each location. If you do hide an object (like the crowbar in the grass in Bomb Squad), you will have to keep track of what is visible and what isn't. We will show you how to do this in our analysis of of the Adventure Creator program. It will help a lot if you write in each object on your grid diagram.

As you plan your game, keep your player in mind. The actions, puzzles, and events should make some kind of consistent sense. Otherwise the game will be impossibly frustrating to play. Of course it's OK to use magic, if your world includes magic, but be sure the rules can be figured out. Random magic is maddening. Also, try to anticipate the verbs and nouns your player might use for various situations.



ANALYZING THE ADVENTURE CREATOR PROGRAM

Once you have your story planned, you are ready to start programming. The purpose of Adventure Creator is to give you a "framework program" in which the hard parts of the program are already done. Your main job will be to provide the details of the "world" of your adventure, and perhaps to make up your own graphics scenes. This will still be a complicated job, but it should be a fascinating process, in which you will learn a great deal.

The next section gives you a complete listing of the "framework" program, and subsequent sections analyze the program in detail. This is probably the best way to learn programming. We are assuming that you already know something about BASIC programming, so you might have to study your computer manual or some other book, if there are details of BASIC that you don't understand.

This "framework" program formed the basis for both Bomb Squad and The Visitor, the two games on this tape, and we will be using examples from Bomb Squad to illustrate various points. You can, of course, list out the relevant parts of Bomb Squad and The Visitor if you want more detailed examples.

We will now list the entire Adventure Creator program; you will need to refer back to this listing as you read the analysis of it. The analysis will make frequent reference to the line numbers in the listing, as we explain each step of the program.

Program listing

```
]
  1 LOMEM :29650
  50 w = 47: g = 18: rm = 34: b1 = 1: b2 = 1: b3 = 1: t1 = 30: t2 = 60: t3 = 90
: dr$ = "1": tl = 4: tc = 1
  60 GOSUB 19900: REM  set scene
  65 GOSUB 8500: GOSUB 2160: REM     initialize
  70 HGR: CALL sr: TEXT: REM  clear out sprites
  90 GOSUB 500: REM  feedback
 100 GOSUB 160: REM  input
 110 GOSUB 700: REM  condition checks
 120 GOSUB 2000: REM  verb action routines
 130 GOTO 90
 159 REM  *****input and analysis of input
 160 PRINT " What will you do now?": INPUT " "; q$: q$ = q$+" "
 170 IF q$ = " " GOTO 160
 240 pk = 27720
 250 FOR 1 = 1 TO LEN(q$): POKE pk, ASC(MID$(q$, 1, 1)): pk = pk+1: NEXT 1
 260 POKE pk, ASC("^")
 280 GOSUB 8630: CALL sr: TEXT
 410 vb = 0: ob = 0: CALL 27430
 450 vb = PEEK(27409): ob = PEEK(27410): w$ = ""
 455 IF ob < vn GOTO 490
 458 ob = ob-vn: RESTORE
 460 READ a$: IF a$ <> "load" GOTO 460
 470 FOR 1 = 1 TO ob: READ w$: NEXT 1
 490 RETURN
 499 REM  *****feedback
 500 IF b1 > 0 OR b2 > 0 OR b3 > 0 GOTO 510
 505 HOME: PRINT " YOU'VE DONE IT! You must, of   course, slip away quietly,but
"
 507 PRINT " you have the personal satis-   faction of a job well done!    It t
ook you "; t; " moves.": END
 510 IF b1 = 1 OR b2 = 1 OR b3 > 0 GOTO 516
 512 PRINT " Your job is done! You didn't   get them all, but the embassy  is s
till there.We may call on  you again.": END
 516 HOME: GOSUB 7900
 518 IF tl = 0 THEN  PRINT " You're a little nutso.The dog  has killed you.": E
ND
 520 PRINT " Visible exits are ";
 530 FOR 1 = 1 TO LEN(r$(rm)): PRINT MID$(r$(rm), 1, 1); ","; : NEXT 1: PRINT
 535 RESTORE
 537 READ a$: IF a$ <> "load" GOTO 537
 540 FOR 1 = 1 TO g: READ o$
 550 IF 1(1) = rm AND f(1) = 0 THEN  PRINT " You can see "; o$; " here."
 560 NEXT 1: PRINT " "; m$: m$ = "What?"
 600 IF rm = 34 THEN  GOSUB 2160: GOSUB 6200
 610 IF rm = 13 THEN  GOSUB 2160: GOSUB 6100
 620 IF rm = 8 THEN  GOSUB 2160: GOSUB 6500
 630 IF rm = 29 THEN  GOSUB 2160: GOSUB 6400
 690 RETURN
 699 REM  *****condition checks
 700 IF ob = 0 THEN  m$ = " That's silly."
 730 IF vb = 0 OR vb > vn OR (ob > 0 AND ob < vn) OR w$ = "" THEN  m$ = " You
an't '"+q$+"'."
 740 IF vb < vn AND ob > 0 AND ob <= g AND c(ob) = 0 THEN  m$ = "You don't hav
'"+w$+"'."
 825 t = t+1
 840 IF t > t1 THEN  GOSUB 5100
 850 IF t > t2 THEN  GOSUB 5200
 860 IF t > t3 THEN  GOSUB 5300
 900 vj = vb: IF vb > 2 AND vb < 11 THEN  vj = 3
 910 IF vb > 10 AND vb < 18 THEN  vj = 4
 915 IF vb = 18 THEN  vj = 5
 920 IF vb > 18 AND vb < 25 THEN  vj = 6
 925 IF vb = 25 THEN  vj = 7
 930 IF vb = 26 THEN  vj = 8
 935 IF vb > 26 AND vb < 30 THEN  vj = 9
 940 IF vb = 30 OR vb = 31 THEN  vj = 10
 945 IF vb > 31 AND vb < 37 THEN  vj = 11
 950 IF vb = 37 THEN  vj = 12
 955 IF vb = 38 THEN  vj = 13
 960 IF vb > 38 AND vb < 44 THEN  vj = 14
 965 IF vb = 44 THEN  vj = 15
 970 IF vb > 44 AND vb < 49 THEN  vj = 16
 975 IF vb = 49 OR vb = 50 THEN  vj = 17
 980 IF vb > 50 AND vb < 58 THEN  vj = 18
 985 IF vb > 57 AND vb < 66 THEN  vj = 19
 990 IF vb = 66 OR vb = 67 THEN  vj = 20
 992 IF vb = 68 OR vb = 69 THEN  vj = 21
 994 IF vb = 70 OR vb = 71 THEN  vj = 22
 996 IF vb = 72 THEN  vj = 23
 997 IF vb = 73 THEN  vj = 24
 999 RETURN
 1999 REM  *****verb action routines
 2000 IF vj = 0 THEN  RETURN
 2005 IF tc > tl AND vj <> 11 AND vj <> 2 THEN  m$ = " You must drop somethin
: RETURN
 2010 ON vj GOTO 2100, 2200, 2300, 2400, 2500, 2600, 2700, 2800, 2900, 3000, 3
0, 3200, 3300, 3400, 3500
 2020 ON vj-15 GOTO 3600, 3700, 3800, 3900, 4000, 4100, 4200, 4600, 4700
 2100 RESTORE: PRINT " Words I know:": hp = 0: m$ = ""
 2110 READ a$: IF a$ <> "help" GOTO 2110
 2120 READ a$: IF a$ = "zz" GOTO 2160
 2130 PRINT a$; ","; : wc = wc+1: IF wc > 3 THEN  wc = 0: PRINT
 2140 np = hp+1: IF hp = 64 THEN  PRINT: hp = 0: GOSUB 2160
```

```
 2150 GOTO 2120
 2160 PRINT: INPUT " Push return to continue."; a$: RETURN
 2200 PRINT " You are carrying:"; : RESTORE
 2210 READ a$: IF a$ <> "load" GOTO 2210
 2220 FOR l = 1 TO g: READ o$: IF c(l) = 1 THEN  PRINT o$; ",";
 2230 NEXT l: m$ = "": GOTO 2160
 2300 d = 0: IF ob = 0 THEN  d = vb-3
 2303 IF ob = 19 THEN  d = 1
 2306 IF ob = 20 THEN  d = 2
 2309 IF ob = 21 THEN  d = 3
 2312 IF ob = 22 THEN  d = 4
 2344 IF rm = 34 AND (d = 1 OR d = 4) AND f(45) = 0 THEN  m$ = " The guard won't
 let you pass.": RETURN
 2370 f(19) = 0: rl = LEN(r$(rm))
 2372 FOR l = 1 TO rl
 2374 u$ = MID$(r$(rm), l, 1)
 2376 IF (u$ = "N" AND d = 1 AND f(19) = 0) THEN  rm = rm-6: f(19) = 1
 2378 IF (u$ = "S" AND d = 2 AND f(19) = 0) THEN  rm = rm+6: f(19) = 1
 2380 IF (u$ = "W" AND d = 3 AND f(19) = 0) THEN  rm = rm-1: f(19) = 1
 2382 IF (u$ = "E" AND d = 4 AND f(19) = 0) THEN  rm = rm+1: f(19) = 1
 2384 NEXT l
 2386 m$ = " OK."
 2388 IF f(19) = 0 THEN  m$ = " Can't go that way!"
 2390 IF d < 1 THEN  m$ = " Go where?"
 2399 RETURN
 2400 IF ob > g THEN  m$ = " You can't get "+w$+".": RETURN
 2420 IF l(ob) <> rm THEN  m$ = " It isn't here."
 2430 IF f(ob) <> 0 THEN  m$ = " What "+w$+"?"
 2440 IF c(ob) = 1 THEN  m$ = " You already have it."
 2452 IF tc >= tl THEN  m$ = " You can't carry more than "+STR$(tl)+"    things.
": RETURN
 2460 IF ob > 0 AND l(ob) = rm AND f(ob) = 0 THEN  c(ob) = 1: l(ob) = 38: m$ = "
 OK. You have the "+w$+".": tc = tc+1
 2470 IF q$ = "take pictures" THEN  m$ = "using what?"
 2499 RETURN
 2500 IF rm = 30 AND c(1) = 0 AND c(2) = 0 AND c(3) = 0 AND ob = 47 THEN  m$ = "
 You don't have anything to     open it.": RETURN
 2530 IF rm = 30 THEN  m$ = " The door is open. You had the  tools for the job."
: r$(30) = "NSW"
 2590 RETURN
 2600 IF ob <= w AND (l(ob) = rm OR c(ob) = 1) THEN  m$ = " Nothing special--jus
t a "+w$+"."
 2630 IF rm = 19 AND ob = 36 THEN  m$ = " That's disgusting!"
 2640 IF rm = 25 AND ob = 39 AND f(2) = 0 THEN  m$ = " You've discovered a crowb
ar in the grass!": f(2) = 0
 2699 RETURN
 2700 IF ob = 12 AND c(12) = 1 THEN  m$ = " It says,'Moveable furniture is decep
tive.'"
 2730 IF (ob = 11 OR ob = 12) AND c(11) = 0 AND c(12) = 0 THEN  m$ = " How can y
ou read what you're   not holding?"
 2799 RETURN
 2800 IF ob = 11 AND c(15) = 1 THEN  m$ = " It says,'The bookcase moves.   The a
mbassador will die'"
 2820 IF c(15) = 0 THEN  m$ = " You don't have the codebook."
 2900 IF b1 = 1 AND rm = 15 AND (c(6) = 1 OR c(17) = 1) THEN  m$ = " Good work.
The ambassador is   safe for now.": b1 = 0
 2940 IF c(6) = 0 AND c(17) = 0 THEN  m$ = " You can't defuse anything with no k
nife or tools."
 2950 IF rm = 8 AND f(26) = 0 THEN  m$ = " A crate's in the way."
 2999 RETURN
 3000 IF rm = 30 AND (c(3) = 1 OR c(2) = 1) THEN  m$ = " The cell is open.": rS(
30) = "NSW": f(46) = 1
 3010 IF c(3) = 0 AND c(6) = 0 AND rm = 30 THEN  m$ = "You have no keys or crowb
ar."
 3099 RETURN
 3100 IF c(ob) = 0 THEN  m$ = " You're not carrying it."
 3110 IF c(ob) = 1 THEN  c(ob) = 0: l(ob) = rm: m$ = " Done.": tc = tc-1
 3199 RETURN
 3200 IF ob = 5 AND c(5) = 1 THEN  m$ = " It's lit.": f(5) = 1
 3210 IF ob = 5 AND c(5) = 0 THEN  m$ = " You don't have the flashlight."
 3299 RETURN
 3300 IF ob = 5 AND c(5) = 1 THEN  m$ = " It's turned off."
 3310 IF ob = 5 AND c(5) = 0 THEN  m$ = " You don't have the flashlight."
 3399 RETURN
 3400 IF rm = 15 AND ob = 35 THEN  m$ = " What good did that do?She's   unconsc
ous now.": f(35) = 1
 3499 RETURN
 3500 IF rm = 8 AND (ob = 1 OR ob = 2) THEN  m$ = " Great.The crate moves. Ther
's the bomb!": f(26) = 1
 3510 IF c(ob) = 0 THEN  m$ = " You don't have it."
 3599 RETURN
 3600 IF rm = 8 AND ob = 26 AND f(26) = 0 THEN  m$ = " It's extremely heavy.Wha
will you use to move it?"
 3605 IF rm = 29 AND ob = 41 AND f(ob) = 1 THEN  m$ = " It's already moved."
 3610 IF rm = 29 AND ob = 41 AND f(ob) = 0 THEN  m$ = " IT MOVES! Stairs lead d
wn!": f(41) = 1: r$(29) = "WE"
 3699 RETURN
 3700 IF ob = 9 AND c(9) = 1 THEN  m$ = " You doze off for 20 precious   minute
!": t = t+10
 3799 RETURN
 3800 IF ob = 44 AND (b1 = 1 AND rm = 15) OR (b2 = 1 AND rm = 8) OR (b3 = 1 AND
rm = 24) THEN  mS = " Don't do that!": GOTO 6000
 3810 IF rm = 30 AND ob = 46 THEN  m$ = " I's so strong. What will you use?"
 3899 RETURN
```

```
 3900 IF rm = 30 AND f(46) = 1 AND ob = 43 GOTO 3902
 3910 IF rm = 15 AND ob = 35 THEN  m$ = " She is charmed. As you inspect the roo
m,you see a bomb"
 3999 RETURN
 4000 IF rm = 14 AND ob = 34 AND (c(4) = 0 OR c(14) = 0) THEN  m$ = " You don't
have anything he     wants."
 4010 IF rm = 14 AND ob = 34 AND (c(4) = 1 OR c(14) = 1) THEN  m$ = " Clever. He
seems to like you.": f(34) = 1: c(4) = 0: c(14) = 0
 4099 RETURN
 4100 IF f(20) = 1 AND rm = 34 AND f(45) = 0 THEN  m$ = " Devious but effective.
The com- promising pictures got him.": f(45) = 1
 4110 IF f(20) = 0 AND rm = 34 AND ob = 45 AND f(45) = 0 THEN  m$ = " What will
you use?"
 4199 RETURN
 4200 IF rm = 11 AND ob = 18 AND c(18) = 1 THEN  mS = " He's fooled and lets you
pass": f(30) = 1
 4299 RETURN
 4600 INPUT " Tape or disk ready?(y/n)"; a$: IF a$ <> "y" THEN  RETURN
 4610 PRINT CHR$(4); "open bombgame,d"; dr$
 4620 PRINT CHRS(4); "write bombgame"
 4630 PRINT rm: PRINT b1: PRINT b2: PRINT b3: PRINT t1: PRINT t2: PRINT t3: PRIN
T tl: PRINT tc
 4640 FOR l = 1 TO 36
 4650 PRINT r$(l)
 4660 NEXT l
 4670 FOR l = 1 TO w
 4675 PRINT f(l)
 4680 NEXT l
 4685 FOR l = 1 TO g
 4688 PRINT l(l): PRINT c(l)
 4690 NEXT l
 4695 PRINT CHR$(4); "close bombgame,d"; dr$
 4699 RETURN
 4700 INPUT " Tape or disk ready? (y/n)?"; a$: IF a$ <> "y" THEN  RETURN
 4710 PRINT CHR$(4); "open bombgame,d"; dr$
 4720 PRINT CHRS(4); "read bombgame"
 4730 INPUT rm: INPUT b1: INPUT b2: INPUT b3: INPUT t1: INPUT t2: INPUT t3: INPU
T tl: INPUT tc
 4740 FOR l = 1 TO 36
 4750 INPUT r$(l)
 4760 NEXT l
 4770 FOR l = 1 TO w
 4775 INPUT f(l)
 4780 NEXT l
 4785 FOR l = 1 TO g
 4788 INPUT l(l): INPUT c(l)
 4790 NEXT l
```

```
4795 PRINT CHR$(4); "close bombgame,d"; dr$
4799 RETURN
5100 RETURN: REM  dummy unusual action routine
5200 RETURN: REM  dummy routine
5300 RETURN: REM  dummy routine
5999 REM  *****graphics
6000 HPLOT 0, 12 TO 80, 40 TO 79, 90 TO 0, 158: HPLOT 80, 90 TO 200, 90 TO 199,
39 TO 80, 40
6010 HPLOT 200, 40 TO 250, 15: HPLOT 200, 90 TO 250, 150: RETURN
6030 HPLOT x, y TO x+30, y TO x+29, y+20 TO x-1, y+19 TO x, y: HPLOT x+15, y TO
x+15, y+20: HPLOT x, y+10 TO x+30, y+10: RETURN
6040 HPLOT 130, 90 TO 130, 60 TO 146, 61 TO 145, 90: RETURN
6050 HPLOT x, y TO x+25, y-5 TO x+24, y+30 TO x-1, y+20 TO x, y: HPLOT x, y+11
TO x+25, y+11: HPLOT x+11, y-2 TO x+11, y+23
6055 RETURN
6100 RETURN: REM  dummy graphics routine
6200 HGR
6203 HCOLOR  = 12: x = 8: y = 22: GOSUB 6030: x = 60: y = 22: GOSUB 6030: x = 1
75: y = 22: GOSUB 6030
6205 x = 60: y = 3: GOSUB 6030
6210 HCOLOR  = 14: HPLOT 0, 70 TO 255, 70: HPLOT 120, 70 TO 120, 45 TO 143, 46
TO 142, 70: HPLOT 131, 46 TO 131, 70
6220 HPLOT 70, 158 TO 120, 70: HPLOT 180, 158 TO 142, 70: HPLOT 48, 158 TO 35,
120
6230 HCOLOR  = 2: HPLOT 0, 80 TO 12, 77 TO 35, 85 TO 34, 120 TO 27, 119 TO 28,
93 TO 0, 94
6250 POKE sa, 95: POKE sa+1, 160: POKE sa+3, 14: POKE sa+16, 79: POKE sa+17, 1
0: POKE sa+60, 79: POKE sa+61, 160
6255 POKE sa+64, 79: POKE sa+65, 160: POKE sa+52, 95: POKE sa+53, 160: POKE sa
56, 95: POKE sa+57, 160
6260 POKE sa+44, 101: POKE sa+45, 10: POKE sa+48, 101: POKE sa+49, 10: POKE sa
24, 50: POKE sa+25, 200
6265 POKE sa+32, 64: POKE sa+33, 200: POKE sa+28, 80: POKE sa+29, 50
6270 CALL sr
6275 IF f(45) = 1 THEN  RETURN
6280 FOR 12 = 1 TO 2: FOR l = 95 TO 79 STEP -3
6283 IF l = 95 THEN  FOR lp = 1 TO 4: GOSUB 7800: NEXT lp
6286 POKE sa+52, l: POKE sa+56, l: CALL sr: GOSUB 7800
6290 NEXT l, 12
6295 POKE sa+52, 95: POKE sa+56, 95: CALL sr
6299 RETURN
6300 RETURN: REM  dummy graphics routine
6400 RETURN: REM  dummy graphics
6500 RETURN: REM  dummy graphics
7799 REM  *****time delay routine
7800 FOR de = 1 TO 100: NEXT de: RETURN
```

```
7899 REM  *****room descriptions
7900 ON rm GOTO 8010, 8020, 8030, 8040, 8050, 8060, 8070, 8080, 8090, 8100, 811
0, 8120, 8130, 8140, 8150
7920 ON rm-15 GOTO 8160, 8170, 8180, 8190, 8200, 8210, 8220, 8230, 8240, 8250,
8260, 8270, 8280, 8290, 8300
7930 ON rm-30 GOTO 8310, 8320, 8330, 8340, 8350, 8360
8010 PRINT " room 1": RETURN
8020 PRINT " room 2": RETURN
8030 PRINT " room 3": RETURN
8040 PRINT " room 4": RETURN
8050 PRINT " room 5": RETURN
8060 PRINT " room 6": RETURN
8070 PRINT " room 7": RETURN
8080 IF b2 = 2 THEN  PRINT " No wonder you heard an explo-  sion! This storage
room is a   wreck.": RETURN
8084 PRINT " A storeroom with big crates.": RETURN
8090 PRINT " room 9": RETURN
8100 PRINT " room 10": RETURN
8110 PRINT " room 11": RETURN
8120 PRINT " room 12": RETURN
8130 PRINT " room 13": RETURN
8140 IF f(34) = 0 THEN  PRINT " That dog will tear you to bits if you try to cr
oss this yard."
8142 IF f(34) = 1 THEN  PRINT " The dog peacefully munches his food."
8149 RETURN
8150 IF b1 = 2 THEN  PRINT " Smoke and the smell of death.  The ambassador's su
ite is wrecked.": RETURN
8151 PRINT " You have burst in on the        ambassador herself."
8152 IF f(35) = 1 GOTO 8158
8154 IF f(35) = 0 AND b1 = 1 THEN  PRINT " 'Who are you?' she challenges  'Get
out!"
8156 IF f(35) = 0 AND b1 = 0 THEN  PRINT " She welcomes you but asks why  you h
ave returned to her suite"
8158 IF f(35) = 1 THEN  PRINT " She's still unconscious."
8159 RETURN
8160 PRINT " room 16": RETURN
8170 PRINT " room 17": RETURN
8180 PRINT " room 18": RETURN
8190 PRINT " room 19": RETURN
8200 PRINT " room 20": RETURN
8210 PRINT " room 21": RETURN
8220 PRINT " room 22": RETURN
8230 PRINT " room 23": RETURN
8240 PRINT " room 24": RETURN
8250 PRINT " room 25": RETURN
8260 PRINT " room 26": RETURN
8270 PRINT " room 27": RETURN
```

```
8280 PRINT " room 28": RETURN
8290 PRINT " room 29": RETURN
8300 PRINT " room 30": RETURN
8310 PRINT " room 31": RETURN
8320 PRINT " room 32": RETURN
8330 PRINT " room 33": RETURN
8340 PRINT " room 34": RETURN
8350 PRINT " You're at the door of the       guard house.": RETURN
8360 PRINT " room 36": RETURN
8499 REM  *****initialize
8500 DIM c(w), l(w), f(w), r$(36)
8600 DATA  200,200,0,09,200,200,4,09,200,200,4,14,200,200,8,14,200,200,16,14,2
C,200,20,6,200,200,24,3,200,200,28,12
8615 DATA  200,200,32,06,200,200,36,08,200,200,40,14,200,200,44,04,200,200,48
5,200,200,52,10,200,200,56,9,200,200,60,6
8625 DATA  200,200,64,09,200,200,68,15,200,200,72,15,200,200,76,13
8630 sa = 29500: RESTORE
8640 FOR l = 0 TO 79: READ a: POKE sa+l, a: NEXT l
8650 IF ret = 1 THEN  RETURN
8660 ret = 1
8750 pk = 27850
8760 READ a$: IF a$ = "zz" GOTO 8790
8765 wnum = wnum+1: IF a$ = "autojack" THEN  vx = 1
8766 IF vx = 0 THEN  vnum = vnum+1
8770 FOR l = 1 TO LEN(a$): POKE pk, ASC(MID$(a$, l, 1)): pk = pk+1: NEXT l
8780 POKE pk, ASC("^"): pk = pk+1: GOTO 8760
8790 POKE pk, ASC("]")
9115 DATA  help,carrying?,go,N,S,W,E,walk,run,exit,get,take,grab,lift,seize,p
k,steal
9150 DATA  open,examine,look,inspect,search,investigate,explore,read,decode
9155 DATA  defuse,dismantle, disarm,unlock,pry,drop,throw,dump,release,leave,
light",extinguish
9200 DATA  fight,punch,kick,attack,hit,use,move,push,shove,pull,consume,drink
reak,bend,split,shatter,destroy,wreck,burn
9240 DATA  talk,persuade,charm,threaten,convince,flatter,deceive,plead
9250 DATA  feed,distract,bribe,blackmail,show,"flash ",save,load
9310 DATA  autojack,crowbar,keys,meat,flashlight,tools,camera,money,wine
9320 DATA  rope,letter,"book ",matches,bones,codebook,tirepump,knife,badge,no
h,south,west,east
9360 DATA  television,pool,furnace,crate,bed,king,furniture,aide,maps,window,
nce
9370 DATA  dog,ambassador,garbage,stairs,weapons,grass,staff,bookcase,car,pri
ner,bomb,guard,door,room,zz
9510 DATA  175,14,0,50,17,107,50,18,107,50,19,107,50,15,107,50,16,107,33,202
8
9515 DATA  237,99,20,107,58,16,107,60,50,16,107,33,52,108,237,91,20,107,26,1
9520 DATA  254,94,202,92,107,254,93,200,35,19,195,77,107,19,237,83,20,107,17
,103
```

```
9525 DATA  14,0,33,52,108,6,0,235,167, 237,66,235,14,0,26,71,126,254,94,202,150
,107
9530 DATA  50,22,107,120,254,94,202,63,107,58,22,107,184,202,145,107,33,52,108
9535 DATA  19,195,105,107,35,19,12,195,114,107,58,17,107,230,255,194,180,107,58
,16,107
9540 DATA  50,17,107,33,72,108,235,167, 237,82,235,123,50,15,107,195,63,107,58,
18,107
9545 DATA  230,255,194,195,107,58,16,107,50,18,107,195,63,107,58,16,107,50,19,1
07,201,256
9570 pk = 27430
9580 READ a: IF a < 256 THEN  POKE pk, a: pk = pk+1: GOTO 9580
9610 DATA  33,25,18,20, 7,12,00,12, 1, 5,29,29, 4,19, 5,33,20,17
9612 DATA  0,0,0,0,2,4,7,8,10,10,10,11,12,18,0,14,15,19,22
9614 DATA  24,25,26,29,33,36,24,34,30,0
9620 FOR l = 1 TO w: READ l(l): NEXT l
9660 DATA  S,SE,WE,WE,SWE,W
9670 DATA  NE,NSW,SE,SW,NSE,SW
9680 DATA  SE,NSW,NE,NSW,NSE,NW
9690 DATA  NS,NSE,SW,NSE,NW,S
9700 DATA  NS,N,NE,NSWE,W,NW
9710 DATA  NE,WE,WE,NWE,W,N
9720 FOR l = 1 TO 36: READ r$(l): NEXT l
9760 f(2) = 1: f(43) = 1: c(7) = 1
9901 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9902 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9904 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9905 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9907 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9908 DATA  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255
9910 DATA  255,255,192,192,192,192,192,192,192,192,192,192,192,192,255,255
9911 DATA  255,255,3,3,3,3,3,3,3,3,3,3,3,255,255
9913 DATA  255,255,128,128,128,128,128,128,128,128,128,128,128,128,128,255
9914 DATA  255,255,1,1,1,1,1,1,1,1,1,1,1,1,255
9916 DATA  255,191,223,224,239,239,239,224,223,128,239,239,239,224,223,128
9917 DATA  255,253,251,7,247,247,247,7,251,1,247,247,247,7,251,1
9919 DATA  3,7,13,55,63,127,251,239,255,247,253,127,59,15,15,3
9920 DATA  224,240,252,252,126,237,255,247,255,191,251,254,252,252,240,224
9922 DATA  9,45,180,82,86,112,41,237,228,179,211,87,123,63,55,23
9923 DATA  80,220,153,166,166,77,125,136,190,246,83,215,204,200,246,214
9925 DATA  25,15,7,3,2,3,3,3,3,3,3,3,3,3,15,124
9926 DATA  136,152,240,224,96,224,224,224,32,224,224,224,224,224,248,158
9928 DATA  255,248,240,224,224,224,240,224,192,192,192,192,192,128,128,128
9929 DATA  255,15,31,127,127,15,3,3,3,3,3,3,3,3,3,3
9931 DATA  0,0,0,0,0,1,2,13,59,54,61,63,63,63,62,60
9932 DATA  0,0,0,0,0,128,128,128,128,128,128,128,31,50,124
9934 DATA  0,0,0,7,8,16,16,63,63,76,204,248,255,0,0,0
9935 DATA  0,0,0,240,8,6,6,252,252,50,51,31,255,0,0,0
```

```
9937 DATA  0,0,0,0,0,0,0,0,50,48,3,0,48,48,48
9938 DATA  4,4,4,4,4,32,80,0,0,76,140,224,0,12,12,12
9940 DATA  0,3,7,6,12,12,12,108,108,60,28,12,0,0,0,0
9941 DATA  0,252,252,14,6,6,6,6,6,7,7,6,0,0,0,0
9943 DATA  0,0,0,1,3,3,3,3,1,0,0,0,15,15,25,49
9944 DATA  0,0,0,240,88,248,248,16,224,192,224,224,252,252,246,243
9946 DATA  0,3,4,4,1,0,0,1,0,0,0,31,63,51,99,195
9947 DATA  0,240,8,4,0,0,0,224,0,0,0,28,254,242,243,243
9949 DATA  0,0,3,3,6,7,7,4,1,0,1,0,0,0,0,0
9950 DATA  0,0,240,248,216,248,248,16,224,192,224,224,0,0,0,0
9952 DATA  0,0,0,0,0,8,8,24,104,254,31,31,63,100,68,194
9953 DATA  0,0,0,0,0,0,1,1,1,62,254,255,254,14,9,17
9955 DATA  0,0,0,0,0,8,8,24,248,62,111,143,15,14,20,18
9956 DATA  0,0,0,0,0,0,1,1,1,62,254,254,255,31,18,20
9958 DATA  0,0,15,24,48,96,255,248,232,230,194,198,232,224,248,255
9959 DATA  0,0,255,3,7,13,249,249,57,57,25,25,57,58,252,248,256
9996 sd = 28850: 1 = 0
9997 READ a: IF a = 256 GOTO 10030
9998 POKE sd+1, a: 1 = 1+1: GOTO 9997
10010 DATA  14,226,6,1,205,32,253,58,24,252,253,33,80,00,33,178,112,17,00,00,20
,44,253
10020 DATA  58,23,252,253,33,20,00,17,00,00,33,60,115,205,44,253,201,256
10030 sr = 29600: 1 = 0
10040 READ a: IF a = 256 GOTO 10200
10050 POKE sr+1, a: 1 = 1+1: GOTO 10040
10200 RETURN
19900 HOME: PRINT TAB(6); "OPENING DESCRIPTION"
19910 PRINT: PRINT " description.": PRINT " Please wait a moment.": RETURN
```

General structure of the program

Memory map

In the ADAM, BASIC uses up memory up to memory location 27407. In Adventure Creator, we have to reserve a part of memory for two important functions that are handled in machine language, rather than BASIC, because machine language is so much faster. The area from 27407 is reserved by the command LOMEM:29650. Thus, the area from 27407 to 29650 will not be used by BASIC.

Each of these program parts will be explained in detail later. In this section, we will just locate them for you.

The PARSER analyzes the player's input to see which verb and noun were used. It takes memory from 27407 to 27849. (Actually, a little memory is left over unused to make it easier to expand the program later if desired). From 27850 to 28849, the vocabulary understood by the program is placed. 28850 to 29490 hold the sprite data that define the shapes of the 20 sprites used in the program. 29500 to 29580 hold 80 bytes that define the attributes of the 20 sprites--4 bytes per sprite. From 29600 to 29640 is the machine language routine that controls the sprites.

BASIC program map.

This listing uses program line numbers.

1-150 overall supervisor section
160-499 get input and analyze input (parser)
500-999 description and feedback
2000-4799 verb action routines
5000-5399 unusual actions routines (explosions in Bomb Squad)
6000-7000 graphics
7900-8400 location descriptions
8500-10010 initialization routines.
19900-end set opening scene

Note that the general strategy is to place often used parts of the program near the beginning. This is because, when BASIC is looking for a subroutine or a place to GOTO, it starts at the beginning of the program. The extensive initialization and the opening description are only used once, so they're at the end. This greatly speeds up the program and makes it more fun to play.

Initialization

There are many things to be done before the game is ready to play. In Adventure Creator, this takes about 13 seconds. First the opening description comes up on the screen, so the player has something to read while waiting for the initialization to finish. This is accomplished by line 60.

Line 50 initializes a number of variables necessary for the game. w=47 sets the number of objects or nouns. g=18 sets the number of "gettable objects". rm=34 sets the room number for the player's location at the start of the game. b1=1 b2=1 b3=1 are unique to Bomb Squad, but they illustrate a programming method. If b1, for example, equals 1, we know that the first bomb is still ticking away somewhere; if it has been set to zero by the program, the bomb has been defused; if it has been set to 3, the bomb has exploded. t1=30 t2=60 t3=90 set time limits in Bomb Squad. The program keeps track of "time" by counting the number of moves made. Thus it is easy to check if t1 (time #1) has been exceeded and take the appropriate action.

Still on line 50, dr$=1 determines which storage device is used for saving games. As long as dr$=1, the program will SAVE to and LOAD from the tape drive. If dr$=5, the disk drive will be used. tl=4 sets the player's "total load"--the number of objects that can be carried. tc=1 is the number of objects actually being carried at the moment. It is set to 1 here, because the player starts out with the camera in Bomb Squad. You might want to change it to zero, depending on how your game starts.

Line 8500 reserves room for 4 arrays. If you don't understand arrays, you should study a book about BASIC, but in general, an array is a block of memory that operates like numbered boxes. The DIM command, dimensions or reserves the arrays you want.

Array c(w) will keep track of what the player is carrying. The variable w is the number of nouns or objects in the vocabulary; w was set to 47 in this game, back in line 50. When a program is RUN, all variables are set to zero, so there is no need to set any of the numbers in c(w), unless the player is carrying something to start with. In Bomb Squad, the player starts out carrying a camera, so in line 9760, you will find that c(7) is set to 1. This is because the camera is item #7 in the vocabulary of nouns (see lines 9310 to 9380). Notice that only the first 18 objects are things that the player <u>can</u> carry--these are "gettable" objects, so only c(1) through c(18) will ever actually be used for items being carried, but we still reserve space in the c(w) array to check in case the player says something like "throw car".

The array l(w) keeps track of the location of each object. In lines 9610-9620 we will set each value in array l(w) depending on which room each object is in. For example, if the first object in our object list is "autojack" and it starts out in the garage, which is room 33, then l(1) will be set to 33. During play, if the player picks up an object, the location of that object will be set to zero. Later the player may put the object down, so we will keep track of where the object is by changing its value in the l(w) array.

The array f(w) keeps track of "flags" for each object. These flags make the game much more interesting. A value of zero indicates that the object is in its normal state. For example, in Bomb Squad, the ambassador is object number 35. The game starts with f(35)=0. If the ambassoador gets knocked out, f(35) is set to 1, indicating a changed status. As another example, the crowbar is object number 2. At the beginning of the game, it is hidden in the grass. f(2)=1 until it becomes visible; then f(2) is changed to zero, because that would be a "normal" state.

Array r$(36) stores the visible exits for each room. In lines 9660-9720, each location in the array is filled. For example, the visible exits for room 34 (the starting location in Bomb Squad) are north, west and east. Thus, the 34th value (in this case, the 4th piece of data in line 9710) is NWE.

Lines 8600-8660 POKE into memory, starting at memory location 29500, the 80 attributes of the sprites. These are the first of many pieces of data to be POKEd into memory. The sprite attributes are first because we are going to be poking them into memory many times in the program--in fact, each time we set up a new graphics scene. Look at line 8650. The first time through, the variable "ret" will be zero (all variables are set to zero when a program is RUN), so this line will be ignored. However, the next line sets "ret" to 1. Thus, we can now use lines 8630-8650 as a subroutine that POKEs in the original sprite attribute data. When we analyze the graphics routines, you will see the command GOSUB 8630. This sets up 20 sprites in a location that is not visible on the screen. Then with the screen clear of sprites, we can move the ones we want onto the screen by changing some of the attributes. (As you will soon see, the first two numbers of each set of four places the sprite initially at coordinates x=200 and Y=200, which is off the screen.)

Lines 8750-8790 POKE the next data into memory. This data is the vocabulary of the game. The words will be POKEd in starting at memory location 27850. Each word in the data is read and then POKEd into memory, one letter at a time by line 8770. At the end of each word, the character "^" is POKEd in because the parsing routine needs some way to recognize the end of a word or phrase. (As you will see later, the vocabulary could contain phrases rather than just words, if you want to get fancy.) At the end of the vocabulary data section, the last "word" is "zz". When line 8760 encounters "zz", it jumps to line 8790 and POKEs in "]" to mark the end of the vocabulary. Note that the first words are the verbs and the objects (nouns) come second. If you add or subtract verbs, be sure that "save" and "load" are always your last verbs. Line 8765 keeps track of the total number of words by incrementing variable wnum. Line 8766 keeps track of the number of verbs. It stops counting after the word "autojack" is encountered--which happens to be the first noun in this list.

The data in lines 9510-9560 are the values for the machine language parsing routine (see the parser section for details). Lines 9570-9580 POKE these values in, starting at memory location 27430.

Line 9620 reads in the locations of each object into array l(w), as discussed before. The data in lines 9610-9614 show which rooms hold each object. For example, object #1 is in room #33. Object # 2 is in room # 25 etc. In this list, object #7 has a location of zero, because the player is carrying it at the start of the game. Object # 33 is zero because, in this example, it is a fence that is in many locations. Objects 19,20,21, and 22 are also zero because they are not really objects at all. Look at the vocabulary list, and you will see that they are the directions north, south, west and east. We need to include them as nouns because the player can indicate a direction to go with the sentence "go south", for example.

Line 9720 fills the r$(36) array, as discussed previously.

Lines 9901-9959 contain the 640 values needed to define the shapes of the 20 sprites used. The section on sprites will explain all this. Lines 9997-9998 POKE these values in, starting at memory location 28850.

Lines 10010-10020 are the values for the machine language routine that controls the sprites. Lines 10030-10050 POKE this routine in, starting at memory location 29600. The variable sr (for sprite routine) is set at 29600, so we can CALL sr---later when we do graphics.

Lines 19900-19955 set the first scene for the player.

Description and feedback

The lines from 500-699 and the "room descriptions" in lines 7900-8360 write to the screen after each move to tell the player where he or she is and what happened as a result of the last action.

Lines 500-512 check each time to see if the game has been completed. In Bomb Squad, it checks the status of each bomb. If they all have been defused, congratulations are offered. If bomb 3 is defused but at least one of the others has exploded, and the other one is defused or exploded, lukewarm congratulations are offered. You will see elsewhere in the game that if bomb 3 explodes, the game ends in total disaster.

Line 516 clears the screen and prints the description of the player's location. Let's skip to the room description routine starting at line 7900.

The variable rm always represents the number of the room the player is in at the time. Lines 7900,7920, and 7930 will cause a jump to the lines that describe the current room. Notice how the lines are numbered to help you keep track of your room and find the right lines quickly for debugging. 8010 is for room #1, 8220 is for room #22 etc. You always know the middle two numbers of the line number correspond to the room being described. Of course, we don't know what each of your rooms will be like, so in most cases we have just put " room 1" etc. where you will put the actual descriptions of your rooms. Notice that in the descriptions, the first character in the string is a space. This is necessary because on some TV sets, the first character is displayed off the left side of the screen.

We have, however, included some descriptions from Bomb Squad to illustrate some techiniques. For example, lines 8080-8084 describe room #8. Bomb #2 (whose status is kept track of by variable b2) is in room 8. If it has already exploded (b2=2) we will want a much different description than if it has not (b2=1 or =0).

In room #14 (lines 8140-8149) we see the use of "flags". If the dog is in his normal state--that is, hungry and mean--his flag, f(34), still equals zero. However, when he is fed, the program sets his flag to 1, and we get a much different room description.

In the ambassador's suite (room #15) we get even more complicated. Several different things might be said, depending on the status of the bomb in her suite (what does b1 equal?) and on whether the player has knocked her out, changing the status of her flag, f(35).

One last point, in line 8350, the spacing looks odd on paper. However, when you list this line on the screen, you will see that the extra spaces are there to prevent the word "guard" from being split at the end of the line.

Now let's return to the feedback section at line 518. This is a condition check that only works placed here in the program. The only way variable tl (total load) can be reduced to zero is if the player repeatedly tries to get past the dog without using strategy. Obviously, this line is unique to Bomb Squad, but you might have similar conditions arise in your games.

Lines 520-530 are absolutely essential. They take the letters in r$( ) for this room and list them one by one to show the player what exits are visible from the room the player is in. If an exit becomes visible during the game, we will change the letters for that room in r$( ). We will discuss this in more detail later, but for example, if the player's flashlight reveals a hidden door in room # 12 on the south wall, when that happens r$(12) will be changed. If previously there were three visible exits, r$(12) would have been "NWE". Now we just include the statement r$(12)="NSWE", and the next time the player is in room #12, all four directions will be listed as visible. Similarly, if a door locks behind the player, we might subtract a legal exit.

Lines 535-560 add to the room description by listing all the "gettable" objects in the room. Line 535 RESTOREs to the beginning of the data statements. Line 537 simply reads from the data statements to skip over the ones we're not interested in here. When it reads the word "load" it knows it is at the end of the verbs. (This is why the verb "load" must always be your last verb.) Line 540 loops through all the gettable objects, reading each noun, one at a time. Line 550 checks if the location of each object is in the current room and if the object is visible--that is, its flag=0. If both conditions are met, the name of the object is printed out.

Finally, line 560 ends the loop and prints out m$, which is the feedback (message string) that much of the rest of the program is devoted to. m$ is changed many times in the program and should end up with a meaningful message for the player--such as "You can't go that way." or "Excellent move! The amulet weakened the monster" etc. As soon as the feedback is given, m$ is set to "What?". This is the "default" feedback. If nothing else happens anywhere else in the program (very unlikely), the player will get this feedback, and will have to try some other command. Notice that an extra space is printed just before m$ is printed. This is also because some TVs don't show the first column of each line with the ADAM.

Lines 600-630 present the graphic illustrations for the four rooms that have graphics. In each line, GOSUB 2160 causes the program to print "Push enter to continue" and then wait for the player to push enter. This permits the verbal descriptions and feedback to stay on the screen until the player is ready to see the picture. Notice that 2160 is really part of one of the verb action routines, but it serves nicely whenever we want this kind of pause inserted. Then the final GOSUB in each line calls the graphics routines.

Input

After the player is given feedback and a description of where he or she is, the question "What will you do now?" appears on the screen. Line 160 gets the player's input as q$. The player is not permitted to use commas in the input, since the INPUT statement ignores everything after the comma. This is no problem in most adventure games, since only two word commands are permitted. If you want to permit commas, use the input routine given later in the section on "the parser".

For the parser to analyze this input, the input must be POKEd into memory starting at location 27720 and be terminated with the character "^". Lines 240-260 take care of this. Line 280 clears the screen in preparation for the next feedback.

As soon as the input phrase is in memory, lines 410-499 set the verb number (vb) and object number (ob) to zero and CALL the parser routine.

The parser

A "parser" separates a sentence into words and permits the analysis of those words. In most adventure games written in BASIC, the parser uses string manipulation and is so slow that the game is limited to a vocabulary of about 20 words. Adventure Creator uses a machine language parser that permits a large vocabulary (Bomb Squad has 73 verbs and 43 objects, for example) and is very fast. In fact, this parser has features that are not used in Bomb Squad. We will describe its features, since you might want to use them in your own games or for other programs.

How it operates.

Remember that the input routine POKEs the player's input into memory at 27720. The initialization routine POKEd the whole vocabulary into memory at location 27850. When you CALL 27430, the parser routine takes each word in the vocabulary and tries to match it to the input sentence. When it finds the first match, it puts the number of that word into memory location 27409. That is, if the player's input was "persuade ambassador", the parser routine will try to match each of the vocabulary words to the input sentence, and when it reaches the word "persuade" in the vocabulary (this is word #59 in the vocabulary) it will find a match and will put the number 59 in location 27720. Then it will try to match the second word in the input sentence and put its number in location 27410. If no match is found, those locations will be zero. In our example, "ambassador" is word #108.

Thus, line 450 can set the value for the verb (vb) and object (ob). (Note that line 458 subtracts the number of verbs from ob, so we know which object this is, rather than which word from the entire list.)

There is a problem you will have to watch out for. Look at line 9195. The word " light" is in quotation marks and has a leading space. Otherwise, the parser would be confused by a word like "flashlight" because it would find a match btween the word light in its vocabulary and the input word flashlight. For the same reason, the directions N, S, W, and E must be in capital letters, or the parser would match them with any word containing the letter.

<u>Extra features of the parser</u>.

Although most adventure games use two word commands as in Bomb Squad, the parser routine can include phrases in its vocabulary as well. In the data statements containing the vocabulary, you might include the phrase "why me?" as one entry in your vocabulary data. The parser will look for this phrase in the player's input.

<u>Fancier input</u>.

If you do use the parser for some application that permits the use of phrases, the player will often enter sentences that include commas. This possibility requires you to use a more complicated input routine than included in Adventure Creator, which simply uses an INPUT statement. The INPUT command ignores anything typed in after a comma. Change line 160 and add the following lines to the input routine in Adventure Creator

```
160 print " What will you do now?"
180 q$= ""
185 get p$:print p$;: if p$="," then p$=" "
190 if asc(p$)=13 goto 240
195 if asc(p$)=8 and len(q$)=1 then q$="":goto 185
200 if asc(p$)=8 then q$=left$(q$,len(q$)-1):goto 185
210 q$=q$+p$:goto 185
220 if q$="" goto 185
```

These lines convert commas to spaces, look for the carriage return (asc(13)) and handle backspaces (asc(8)) correctly.

<u>Two more extra parser features</u>.

The parser can actually find as many as three words or phrases. With most adventure games, only two words, a verb and a noun, are permitted, but if you ever have a use for it, our parser will look for a third phrase. If it finds a third match to the vocabulary list, it will put the number of that word or phrase in memory location 27411.

Finally, the parser also keeps track of <u>where</u> in the player's input phrase the first match was found. In other words, if the player input "don't go home", and the first matching word found was "go", the location of "go" in the sentence will be indicated as 8. That is, the end of the matched word or phrase is at the eighth character. The parser will place this number in memory location 27407. You will have to use your imagination to find uses for this feature, but if you are trying to analyze a sentence input by a player, this information is often useful.

<u>Assembly language listing of the parser</u>.

Most users will probably not be familiar with assembly language, but for those who are, here is a commented listing

of the parser routine. The first number in each line is the decimal address of the code listed in that line. The second number is the same address in hexadecimal. The actual program values (in hexadecimal) are next, follwed by the assembly language code.

```
27407:6B0F 0            PH1END DEFB 0; end of phrase 1
27408:6B10 0            CURPHR DEFB 0
27409:6B11 0            PHCNT1 DEFB 0; # of 1st phrase
27410:6B12 0            PHCNT2 DEFB 0
27411:6B13 0            PHCNT3 DEFB 0
27412:6B14 0            VOCADD DEFB 0;current address in vocabulary
27413:6B15 0                   DEFB 0
27414:6B16 0            TMPSTO DEFB 0
27430:6B26                     ORG 27430
27430:6B26 AF                  XOR A; clear
27431:6B27 0E d1               LD C,$00
27433:6B29 32 11 6B            LD (PHCNT1),A
27436:6B2C 32 12 6B            LD (PHCNT2),A
27439:6B2F 32 13 6B            LD (PHCNT3),A
27442:6B32 32 0F 6B            LD (PH1END),A
27445:6B35 32 10 6B            LD (CURPHR),A
27448:6B38 21 CA 6C            LD HL,S6CCA
27451:6B3B ED 63 14 6B         LD (VOCADD),HL
27455:6B3F 3A 10 6B     NXTPHR LD A,(CURPHR);count phrase
27458:6B42 3C                  INC A;update and store
27459:6B43 32 10 6B            LD (CURPHR),A
27462:6B46 21 34 6C            LD HL,S6C34;phrase buffer
27465:6B49 ED 5B 14 6B         LD DE,(VOCADD);current location in voc
27469:6B4D 1A           PHRMOV LD A,(DE); get character to move
27470:6B4E 77                  LD (HL),A; store it
27471:6B4F FE 5E               CP $5E:"^"?
27473:6B51 CA 5C 6B            JP Z,MOVDON;phrase moved
27476:6B54 FE 5D               CP $5D;"]"?
27478:6B56 C8                  RET Z; yes-end of phrases
27479:6B57 23                  INC HL; inc to store
27480:6B58 13                  INC DE; inc to get
27481:6B59 C3 4D 6B            JP PHRMOV; move another
```

```
27484:6B5C 13          MOVDON INC DE;uodate vocab addr
27485:6B5D ED 53 14 6B        LD (VOCADD),DE ;store pointer
27489:6B61 11 48 6C           LD DE,36C48;input addr
27492:6B64 0E 00              LD C,S00; zero match counter
27494:6B66 21 34 6C           LD HL,36C34;phrase buffer
27497:6B69 06 00       NXTCHR LD B,S00;only C w/#'s
27499:6B6B EB                 EX DE,HL;false start-back up
27500:6B6C A7                 AND A; clear carry
27501:6B6D ED 42              SBC HL,BC
27503:6B6F EB                 EX DE,HL
27504:6B70 0E 00              LD C,S00; zero match counter
27506:6B72 1A          NXTMAC LD A,(DE);get from in buff
27507:6B73 47                 LD B,A
27508:6B74 7E                 LD A,(HL);get from phrase buff
27509:6B75 FE 5E              CP S5E;"^"?
27511:6B77 51 97 6B           JP Z,PHRDON;phrase done-a  match
27514:6B7A 32 16 6B           LD (TMPSTO),A
27517:6B7D 78                 LD A,B
27518:6B7E FE 5E              CP S5E;input a "^"?
27520:6B80 CA 3F 6B           JP Z,NXTPHR;yes-done input
27523:6B83 3A 16 6B           LD A,(TMPSTO)
27526:6B86 B8                 CP B;match?
27527:6B87 CA 91 6B           JP Z,MATCH1;yes-do more
27530:6B8A 21 34 6C           LD HL,S6C34;no-restart phrase buff
27533:6B8D 13                 INC DE;next input char
27534:6B8E C3 69 6B           JP NXTCHR
27537:6B91 23          MATCH1 INC HL;next 2 chars
27538:6B92 13                 INC DE
27539:6B93 0C                 INC C;count matches
27540:6B94 C3 72 6B           JP NXTMAC
27543:6B97 3A 11 6B    PHRDON LD A,(PHCNT1);already matched?
27546:6B9A E6 FF              AND SFF
27548:6B9C C2 B4 6B           JP NZ,SCNDPH;yes
27551:6B9F 3A 10 6B           LD A,(CURPHR);no-get phr #
27554:6BA2 32 11 6B           LD (PHCNT1),A;store it
27557:6BA5 21 48 6C           LD HL,$6C48;input buffer
27560:6BA8 EB                 EX DE,HL;subtract start of input buffe
```

```
27561:6BA9 A7                 AND A;(clear carry)
27562:6BAA ED 52              SBC HL,DE
27564:6BAC ED                 EX DE,HL;location of phrase 1
27565:6BAD 7B                 LD A,E;store it
27566:6BAE 32 0F 6B           LD (PH1END),A
27569:6BB1 C3 3F 6B           JP NXTPHR
27572:6BB4 3A 12 6B    SCNDPH LD A,(PHCNT2);already 2?
27575:6BB7 E6 FF              AND SFF
27577:6BB9 C2 C5 6B           JP NZ,THRDPH
27580:6BBC 3A 10 6B           LD A,(CURPHR)
27583:6BBF 32 12 6B           LD (PHCNT2),A
27586:6BC2 C3 3F 6B           JP NXTPHR
27589:6BC5 3A 10 6B    THRDPH LD A,(CURPHR);3rd match
27592:6BC8 32 13 6B           LD (PHCNT3),A
27595:6BCB C9                 RET ;3 plenty
27700:6C34                    ORG 27700
27700:6C34 0           VOCBUF DEFB 0;phrase buffer
27720:6C48                    ORG 27720
27720:6C48 0           INBUFF DEFB 0;input buffer
27850:6CCA                    ORG 27850
27850:6CCA 0           VOCABU DEFB 0;vocabulary
```

Error messages and condition checks

An essential part of adventure games is that various events depend on what the player has already done. Lines 700-999 check for whether the player's input is valid and whether certain conditions have been met.

It is here that we will start assigning a value to the string called m$. Remember in the feedback section that m$ (for "message string") was printed out to tell the player what was happening.

First, line 700 checks to see if two words were used. If there was no second word that exists in the vocabulary, line 700 sets m$ to "That's silly." Remember that we will have plenty of chances to change m$ before we report back to the player. Thus, if the player enters a single letter, such as N, to go north, line 700 will set m$ to "That's silly" even though the input was valid. Later we will override this with a new version of m$.

Line 730 checks several things. If there is no verb or the first word in the input is not in the verb vocabulary or if the second word in the input (ob) is in the verb list, m$ says "You can't (whatever the player input)."

Line 740 checks if the player is carrying the named object. This m$ message will be replaced later in most cases.

Lines 825-860 are one condition check from Bomb Squad and are included here as an example of the kinds of things you should put in this part of the program. Bomb Squad is a time-limited game, so t (for time) is incremented each move the player makes. As soon as t1 (for time#1) is passed, the first bomb explodes, with GOSUB 5100. Similarly for t2 and t3. In Bomb Squad, the third bomb is a killer, so the routine at 5300 ends the game. This kind of condition check permits a lot of interesting variation. For example, in Bomb Squad, there are actions the player can take that cause time to be wasted (you'll have to figure out how yourself). The passage of time can be simply marked by adding some number to the variable t.

You will need lines 900-999. The parser has given you the number of the right verb, and in a moment you will want to branch to the routine that handles the activity called for by that verb. We have lots of verbs available, in order to make the game more interesting, but many of the verbs have roughly the same meaning and will call for the same verb action routines. In other words, the player might want to "search" or "examine" or "inspect" or "explore" etc. If the game permits only one of these words, it can be very frustrating to the player who has to make many guesses to get exactly the one word used by the author. Adventure Creator permits many such words to be used, but the parser will give each of these words a different value (in the variable vb). We need to see that all of these similar words have the same value for getting to the right verb action routine. Thus, for example, in the verb vocabulary, the words like "search" are verbs numbered 19-24. If the input verb was one of these, line 920 will set vj (for verb jump number) to 6. You will soon see that verb action routine #6 deals with all "searching" verbs. You can apply the same prinicple to each of the lines that sets a value for vj. To figure this out, you will have to keep referring to the vocabulary data list in lines 9115-9370.

Verb action routines

In many ways, this section is the heart of the adventure game. The player's verb input tells what is to be done. Each verb (or similar group of verbs) has its own set of permitted actions.

Line 2000 returns with no action if no verb was found--m$ will say "You can't (whatever the player input)".

Line 2005 is a condition check from Bomb Squad. This condition check had to be put here because it is one that permits only two actions--dropping something or checking what one is carrying. If the player got weakened in the game, his or her carrying capacity (tl for "total load" in Bomb Squad) was reduced. Thus, in order to continue, it may be necessary to drop something. The condition check must permit the player to get to the verb action routines in order to drop something, but line 2005 permits only verb action routine 2(carrying?) or 11 (drop,throw,dump,release, or leave). We have gone into detail on this line as an example of the kind of condition checks you will want to develop in your own games.

Jumps to verb routines.

Lines 2010 and 2030 use vj to jump to the appropriate verb action routine. Note how we have numbered the lines here. The first verb ("help") is at 2100, the second ("carrying?") is at 2200 etc. This permits you to find your way around your program more easily. A hundred line numbers are plenty for each verb routine, and your program will be much more readable and easy to modify if you use some rational line numbering system.

Line 2100- Help.

If you're hard hearted, you may want to leave out this routine. The player can ask for help and have the entire vocabulary listed on the screen. This might be considered cheating, since seeing the vocabulary usually gives huge hints on solutions to problems. For example, in Bomb Squad, seeing the verb "blackmail" practically gives away one solution.

Line 2110 skips through all the data statements until it finds the first vocabulary word. Then line 2120 reads the vocabulary until it finds "zz", at which point it quits. The variables wc (for word count) and hp (for help) keep track to be sure only 4 words per line and 16 lines per screenful are printed, so the screen will be readable.

Line 2200- Carrying?.
Line 2210 skips through the data statements to the last verb. Then line 2220 checks the c( ) array to see if each "gettable" item is being carried. If it is, the object name is printed.

Line 2300- Go.
The movement routine is so important, it is practically a small program in its own right. First, we must determine the direction the player wants to go. If the player input just N, S, W, or E, line 2300 will set variable d (for direction) to 1, 2, 3, or 4 by subtracting 3 from the verb number vb. These letters are items 4-7 in the verb list.

Now you will see why the words "north" "south", "west", and "east" are included in the vocabulary as "objects" or nouns. We permit the player to say things like "go south" or "crawl east". Thus, the verb will be "go" and the object will be one of the directions. In this case, lines 2303-2312 will set d correctly.

Following the setting of d, you should include some condition checks relating to player movement. Line 2344 is one such check included from Bomb Squad as an example. (If you want more examples, list out Bomb Squad.) In this example, the player is at the front door and has not yet "disabled" the guard, which the program knows because the "flag" for the guard (f(45)) is still zero. If the player tries to go north (d=1) or east (d=4) under these circumstances, m$ is set to "The guard won't let you pass", and nothing is permitted to happen because the RETURN, terminates the verb action.

Lines 2370-2388 check to make sure the player isn't trying to walk through a wall. Remember that the route array r$( ) contains the permitted directions of movement for each location. We are going to use "flag" 19 (f(19)) temporarily here because we won't ever need it to keep track of events--why?, because "object" #19 is really a direction name. The FOR NEXT loop in lines 2372 take each letter from r$( ) for the current location. For example, if the player is in "room" #12, and room #12 has only doors at north and south, then r$(12) will be "NS". If the player is trying to go north, line 2376 will determine that d=1 and N is permitted in room #12; f(19) will be set to 1. However, if the player is trying to go east, d will equal 4 and line 2382 will fail, since the letter E is not in r$(12). Thus, f(19) will not get set to 1. If this is the case, f(19) will still be zero, and line 2388 will set m$ to "Can't go that way!".

Now return to lines 2376-2382. When the IF condition succeeds, the room number gets changed. Now you can see why the "world" of your adventure game is laid out in a square grid. When the player moves north, the new room number is exactly 6 less than the old one (assuming you're using a 6x6 grid, as in Bomb Squad). Moving south adds six to the room number, east adds one, and west subtracts one.

Line 2400- Get.
The "get" verbs are another essential verb routine. The player must acquire and discard items to solve problems.

If the player says something like "steal guard" and you have not included the guard as a "gettable" object, line 2400 will prevent the action. Line 2420 checks if the player is in the same room as the object. Line 2430 checks if the object is visible--if the object's flag is set to it is likely hidden or invisible, if magic is at work. If the player is already carrying the object, line 2440 takes care of things.

Line 2452 is usually necessary to force the player to use strategies. If there is no limit to the objects that can be carried, the player will simply pick up everything. Here variables tc (for total carried) and tl (for total load) keep track of things. If the load is at maximum, the action is prevented by RETURNing the player without taking action. In this example, tl may change depending on whether the player has been weakened in previous game action. Bomb Squad starts with tl set at 4 items maximum to be carried. This gives some flexibility but also requires careful decision making, since the player isn't sure what circumstances will be encountered.

Line 2460 checks all the required conditions and then grants the player the item being sought.

Line 2470 is specific to Bomb Squad, but it is included here because it illustrates the solution to a common problem. The verb "take" is included as one of the "get" verbs, but in Bomb Squad, it can also be sensibly used in "take pictures". This line gets around the problem. Your problem is going to be that you will have to anticipate how your players might use different words in ways you don't anticipate. Probably your best strategy here is to play your own game many times, putting yourself in others's shoes. Then have some friends play the game while you take notes on such problems.

Line 2500-- Open.
A common adventure verb, open is used in many ways dependent on your particular story line. Lines 2500-2590 demonstrate several condition checks, to see if the player has

the necessary tools (or magic spells or keys or whatever) to open the door. Line 2530 is included to illustrate a new technique. In this instance, the only permitted directions of movement in room #30, were N and W. However, after the player unlocks a door, south is also permitted, so r$(30), which holds the permitted routes for room #30 is changed from "NW" to "NSW".

<u>Line 2600-- Examine</u>.
Another very useful set of verbs are the search/examine ones. Line 2600 checks if the player is in the same room as the object and sets m$ to "Nothing special--just a (whatever the object is)". Of course, if it is something special, we will soon change m$ to something else. For example, line 2630 determines that the player said, "examine garbage", so m$="That's disgusting!". (Try to catch the player off guard and be amusing sometimes.) Line 2640 illustrates what happens if something hidden is discovered. The player has said, "search grass" and discovers a crowbar. Thus f(2) --the crowbar "flag"-- is set to zero, since it is now visible--that is, "normal".

<u>Lines 2700-3000</u>.
These verbs (read, decode, defuse, unlock/pry) are fairly specific to the Bomb Squad game. You may be using other verbs here. The sample lines provided are fairly easy to figure out.

<u>Line 3100-- leave</u>.
Verbs like "leave" and "drop" are essential, if you have used "get" verbs and limit the amount the player can carry.

<u>Lines 3200-3400</u>.
More specific verbs (light, extinguish, fight). One interesting point here is that the verb "unlight" is ungrammatical but commonly accepted in adventure games, since we don't have a common verb that means turn off the light. However, you and we are erudite gamers and prefer to use accurate- even if esoteric--words, like "extinguish". Impressed? Another point is that this action is absolutely useless to the player in Bomb Squad. However, useless actions and useless objects must be scattered through the game to force the player to think through what really matters.

<u>Line 3500-- Use</u>.
Although this verb is absolutely essential, try to use it sparingly. Adventure games sometimes get simplistic and

boring by making the player say things like "use keys" and not permitting "unlock door". The more specific language is more interesting. However, some things do not lend themselves easily to one-verb commands. For example, you might have to say "use crowbar", since we don't have a good crowbar verb. Bomb Squad gets around this at one point by permitting "pry door" if the player has the crowbar.

<u>Lines 3600-4100</u>.
Some lines from Bomb Squad are included for these verbs to illustrate the verbs--move, drink, break, talk, feed, blackmail/bribe, and show.

<u>Line 4600-- SAVE</u>.
The SAVE command is not essential, but better games include it, and it greatly adds to the player's pleasure. Some people actually have to work for a living, and it can be pretty frustrating to have defeated the Lord of Darkness, cross the Mystic Threshhold, prepare to decode the magic code and have the boss call to find out why you're late again. A game in progress can be SAVEd to be completed later. It is also smart to SAVE a game as you are playing it. Then if you get killed or are in a hopeless situation (perhaps you can no longer carry <u>anything</u>), you can quit, RUN the game again, and reLOAD the game as you were when you did the last SAVE.
This and the "load" routine illustrate the general procedures for interacting with tape or disk drives with the ADAM. Line 4610 opens the file. In this case, the file will be named "bombgame", but you will want to use some other name. If you want to permit the player to SAVE the game under many different names, include an INPUT line somewhere that permits the player to assign a name (up to 10 characters long) to the variable f$ (for file string, although you can use any variable name, or course). In this case, line 4610 would be exactly as follows.

```
4610 print chr$(4);"open ";f$;",d";dr$
```

This creates the equivalent of the current line 4610 with a different file name. Remember that dr$ determines whether you will be saving to tape or disk. In line 50, we set dr$=1. This is for tape. If you want a disk version of your game, set dr$ to 5 in line 50.
Line 4620 prepares BASIC to write to the file. Lines 4630 use PRINT statements to write all the important variables. In this circumstance, the PRINT command writes to tape or disk because of the PRINT chr$(4) in line 4620.
Note that line 4695 terminates the SAVE and has the same general format as the opening line, 4610. If you permit f$ for different file names, make line 4695:

```
4695 print chr$(4);"close ";f$;",d";dr$
```

Line 4700-- LOAD.

When the player inputs simply the verb "load", the program looks for a file named "bombgame" or whatever you change this to for your own games.

Note that this routine is exactly like the SAVE routine, except we say "read" instead of "write", and INPUT rather than PRINT. It is essential that the variables be INPUT in exactly the same order as they were SAVEd or, of course, they will have the wrong values. If you use f$ for the file name, make Line 4710 just like 4610 and line 4795 just like 4695. (Notice how we are using parallel line numbering to make the program easier to understand and modify.)

GRAPHICS

Lines 6000-7799 have been reserved for graphics routines. Graphics routines in BASIC take up quite a bit of memory, but they are really worth it. Using visual clues in pictures adds a lot to adventure games. The ADAM has some pretty spectacular graphics capabilities, but BASIC makes them hard to use. Once you understand these sections on graphics, you should be able to draw high resolution pictures and use up to 32 sprites--but more on this later.

Using HPLOT for "building blocks"

Your general strategy is going to be to draw line drawings using small subroutines as "building blocks", place sprites around your picture, and then animate the sprites.

Examples of "building blocks" can be found in lines 6000-6055. Lines 6000 and 6010 draw the interior lines of a room in "3D" perspective. These two lines assume that somewhere else in the program you have gotten into high resolution mode (with lines of text at the bottom of the picture) with a HGR command and have set HCOLOR to some value. Then you use GOSUB 6000 to draw the room walls. To portray different rooms, just use different HCOLOR's.

Line 6030 is a similar routine that draws a 4-pane window in whatever HCOLOR was set last. The upper left-hand corner of the window will be at coordinates x and y. X and y must be set before this line is called with a GOSUB. X is the horizontal value and Y is the vertical value. This permits you to draw windows anywhere on the screen and to "stack" these windows on top of each other for big windows.

By the way, let us save you some frustration--or at least prepare you for it. The command HPLOT 100,50 to 150,50 should always draw a straight horizontal line. Sometimes, for reasons best known to the writers of smartBASIC, there is a jog in the line, so you have to use something like

HPLOT 100,50 to 150,49 to get a straight line. The subroutines included in Adventure Creator are adjusted for this, but you will probably run into the problem when you make your own "building blocks".

Line 6040 draws a door at the back of the room drawn by line 6000.

Line 6050-6055 draws a side window, in perspective, on the right hand wall of the room drawn by line 6000. As with the rectangular window, you need to set x and y coordinates before calling this subroutine.

Plotting your drawings.

In order to use HPLOT to draw pictures it is essential to prepare graph paper marked from 0 to 255 along the horizontal axis and 0 (at the top) to 159 along the vertical axis. Draw in the major lines of your scene and determine the points for starting and ending your HPLOT commands.

Drawing the scene

The more "building blocks" you have, the easier it is to draw a scene, but there will usually be unique parts to be drawn for each scene. Bomb Squad and The Visitor both use graphics to illustrate four scenes. One of the scenes--the view of the front of the embassy--will be analyzed here as an example. If you want more examples, of course, you can list the sections from 6000-7799 in each of the other programs.

Line 6200 sets BASIC to the HGR mode, which is high resolution graphics, with room reserved at the bottom of the screen for lines of text. For our purposes, this is the best mode, so we can ask the player for input while the picture is still on the screen.

Line 6203-6205 sets HCOLOR and x and y and draws windows by GOSUB 6030. Lines 6210-6230 use HPLOT's to draw in building outlines, driveways, and a garage.

Using sprites

Now comes one of the most interesting (and probably most complicated) parts. But hang in there, sprites will be worth understanding.

One of the most powerful graphic tools on the ADAM is the 32 sprite capability available. A sprite is a high resolution figure that can be 8x8 bits or 16x16 bits in size. Each bit is one dot on the screen, and remember that in high resolution, there are 256 dots horizontally and 159 dots vertically. So a 16x16 sprite will occupy about 10% of the picture from bottom to top. We find that a human figure should thus be made up of two 16x16 sprites (one on top of the other) to be proportional to an interesting picture on the screen. The sprites can also be in a magnified mode, so that an 8x8 becomes 16x16 and the 16x16 sprite becomes 32x32. The problem with the magnified mode is that the resolution looks much cruder--that is, each dot now looks like a small square, so the pictures don't have as professional a look.

For now, don't worry about how to make a sprite a particular shape; we will deal with that later. Just understand how they look on the screen. Each sprite is a small picture that can be instantly moved anywhere on the screen simply by setting two numbers, which will be the coordinates of the upper left hand corner of the sprite. In addition to being instantly moveable, each sprite has a certain priority of being visible. The sprites are numbered from 0-31, and the lower the number, the higher the priority of being visible. Thus, if sprite 1 and sprite 7 are moved to the same place on the screen, sprite 7 will be hidden "behind" sprite 1. Even better, if part of sprite 1 is not colored in, that part will seem transparent, and we will be able to catch glimpses of sprite 7 behind it, through the transparent parts of sprite 1. These characteristics permit very complex "3D" effects without complex programming.

Any one sprite can be only one color, but these characteristics give us ways to make multicolored objects on the screen. An example is the woman who appears and disappears in the guard house in Bomb Squad (and is included as an example in Adventure Creator). She has peach-colored skin and yellow hair. She is made up of two sprites, one yellow and one peach. She is drawn so that the "skin" sprite is transparent where her hair goes and the hair sprite is transparent where the skin goes. Actually, both sprites are transparent around the edges too, so you can see the guard "behind" her. Obviously, this means that the guard is made up of sprites with higher numbers than the sprites that make up the woman. When she moves, we simply move both the "skin" and "hair" sprites simultaneously. If you want to get cute, you could draw her as bald in sprite 14 and draw her hair in sprite 13. Then as long as both sprites move together, she will have her hair, because sprite 13 has priority for being visible, and her bald head would be hidden behind the hair. Then if you move only the "hair" sprite, her "wig" would come off. Then you could change her skin color, as she flushes with embarrassment. Let your imagination run.

You probably won't run into this, but only 4 sprites can be on any one line of the screen at a time. Nothing disastrous occurs, but if more than 4 sprites have the same vertical coordinate, parts of some of them will fade in unpredicatable ways.

Sprite data

Learning how to create sprite shapes will not be easy, unless you already understand things like hexadecimal numbers and the bit patterns used by computers. We will try to make the necessary information understandable.

Start with a grid on paper that corresponds to the size of sprite you want. We much prefer the 16x16 sprites without double-size magnification. They do consume a lot of memory, but they are also of a more useful size on the screen than the 8x8 sprites, and they look better than the magnified sprites. Our example is of a 16x16 sprite, but the same principles will apply to 8x8 if you prefer them.

The 16x16 grid in the following figure shows how we created the two sprites for the woman. First sketch in the parts, in this case the skin and hair, lightly. The squares of the grid containing hair we filled in with the letter H. (Actually, we did the original picture with colored pencils, which made it easier to visualize.) The skin squares are marked S. Since the woman is to appear only in the top of the guard house, we will need only the top part of her picture--otherwise we would have needed another two sprites (for skin and clothing) for the bottom half of the picture.

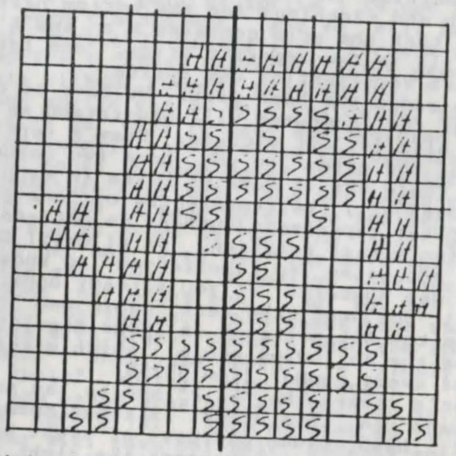
Now that the grid contains H's and S's and blank spots we can fill in two separate grids, one for hair and one for skin, each representing one sprite.

Now comes the hard part. We have to determine the numbers that the computer will understand as the correct pattern of bits for our picture. Draw a vertical line down the middle of the grid, so you have two columns, each 8 boxes wide. There are 16 rows of boxes in each column. Each row of 8 boxes will be represented by one number from 0-255. The computer represents numbers as patterns of 8 ones or zeros, using binary notation. Wait! Wait! Don't stop reading. We are not going to require you to understand binary and hexadecimal numbers. If you already understand these things you will not need the next table and can assign numbers based on the bit patterns in your drawings. If you don't, just use the following table, which gives you the "bit pattern" of every number from 0-255.

Let's create the "hair" sprite as an example. We need 32 numbers--in this order. Sixteen numbers that represent the left hand column of rows of eight boxes, followed by sixteen numbers that represent the right hand column. (Each of these numbers will be a "byte".) The first row in the left hand column is all blanks, so the first number will be zero. The second number will be 3 (we will provide the hexadecimal numbers in parentheses here--03H for this one). The third number is 7 (07H), fourth is 6 (06H), and fifth is 12 (0CH). You can refer to the diagram and to the bit pattern table to understand how the rest of the numbers were determined for the hair sprite and for the skin sprite.

As an example of how to use the bit pattern table, look at the fifth row of boxes in the "hair sprite". The left 4 boxes are empty, the next two are filled with the color, and the right 2 boxes are empty. In the bit table, 0 represents an empty box and 1 represents a colored one. Thus, the pattern we are looking for here is:
00001100

With a little practice, it will become easy to find a particular pattern in the bit pattern table; here we see that the number 12 gives us the pattern we want, so the fifth number in the data for our hair sprite will be 12. In the program listing, this is the fifth piece of data in line 9940.


Initial sketch of woman's head for making sprites #13 and #14. The letter H shows where hair color is to go, and the letter S shows where skin color is to go.

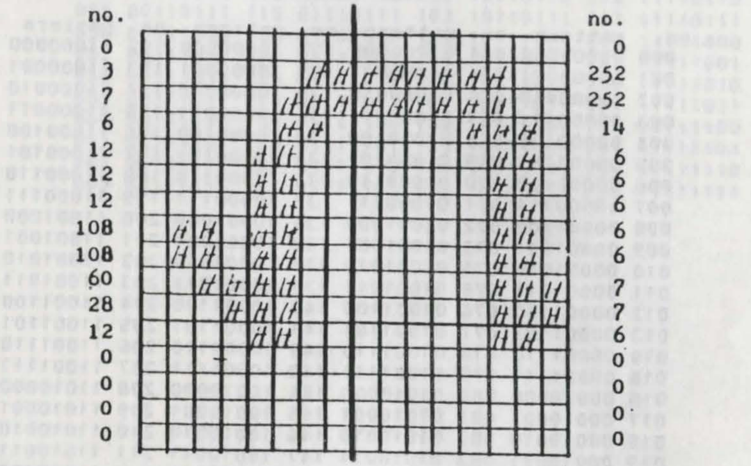


Diagram of grid for "hair" sprite. Numbers are taken from The Bit Pattern Table and correspond to the DATA in line number 9940-9941.

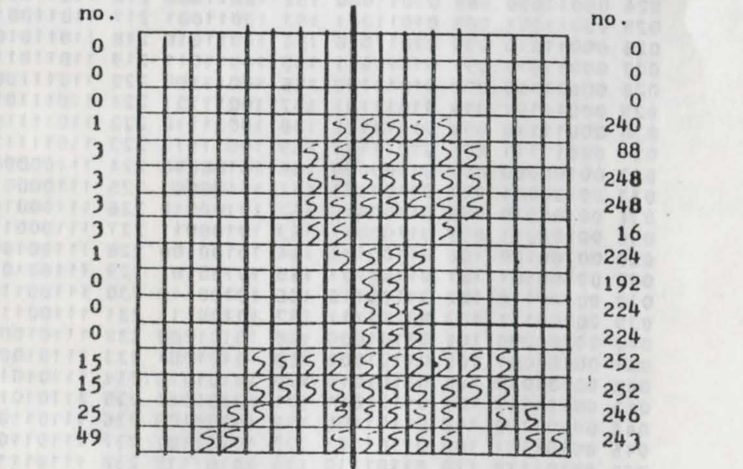


Diagram of grid for ׀skin" sprite. Numbers correspond to the DATA in line number 9943-9944.

BIT PATTERN TABLE

no.	pattern	no.	pattern	no.	pattern	no.	pattern
000	00000000	064	01000000	128	10000000	192	11000000
001	00000001	065	01000001	129	10000001	193	11000001
002	00000010	066	01000010	130	10000010	194	11000010
003	00000011	067	01000011	131	10000011	195	11000011
004	00000100	068	01000100	132	10000100	196	11000100
005	00000101	069	01000101	133	10000101	197	11000101
006	00000110	070	01000110	134	10000110	198	11000110
007	00000111	071	01000111	135	10000111	199	11000111
008	00001000	072	01001000	136	10001000	200	11001000
009	00001001	073	01001001	137	10001001	201	11001001
010	00001010	074	01001010	138	10001010	202	11001010
011	00001011	075	01001011	139	10001011	203	11001011
012	00001100	076	01001100	140	10001100	204	11001100
013	00001101	077	01001101	141	10001101	205	11001101
014	00001110	078	01001110	142	10001110	206	11001110
015	00001111	079	01001111	143	10001111	207	11001111
016	00010000	080	01010000	144	10010000	208	11010000
017	00010001	081	01010001	145	10010001	209	11010001
018	00010010	082	01010010	146	10010010	210	11010010
019	00010011	083	01010011	147	10010011	211	11010011
020	00010100	084	01010100	148	10010100	212	11010100
021	00010101	085	01010101	149	10010101	213	11010101
022	00010110	086	01010110	150	10010110	214	11010110
023	00010111	087	01010111	151	10010111	215	11010111
024	00011000	088	01011000	152	10011000	216	11011000
025	00011001	089	01011001	153	10011001	217	11011001
026	00011010	090	01011010	154	10011010	218	11011010
027	00011011	091	01011011	155	10011011	219	11011011
028	00011100	092	01011100	156	10011100	220	11011100
029	00011101	093	01011101	157	10011101	221	11011101
030	00011110	094	01011110	158	10011110	222	11011110
031	00011111	095	01011111	159	10011111	223	11011111
032	00100000	096	01100000	160	10100000	224	11100000
033	00100001	097	01100001	161	10100001	225	11100001
034	00100010	098	01100010	162	10100010	226	11100010
035	00100011	099	01100011	163	10100011	227	11100011
036	00100100	100	01100100	164	10100100	228	11100100
037	00100101	101	01100101	165	10100101	229	11100101
038	00100110	102	01100110	166	10100110	230	11100110
039	00100111	103	01100111	167	10100111	231	11100111
040	00101000	104	01101000	168	10101000	232	11101000
041	00101001	105	01101001	169	10101001	233	11101001
042	00101010	106	01101010	170	10101010	234	11101010
043	00101011	107	01101011	171	10101011	235	11101011
044	00101100	108	01101100	172	10101100	236	11101100
045	00101101	109	01101101	173	10101101	237	11101101
046	00101110	110	01101110	174	10101110	238	11101110
047	00101111	111	01101111	175	10101111	239	11101111
048	00110000	112	01110000	176	10110000	240	11110000
049	00110001	113	01110001	177	10110001	241	11110001
050	00110010	114	01110010	178	10110010	242	11110010
051	00110011	115	01110011	179	10110011	243	11110011
052	00110100	116	01110100	180	10110100	244	11110100
053	00110101	117	01110101	181	10110101	245	11110101
054	00110110	118	01110110	182	10110110	246	11110110
055	00110111	119	01110111	183	10110111	247	11110111
056	00111000	120	01111000	184	10111000	248	11111000
057	00111001	121	01111001	185	10111001	249	11111001
058	00111010	122	01111010	186	10111010	250	11111010
059	00111011	123	01111011	187	10111011	251	11111011
060	00111100	124	01111100	188	10111100	252	11111100
061	00111101	125	01111101	189	10111101	253	11111101
062	00111110	126	01111110	190	10111110	254	11111110
063	00111111	127	01111111	191	10111111	255	11111111

Sprite data strategies.

Adventure Creator includes the sprites actually used in the Bomb Squad game to illustrate some programming principles and to provide you with a starting point. Once you understand this part, you will be able to use some of the sprites from The Visitor if you prefer them.

The lines 9901-9959 contain the data for drawing the 20 sprites from Bomb Squad. Each two lines of data represent one sprite. Each data line contains 16 values. the first 6 lines of data are all 255. This creates three sprites that are solid blocks--sprites #0, #1, and #2. These three blocks can be made various colors for use in many different scenes. For example, one block is the bottom half of the guard house in Bomb Squad, and all three blocks are used to make up 3/4 of the gate in the picture of the dog's yard scene. Two blocks are used as furniture in the library scene, and they also serve as crates in the storage room scene. These blocks are put in low-numbered sprites so we can hide things behind them. In the guard house scene, for example, the woman is hidden in the bottom of the guard house until she appears.

The next two sprites (lines 9910-9914) are empty squares that can also be used for different scenes. For example, as the top of the guard house and as crates in the storage room.

Lines 9916-9917 are the top of a bookcase that is set on top of one of the solid blocks to make a bookcase in the library.

Lines 9919-9920 are a round bush that can also be used as the top of a tree. 9922-9923 are a leafy bush that can also be the top of a tree. 9925-9926 is a tree trunk. Of course, these three sprites can be used in different combinations with each other and in different colors to create different species of trees.

9928-9929 is the "broken part" of the gate (the lower left corner) in the dog yard scene. 9931-9932 are the book and letter on the shelf in the library. 9934-9935 make up the main body of the car in the garage; 9937-9938 make up the trim, wheels and steering wheel of the car. Of course these two sprites always are placed in the same location.

9940-9941 is sprite #13 (remember to start counting with 0), which is the woman's hair. 9943-9944 make up her face and shoulders.

9946-9947 are the man's hair and shirt, and 9949-9950 are his face. Note that these "face/hair" sprites could be used to make several different characters in different scenes by changing the color of hair and skin.

9952-9953 is one picture of a dog, and 9955-9956 is the same dog with his legs and mouth in a different position. By switching between these two sprites and moving horizontally, you can animate the dog to be walking and biting.

9958-9959 are the bomb in Bomb Squad.

As we noted in the section on initializing, all of these numbers must be POKEd into memory in exactly this order, starting at memory location 28850.

Sprite attributes

The control of each sprite depends on 4 numbers--the attributes of the sprite. Thus, for our 20 sprites, we will need 80 attributes. The first two numbers determine the location of the sprite on the screen; they are the coordinates of the upper left corner of the sprite. The first number is the vertical coordinate, and the second is the horizontal coordinate. (Note that this is opposite of using coordinates with HPLOT, where the first number is the horizontal coordinate.) The attributes of the 20 sprites in Bomb Game are in the data lines 8600-8625. Notice that in each group of 4 numbers, the first two numbers are 200. This means that when the game is initialized, each sprite is at location 200 (vertical) and 200 (horizontal). This is below the visible part of the screen, so the sprites are hidden, waiting to be used.

The third attribute is the sprite number. If you are using 8x8 sprites, this number will simply be 0, 1, 2, or whatever the actual number of the sprite is. However, if you are using 16x16 sprites (as we are), the actual number you must use is the sprite number multiplied by 4. The reason for this is that ADAM uses this number to know where to look for the correct sprite data in the data table for determining shapes. The 8x8 sprites each use 8 bytes for data, but the 16x16 sprites use 32 bytes each. Thus, in the data statements in lines 8600-8625, the "sprite number attribute" of the first sprite is zero (sprite #0 times 4). The second sprite in the list is sprite #1, so the "sprite number attribute" must be 4. The next "sprite number attribute" is 12 etc.

The last attribute determines the color of the sprite. Unfortunately, the colors do not correspond directly to the color-numbers used in BASIC.

Sprite Color Attribute Table

Attribute #	Color	Attribute #	Color
0	transparent	8	med. red
1	black	9	lt. red/peach
2	med. green	10	dk yellow
3	lt. green	11	lt yellow
4	dk. blue	12	dk. green
5	lt. blue	13	magenta
6	dk. red/orange	14	white
7	cyan	15	gray

Notice in the program that line 8630 sets the variable sa (for sprite attributes) to 29500, and then the sprite attributes are POKEd into memory starting at this location. We now have our "sprite attribute table" in place. Remember also that lines 8630-8660 are set up so that in the program we can use GOSUB 8630 to set all the sprite attributes to their initial state--with all of them hidden off the screen. The main way we do this is in line 280--in the input section of the program. This line restores the screen to text, so if there is a picture there it will be erased. Just before it does this, it reinitializes all the sprite attributes and hides the sprites off the screen. If we did not do this, the next time we draw a scene, the old sprites would be visible for an instant before we draw the new scene.

You will also notice that line 280 includes CALL sr. This means to call the "sprite routine", which we will discuss in a moment.

Placing the sprites in a scene.

We are finally ready to put sprites in our picture. We can now return to line 6250 in the "framework program". Up to this point, we had used HPLOT to outline our picture. Line 6250 is going to POKE new numbers into the "sprite attribute table" at memory location 29500; then we will call the "sprite routine" subroutine, which will move the sprites around and change their colors, depending on the numbers in the "sprite attribute table".

The address of the "sprite attribute table" is given to variable sa (for sprite attribute). Thus, in line 6250, we see the command

POKE sa,95: poke sa+1, 160

Let's understand these two POKEs before looking at the rest of this line. POKE sa,95 puts the number 95 into the first location in the sprite attribute table. This will then become the vertical coordinate of the first sprite. POKE sa+1, 160 changes the second number in the table, which is the horizontal coordinate of the first sprite (remember that the first sprite is #0). (Later, in line 6270, we will CALL sr, and sprite #0 will move onto the screen to become the bottom of the guard house).

The next POKE sa+3, 14 changes the color of this sprite to white, which is the color we want for the guard house. Notice that we skipped one number in order to set the color. In each set of four attributes, the first two are always the location coordinates, the third is the sprite number (multiplied by 4 if you are using 16x16 sprites) which you never change, and the fourth is the color number. Since sprite #0 is used for many different purposes, we will have to change its color each time, depending on what it is supposed to be in the picture.

The next POKE sa+16, 79 and POKE sa+17, 160 sets new coordinates for sprite # 4. How do we know that these are the coordinates for sprite #4? Simply by mutliplying the sprite number by 4. Then this number and the next one are the location coordinates. In this case we don't have to change the color number, because we set the color of sprite #4 to white when we initialized the sprite attributes.

After lines 6250-6265 POKE in all the coordinates and new colors wanted, line 6270 finishes the picture by CALL sr, the sprite routine.

Animating the sprites

In our example scene, line 6275 checks to see if the guard has been "disabled". If he has, nothing further happens (IF f(45)=1). However, if he is in his "normal" state (f(45)=0), then we have to animate the picture, to show that something fishy is going on in the guard house. The animation shows the woman standing up to look out the window, quickly hiding and then looking out again before hiding for good. Very suspicious.

Lines 6280-6295 animate the woman. Line 6280 sets up a FOR NEXT loop, so she will go through the movements twice. Then it sets up a FOR NEXT loop that will automatically change the vertical coordinates for the "hair" and "face" sprites that make up the woman. This loop goes from 95 to 79 in steps of -3.

In line 6286 we POKE sa+52, L and POKE sa+56, L ;then we call the sprite routine; then we stall briefly to smooth out the movement. Thus, the first time through, L will equal 95 and both the "hair" and "face" will still be hidden. The second time through this FOR NEXT loop, L will be 3 less, that is:92. Then the vertical coordinates of the "hair" and "face" sprites will be 3 higher,and the woman will start to rise in the window.

As soon as the loop finishes a second one starts, with the woman hidden again, so it looks like she quickly ducked down.

Line 6283 is simply included to put in a delay when she is hidden.

Line 6295 returns her to the hidden position before the RETURN from this graphics routine.

The general principle.

Bascially, then, animation is easy; it is just tedious, because you have to keep changing various coordinates and then CALLing the sr routine.

Assembly language listing for sprite control

We have made frequent reference to the "sprite routine" which uses the sprite attributes to move around the sprites. We will describe this routine in some detail. Those of you familiar with assembly language should end up with an intimate knowledge of sprite control. But even if you don't know assembly language, we will try to explain things so you can use sprites more flexibly.

The sprite routine consists of the numbers in the data statements in lines 10010 and 10020. Lines 10030-10050 POKE these numbers into memory starting at memory location 29600. This is why the variable sr equals 29600; whenever we CALL sr, this routine is called. We will give you an assembly language listing of the program with extensive comments.

```
Assembly Code      Decimal values from data statements
LD C,E2H           14, 226
LD B,1             6,1
CALL FD20          205,32,253
```

****comment: these lines set the magnification and size of the sprites. The critical value is the underlined one--in this case 226 will make the sprites 16x16 with no magnification. Changing this number to 227 will give 16x16 with double magnification. 224 gives 8x8 with no magnification. 225 gives 8x8 with double magnification.

```
LD A,(FC18H)       58,24,252
LD IY,0050H        253,33, 80, 00
```

****comment: this sets the number of entries to be used. In our examples, we have 20 sprites of 16x16, so we have to use the number 80 in the underlined value here--that is, the number of sprites multiplied by 4. If one were using 20 8x8 sprites, this number would simply be 20.

```
LD HL,70B2H        33,178,112
```

****comment: load HL with the address of the sprite data (28850 decimal). You will really need to understand assembly language to change this address, so you may just want to use it as in the sample program.

```
LD DE,0000H        17,00,00
```

****comment: which entry should the routine start writing to? We find it very confusing to try to change this value, so we always start with entry #0 and rewrite all of the sprites every time, rather than trying to pick out just a few to rewrite. The process is so fast that it makes no practical difference to start at entry #0 each time. We recommend leaving this value alone.

```
CALL FD2CH         205,44,253
```

****comment: write these entries into VRAM (video RAM)

```
LD A,(FC17H)       58,23,252
```

****comment: set up table#0--the entry point for attribute setting.

```
LD IY,0014H        253,33, 20, 0
```

****comment: enter the number of sprites being used. Obviously the underlined number in this line would be changed to change the number of sprites.

```
LD DE,0000H        17,00,00
```

****comment: entry to write to again

```
LD HL,733CH        33,60,115
```

****comment: location address of the sprite attributes data (29500 decimal)

```
CALL FD2CH         205,44,253
```

****comment: write attributes table to VRAM

```
RET                201
```

In the data statements in the framework program, the last number is 256, but this is not part of the sprite routine. It is just there to signal the end of the data.

This should give you the information you need to add dramatic graphics to all of your programs.

Problems with BASIC

We need to warn you about two problems with smartBASIC. The first one seems to be a bug that appears with large programs that push the limits of memory--which your program probably will do. In the program, the string variable m$ is used to give feedback to the player. Occasionally, the first several letters of m$ will be skipped and random letters added on to the end. We can find no way around this, and just warn the player to simply try the command again.

The second problem can be a real nuisance. For reasons we will never understand, each time a program is LOADed from tape or disk, BASIC adds a space immediately after each DATA statement and each REM statement. If you then make changes to the program and SAVE the modified version--as you often will when you develop a program--the extra spaces are also SAVEd. The next time you LOAD the program, another space will be added. In a program with many DATA statements, several SAVEs and LOADs can waste a lot of memory, and adventure games usually have little memory to spare. We find it necessary, every now and then, to list each DATA line and edit it. To do this, move the cursor under the line number, erase the "DATA" with the space bar, move the cursor along until it is five spaces from the actual data, and type in a new DATA command.

Another problem is not a shortcoming of BASIC, but it can be the source of a hard-to-find bug. At the end of line 160 we add one space to q$, which is the player's input. We put the space at the end of q$ because later the parser won't be able to tell the difference between "book" and "bookcase". In the vocabulary, book is listed with a space at each end:

" book ". Now it won't be confused with codebook or bookcase, as long as there is a space at the end of the player's input like "read book ". However, now we have a problem to watch for in line 2470, where we want to know if the player said "take pictures". We must say

```
  if q$="take pictures " then m$="using what?"
```

being careful to include the space at the end of "take pictures ", because q$ will have a space there and BASIC is very fussy. When you ask if q$ equals something, it must be exactly like q$.

BOMB SQUAD

In this game, you have been appointed to find and defuse three bombs that have been placed in the embassy of tiny Lunaria, the only country with large known reserves of kryptonite. As with most adventure games, you will be entering two word English commands to find your way around the embassy, to gather whatever supplies you might need, and to deal with any situation a good intelligence/explosives expert might meet.

You can move around any of the four directions, N, S, W, or E-- if there is a visible exit available. You can do this either by typing in a command like "go south" or "walk east" or, to save time and typing, you can simply enter one of the letters N, S, W, or E. These one-letter commands must be in upper case letters.

The computer will describe your location and what you can see at each turn. You might try to accumulate objects for later use, but there is a limit to what you can carry. If you leave something in a room, it will be there waiting for you when you return.

If you try to do something, and the computer tells you that you can't do that, try a synonym. Remember that each command you enter should include exactly two words--a verb and a noun (with the exception of the one-word direction commands or "save", "load", "help", or "carrying?".)

Unlike most adventure games, Bomb Squad puts you under some time pressure. If you don't find the bombs fast enough, they will start exploding one by one. Even if this happens, though, remember what you learned from your experiences, so next time you will have a better chance of finding all three bombs.

As a general strategy, it is essential to draw a map of your searches as you go.

Saving and loading.

If you are in the middle of a game and have to stop, you can enter the one word "SAVE" (be sure there is a tape or disk in place) and your current position will be SAVEd as a file named "bombgame". When you want to resume the game, LOAD the program as usual, RUN it, and then enter the one word command "load". This way, you don't have to start over from the beginning of the game.

You may also want to SAVE the game occasionally as you play, so if you get killed or get into a hopeless situation, you won't have to start over.

A bug in BASIC.

Occasionally, you might read a sentence with part of the front cut off and gibberish at the end. This seems to be a string handling bug in BASIC. Just try your command again if you can't figure out what the feedback sentence means.

Graphics clues.

Some of the scenes are presented graphically, so be sure you study the scene and the action in the scene to help you figure out the best course of action.

Getting help in the game.

If you need to know what you are carrying, simply enter the word "carrying?", and the program will list your possessions. Although it is considered bad form, you can even ask for a list of the words that the game understands by entering the one-word command "help". It's bad form because it makes the game too much easier, but if you really need to, far be it from us to make you feel guilty--just because you lack character.

A deadly warning.

It is possible to survive some of the explosions if you don't find certain bombs in time--but it is also possible that you might be in the same room at the very moment one goes off. In that unlkely event, there is nothing to be done but start the game again.

A Bug in BASIC-
Occasionally, you might read a sentence with part of the front cut off and gibberish at the end. This seems to be a string handling bug in BASIC. [illegible] your command again if you can't figure out what the [illegible] sentence means.

Graphics Clues-
Some of the scenes are presented graphically. Be sure you study the scene and the action in the scene to help you figure out the best course of action.

Getting help in the game-
If you need to know what you are carrying, simply enter the word "carrying" and the program will list your possessions. Although it is considered bad form, you can even ask for a list of the words that the game understands by entering the one word command "help". [illegible] bad form because it makes the game too much easier. [illegible] if you really need to, [illegible] of 50 moves you [illegible] because you lack character.

A Little Caution-
It is possible to [illegible] of the [illegible] if you don't [illegible] also possible that you might [illegible] and wander off. In [illegible] start the game again.