Bill Budge's

3-D GRAPHICS SYSTEM

AND

GAME TOOL

Top Of The Orchard

software

N O T I C E

A credit notice is required for use of this software to create programs for resale.

Bill Budge and California Pacific Computer Co. intend that this program be used by programmers to create games and simulations. Should anyone wish to market or sell product created with the use of this program permission is hereby given providing that a credit notice is prominently placed on the packaging, on the documentation, and on a text screen in such a manner that users of the program shall, in normal operation of the program, see said credit notice. Credit notice shall read: "3-D effects and animation created with the use of Bill Budge's 3-D Graphics System & Game Tool, copyright 1980, California Pacific Computer Co.".

<u>Prohibition</u>: Use of this program, or any part thereof, in a graphics software package or program for sale or resale is expressly prohibited.

INTRODUCTION TO BILL BUDGE'S APPLE II GAME TOOL

The Apple II Game Tool is an interactive three-dimensional graphics system that has been specialized to produce displays for video games. With it, any programmer can add 2-D or 3-D animations to his or her programs to create games and simulations that are as good as anything now being sold for the Apple II. How is this possible?

First, the Game Tool contains 3-D drawing software that is fast. In order to achieve the illusion of smooth motion, an animation should run at 10 frames per second or faster. The Game Tool graphics software can go that fast even when drawing fairly large objects, and small animations have been clocked at over 40 frames per second. To round out the package, special utilities are provided to draw missiles and characters on the screen.

Second, the Game Tool includes interfaces to Integer and Applesoft Basics that are clean and easy to use. This means that Apple users who don't know assembly language can use the graphics software almost as effectively as assembly language programmers can. The speed measurements quoted above were made using Integer Basic driver programs.

The capabilities of the Game Tool will make it useful to advanced and even professional programmers. What is really exciting, however, is that you don't have to be an expert to use it. Now even novice programmers can create impressive animated graphics. Never before has a 3-D graphics package been optimized specifically for game graphics and then made available to such a wide range of users. The results should be interesting.

TURNING ON THE GAME TOOL

The Game Tool will run on any Apple II computer with 48K of RAM. Just boot the disk to get started. If all is well, this menu will appear on the screen:

```
ENTER COMMAND:

E -- EDIT A GRAPHICS DATA BASE
M -- BUILD A GAME MODULE
D -- RUN THE DEMO PROGRAM
```

If you want to see what the graphics software can do, type 'D' to run the Demo included on the disk. This is a binary program that shows off the 3-D drawing subroutines. It was created with the Game Tool. To stop it, just hit a key on the Apple keyboard.

The best way to get started with the Game Tool is to actually use it to build a game. To help you, the following tutorial is provided. This will give you a good idea of what the system is and what it does. Following the tutorial is a more detailed reference section which should be read whenever questions about specific parts of the system arise.

A TUTORIAL INTRODUCTION TO THE APPLE II GAME TOOL

It is assumed that the reader is already familiar with at least one of the Apple II Basic languages (Integer or Applesoft). Though an uninitiated user can create and view shapes using the Game Tool, this software is intended as a development tool for programmers. This tutorial begins by showing how to create a version of the classic "Lunar Lander" video game. This game was chosen because it requires only simple 2-D graphics, which makes the discussion as simple and uncluttered as possible. However, you will see that the Game Tool is very effective even when restricted to two dimensions. Most video games use only 2-D animation. Even when the third dimension is required, it can often be faked with a 2-D software package. Since the Game Tool has true 3-D capability, however, the tutorial ends with a simple 3-D example: a real-time simulation of complex rotations of the Space Shuttle Enterprise.

The first step in creating our Lunar Lander game is to form descriptions of the graphics objects we want drawn. These descriptions must be in a form that the drawing programs understand; together the descriptions are called the graphics data base. An editor is provided as part of the system to make this step easier.

The next step is to put the newly created data base together with the 3-D graphics software to get a binary module that will support our application program. This module will do all the hi-res graphics for us. Making the module is trivial.

The last step is to program the dynamics of "lunar landing" in Integer Basic, Applesoft, or assembly language. This program is rather small becauae it doesn't have to do any graphics; about all it does is maintain a few variables describing the lander's velocity and position. The module does the rest.

These three steps are necessary for any application of the Game Tool, and several iterations may be necessary before the game or simulation is done. This is the model for using the system:

Step 1: Using the editor, create or modify a graphics data base.

Step 2: Build a module by adding graphics software.

Step 3: Write the application program.

(If more or different objects are needed, go back to Step 1.)

Let's create the graphics data base for Lunar Lander. We will need two objects: a landing vehicle (LEM), and a cross section of the Lunar surface to serve as a background. A sketch is a good first step in defining these objects.

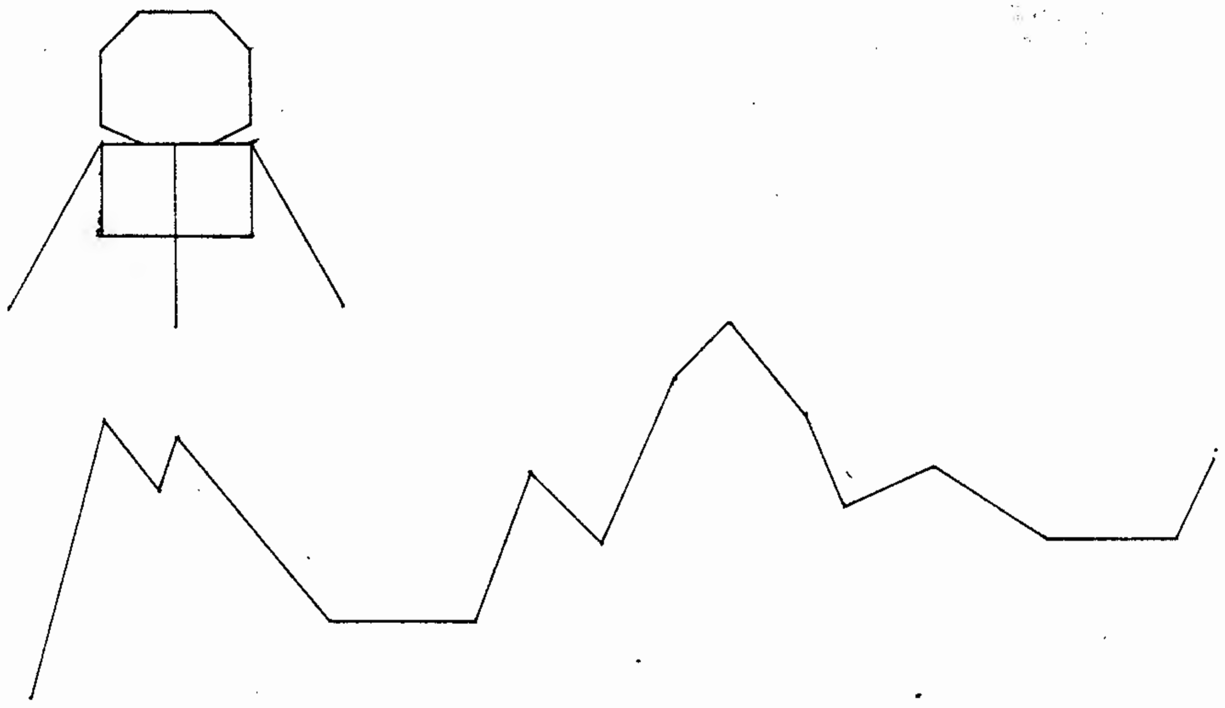

Figure 1

The LEM object will move around on the screen depending on how much thrust the game player applies (using a game button). To make the game interesting, it will be necessary to control the LEM's approach angle with the game paddle (to draw the 'tilted' LEM we will use just one of the three rotations that the 3-D package can perform). The lunar cross section will serve as a background. Once it is drawn, it will not move or rotate or change size, though these are interesting and possible alternatives.

As it now stands, the system cannot accept information in sketch form. Instead, we must break the sketch down into its component pieces, and enter those pieces via the keyboard. This process is more tedious that entering the data base with a graphics tablet or a light pen, but it has the advantage that it will work with any Apple computer system. There are three kinds of pieces that we need to consider: (1) objects, like the LEM and the lunar surface, (2) lines, and (3) points. Breaking the drawing of Figure 1 into objects, points and lines is easy; the result (with each part numbered for identification) is shown in Figure 2.

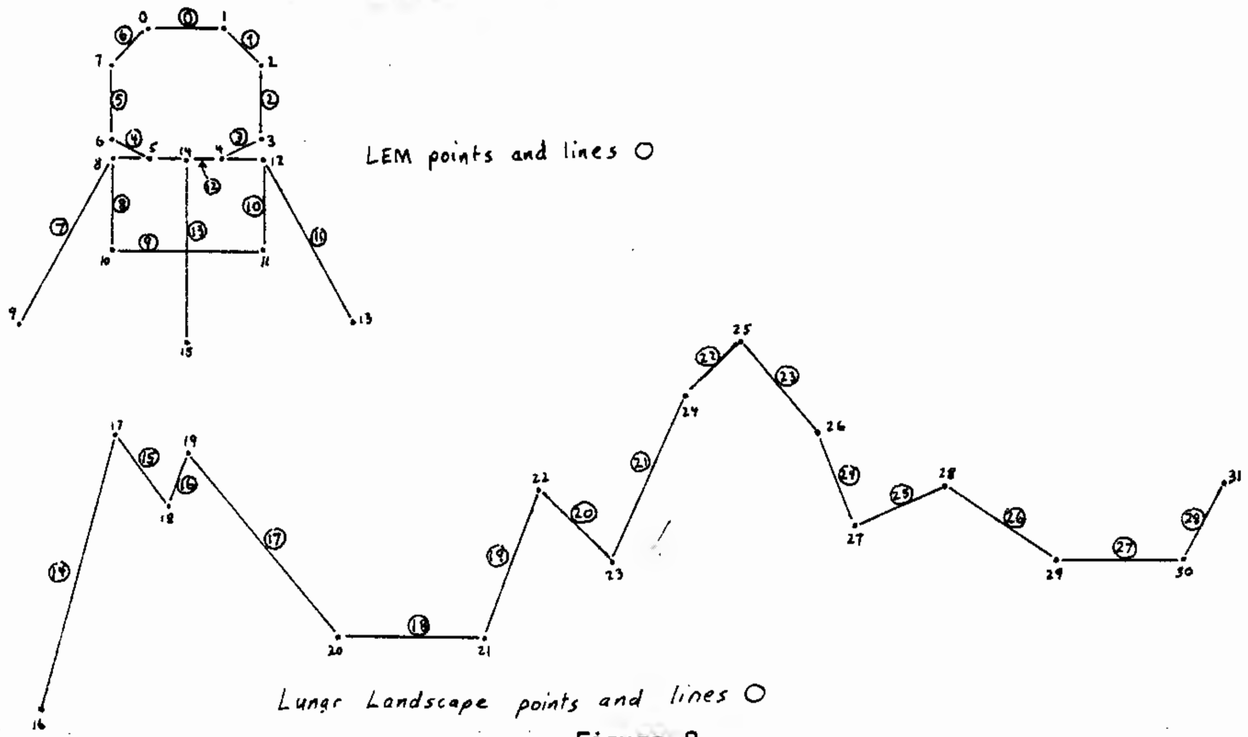

Figure 2

All that we have to do now is form a description of each object, point and line.

An object is described by giving the range of points and lines that it contains, ie., Object 4 contains points 20 through 40, and lines 25 through 50.

A line is described by giving its two endpoints, ie., Line 4 connects points 8 and 9.

A point is described by specifying its position in Cartesian Coordinates, ie., Point 78 has coordinates -10,10. (If you are unfamiliar with coordinate systems, see Appendix A.)

Figure 3 shows the LEM and lunar surface objects with superimposed coordinate axes and with the coordinates for each point written in. Note that each object has its own set of axes, which can be positioned in many different ways. In the case of the LEM, the axes meet at its center of gravity. Since the 3-D package always rotates around the origin, this insures that the LEM will rotate in a convincing way. The completed data base for Lunar Lander is given in Table 1.

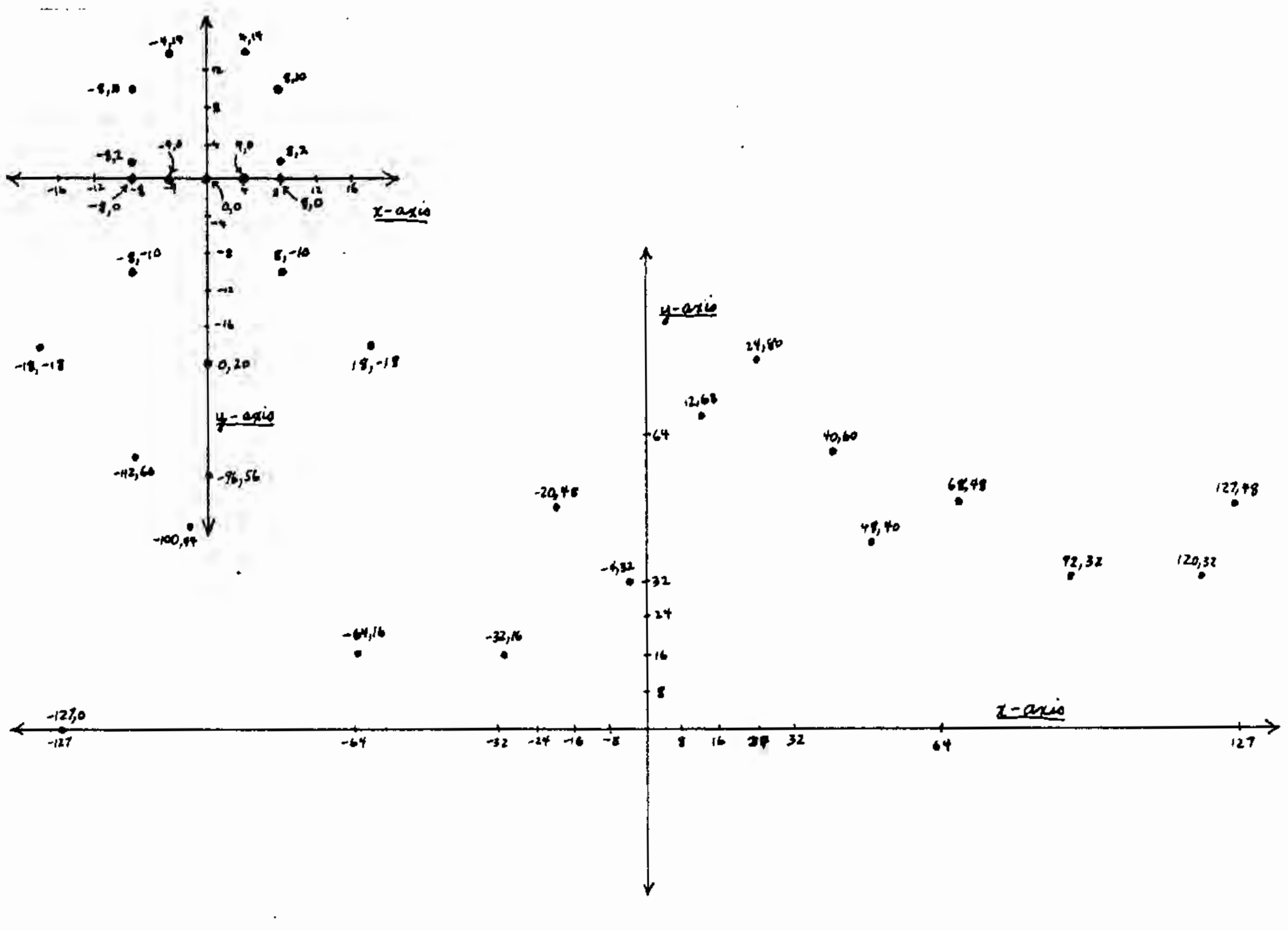

Figure 3

Object 0 contains points 0-15, lines 0-13.
Object 1 contains points 16-31, lines 14-28.

Point 0 is at -4, 14	Line 0 connects points 0 and 1
Point 1 is at 4, 14	Line 1 connects points 1 and 2
Point 2 is at 8, 10	Line 2 connects points 2 and 3
Point 3 is at 8, 2	Line 3 connects points 3 and 4
Point 4 is at 4, 0	Line 4 connecte points 5 and 6
Point 5 is at -4, 0	Line 5 connects points 6 and 7
Point 6 is at -8, 2	Line 6 connects points 7 and 0
Point 7 is at -8, 10	Line 7 connects points 8 and 9
Point 8 is at -8, 0	Line 8 connects points 8 and 10
Point 9 is at -18,-18	Line 9 connects points 10 and 11
Point 10 is at -8,-10	Line 10 connects points 11 and 12
Point 11 is at 8,-10	Line 11 connects points 12 and 13
Point 12 is at 8, 0	Line 12 connects points 8 and 12
Point 13 is at 18,-18	Line 13 connects points 14 and 15
Point 14 is at 0, 0	
Point 15 is at 0,-20	

Point 16 is at -127, 0	Line 14 connects points 16 and 17
Point 17 is at -112, 60	Line 15 connects points 17 and 18
Point 18 is at -100, 44	Line 16 connects points 18 and 19
Point 19 is at -96, 56	Line 17 connects points 19 and 20
Point 20 is at -64, 16	Line 18 connects points 20 and 21
Point 21 is at -32, 16	Line 19 connects points 21 and 22
Point 22 is at -20, 48	Line 20 connects points 22 and 23
Point 23 is at -4, 32	Line 21 connects points 23 and 24
Point 24 is at 12, 68	Line 22 connects points 24 and 25
Point 25 is at 24, 80	Line 23 connects points 25 and 26
Point 26 is at 40, 60	Line 24 connects points 26 and 27
Point 27 is at 48, 40	Line 25 connects points 27 and 28
Point 28 is at 68, 48	Line 26 connects points 28 and 29
Point 29 is at 92, 32	Line 27 connects points 29 and 30
Point 30 is at 120, 32	Line 28 connects points 30 and 31
Point 31 is at 127, 48	

Table 1 (Lunar Lander Data Base)

This description is a lot like a connect-the-dots puzzle. It is in a very convenient form as far as the 3-D drawing software is concerned. Coding this description can be a nuisance for large objects, but for games in general, this phase of development takes only an hour or two. To summarize, the graphics data base went through the following stages:

Stage 1: A drawing or mental image of the graphics for the application.

Stage 2: The same representation, divided into objects, points and lines.

Stage 3: A textual description of each object, point and line.

The data base we created can now be entered into the Apple. It would be a lot of trouble to load this data into the Apple's memory by hand, and so an editor is provided. It makes entering and changing shapes much easier, and provides an easy way to examine shapes and transfer them to and from a disk. If you have booted the Game Tool disk, you can now type 'E' to run the editor.

One of the first things you will notice is that the editor uses the top lines of the screen to display its status. Whenever it expects a prompt, it will list the current commands or modes which can be selected; whenever you issue an illegal or impossible command, it will tell you so. Another convention is that the <esc> key will always get you out of whatever command you are executing, if it is the first character in an input line.

Type 'P' to see an error message. No command begins with a 'P'. Now type 'A' to append, followed by <esc>, which gets you back to the top command level. What we want to do is create a brand new data base, so we will append objects, points and lines to the editor's empty memory. Type 'AP' to get into append-point mode. The editor will prompt you for point 0 (printing '0:') and then accept the input line you type to describe point 0. Because this is a 3-D system, it will expect the point to have three coordinates, so follow the X and Y coordinates for each point with a zero (and end each line with a <cr>). Enter Lunar Lander's 32 points as follows: (the numbers come from Table 1)

```
 0:-4,14,0
 1:4,14,0
     .
     .
     .
31:127,48,0
```

Type <esc> to break out of this seemingly infinite loop. Remember that the <esc> has to be the first character you type in an input line. You should still be in append mode; type 'L' to append lines. Enter all the lines in the data base just like you appended the points:

```
 0:0,1
 1:1,2
    .
    .
    .
28:30,31
```

Type <esc> when the editor prompts you for line 29. Then type 'O' to append the objects to the data base:

```
0:0,15,0,13
1:16,31,14,28
```

Type <esc> <esc> to get back to the top command level of the editor. To list the data base, type **'L'**. This command requires a range, so it gives you a prompt:

RANGE: "X" "X,Y" <cr> <esc>

You can list one point and one line by typing a single number **"X"**, or you can list a range of points and lines by typing the range **"X,Y"**. To list everything just hit <cr>, and to escape - you guessed it, <esc>. List the whole data base now to make sure you entered everything correctly; use the spacebar to keep the text from flying past; the space bar allows you to list line by line.

Before going any further, we should see what our shapes look like. Ordinarily, this is a very important step, as this is the easiest way to make sure that the data base is as we want it; in this contrived example though, the shapes were correct before you typed them in. Get into the show mode (type **'S'**) and position the objects on the screen (type **'P'**). This is important, because we want the lunar surface at the bottom of the screen, and we don't want either object to be going off an edge of the screen. The fast graphics software does no "clipping" of lines that extend off screen, and the resulting effects aren't usually what we want. The position mode prompts you for the number of the object which you are positioning. Type '0' following by <cr>. The screen should now show the lunar lander somewhere. Use the game paddles to move it around; notice what happens when part of it hits a screen boundary. This is something to be avoided.

Position the LEM near the center of the screen. Type <esc>, then position object 1, the lunar surface. Put it down near the bottom of the screen - it has to be precisely centered, or you will get unwanted effects. Type <esc> to get to show mode again. This time, type 'T' to perform 3-D transforms on the LEM. In this mode, paddles 0 and 1 control the two kinds of rotation that don't make sense in 2-D worlds. Play with the paddles but then leave them at their zero points (all the way counterclockwise). The rotation we are interested in is controlled by the **'Z'** key (Z for rotation around the Z-axis). Hit this key repeatedly to rotate the LEM. Hit the <u>cursor keys</u> to change the scale. If you set the paddles correctly, and entered the shape description with no mistakes, the LEM should look just like it did in the drawing in Figure 1. (If you made a mistake, use the 'change' command. See also the editor technical section.)

Type <esc> <esc> to get out of the show mode. Try listing the data base. It is unchanged, even though you probably left the LEM in a rotated or scaled state. The 3-D transformations never change the actual data base.

That's enough editing for now. Save this data base using the disk write command (type **'W'**). Replace the Game Tool system diskette with one of your own initialized diskettes, and for the filename prompt type **LUNAR LANDER**. This completes the definition of the graphics for our game, until we wish to add more objects, or change the ones we just entered.

Put the Game Tool disk back in the drive and quit the editor by typing <esc> and then **'Y'** to verify that you want to quit.

PUTTING THE PIECES TOGETHER: BUILDING A MODULE

When you quit the editor, the executive prompts you for your next command. We want to build a module, so type '**M**'. The program now loads one of two drawing methods a module can use. The simplest method is 'OR' drawing. The other method, however, called 'XOR' (exclusive-or) drawing has the nice property that it does not wipe out any background designs on the screen. Lunar Lander requires a moonscape in the background so our choice is XOR-drawing. If we used OR drawing, we would have to refresh the background each frame, because the LEM might have been drawn over it. This would slow down the animation.

Include the XOR-drawing version of the drawing software by typing '**X**'.

The next choice regards the interface. Since there are differences in the way arrays are implemented in Integer Basic, Applesoft and assembly language, the interface programs for these languages are not the same. Type '**I**' if you are using Integer Basic, '**F**' if you are using Applesoft. Assembly language programmers should pick their favorite Basic.

Leave out hi-res text and missiles for now (type '**N**' to both prompts).

Finally, add our data base, by typing **LUNAR LANDER** for the shape and inserting your data base diskette. For the module name, type **LUNAR LANDER** again. Save the module on the same disk that the shapes are on. That's all there is to building a module.

WRITING THE GAME PROGRAM

Before we write the Lunar Lander program, let's play with the module. The first thing to do is type in a Basic "header". Remove the Game Tool diskatte and boot up a normel Basic diskette. A header is just a series of statements which help you to interface with the graphics module. When you write your game program in Basic, you must add it to these header statements if it is to communicate correctly with a module. There is a header for Integer Basic and one for Applesoft. If you are going to use Applesoft for your programs, skip the rest of this section, and read the next one entitled "Writing the Game Program in Applesoft".

Here is the Integer Basic header program:

```
0 POKE 74,0:POKE 75,96:POKE 204,0:POKE 205,96
1 DIM CODE(15),X(15),Y(15),SCALE(15),XROT(15),YROT(15),
    ZROT(15),SCRNX(15),SCRNY(15)
2 DIM MCODE(15),MX(15),MY(15),MDX(15),MDY(15)
3 RESET=7932:CLEAR=7951:HIRES=7983:CRUNCH=7737:
    MISSILE=7993:TXTGEN=768
4 D$="":REM SET D$ TO CTRL-D
5 PRINT D$;"BLOAD MODULE."
```

You might want to **SAVE** this program as **INT/HEADER**, to avoid having to type it in again the next time you start a game.

The first thing to change is statement 5 so the header will load the module we defined for our game.

Type **5 PRINT D$; "BLOAD MODULE.LUNAR LANDER"**.

Run the program. Integer Basic will give you a *** NO END ERR, but don't worry about it. The header forces your program to reside at a location in memory which is above the second hi-res page, so it won't get wiped out by tha graphics. It also sets up some important arrays in a way that makes them available to the graphics module. Do not change any part of the header; you will probably change the locations the array elements, which will confuse the module. Statement 3 defines some labels; these are the calls you can make to the module. The first routine, RESET, must be called before you can use any of the other subroutines. It clears both hi-res screens and initializes the module.

Try this now. Type **CALL RESET** in immediate mode. This has the side effect of turning on the hi-res graphics, so type **TEXT** to get back. Now you can play with your shapes. From Basic, shapes are controlled by changing the entries of the arrays dimensioned in statements 1 and 2. Each array has sixteen entries, one for each object. CODE(0), for example, is the command code for object 0, the first object in the data base.

The arrays are the only way you can tell the graphics module what you want, and the only way it can communicate with your program. There are no PEEKS or POKES required. Here is what some of the arrays are for:

CODE(i) - tells the graphics module what to do with object i.

X(i) - tells the graphics module where to place object i on the hi-res screen, horizontally.

Y(i) - tells the graphics module where to place object i, vertically.

SCALE(i) - tells the graphics module how big object i should be.

XROT(i) - We won't use this just now.

YROT(i) - Or this.

ZROT(i) - tells the graphics module how much object i should be rotated around the z-axis.

SCRNX(i) - this entry is filled in by the module. It tells you where on the screen the last point of object i fell, horizontally.

SCRNY(i) - same as SCRNX, only it's the vertical position of the last point plotted for object i.

These arrays provide a convenient place to put parameters for the subroutines in the module. The subroutines get invoked by calls using the labels defined in Statement 3. The following animation functions are provided:

RESET - initializes the module, clears the CODE array and clears both hi-res screens.

CLEAR - clears both hi-res screens.

HIRES - turns on the hi-res graphics.

CRUNCH - creates a new frame of animation on one hi-res page, while displaying the other page.

MISSILE - adds missile-like objects to the animation.

TXTGEN - turns on a "built-in" hi-res character generator, so you can put text into your animation.

The real workhorse of the command group is CRUNCH, which does all the 3-D drawing. Before calling it, you should initialize the array entries for each shape. To see how CRUNCH works, let's draw in the background for Lunar Lander. First, do these assignments (in immediate or deferred mode):

```
>CODE(1)=1

>X(1)=127

>Y(1)=191

>SCALE(1)=15

>XROT(1)=0

>YROT(1)=0

>ZROT(1)=0
```

Setting CODE to 1 is the usual way to introduce an object to the screen. This code causes CRUNCH to transform the object first, and then to draw it. It will not erase any old image of the object. This is exactly what we want, since nothing has been drawn yet. In fact, erasing would not only be wasted effort, but a mistake, since we are using the XOR drawing method. (Remember that XOR toggles the bits to erase and to draw.)

The position of the object is controlled by the X and Y array entries and with the above assignments, the lunar surface will be drawn horizontally centered, and at the bottom of the screen. (More precisely, the origin of the lunar surface's coordinate system is centered, and at the bottom of the screen.)

The scale of the object is set to full size.

The rotation variables are set for a normal orientation (no rotation).

There is no assignment to the SCRNX and SCRNY arrays. These are filled in by CRUNCH.

Now the moonscape can be drawn. We call CRUNCH twice, in order to draw it on both hi-res screens. This way, the object will not flicker when the screens switch during animation. Perform the following calls:

```
>CALL HIRES       Do this to get back to the hi-res mode.

>CALL CRUNCH      Type carefully, since you can't see
                  what you're typing.

>CALL CRUNCH      Nothing appears to happen; the screen
                  switch is quick.
```

```
>POKE -16299,0   Switch pages to make sure it's on both
                 screens.

>POKE -16300,0   Back to page 1.

>CALL CRUNCH     Just for fun, see how XOR drawing can
                 be weird?

>CALL CRUNCH     Now both screens are blank.

>CALL CRUNCH     Draw the moon all over again.

>CALL CRUNCH     Now both screens again contain the
                 drawing.

>TEXT
```

Now that our background is on the screen, we can forget about it; this is the nice thing about XOR drawing. Set CODE(1)=0 to prevent CRUNCH from drawing it any more. Let's introduce the LEM in exactly the same way as we introduced the moonscape:

```
>CODE(0)=1

>X(0)=227        This will put the LEM on the right
                 side of the screen.

>Y(0)=48         This puts it near the top.

>SCALE(0)=15

>XROT(0)=0

>YROT(0)=0

>ZROT(0)=3       Tilt it a bit.

>CALL HIRES      Turn on graphics again.

>CALL CRUNCH

>CALL CRUNCH
```

Now the LEM and the moon are on both hi-res pages. To animate the LEM, we change its CODE entry to 2. Now each time CRUNCH is called it will first erase the LEM's old image, transform it and then draw a new image. If the code was left equal to 1, the lander would either flicker (from the toggling action of XOR), or leave a trail of old images. Once we introduce an object, normal animation requires that we do the following for each frame:

1) ERASE THE OLD IMAGE OF THE OBJECT

2) TRANSFORM THE OBJECT

3) DRAW THE NEW IMAGE

Set **CODE(0)=2**. Now you can call CRUNCH whenever you want with no unwanted effects. Try changing the X array entry for the LEM and calling CRUNCH. Do this a few times. You can't see any screen switching, and the erasing/drawing is always done on the invisible hi-res page; thus, the animation is flicker free. One thing, though - the motion is much too slow because each frame requires a lot of typing. It's time to write a program.

First type **TEXT**; then do all the initialization steps we did before, but in deferred mode:

```
10 CALL RESET
100 REM
110 REM  DRAW IN THE LUNAR SURFACE
120 REM
130 CODE(1)=1
140 X(1)=127:Y(1)=191:SCALE(1)=15
150 XROT(1)=0:YROT(1)=0:ZROT(1)=0
160 CALL CRUNCH:CALL CRUNCH
170 CODE(1)=0
180 REM
190 REM  INTRODUCE THE LEM
200 REM
210 CODE(0)=1
220 X(0)=227:Y(0)=28:SCALE(0)=15
230 XROT(0)=0:YROT(0)=0:ZROT(0)=3
240 CALL CRUNCH:CALL CRUNCH
250 CODE(0)=2
260 REM
270 REM  NOW THE LEM IS READY FOR ANIMATION
280 REM
```

There are a lot of things we could do at this point. To get the LEM to move horizontally, and to wrap around the screen when it reaches the edge, we could execute the following simple animation loop:

```
290 FOR I=227 TO 28 STEP -1
300 X(0)=I
310 CALL CRUNCH
320 NEXT I
330 GOTO 290:REM  LOOP FOREVER - TYPE CTRL-C TO STOP
```

RUN the program. The important thing to notice is that the program puts the LEM at X-coordinates that guarantee it won't straddle the edge of the screen. By a straightforward analysis, it can be shown that no point of the LEM will ever have an X-coordinate greater than 28 or less than -28.

Statement 290 will give the LEM X-coordinates that keep it at least 28 plot positions from the right or the left sides of the screen. This is a simple but important concept.

We can put any number of assignments into this loop. Suppose we want paddle #0 to control the approach angle of the LEM. Then we put this statement into our animation loop:

```
305 ZROT(0)=(PDL(0)/9) MOD 28
```

<u>NOTE:</u> ZROT must be in the range 0-27.

Add this to your program and try it out. There is virtually no limit to the things you can do, although the loop should be tight enough that the animation will be fast. The final version of the Lunar Lander game is just such an expansion of the basic animation loop developed above. It contains statements that simulate the affects of lunar gravity and the LEM's controls, and update X, Y, and ZROT to display those effects on the graphics screen:

```
0 POKE 74,0:POKE 75,96:POKE 204,0:POKE 205,96
1 DIM CODE(15),X(15),Y(15),SCALE(15),XROT(15),YROT(15),
    ZROT(15),SCRNX(15),SCRNY(15)
2 DIM MCODE(15),MX(15),MY(15),MDX(15),MDY(15)
3 RESET=7932:CLEAR=7951:HIRES=7983:CRUNCH=7737:
    MISSILE=7993:TXTGEN=768
4 D$="":REM  SET D$ TO CTRL-D
5 PRINT D$;"BLOAD MODULE.LUNAR LANDER"
10 CALL RESET
20 GOSUB 1000:REM  THIS SETS UP SOME TABLES
100 REM
110 REM  SET UP THE SCREENS
120 REM
130 CODE(1)=1
140 X(1)=127:Y(1)=191:SCALE(1)=15
150 XROT(1)=0:YROT(1)=0:ZROT(1)=0
160 CALL CRUNCH:CALL CRUNCH
170 CODE(1)=0
180 REM
190 REM
200 CODE(0)=1
210 X(0)=227:Y(0)=28:SCALE(0)=15
220 XROT(0)=0:YROT(0)=0:ZROT(0)=3
230 CALL CRUNCH:CALL CRUNCH
240 CODE(0)=2
250 REM
260 REM  INITIALIZE THE SIMULATION - SET UP AN INITIAL
270 REM  VELOCITY AND POSITION IN SOME IMAGINARY WORLD
280 REM
290 HSPEED=-20:VSPEED=0:HPOS=29184:VPOS=3584
300 REM
310 REM  NOW READ THE CONTROLS (PADDLE #0 CONTROLS TILT,
320 REM  WHILE SWITCH #0 TURNS ON THE ROCKET ENGINE)
```

```
330 REM
340 TILT=(PDL(0)/6) MOD 28
350 IF PEEK(-16287)<128 THEN 460
360 REM
370 REM  UPDATE THE VELOCITY VARIABLES
380 REM
390 HSPEED=HSPEED+HTHRUST(TILT)
400 IF HSPEED<-512 OR HSPEED>512 THEN HSPEED=HSPEED-HTHR
     UST(TILT)
410 VSPEED=VSPEED+VTHRUST(TILT)
420 IF VSPEED<-512 OR VSPEED>512 THEN VSPEED=VSPEED-VTHR
     UST(TILT)
430 REM
440 REM  SIMULATE THE EFFECTS OF GRAVITY
450 REM
460 VSPEED=VSPEED+10:REM  A CONSTANT ACCELERATION
470 IF VSPEED>512 THEN VSPEED=VSPEED-10
480 REM
490 REM  UPDATE POSITION VARIABLES
500 REM
510 HPOS=HPOS+HSPEED:IF HPOS<3584 OR HPOS>29184 THEN
     HPOS=HPOS-HSPEED
520 VPOS=VPOS+VSPEED:IF VPOS<3584 OR VPOS>20992 THEN
     VPOS=VPOS-VSPEED
530 REM
540 REM  UPDATE THE GRAPHICS ARRAY ENTRIES ACCORDINGLY
550 REM
560 ZROT(0)=TILT
570 X(0)=HPOS/128:Y(0)=VPOS/128
580 REM
590 REM  NOW DRAW THE NEW FRAME
600 REM
610 CALL CRUNCH
620 GOTO 340
1000 DIM VTHRUST(28),HTHRUST(28)
1010 REM
1020 REM  SET UP SOME SIN/COS TABLES FOR EFFECTS OF
1030 REM  THRUST ON ROTATED LEM
1040 REM
1050 VTHRUST(0)=-30:VTHRUST(1)=-29:VTHRUST(2)=-28
1060 VTHRUST(3)=-26:VTHRUST(4)=-22:VTHRUST(5)=-16
1070 VTHRUST(6)=-8:VTHRUST(7)=0
1080 FOR I=0 TO 7:VTHRUST(14-I)=-VTHRUST(I):NEXT I
1090 FOR I=0 TO 14:VTHRUST(28-I)=VTHRUST(I):NEXT I
1100 FOR I=0 TO 28:HTHRUST((I+7) MOD 28)=VTHRUST(I)
1110 NEXT I
1120 RETURN
```

Enter this version and **RUN** it if you like. To stop the program, you'll heve to type **CTRL-C**. Type **TEXT** to turn off graphics; a **POKE -16300,0** may be necessary if the program was interrupted while displaying the secondary hi-res page.

The game is really nothing more than an elementary simulation of a lunar lander. There are of course many features missing; some of these would make the game much more interesting. We could, for instance, have several "moonscapes", and when the LEM hit a screen edge, the program would scroll from one scene to another. A particularly nice feature is "auto-zoom", where the LEM and a portion of the moon are automatically magnified when landing (or crashing) is imminent. If we really wanted to get fancy, we could expand the game by adding meteors, and other space vehicles (friendly or otherwise). These and other special effects can be achieved either as direct extensions of the techniques we have covered, or by clever tricks. See Appendix B for a list of special effects and how to create them.

We won't develop this game any further, such extensions being outside the scope of a tutorial. The important point is that you can extend the game to include many more features with only a modest effort. Games are much easier when the graphics are taken care of.

WRITING THE GAME PROGRAM IN APPLESOFT

Here is the Applesoft header:

```
1 DIM CODE%(15),X%(15),Y%(15),SCALE%(15),XROT%(15),
    YROT%(15),ZROT%(15),SX%(15),SY%(15)
2 RESET%=7932:CLR%=7951:HIRES%=7983:CRNCH%=7737:
    TXTGEN%=768
3 D$=CHR$(4):REM  SET D$ TO CTRL-D
4 PRINT D$;"BLOAD MODULE."
```

Type these statements in and **SAVE** them under the name **FP/HEADER**, so you don't have to retype them when you start another game.

Before running the program, Applesoft's memory must be reconfigured. This will keep the module and your Applesoft program from colliding. The following series of pokes will accomplish this:

```
]POKE 24576,0
]POKE 103,1
]POKE 104,96
]POKE 175,1
]POKE 176,96
```

This unfortunately erases the program, so **LOAD FP/HEADER** (if you saved it earlier!). Before you RUN it, change statement 4 so it will load in the module we created for our game. Type **4 PRINT D$"BLOAD MODULE.LUNAR LANDER"**. Now **RUN** the program. The header places itself and the interface variables above the second hi-res page, so nothing will get wiped out by the graphics. It also defines some arrays which interface your program to the graphics module. Don't change any part of the header; you will probably change the addresses of the array elements and confuse the module. Statement 2 defines some labels; these are the calls you can make to the module. The first routine, RESET%, must be called before you can use any of the other subroutines. It clears both hi-res screens and initializes the module.

Try this now. Type **CALL RESET%** in immediate mode. This has the side effect of putting you into hi-res mode, so type **TEXT** to get back. Now you can play with your shapes. From Basic, shapes are controlled by changing the entries of the arrays dimensioned in Statement 1. Each array has sixteen entries, one for each object. CODE%(0), for example, is the command code for object 0, the first object in the data base.

The arrays are the only way you can tell the graphics module what you want, and the only way it can communicate with your program. There are no PEEKS or POKES required. Here is what some of the arrays are for:

<u>CODE%(i)</u> - tells the graphics module what to do with object i.

<u>X%(i)</u> - tells the graphics module where to place object i on the hi-res screen, horizontally.

<u>Y%(i)</u> - tells where to place object i, vertically.

<u>SCALE%(i)</u> - tells the graphics module how big object i should be.

<u>XROT%(i)</u> - We won't use this just now.

<u>YROT%(i)</u> - Or this.

<u>ZROT%(i)</u> - tells the graphics module how much object i should be rotated around the z-axis.

<u>SX%(i)</u> - this entry is filled in by the module. It tells you where on the screen the last point of object i fell, horizontally.

<u>SY%(i)</u> - same as SX%, only it's the vertical position of the last point plotted for object i.

These arrays provide a convenient place for parameters to the subroutines in the module. The subroutines are called using the labels defined in Statement 2. The following <u>animation functions</u> are provided:

<u>RESET%</u> - initializes the module, clears the CODE% array and clears both hi-res screens.

<u>CLR%</u> - clears both hi-res screens.

<u>HIRES%</u> - turns on the hi-res graphics.

<u>CRNCH%</u> - creates a new frame of animation on one hi-res page, while displaying the other page.

<u>TXTGEN%</u> - turns on a "built-in" hi-res character generator, so you can put text into your animation.

The real workhorse of the group is CRNCH%, which does all the 3-D drawing. Before calling it, you should initialize the array entries for each shape. To see how CRNCH% works, let's draw in the background for Lunar Lander. First, do these assignments (in immediate mode):

```
]CODE%(1)=1

]X%(1)=127

]Y%(1)=191

]SCALE%(1)=15

]XROT%(1)=0

]YROT%(1)=0
```

```
]ZROT%(1)=0
```

Setting CODE% to 1 is the usual way to introduce an object to the screen. This code causes CRNCH% to transform the object first, and then to draw it. It will not erase any old image of the object. This is exactly what we want, since nothing has been drawn yet. In fact, erasing would not only be wasted effort, but a mistake, since we are using the XOR drawing method. (Remember that XOR toggles the bits to erase and to draw.)

The position of the object is controlled by the X% and Y% array entries and with the above assignments, the lunar surface will be drawn horizontally centered, and at the bottom of the screen. (More precisely, the origin of the lunar surface's coordinate system is centered, and at the bottom of the screen.)

The scale of the object is set to full size.

The rotation variables are set for a normal orientation (no rotation).

There is no assignment to the SX% and SY% arrays. These are filled in by CRNCH%.

Now the moonscape can be drawn. We call CRNCH% twice, in order to draw it on both hi-res screens. This way, the object will not flicker when the screens switch during animation. Perform the following calls:

]CALL HIRES%	Do this to get back to the hi-res mode.
]CALL CRNCH%	Type carefully, since you can't see what you're typing.
]CALL CRNCH%	Nothing appears to happen; the screen switch is quick.
]POKE -16299,0	Switch pages to make sure it's on both screens.
]POKE -16300,0	Back to page 1.
]CALL CRNCH%	Just for fun, see how XOR drawing can be weird?
]CALL CRNCH%	Now both screens are blank.
]CALL CRNCH%	Drew the moon all over again.
]CALL CRNCH%	Now both screens again contain the drawing.

```
]TEXT
```

Now that our background is on the screen, we can forget about it; this is the nice thing about XOR drawing. Set CODE%(1)=0 to prevent CRNCH% from drawing it any more. Let's introduce the LEM in exactly the same way as we introduced the moonscape:

```
]CODE%(0)=1

]X%(0)=227        This will put the LEM on the right
                  side of the screen.

]Y%(0)=48         This puts it near the top.

]SCALE%(0)=15

]XROT%(0)=0

]YROT%(0)=0

]ZROT%(0)=3       Tilt it a bit.

]CALL HIRES%      Turn on graphics again.

]CALL CRNCH%

]CALL CRNCH%
```

Now the LEM and the moon are on both hi-res pages. To animate the LEM, we change its CODE% entry to 2. Now each time CRNCH% is called it will first erase the LEM's old image, transform it and then draw a new image. If the code was left equal to 1, the lander would either flicker (from the toggling action of XOR), or leave a trail of old images. Once we introduce an object, normal animation requires that we do the following for each frame:

1) ERASE THE OLD IMAGE OF THE OBJECT

2) TRANSFORM THE OBJECT

3) DRAW THE NEW IMAGE

Set **CODE%(0)=2.** Now you can call CRNCH% whenever you want with no unwanted effects. <u>Try changing the X array entry</u> for the LEM <u>and calling CRNCH%</u>. Do this a few times. You can't see any screen switching, and the erasing/drawing is always done on the invisible hi-res page; thus, the animation is flicker free. One thing, though - the motion is much too slow because each frame requires a lot of typing. It's time to write a program.

First type **TEXT**; then do all the initialization steps we did before, but in deferred mode:

```
10 CALL RESET%
100 REM
```

```
110 REM  DRAW IN THE LUNAR SURFACE
120 REM
130 CODE%(1)=1
140 X%(1)=127:Y%(1)=191:SCALE%(1)=15
150 XROT%(1)=0:YROT%(1)=0:ZROT%(1)=0
160 CALL CRNCH%:CALL CRNCH%
170 CODE%(1)=0
180 REM
190 REM  INTRODUCE THE LEM
200 REM
210 CODE%(0)=1
220 X%(0)=227:Y%(0)=28:SCALE%(0)=15
230 XROT%(0)=0:YROT%(0)=0:ZROT%(0)=3
240 CALL CRNCH%:CALL CRNCH%
250 CODE%(0)=2
260 REM
270 REM  NOW THE LEM IS READY FOR ANIMATION
280 REM
```

There are a lot of things we could do at this point. To get the LEM to move horizontally, and to wrap around the screen when it reaches the edge, we could execute the following simple animation loop:

```
290 FOR I=227 TO 28 STEP -1
300 X%(0)=I
310 CALL CRNCH%
320 NEXT I
330 GOTO 290:REM  LOOP FOREVER - TYPE CTRL-C TO STOP
```

RUN the program. The important thing to notice is that the program puts the LEM at X-coordinates that guarantee it won't straddle the edge of the screen. By a straightforward analysis, it can be shown that no point of the LEM will ever have an X-coordinate greater than 28 or less than -28. Statement 290 will give the LEM X-coordinates that keep it at least 28 plot positions from the right or the left sides of the screen. This is a simple but important concept.

We can put any number of assignments into this loop. Suppose we want paddle #0 to control the approach angle of the LEM. Then we put this statement into our animation loop:

```
305 ZROT%(0)=(PDL(0)/9)
```

<u>NOTE:</u> ZROT% must be in the range 0-27.

Add this to your program and try it out. There is virtually no limit to the things you can do, although the loop should be tight enough that the animation will be fast. The final version of the Lunar Lander game is just such an expansion of the basic animation loop developed above. It contains statements that simulate the effects of lunar gravity and the LEM's controls, and update X%, Y%, and ZROT% to display those effects on the graphics screen:

```
1 DIM CODE%(15),X%(15),Y%(15),SCALE%(15),XROT%(15),
    YROT%(15),ZROT%(15),SX%(15),SY%(15)
2 RESET%=7932:CLR%=7951:HIRES%=7983:CRNCH%=7737:
    TXTGEN%=768
3 D$=CHR$(4):REM  SET D$ TO CTRL-D
4 PRINT D$;"BLOAD MODULE.LUNAR LANDER"
10 CALL RESET%
20 GOSUB 1000:REM  THIS SETS UP SOME TABLES
100 REM
110 REM  SET UP THE SCREENS
120 REM
130 CODE%(1)=1
140 X%(1)=127:Y%(1)=191:SCALE%(1)=15
150 XROT%(1)=0:YROT%(1)=0:ZROT%(1)=0
160 CALL CRNCH%:CALL CRNCH%
170 CODE%(1)=0
180 REM
190 REM
200 CODE%(0)=1
210 X%(0)=227:Y%(0)=28:SCALE%(0)=15
220 XROT%(0)=0:YROT%(0)=0:ZROT%(0)=3
230 CALL CRNCH%:CALL CRNCH%
240 CODE%(0)=2
250 REM
260 REM  INITIALIZE THE SIMULATION - SET UP AN INITIAL
270 REM  VELOCITY AND POSITION IN SOME IMAGINARY WORLD
280 REM
290 HSPEED%=-20:VSPEED%=0:HP%=29184:VP%=3584
300 REM
310 REM  NOW READ THE CONTROLS (PADDLE #0 CONTROLS TILT,
320 REM  WHILE SWITCH #0 TURNS THE ROCKET ENGINE ON)
330 REM
340 TILT%=PDL(0)/9
350 IF PEEK(-16287)<128 THEN 460
360 REM
370 REM  UPDATE THE VELOCITY VARIABLES
380 REM
390 HSPEED%=HSPEED%+HTHRUST%(TILT%)
400 IF HSPEED%<-512 OR HSPEED%>512 THEN
       HSPEED%=HSPEED%-HTHRUST%(TILT%)
410 VSPEED%=VSPEED%+VTHRUST%(TILT%)
420 IF VSPEED%<-512 OR VSPEED%>512 THEN
       VSPEED%=VSPEED%-VTHRUST%(TILT%)
430 REM
440 REM  SIMULATE THE EFFECTS OF GRAVITY
450 REM
460 VSPEED%=VSPEED%+10:REM  A CONSTANT ACCELERATION
470 IF VSPEED%>512 THEN VSPEED%=VSPEED%-10
480 REM
490 REM  UPDATE POSITION VARIABLES
```

```
500 REM
510 HP%=HP%+HSPEED%:IF HP%<3584 OR HP%>29184 THEN
      HP%=HP%-HSPEED%
520 VP%=VP%+VSPEED%:IF VP%<3584 OR VP%>20992 THEN
      VP%=VP%-VSPEED%
530 REM
540 REM  UPDATE THE GRAPHICS ARRAY ENTRIES ACCORDINGLY
550 REM
560 ZROT%(0)=TILT%
570 X%(0)=HP%/128:Y%(0)=VP%/128
580 REM
590 REM  NOW DRAW THE NEW FRAME
600 REM
610 CALL CRNCH%
620 GOTO 340
1000 DIM VTHRUST%(28),HTHRUST%(28)
1010 REM
1020 REM  SET UP SOME SIN/COS TABLES FOR EFFECTS OF
1030 REM  THRUST ON ROTATED LEM
1040 REM
1050 VTHRUST%(0)=-60:VTHRUST%(1)=-58:VTHRUST%(2)=-56
1060 VTHRUST%(3)=-52:VTHRUST%(4)=-44:VTHRUST%(5)=-32
1070 VTHRUST%(6)=-16:VTHRUST%(7)=0
1080 FOR I=0 TO 7:VTHRUST%(14-I)=-VTHRUST%(I):NEXT I
1090 FOR I=0 TO 14:VTHRUST%(28-I)=VTHRUST%(I):NEXT I
1100 FOR I=0 TO 21:HTHRUST%(I+7)=VTHRUST%(I):NEXT I
1110 FOR I=22 TO 28:HTHRUST%(I-22)=VTHRUST%(I):NEXT I
1120 RETURN
```

Type in this program and run it if you like. To stop the program, you'll have to type **CTRL-C**, and then do a **TEXT** command in immediate mode to get the text screen back.

The game is really nothing more than an elementary simulation of a lunar lander. There are of course many features missing; some of these would make the game much more interesting. We could, for instance, have several "moonscapes", and when the LEM hit a screen edge, the program would scroll from one scene to another. A particularly nice feature is "auto-zoom", where the LEM and a portion of the moon are automatically magnified when landing (or crashing) is imminent. If we really wanted to get fancy, we could expand the game by adding meteors, and other space vehicles (friendly or otherwise). These and other special effects can be achieved either as direct extensions of the techniques we have covered, or by clever tricks. See Appendix B for a list of special effects and how to create them.

We won't develop this game any further, such extensions being outside the scope of a tutorial. The important point is that you can extend the game to include many more features with only a modest effort. Games are much easier when the graphics are taken care of.

SOME FUN WITH 3-D GRAPHICS

The 3-D world is a simple extension of the 2-D world, but it is much more fun. While 2-D objects and transformations are easy to visualize, things are much less intuitive in 3-D. Suddenly, instead of one kind of rotation, there are three kinds which can be combined to get some very complex motions. Three dimensional objects also look much more realistic than flat shapes. These are just some of the reasons why 3-D graphics are so appealing.

The biggest problem with 3-D graphics is the difficulty of visualizing objects and describing them to the graphics editor. It's harder to draw 3-D objects on paper, and easy to get the different coordinates and axes confused. If you thought 2-D coordinate systems were difficult to use, you'll find 3-D coordinates even worse.

One way to represent complicated 3-D objects is to create several different 2-D "views". This is the architect's approach: draw the object from the side, from head on and from above. The Space Shuttle in Figure 4 is a good example:

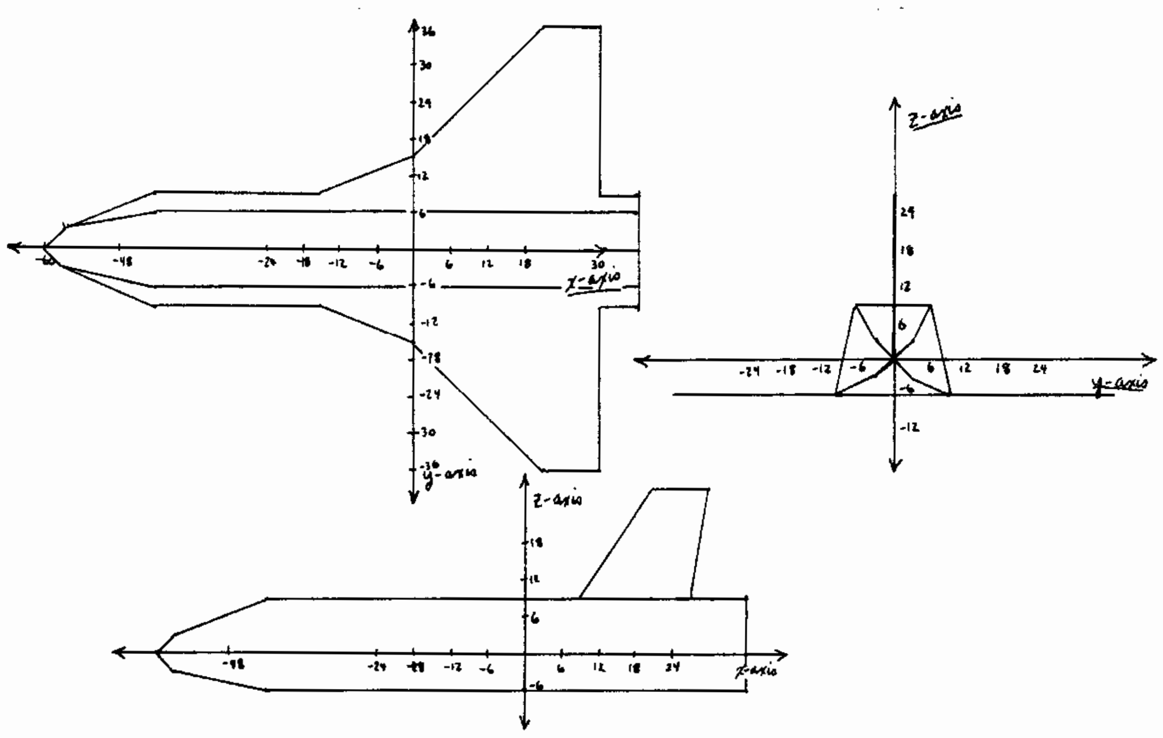

Figure 4

We need all three drawings to eliminate the ambiguities present in each drawing by itself. However, we can orient many objects so that a single drawing will show where everything is. This drawing is tilted so that there are no hidden points or lines. The Box in Figure 5 illustrates this technique:

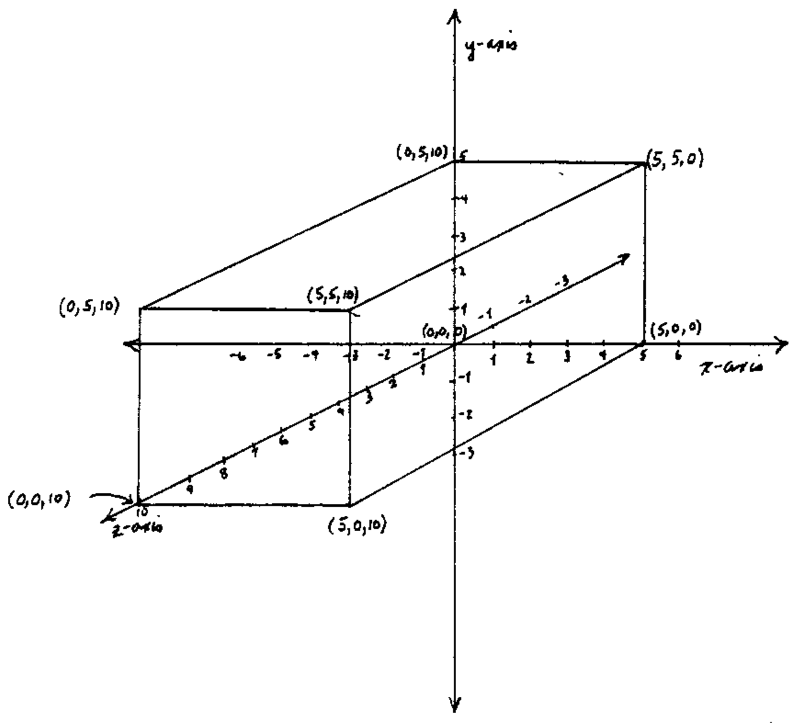

Figure 5

Another approach is to slice the object into layers. Usually, the slicing is perpendicular to one of the axes, so that all of the points in a slice have one coordinate in common. How convenient this method will be depends on the object. For a certain class of shapes, this method is ideal; consider the Prism in Figure 6:

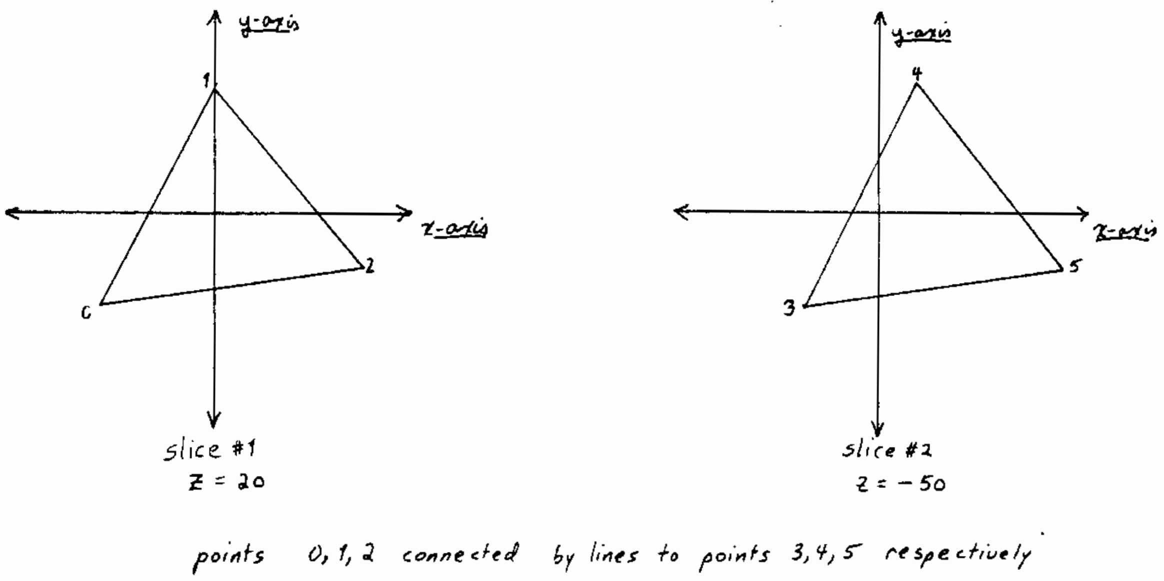
y-axis
1
x-axis
2
0
slice #1
z = 20
y-axis
4
x-axis
5
3
slice #2
z = −50
points 0, 1, 2 connected by lines to points 3, 4, 5 respectively

Figure 6

These techniques are only suggestions. Often, very informal methods, relying heavily on trial and error, work best of all. Feel free to experiment with the editor – that's the best way to learn. (The editor is more than adequate for most conceivable game applications, but it is by no means the last word. You may want to write your own custom editor to help you with complex 3-D shapes. See Appendix C for some ideas.)

At this point we present the data base for the Space Shuttle. You should make sure you understand where it came from (see Appendix A).

Object 0 contains points 0-26, lines 0-28

Point 0 is at -60, 0, 0		Line 0 connects points 0 and 1
Point 1 is at -57, 3, 3		Line 1 connects points 0 and 2
Point 2 is at -57, 3, -3		Line 2 connects points 0 and 3
Point 3 is at -57, -3, 3		Line 3 connects points 0 and 4
Point 4 is at -57, -3, -3		Line 4 connects points 1 and 5
Point 5 is at -42, 6, 9		Line 5 connects points 2 and 6
Point 6 is at -42, 9, -6		Line 6 connects points 3 and 7

```
Point  7 is at -42, -6,  9
Point  8 is at -42, -9, -6
Point  9 is at -15,  9, -6
Point 10 is at -15, -9, -6
Point 11 is at   0, 15, -6
Point 12 is at   0,-15, -6
Point 13 is at  21, 36, -6
Point 14 is at  21,-36, -6
Point 15 is at  30, 36, -6
Point 16 is at  30,-36, -6
Point 17 is at  30,  9, -6
Point 18 is at  30, -9, -6
Point 19 is at  36,  9, -6
Point 20 is at  36, -9, -6
Point 21 is at  36,  6,  9
Point 22 is at  36, -6,  9
Point 23 is at   9,  0,  9
Point 24 is at  27,  0,  9
Point 25 is at  21,  0, 27
Point 26 is at  30,  0, 27
```

```
Line  7 connects points  4 and  8
Line  8 connects points  6 and  9
Line  9 connects points  8 and 10
Line 10 connects points  9 and 11
Line 11 connects points 10 and 12
Line 12 connects points 11 and 13
Line 13 connects points 12 and 14
Line 14 connects points 13 and 15
Line 15 connects points 14 and 16
Line 16 connects points 15 and 17
Line 17 connects points 16 and 18
Line 18 connects points 17 and 19
Line 19 connects points 18 and 20
Line 20 connects points 19 and 20
Line 21 connects points  5 and 21
Line 22 connects points  7 and 22
Line 23 connects points 19 and 21
Line 24 connects points 20 and 22
Line 25 connects points 21 and 22
Line 26 connects points 23 and 25
Line 27 connects points 25 and 26
Line 28 connects points 26 and 24
```

You'll have to type this in to play with the space shuttle (it's not available on the disk because of the copy protection). Use the editor exactly as you did before when you created the Lunar Lander shapes. When you've finished, get into **display mode** (type 'S') and then into spin mode (type 'S' again). You should now see the Space Shuttle on the screen.

Spin mode is a special feature of the editor that can give you a good idea of what an object will look like as it rotates around the different coordinate axes. In this mode the **X**, **Y**, and **Z** keys act like toggle switches to control the spinning around the X, Y, and Z axes, respectively. For instance if you hit the 'Z key, the Space Shuttle will begin to rotate around the Z-axis of its coordinate system. Hitting it again will stop the rotation. Try it.

Stop the Space Shuttle so that it is rotated a bit. Now hit the **'Y'** key. The Shuttle will turn around the Y-axis, but note that the Y-axis has itself been rotated (normally it is straight-up-and-down). This is because of the way the 3-D package transforms objects. The first thing it does is the rotation around the Z-axis; then it takes the rotated object and rotates it some more, but this time around a transformed Y-axis. Finally, it rotates again, around the X-axis. Many 3-D packages work in this way. This allows you to rotate any object around its Z-axis in any orientation; you should keep this in mind when you lay out your shapes.

Try this. Use the **'X'** and **'Y'** keys to get the Space Shuttle into different orientations. Then, when you have stopped the rotation, hit the **'Z'** key. The Shuttle will spin around its transformed Z-axis.

This is definitely not the easiest concept to visualize. To make things worse, you can have several (up to 3) distinct rotations going on at the same time. Experiment with this. Some of the resulting effects are interesting.

Coding this simple simulation in Basic would be very easy. Since we are now in the 3-D world we would use the XROT (XROT%) and YROT (YROT%) arrays, which were zeroed out in the 2-D Lunar Lander. Every drawing technique carries over directly from the 2-D to the 3-D case.

Some final comments: The 3-D package has no perspective or hidden-line capability, two features of the real world that humans use to understand complex scenes. This means that many orientations of objects will have two interpretations, as in an opticial illusion; this could make a game confusing. Another limitation has to do with the limited number of paddles on the average Apple II. It is difficult to create a 3-D game that allows the player to control a space ship (or whatever) without some keyboard commands. These are almost always hard to master. These comments are not meant to discourage you, but rather to challenge you to create a first class 3-dimensional game.

GAME TOOL REFERENCE SECTION

This part of the manual contains all of the information about each part of the Game Tool in a spare and orderly manner and covers the parts of the system that were not covered in the tutorial. You should consult this section when you have specific questions about some part of the system. It will be assumed that you have read at least a part of the tutorial.

GAME TOOL EDITOR COMMANDS

A/PPEND — Adds to the current graphics data base. There are three modes of appending: P/OINT, L/INE, and O/BJECT. A data base can have a maximum of 255 points, 255 lines, and 16 objects. Any attempt to append to a part of the data base that is full will result in an error message.

I/NSERT — Inserts into the data base. As with append there are three modes: P/OINT, L/INE, and O/BJECT. When points are inserted, some lines may be changed so as to refer to their original points. Inserting points or lines never changes objects. Make sure to change object descriptions when you insert points or lines. Any attempt to insert in part of the data base that is full will result in an error message.

D/ELETE — Deletes from the data base. Again, there are three modes: P/OINT, L/INE, and O/BJECT. When points are deleted, some lines may change as in insert. Objects are never modified when points or lines are deleted. Make sure to change object descriptions when deleting points or lines.

C/HANGE — Changes P/OINTS, L/INES, and O/BJECTS. Requires a range which specifies how many to change. A single number "X" means change only point, line or object X, a pair "X, Y" means change points, lines or objects X through Y, and a <cr> means change all of the points, lines or objects in the data base. Old points, lines or objects are displayed on the screen as prompts.

L/IST — Lists the current data base. A range is required as in C/HANGE. The data base is listed starting with all of the objects, followed by a range of points and lines.

R/EAD — BLOAD's a previously defined data base from the disk into the data base area. The standard Apple DOS (not copy-protected 3-D DOS) is used for the read.

W/RITE — BSAVE's the current data base to the disk. You are prompted for a filename which is appended to the string "SHAPE." to get the actual file name. Lunar Lander becomes SHAPE.LUNAR LANDER in the disk catalog. The standard Apple DOS is used for the write.

M/ESSAGE — Sends a message through the Apple II DOS. This is useful for CATALOG'ing the disk while in the editor. Just type **'CATALOG'** when the flashing cursor appears. Any DOS command can be entered, but the effects of LOAD's and SAVE's are unpredictable.

S/HOW — Displays the data base. This command is really three different commands: P/OSITION, T/RANSFORM, and S/PIN. Each one prompts for an object number. Position will move the specified object to a position on the graphics screen that corresponds to the setting of the game paddles. Transform will perform all of the 3-D transformations on the object. In this mode, paddles 0 and 1 control X- and Y-axis rotation, respectively, the 'Z' key controls rotation around the Z-axis, and the cursor keys control scaling. Finally, spin will continuously rotate an object around any combination of axes. The 'X', 'Y' and 'Z' keys control rotation around the X-, Y- and Z-axis respectively.

<esc> — In any mode, escapes to a higher command level. If it is the first character of an input line (where numbers are entered), it also cancels the command and escapes to a higher command level.

For more information about Editor commands, see Appendix D.

EDITOR ERROR MESSAGES

BAD POINT, REENTER — Points must have three coordinates, seperated by two commas. Each coordinate must be in the range [-127 ... 127].

BAD LINE, REENTER — The endpoints of a line must be currently defined points.

BAD OBJECT, REENTER — The point and line ranges describing an object must be within the range of currently defined points, lines and objects.

BAD INDEX, TRY AGAIN — When specifying a point, line or object, an index must be within the range of currently defined points, lines or objects.

BAD RANGE, TRY AGAIN — When specifying a range of points, lines or objects, the index or indices must be in the currently defined ranges.

NOT A COMMAND, TRY AGAIN — You hit a key that didn't correspond to any of the possible commands.

NOT A GOOD MODE, REENTER —You have three choices: P/OINT, L/INE, and O/BJECT modes. (Note: the I/NSERT command has no O/BJECT mode.)

POINT SPACE FULL — The data base contains 255 points. No more can be entered.

LINE SPACE FULL — The data base contains 255 lines. It can't hold any more.

OBJECT SPACE FULL — The data base contains 16 objects. That's the limit.

CANNOT DELETE SOME POINTS — You are attempting to delete points which are a part of some line descriptions.

THE THREE DIMENSIONAL GRAPHICS SOFTWARE

The first part of the graphics package is the line erasing and drawing program. There are two versions of this program; one draws by ORing the graphics memory with bitmasks, and erases by storing zeroes. This makes the erasing function faster, since it is not necessary to read the screen. The other version of the drawing software draws and erases by XORing the screen with the bitmask. The same subroutine is used for drawing and erasing. It is slightly slower than the OR version since it must read memory to erase, but it has the corresponding advantage of leaving the screen in its original state after each draw-erase cycle.

The drawing software works with a standard coordinate system and does the mapping to the actual memory locations with the aid of tables. The table scheme employed is the fastest possible but has the disadvantage that not all of the hires screen is available. The effective screen size is 256x192, while the actual screen dimensions are 280x192.

Both versions of the drawing software work with both screens, though only one screen will be active at any one time. Usually, the erasing and drawing is done on the invisible screen; this is why you can never actually see any lines being drawn during an animation.

The second part of the package is the 3-D transformation program. This program performs true 3-D rotation and scaling of line drawings, though in the interests of execution speed some limitations are imposed. The first transform is rotation around the Z-axis. There are 28 possible orientations around the Z-axis - plenty for game applications. The next transform is rotation around the Y-axis; it is applied to the result of the previous transform. Again there are 28 different orientations around this axis. The last rotation is around the X-axis; this is applied to the result of the previous rotations. There are 28 different orientations around the X-axis.

After carrying out the requested rotations, the program performs a scaling transformation. There are 16 possible sizes for an object ranging from full scale to 1/16 scale, in increments of 1/16. This should be adequate for game applications.

The final transformation is positioning. The transformed points are given an absolute offset of 0-255 horizontally, and 0-191 vertically. The 0,0 point is the left top corner of the screen.

It is important to realize that this program does not clip lines that are partially or entirely off the screen. In fact, it cannot even detect those lines, and this sometimes gives unwanted results. It is necessary for the application program to give objects absolute positions that keep them suitably far from the screen boundaries. A similar though subtler problem is that when rotated, points can go off the screen if they are far enough from the origin. If possible, points should be defined to be close enough to the origin that they don't fall off the screen in any possible orientation.

HIGH LEVEL INTERFACE FOR BASIC PROGRAMS

The high level interface concept is simple - transfer arguments to and from the 3-D package through fixed locations set up as Basic arrays. The arrays must be defined at the beginning of the program, but this is not a serious loss of flexibility at all. The arrays provide the following interface:

CODE (CODE%) — An array of 16 integers (numbered 0 to 15) which specifies the graphics operations that are to be performed for each of up to 16 objects. The numbering of objects in the editor corresponds to the numbering of the CODE array. An element of this array may take on the following values:

0 — do nothing with the object.
1 — transform the object and draw it.
2 — erase the object, transform it and redraw it.
3 — erase the object.

CODE entries for non-existent objects must be 0. This is done for you when a module is initialized.

X (X%) — An array of 16 integers which specifies the horizontal offset, in pixel positions, of each object. An offset of 0 puts the origin of an object's coordinate system exactly on the left screen boundary, while an offset of 255 puts the object on the right screen boundary.

Y (Y%) — An array of 16 integers which specifies the vertical offset, in pixel positions, of each object. An offset of 0 puts the origin of an object's coordinate system right on the top screen boundary, while an offset of 191 puts the object on the bottom screen boundary.

SCALE (SCALE%) — An array of 16 integers which specifies how big each of the objects should be drawn. Array values can range from 0 (1/16th normal size) to 15 (normal size). Any other value has the effect of turning off the scaling transformation. This can be useful when speed is critical and no scaling is required.

XROT (XROT%) — An array of 16 integers which specifies how much each of the objects should be rotated around the X-axis of its coordinate system. Array values can range from 0 (no rotation) to 27 (rotated almost 360 degrees). Any other value has the effect of turning off the X-rotation transformation. This is useful when speed is critical.

YROT (YROT%) — An array of 16 integers which specifies how much each of the objects should be rotated around the Y-axis of its coordinate system. Array values can range from 0 (no rotation) to

27 (rotated almost 360 degrees). Any other value has the effect of turning off the Y-rotation transformation. This is useful when speed is critical.

ZROT (ZROT%) — An array of 16 integers which specifies how much each of the objects should be rotated around the Z-axis of its coordinate system. Array values can range from 0 (no rotation) to 27 (rotated almost 360 degrees). Any other value has the effect of turning off the Z-rotation transformation. This is useful when speed is critical.

SCRNX (SX%) — An array of 16 integers which are filled in by the transformation program. Each entry holds the horizontal position of an object's last point (as numbered in the data base) on the screen. Values returned range from 0 to 255. This is useful for determining when two separate objects touch.

SCRNY (SY%) — An array of 16 integers which are filled in by the transformation program. Each entry holds the vertical position of an object's last point (as numbered in the data base) on the screen. Values returned range from 0 to 191. This is useful for determining when two seperate objects touch.

The set of graphics operations available to the application is very specialized, though for game development it should be adequate. Three dimensional line drawing is well supported, with the following functions:

RESET (RESET%) — Initializes the graphics module by zeroing out the CODE array and setting up addresses in some self-modifying code. The screen is then cleared and hi-res graphics mode (primary page) turned on.

CLEAR (CLR%) — Clears both hi-res graphics screens and turns on hi-res graphics (primary page).

HIRES (HIRES%) — Turns on hi-res graphics (primary page).

CRUNCH (CRNCH%) — Creates a new frame of animation on the invisible hi-res screen. The graphics operations performed depend on the CODE entries for each of the existing graphics objects. When the new frame is complete, CRUNCH flips the display to that hi-res page. In Applesoft, CRNCH% must also locate the arrays, since they are not absolutely fixed in memory. This must happen on every call of CRNCH% since locations can change at run-time.

The Game Tool provides some very special purpose features that should be very useful in games. The most unusual facility is the missile generator. Missiles are simply plotted points, with speed and direction attributes. This allows the user to fire missiles without having to worry about where they should be plotted and when they have gone off the screen. Because of the added overhead in the CRNCH% routine, Applesoft users cannot use missiles. In Integer Basic the missile interface is similar to the 3-D graphics interface. The following arrays provide for transfer of arguments:

MCODE — An array of 16 integers which specifies what operations are to be performed on each of 16 missiles. The following array entries are defined:

- 0 — (inactive) do nothing with the missile.
- 1 — (active) animate missile and wrap at screen boundaries.
- 2 — (unassigned)
- 3 — (active) animate missile and clip at screen boundaries.

The MCODE array entries are written into by the missile utility as well as read from. In clip mode, when a missile goes off an edge of the screen, its MCODE entry is set to 0. This is quite simple to test for. Putting undefined values into the array entries has undefined effects. Before using the missile utility, it is necessary to zero out the MCODE array. RESET does not do this for you.

MX — An array of 16 integers which specifies the horizontal position of each of 16 missiles. The entries may range from 0 (left screen edge) to 255 (right screen edge). The array entry is given the missile's starting position at the time of activation, but the missile utility maintains it from then on.

MY — An array of 16 integers which specifies the vertical position of each of 16 missiles. The entries may range from 0 (top screen edge) to 191 (bottom screen edge). The array entry is given the missile's starting position at the time of activation, but the missile utility maintains it from then on.

MDX — An array of 16 integers which specifies the horizontal component of the missile's velocity. This amount is added to MX every time the missile utility is called. Array entries may have any value in the range -127 ... 127. Suggested values lie in the range -7 ... 7.

MDY — An array of 16 integers which specifies the vertical component of the missile's velocity. This amount is added to MY every time the missile utility is called. Array entries may have any value in the range -127 ... 127. Suggested values lie in the range -7 ... 7.

One utility handles all the missile functions:

MISSILE — This subroutine attempts to update the MX and MY arrays by the values in the MDX and MDY arrays, respectively, for every currently active missile. MCODE entries are checked to determine if a missile should wrap or be clipped when a screen boundary is encountered. If a missile is clipped, its MCODE entry is set back to zero. During normal animation, MCODE entries will be 128 more than the value originally stored there by the user.

The Game Tool has full hi-res text capability:

TXTGEN — Turns on text. Any subsequent print statement will send text onto the hires screen. The character set sits on the primary text page (1024 to 2047) and will be destroyed if any printing is done between the time that the module is loaded and TXTGEN is called. The character set provides upper and lower case, as well as special game symbols. (Thanks to Christopher Espinosa for the text generator and character set).

For more information about the text generator, see the notes in Appendix E. For a sample program using MISSILE and TXTGEN, see Appendix F.

HIGH LEVEL INTERFACE FOR ASSEMBLY LANGUAGE PROGRAMS

All of the functions available to the Integer Basic programmer are also available to the assembly language programmer. The interface is quite similar too. However, the arrays are more compact since there is flexibility at this level to define byte arrays. (This section will describe how to set up the assembly language interface. Consult the previous section for the behavior of the graphics functions.) Here are the array descriptions for this level:

CODE — 16 consecutive bytes starting at address $6000, containing the command bytes for up to 16 objects. The following commands are defined:

$00 — do nothing with the object.
$01 — transform the object and draw it.
$02 — erase the object, transform it and redraw it.
$03 — erase the object.

X — 16 consecutive bytes starting at address $6010 containing the horizontal positions of the objects. Possible array values range from $00 to $FF.

Y — 16 consecutive bytes starting at address $6020 containing the vertical positions of the objects. Possible array values range from $00 to $BF.

SCALE — 16 consecutive bytes starting at address $6030 containing the sizes of the objects. Possible array values range from $00 to $0F.

XROT — 16 consecutive bytes starting at address $6040 containing the orientations of the objects around the X-axis. Possible array values range from $00 to $1B.

YROT — 16 consecutive bytes starting at address $6050 containing the orientations of the objects around the Y-axis. Possible array values range from $00 to $1B.

ZROT — 16 consecutive bytes starting at address $6060 containing the orientations of the objects around the Z-axis. Possible array values range from $00 to $1B.

SCRNX — 16 consecutive bytes starting at address $6070 which the 3-D transformation program fills in with the absolute horizontal screen position of the last point transformed for each object.

SCRNY — 16 consecutive bytes starting at address $6080 which the 3-D transformation program fills in with the absolute vertical screen position of the last point transformed for each object.

MCODE — 16 consecutive bytes starting at address $6090 which specify which operations to perform for each missile. Possible array entries are:

$00 — (inactive) do nothing with the missile.
$01 — (active) animate the missile, wrap at screen boundaries.
$02 — (unassigned)
$03 — (active) animate the missile, clip at screen boundaries.

During normal animation, the array entries will change from $01 to $81 and from $03 to $83.

MX — 16 consecutive bytes starting at address $60A0 which specify the horizontal position of each missile. Array entries may range from $00 to $FF.

MY — 16 consecutive bytes starting at address $60B0 which specify the vertical position of each missile. Array entries may range from $00 to $BF.

MDX — 16 consecutive bytes starting at address $60C0 which specify the horizontal component of each missile's velocity. Array entries may range from $00 to $FF though suggested values lie in the range $F9 ... 0 ... $07.

MDY — 16 consecutive bytes starting at address $60D0 which specify the vertical component of each missile's velocity. Array entries may range from $00 to $FF though suggested values lie in the range $F9 ... 0 ... $07.

The graphics operations have the following addreeses:

RESET — ($1EFC)
CLEAR — ($1F0F)
HIRES — ($1F2F)
CRUNCH — ($1E39)
MISSILE — ($1F39)
TXTGEN — ($300)

The following source might serve as a header for assembly language application programs:

```
        ORG $6000
;
; Set up the arrays
;
CODE    DS      $10     ; Set aside 16 bytes
X       DS      $10
Y       DS      $10
SCALE   DS      $10
```

```
XROT    DS      $10
YROT    DS      $10
ZROT    DS      $10
SCRNX   DS      $10
SCRNY   DS      $10
MCODE   DS      $10
MX      DS      $10
MY      DS      $10
MDX     DS      $10
MDY     DS      $10
;
; Define subroutines in the graphics package
;
RESET   EQU     $1EFC
CLEAR   EQU     $1F0F
HIRES   EQU     $1F2F
CRUNCH  EQU     $1E39
MISSILE EQU     $1F39
TXTGEN  EQU     $300
;
; Your program follows at address $60E0
;
```

Writing an application program in assembly language will be very much like writing one in Basic. The final result, however, will be a faster program.

Appendix A — The Cartesian Coordinate System

To talk about locations in 2 and 3 dimensional "spaces" without pointing, we need some scheme that is both easy to use and unambiguous. The latter condition is vital if we are conversing with a computer. If two locations are different, they must have a different description or there will be confusion. The Cartesian Coordinate System is a scheme for talking about locations that is both easy to use and unambiguous.

Here is how it works in 2 dimensions. First superimpose two scales over your drawing as in Figure A1. The horizontal scale is called the X-axis and the vertical scale is called the Y-axis.

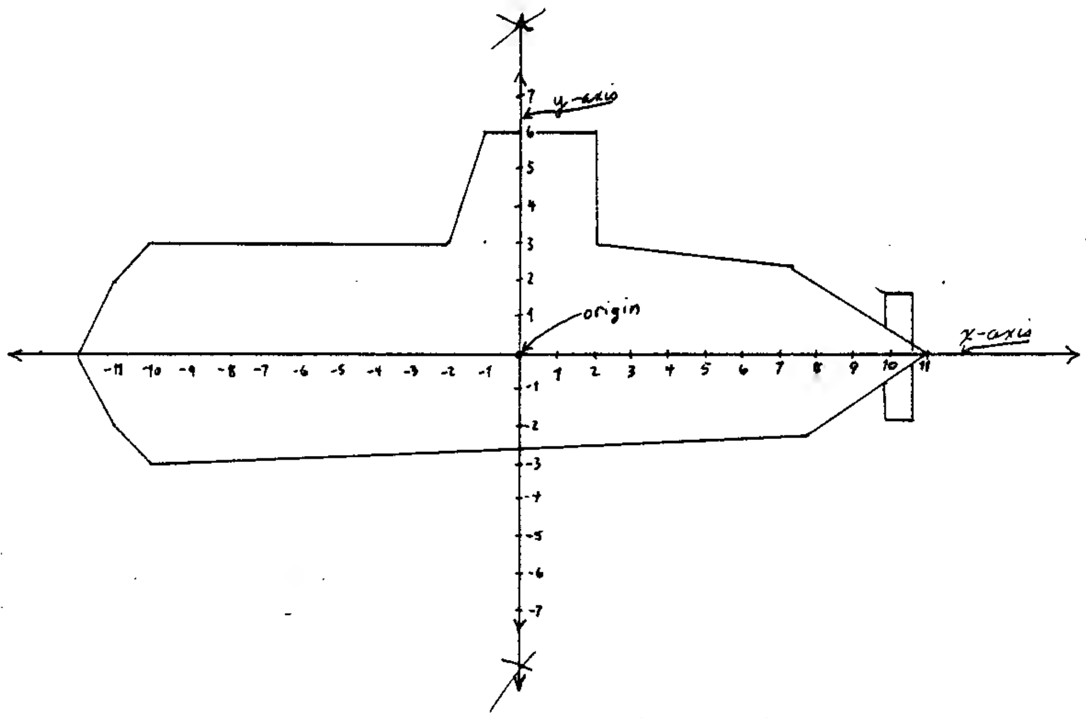

Figure A1

Now, to talk about a point P, draw a rectangle with one corner at the junction of the two axes, and the opposite corner at the point P as in Figure A2. The measure of a horizontal side of the rectangle is the X-coordinate of P, while the measure of a vertical side is the Y-coordinate. The description of P is its X-coordinate and its Y-coordinate. For example, in Figure A2, the X-coordinate of P is −10, while its Y-coordinate is 3. Its description is thus (−10, 3). There is no simpler scheme.

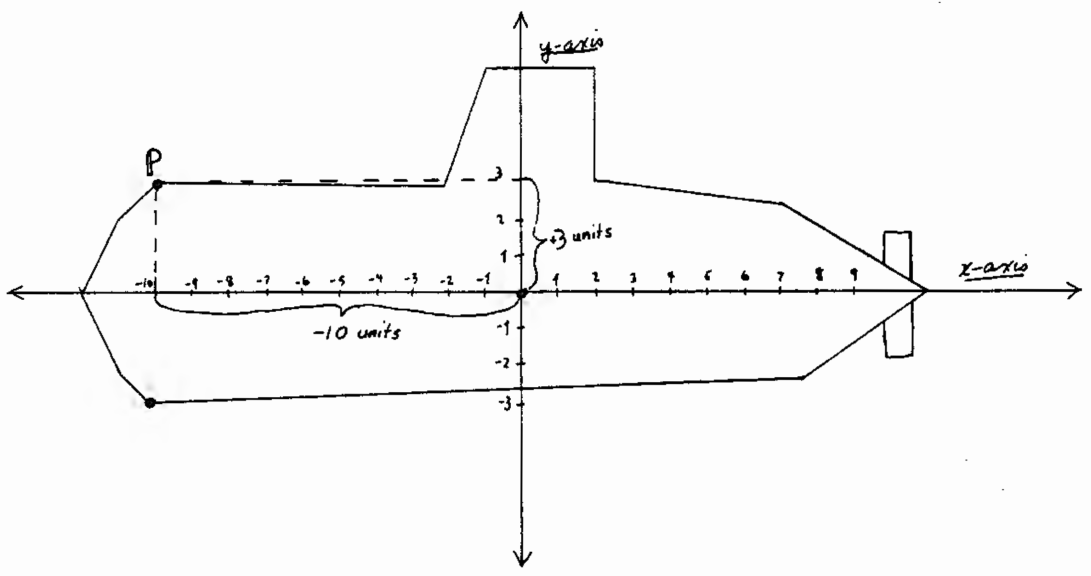

Figure A2

With a few additions it works just as well in a 3-D world. We need another scale, called the Z-axis, and for each point P, instead of drawing in a rectangle, we draw in a rectangular solid as in Figure A3. The X-coordinate is the measure of a side parallel to the X-axis, the Y-coordinate is the measure of a side parallel to the Y-axis, and the Z-coordinate is the measure of a side parallel to the Z-axis. Putting these three coordinates together gives the description of point P, which in Figure A3 is (-10, 3, 20).

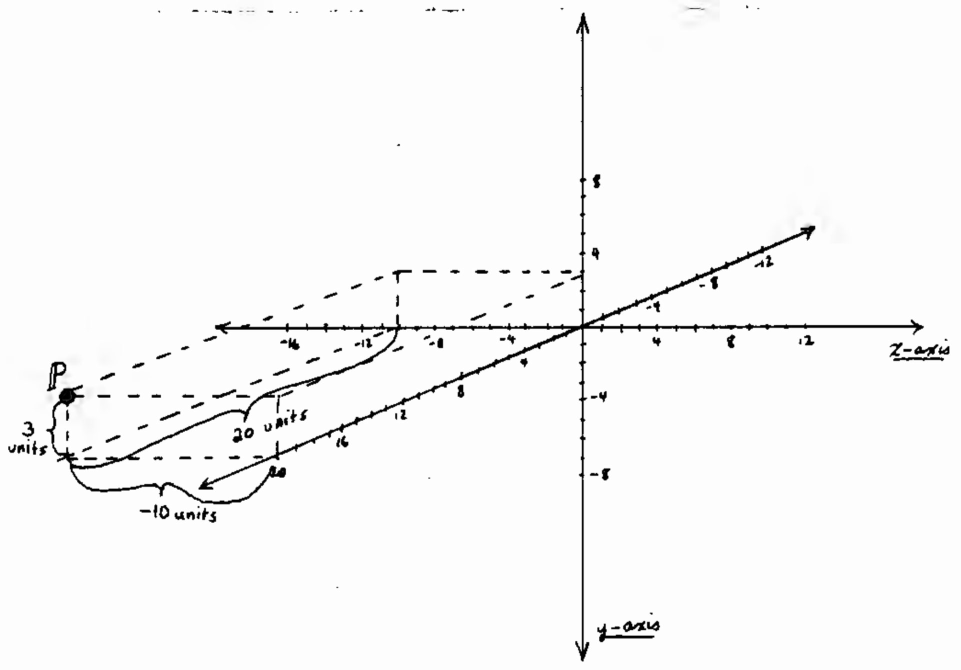

Figure A3

Drawing in the rectangle or solid is really just an explanatory device. Once you get used to coordinates, you will perform the whole process in your head. (Or, after reading this, you may want to write your own editor which interfaces to a graphics tablet or light pen. See Appendix C.)

Appendix B — Special Effects with the Game Tool

The Game Tool was designed to be used in certain standard ways. You do have considerable control over the drawing software, nevertheless, which allows all kinds of special effects. Here is a collection of tricks and hints which will help you to go beyond what was covered in the tutorial.

Flickering Flames

Most outer-space games seem to require a rocket thrust cone. If you draw this just like any other object, though, it doesn't look like a flame. One solution is to draw the thrust cone on every other frame of the animation. To implement this, make sure to introduce the cone-shape on only one of the hi-res pages, and during the animation set the CODE of the cone to 0 every other time through the loop. To pull this off requires close attention to which screen is which.

Orbiting

Sometimes an object must be moved in a circular or eliptical path. This can be implemented by using a table or actually computing the path, but there is an easier way. First you must create a dummy object that consists of but one point. Pick a point that is far away from the origin, but not so far that it will go off the screen when it's rotated. (A point like (80, 0, 0) is a good choice.) Suppose we defined point 11 in that way. Then we would enter the following object using the editor:

```
0:11,11, 1, 0
```

This tells the editor that the object has only a single point, and no lines at all. Now when we rotate this object around the z-axis, we will find that its SCRNX and SCRNY array entries contain the X- and Y-coordinates of a circular orbit. To make any other object follow this path, we simply take the SCRNX and SCRNY values for our "point-object" and put them in the X and Y entries for the other object. To get elliptical orbits, simply rotate around the X- or Y-axis. Experiment to get it right.

Dynamic Objects

Some games require objects to change their shape. Scaling is often all that is required, but what do you do when an object must change its actual form while your program is running? It is possible to change the data base "on the fly", and while this is tricky, it will allow you to achieve all kinds of special effects. Changing the data base is the best way to implement explosions where the object comes apart, laser beams and anything else that requires an object to change radically.

The data you want to get at is in the graphics data base; the memory map of this part of the module is described in Appendix C. From Basic, you can use POKES to change things; assembly language programmers can modify the data by storing new values on top of the old ones. Once you change a data base, it stays changed. If you'll need an object later, copy its description to a safe place before you change it.

Solid 2-D Objects

The Game Tool is limited to vector graphics, but that shouldn't keep you from putting other kinds of graphics on the screen. All you have to do is figure out how to draw your shape using only lines. For example, it takes about 15 small lines to draw one of the Space Invader creatures from that popular Apple II and arcade game. These lines overlap a lot, so you almost have to be OR-drawing your graphics to execute this technique.

Appendix C — Writing Your Own Editor

You can think of the editor as a funny game that is missing its simulation part. It uses a module to do its graphics, just like any program you write. The editor module contains OR-drawing graphics, and the assembly language interface. This means that you can write a program in Basic or assembly language that does everything the Game Tool Editor does, and more. If you are creating a lot of complicated shapes, it might be a good idea to first write a custom editor that will make the whole thing easier. What follows are some hints on how to write your editor.

Any editor must be able to create and modify data bases in their standard form, as the 3-D graphics software won't accept anything else. A standard data base has the following memory map:

$7FD	[2045]	Contains the number of objects in the data base.
$7FE	[2046]	Contains the number of points in the data base.
$7FF	[2047]	Contains the number of lines in the data base.
$800 to $8FF	[2048] [2303]	The X-coordinates for all of the points, in 8-bit 2's compliment form.
$900 to $9FF	[2304] [2559]	The Y-coordinates for all of the points, in 8-bit 2's compliment form.
$A00 to $AFF	[2560] [2815]	The Z-coordinates for all of the points, in 8-bit 2's compliment form.
$B00 to $BFF	[2816] [3071]	The first endpoints for all of the lines, in 8-bit unsigned form.
$C00 to $CFF	[3072] [3327]	The second endpoints for all of the lines, in 8-bit unsigned form.
$D00 to $D0F	[3328] [3343]	The index of the lowest point for all of the objects, in 8-bit unsigned form.

$D10 to $D1F	(3344) (3359)	The index of the highest point PLUS 1, for all of the objects, in 8-bit unsigned form.
$D20 to $D2F	(3360) (3375)	The index of the lowest line for all of the objects, in 8-bit unsigned form.
$D30 $D3F	(3376) (3391)	The index of the highest line PLUS 1, for all of the objects, in 8-bit unsigned form.

It is important to note that there are some dependencies between the different arreys. Objects are made up of points and lines, so if you chenge those arreye, you may get some invelid objects. Similarly, changing the point arrey may make some lines invalid. Your editor should do something sensible in these situations.

Appendix D — Game Tool Editor Notes

The Game Tool Editor has been upgraded to work with both Dos 3.2 and Dos 3.3 systems. When the 3-D disk is booted, it will determine which kind of system it is in and select the corresponding Dos. The user may change versions at any time in the Editor command mode by pressing the '2' key for Dos 3.2 or the '3' key for Dos 3.3 (this will also be the version used by the 3-D Graphics Module Maker). The version of Dos currently in use will be displayed in the top center of the Editor title page:

3/D GRAPHICS EDITOR (3.3) JULY 27, 1980

The Editor **'M'** command can be very useful for controlling your Apple from 3-D Graphics. For example, users with a second drive may select it as the 'data' drive for loading and saving shapes by typing **'M'** and typing the command **'CATALOG,D2'**. All subequent reads and writes will be done with the second drive (the system will still use the first drive for accessing the Game Tool disk). The **'M'** command can also be used to delete and rename files on the data disk.

If the computer has a printer or other output device, it may be turned on from the Editor by typing **'M'** and **'PR#n'**, where n is the slot number of the device. To turn off the device, type **'M'** and **'PR#0'**. For example, to list the database to a printer in slot 1, type the following:

```
MPR#1<cr>                 (turn on the printer)
ML<cr>                    (list the database)
MPR#0<cr>                 (turn off the printer)
```

Some printers can dump the Apple's hires graphics screen after a shape has been displayed on it - to do this with Apple's Silentype printer, type the following commands:

```
MPR#1<cr>                 (turn on the printer)
M<CTRL-Q><cr>             (control-Q dumps the first hires screen)
MPR#0<cr>                 (turn off the printer)
```

Users who are interested in writing their own editors may need to know where shapes and modules are stored in memory. The shape database is described in Appendix C and occupies hex locations $7FD to $D3F. The module consists of this data and the 3-D software, and occupies $7FD to $1FFF. When BSAVEing shapes and modules, use the following ranges:

```
BSAVE SHAPE.NAME ,A$7FD ,L$543
BSAVE MODULE.NAME ,A$7FD ,L$1803
```

To convert a module back into a shape, one may type the following in immediate mode in Basic:

```
BLOAD MODULE.NAME
BSAVE SHAPE.NAME ,A$7FD ,L$543
```

Appendix E — Notes on the TXTGEN Hires Text Utility

The text generator allows the user to place text on the hires graphics screens along with 3-D Graphics shapes. To use it, the user must specify text when creating the module. When such a module is saved to or loaded from disk, the text screen will fill with strange characters - this is the TXTGEN character table, which contains the shapes of the text characters. A program which is going to use the text generator must not erase or print on the text screen. Otherwise the character table will be damaged or destroyed, and the text characters will not print correctly.

The Basic or assembly language program should call the TXTGEN routine before any printing is done. Then, any characters printed will be sent to the hires graphics pages instead of the text screen. The horizontal and vertical tab functions of Integer Basic and Applesoft will work as usual, but the screen will not scroll. To clear the screen, call CLEAR (CLR%) - do not use HOME or CALL -936, as these will damage the character table. See Appendix F for an Integer Basic program using TXTGEN.

The text generator can print upper and lower case characters as well as special graphics symbols. A 'shift' facility is available to allow easy access to characters that aren't available on the Apple keyboard - the following control characters, if printed as part of a string, will shift any subsequent letters as shown (spaces and punctuation are not affected):

CHARACTER	CHR$	FUNCTION	ASCII RANGE
CTRL-@	CHR$(0)	Shift to upper case	64 to 95
CTRL-A	CHR$(1)	Shift to lower case	96 to 127
CTRL-B	CHR$(2)	Shift to graphics symbols	0 to 31
CTRL-C	CHR$(3)	Shift to digits/punctuation	32 to 63

The only other control character recognized by TXTGEN is the carriage return character (ascii 13). All other control codes will print as graphics symbols.

To use the text generator from an assembly language program, first activate it with a JSR TXTGEN (or JSR $300 from the mini-assembler). To print a character, the program should place its ascii code in the A register and call the Monitor routine COUT (JSR $FDED). The character will be printed on the screen at the current 'cursor' location. The only register affected is the A register - it is ANDed with $7F to remove bit 7 from the character.

Other monitor routines which call COUT may also be used. To position the cursor, store the column number in $24 and the row number in $25. To clear the screen, use JSR CLEAR (JSR $1F0F). For printing long strings of text, the user may wish to create a subroutine that calls COUT repeatedly for each character in the string until a terminator (such as $00) is reached. For those wishing to experiment with changing the character table, it is located between $400 and $7F7 in memory - the TXTGEN program itself is at $300 to $3BF.

Appendix F — A Sample Program Using 3-D Graphics, Missiles and Text

The following program should help to illustrate the use of missiles and TXTGEN. Before attempting to run this program, you should first create a module with the missile and text utilities in place. Use the module maker and the Space Shuttle shape data file and specify OR drawing. Now you can run the following program:

```
  0 POKE 74,0: POKE 75,96: POKE 204,0: POKE 205,96
  1 DIM CODE(15),X(15),Y(15),SCALE(15),XROT(15),YROT(15),
        ZROT(15),SCRNX(15),SCRNY(15)
  2 DIM MCODE(15),MX(15),MY(15),MDX(15),MDY(15)
  3 RESET=7932: CLEAR=7951: HIRES=7983: CRUNCH=7737:
        MISSILE=7993: TXTGEN=768
  4 D$="": REM D$=CTRL-D
  5 PRINT D$;"BLOAD MODULE.SPACE SHUTTLE"
  7 POKE 851,209: POKE 862,209: REM  Fix overwrite
 10 CALL RESET
 20 CALL TXTGEN : REM  Turn on the text generator
 40 VTAB 2: PRINT "THE SPACE SHUTTLE"
 50 COUNT=0 : REM  Initialize the animation frame counter
 60 REM
100 CODE(0)=1
110 X(0)=127:Y(0)=96
120 SCALE(0)=15: XROT(0)=2: YROT(0)=5: ZROT(0)=0
130 CALL CRUNCH
140 CALL CRUNCH
150 CODE(0)=2
160 REM  So far, nothing new
170 FOR I=0 TO 15: MCODE(I)=0: NEXT I: REM  Zero the MCODE array
180 REM  Now introduce 4 missiles
190 MCODE(0)=1: MCODE(1)=1: MCODE(2)=1: MCODE(3)=1: REM  Wrap missiles!
200 REM  Set up positions and speeds
210 FOR I=0 TO 3: MX(I)=127: MY(I)=96: NEXT I
220 MDX(0)=6: MDX(1)=0: MDX(2)=-6: MDX(3)=0
230 MDY(0)=0: MDY(1)=6: MDY(2)=0: MDY(3)=-6
240 REM  Start the main animation loop
250 ZROT(0)=(ZROT(0)+1) MOD 28
260 CALL CRUNCH
270 CALL MISSILE
280 REM  Now print the frame number in the upper right corner
290 VTAB 2: TAB 30: PRINT "      ":REM  Erase the old count
295 COUNT=(COUNT+1) MOD 10000
300 VTAB 2: TAB 30: PRINT COUNT
310 GOTO 250
```

* * ATTENTION PROGRAMMERS * *

WE PAY EXCELLENT ROYALTIES

WE HAVE WIDE DISTRIBUTION!

If you have good software, we can do the best job of getting it to the marketplace. Our reputation for only quality offerings is unsurpassed. Customer acceptance of Top of the Orchard Software is overwhelming.

Innovative gaming and well-developed applications and utilities are our interest. Put your software in the hands of professionals and reap the rewards.

Contact us today.

California Pacific Computer Co.
1623 Fifth Street, Suite B
Davis, CA 95616