# GRAPHICS WORKSHOP

## Block Shape Animation — V

*by Robert R. Devine*
*P.O. Box 10*
*Adona, AK 72001*

Hello again !! I think that you'll like what we've got for you this time. When we're finished with this month's topic, you may ask yourself just why we ever bothered with page-flip animation, because this time we'll be dealing exclusively with one-page animation — and our results will be just as smooth and flicker-free as you could want. The reason, however, is very simple: If you're going to be really good at Hi-Res graphics, you'll need more than one method of animation in your bag of tricks.

### INTRODUCING VERTICAL SHIFT ANIMATION

What we're going to look at now isn't true shift animation, as we'll be moving our shapes byte-by-byte rather than bit-by-bit. The results, however, will be so similar that referring to it as shift animation isn't all that far from the truth.

With horizontal shift animation, we found that the only time we needed to DRAW our shape was to place it on the screen. After that, we never needed to DRAW or ERASE it again; instead we simply SHIFTED it across the screen. The same will apply to VERTICAL SHIFT ANIMATION. Using the side-to-side and up-and-down routines together, we'll be able to move our shapes anywhere on the screen without ever drawing the shape again.

Do I have your interest? . . . Good, let's get to work.

The first new concept that you'll need to understand is that with vertical shift animation, you'll always need to have one horizontal row of bytes directly **behind** your shape's direction of travel. This means that VT will be understated by one on a downward-moving shape, and VB will be overstated by one on an upward-moving shape.

Unlike horizontal shifting, which needed the empty column ahead of the shape, with vertical shifting we'll need an empty row **behind** to take care of our ERASING as we move the shape. You can, if you wish, DRAW your shape and then DECrement VT and INCrement VB before starting your shift; or you may choose to make the empty bytes a permanent part of your shape. We'll use both methods.

Since each byte gives us 7 horizontal dots, a shape that is 28 dots wide would only be 4 bytes wide, so adding an empty row **both** ahead and behind would only add 8 bytes to your shape (to allow for changes in direction), and is probably the easiest way to go. In our tests we will make a practice of adding an extra row on both sides.

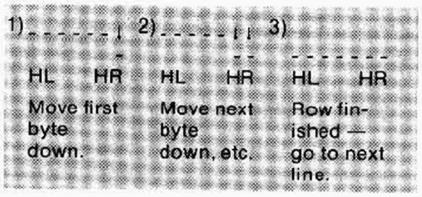
### THE INCRY AND DECRY ROUTINES

Believe it or not, we're going to use some routines that are already built into your Apple!! If you'll remember back to Part I where we talked about YTABLE and how YADDR retrieves our screen addresses, you'll remember that the results were stored in HBASL and HBASH, at $26 and $27, respectively. We did this because those are the normal storage locations for HPOSN, which is your Apple's routine for getting screen addresses. The functions of INCRY $F504 and DECRY $F4D5 are to modify the contents of HBASL and HBASH to reflect the screen address just below (INCRY) or just above (DECRY) the screen byte that we're currently pointing to.

These two routines are the key to our vertical shift routines. The way our vertical shift routines work is to row-by-row, and byte-by-byte, pull our shape forward in its direction of travel. The extra empty row of bytes are used to ERASE the very backmost row of bytes in your shape.

The SHFTDN routine begins at VB/HR and, byte-by-byte, pulls the shape down until finishing at VT/HL. SHFTUP begins at VT/HR and pulls the shape up until finishing at VB/HL.

Let's look at how this works using SHFTDN as an example . . .



Both SHFTUP and SHFTDN step through the block shape as all of our previous routines have. Here's how they move the byte, again using SHFTDN as an example. First, the routine gets the shape byte from the screen (line 1210 of our listing) and stores it in a temporary holder (line 1220). Next, we jump to INCRY, which changes the values in HBASL and HBASH to point to the byte directly below our shape byte. Then we retrieve our shape byte from the holder (line 1240) and place it on the screen at this next lower byte (line 1250). Finally, we jump to DECRY, which moves HBASL and HBASH to point to the Y-coordinate that we started at so we're properly set up for the next byte.

SHFTUP works exactly the same, except that we first jump to DECRY to move up, and then to INCRY to restore the original Y-coordinate.

### THE SHFTDN AND SHFTUP ROUTINES

Listing 1 is the listing for SHFTUP/DOWN which begins at $90AA and fits directly under the shift flip routines (see *Nibble*, Vol. 4/No. 6). As always, you can enter the hex bytes listed (after loading the old driver) and save it to disk with BSAVE BLOCK ROUTINES $90AA, A$90AA, L$556.

Both of the routines have built-in protectors to keep your shape from going off the top or bottom of the screen. SHFTDN won't allow your shape to go lower than VB=189 and SHFTUP won't let you go higher than VT=0, so if your loop accidentally tries to send you off the screen, the routines will simply refuse to execute. If you're going to use these routines on HGR page 1, which only allows vertical coordinates 0–159, then it would be a good idea to enter the following POKEs to modify the driver to keep you on that smaller screen.

Special POKEs for use on HGR page 1:
POKE 37036,157 and POKE 37132,159

Under no circumstances should you allow a shape to go off the top of the screen; however, if you'd like to allow your shape to disappear off the bottom edge of the screen, you may change the two POKEs just mentioned to remove the protection. Be careful that whenever you DRAW your original shape on the screen, you never allow VT or VB to exceed the screen boundaries of 0 or 189, including the empty row of bytes above and/or below your shape.

Now that we've gotten most of the details out of the way, let's see just what it takes to use these new routines.

To run this first test you'll need the Block Routines driver, complete with SHFTUP/DOWN in memory. You'll also need to BLOAD our sample spaceship BLOCK SHAPE #144 from the last issue of *Nibble*. Then simply enter the Applesoft program lines from Listing 2 and RUN the program.

The very first thing that you'll notice is how short and simple the program is, and how smoothly it runs. Once the shape is on the screen, all we need to do is move it !!

Here's how it works . . .

**Lines 10 and 20** take care of the same set-up and DRAWing the shape on the screen which we've done many times before.

**Line 25** adds our extra row of bytes directly above and below the shape. We could have written line 25 as POKE 252,0:POKE 253,13; however, by using the two CALLs (lines 1400 and 1690 of SHFTUP/DOWN respectively) we didn't need to bother figuring out the proper POKEs.

**Line 30** moves the shape down using SHFTDN.

**Line 50** moves the shape back up using SHFTUP.

If we had originally defined our shape as having an extra row of bytes above and below, we could have eliminated line 25 altogether, making our program so short it could probably fit in just one line!!

The reason for the short pause at the bottom and top of the screen is because our loop is really too long; 13(our starting VB)+190=213, which means that the routines are refusing to execute the last 24 trips through each loop.

### USING OUR VERTICAL AND HORIZONTAL SHIFT ROUTINES TOGETHER

Now let's take a little time to get into something a bit more fun. Up until now, every test that we've conducted has run our shapes under program controlled loops, so let's put together a test where you can control your shape with the game paddles.

There are lots of different ways that you can approach any block shape problem, so let's first take a look at some of these.

In our last test we added an extra row of bytes ahead of, and behind our shape because we were going to move in both directions. Big deal, our shape only needed the extra row behind it, but our shape is only three bytes wide, so processing three extra bytes didn't hurt our speed any. We could have changed VT and VB at the end of each loop to maintain the row behind, but it wouldn't have been worth the effort.

If, however, you're moving a number of shapes (or only one for that matter), and they're only moving in one direction, efficiency dictates that you be as speed-conscious as possible and not waste execution time processing unneeded bytes unnecessarily. But if your shape is 15 or 20 bytes wide, perhaps the time spent dealing with extra bytes might make a difference; so you'll need to evaluate each individual situation.

Now we're at a point where we're going to design a program that moves our shape in any one (or combination) of four different directions, so we'll need to allow an extra row of bytes behind (for vertical movement), as well as an extra column of bytes ahead (for horizontal movement). Since our shape is 3 bytes wide by 12 bytes high, this means that we'll need a minimum of 16 extra bytes.

Now we need to decide whether we're going to add just those extra 16 bytes, and change VT and VB, or HR and HL every time we change direction side-to-side, and/or up-and-down, or whether we should simply add the needed cushion of 34 bytes on all four sides of our shape. We're going to either spend execution time processing unneeded bytes (an extra 18), or we'll need to spend time changing which side the empty bytes are on every time we change direction.

As I see it, our major hang-up is that we can't indiscriminately change HR and HL, because we can only do this every seven horizontal shifts; so we would need to have some tests before allowing a right-left-right change of direction. In this case, since we're only dealing with one shape on the screen, we'll build a four-sided cushion, with an extra row of bytes above and below, as well as an extra column right and left.

The first thing we'll need to do is to modify our shape, making the cushion a permanent part of our Shape Table. To do this, let's reload the Block Routines driver, and our original spaceship shape #144. Next, add this new line 26 to Listing 2 and RUN it: Line 26 POKE 254,4: POKE 255,0: STOP.

The shape is now on the screen, with an extra row above and below, as well as right and left.

Finally, CALL 37729 (SCAN) to create your new table, and save it to disk with BSAVE BLOCK +SHAPE+ #144,A$9000,L$45. (See Listing 3.)

Now that your shape is ready to go, enter the Applesoft program (Listing 4) and RUN it.

Spend a few minutes moving the shape around the screen using the game paddles. PDL(0) will move your shape up and down, while PDL(1) will move it from side to side. I know it's rather slow, but that's Applesoft and all those tests that are slowing you down. After we explore how things work we'll try the same program in machine language, which will be quite a speed improvement.

Before we get into the program itself, let's take a moment to see how we've worked our horizontal movement shift tests so that we know when to change HR and HL.

Figure 1 represents our shape which is 3(shape bytes)+2(extra bytes)=5 bytes wide. When we first DRAW our shape on the screen, it fills only the middle three bytes (represented by X's), with the empty bytes on either

## LISTING 1: SHFTDN AND SHFTUP

```
                 1000  .OR $90AA
                 1010  .TA $800
00F9-            1020 HLDR .EQ $F9
00FC-            1030 VT  .EQ $FC
00FD-            1040 VB  .EQ $FD
00FE-            1050 HR  .EQ $FE
00FF-            1060 HL  .EQ $FF
0026-            1070 HBASL .EQ $26
0027-            1080 HBASH .EQ $27
0006-            1090 YO  .EQ $6
00FA-            1100 BASL .EQ $FA
00FB-            1110 BASH .EQ $FB
9391-            1120 YADDR .EQ $9391
F504-            1130 INCRY .EQ $F504
F4D5-            1140 DECRY .EQ $F4D5
90AA- A5 FD      1150 SHFTDN LDA VB        ** CALL 37034 TO ENTER
90AC- C9 BD      1160  CMP #189            ** IS VB>=189 ? (158 FOR HGR)
90AE- B0 2F      1170  BCS RTN1            ** YES-WE'LL GO OFF SCREEN-EXIT
90B0- 85 06      1180  STA YO              ** STORE IN $6 FOR USE BY YADDR
90B2- 20 91 93   1190 L1 JSR YADDR         ** RETURNS-LO=HBASL/HI=HBASH
90B5- A4 FE      1200  LDY HR              ** SET Y-REG TO RIGHTMOST BYTE
90B7- B1 26      1210 L2 LDA (HBASL),Y     ** GET SHAPE BYTE FROM SCREEN
90B9- 85 F9      1220  STA HLDR            ** STORE IN HOLDER
90BB- 20 04 F5   1230  JSR INCRY           ** CHANGE HBASL/HBASH
90BE- A5 F9      1240  LDA HLDR            ** RETRIEVE SHAPE BYTE
90C0- 91 26      1250  STA (HBASL),Y       ** LOAD SHAPE BYTE TO SCREEN
90C2- 20 D5 F4   1260  JSR DECRY           ** RESTORE HBASL/HBASH
90C5- 88         1270  DEY                 ** POINT TO NEXT BYTE <---
90C6- 18         1280  CLC
90C7- C0 FF      1290  CPY #$FF            ** HAS Y-REG PASSED 0 ?
90C9- F0 04      1300  BEQ NXTLN1          ** YES-GO TO NEXT LINE
90CB- C4 FF      1310  CPY HL              ** HAVE WE REACHED HL ?
90CD- B0 E8      1320  BCS L2              ** NO-GOTO NEXT BYTE
90CF- C6 06      1330 NXTLN1 DEC YO        ** MOVE UP TO NEXT LINE
90D1- A5 06      1340  LDA YO              ** GET NEXT Y COORDINATE
90D3- C9 FF      1350  CMP #$FF            ** HAVE WE DONE 0 ?
90D5- F0 04      1360  BEQ J1              ** YES-WE'RE DONE
90D7- C5 FC      1370  CMP VT              ** HAVE WE REACHED VT ?
90D9- B0 D7      1380  BCS L1              ** NO-CONTINUE
90DB- E6 FC      1390 J1 INC VT            ** MOVE VT DOWN 1
90DD- E6 FD      1400  INC VB              ** MOVE VB DOWN 1 (CALL 37085)
90DF- 60         1410 RTN1 RTS             ** DONE-EXIT ROUTINE
90E0- A5 FC      1420 SHFTUP LDA VT        ** CALL 37088 TO ENTER
90E2- C9 01      1430  CMP #1              ** IS VT<1 ?
90E4- 90 33      1440  BCC RTN2            ** YES-WE'LL GO OFF SCREEN-EXIT
90E6- E6 FD      1450  INC VB              ** MOVE VB DOWN 1 LINE
90E8- 85 06      1460  STA YO              ** STORE VT IN $6 FOR USE BY YADDR
90EA- 20 91 93   1470 LP1 JSR YADDR        ** RETURNS-LO=HBASL/HI=HBASH
90ED- A4 FE      1480  LDY HR              ** SET Y-REG TO RIGHTMOST BYTE
90EF- B1 26      1490 LP2 LDA (HBASL),Y  * GET SHAPE BYTE FROM SCREEN
90F1- 85 F9      1500  STA HLDR            ** STORE IN HOLDER
90F3- 20 D5 F4   1510  JSR DECRY           ** CHANGE HBASL/HBASH
90F6- A5 F9      1520  LDA HLDR            ** RETRIEVE SHAPE BYTE
90F8- 91 26      1530  STA (HBASL),Y       ** LOAD SHAPE BYTE ON SCREEN
90FA- 20 04 F5   1540  JSR INCRY           ** RESTORE HBASL/HBASH
90FD- 88         1550  DEY                 ** POINT TO NEXT BYTE <---
90FE- 18         1560  CLC
90FF- C0 FF      1570  CPY #$FF            ** HAS Y-REG REACHED 0 ?
9101- F0 04      1580  BEQ NXTLN2          ** YES-GOTO NEXT LINE
9103- C4 FF      1590  CPY HL              ** HAVE WE REACHED HL ?
9105- B0 E8      1600  BCS LP2             ** NO-GOTO NEXT BYTE
9107- E6 06      1610 NXTLN2 INC YO        ** MOVE DOWN TO NEXT LINE
9109- A5 06      1620  LDA YO              ** GET NEW Y-COORDINATE
910B- C9 BE      1630  CMP #190            ** HAVE WE DONE 190 ? (159 FOR HGR)
910D- F0 04      1640  BEQ J2              ** YES-WE'RE DONE
910F- C5 FD      1650  CMP VB              ** HAVE WE REACHED VB ?
9111- 90 D7      1660  BCC LP1             ** NO-CONTINUE
9113- C6 FD      1670 J2 DEC VB            ** MOVE VB UP 1
9115- C6 FD      1680  DEC VB              ** RESTORE VB
9117- C6 FC      1690  DEC VT              ** MOVE VT UP 1 (CALL 37143)
9119- 60         1700 RTN2 RTS             ** DONE-EXIT ROUTINE
```

Figure 1

```
                                                                 1 1 1 1 1
NS= 0 1 2 3 4 5 6  7                                     7 | 8 9 0 1 2 3 4
    0 0 0 0 0 0 0 | X X X X X X X | X X X X X X X | X X X X X X X | 0 0 0 0 0 0 0
       HL                                                               HR
```

side (represented by 0's). As it stands right now, we can shift back and forth within the same HR/HL as much as we like, as long as we don't go more than seven shifts in any given direction. To keep track of where we are within the HR-HL range, we will set NS (number of shifts) to seven when we first DRAW the shape, and reset NS to seven every time we change HR/HL.

Note that when we are in the exact middle, NS=7. Every time we move right through SHFTR we will INCrement NS (NS=NS+1), and after every left movement through SHFTL we will DECrement NS (NS=NS-1). Once NS reaches 14, it is time to INCrement HR/HL, and when NS reaches 0 it will be time to DECrement HR/HL. Using this method we

## LISTING 3: BLOCK + SHAPE + #144

```
*9000.9045

9000- 00 00 00 00 00 00 00 0C
9008- 00 00 00 00 7F 40 00 00
9010- 03 7F 70 00 00 07 7F 78
9018- 00 00 0F 7F 7C 00 00 1F
9020- 7F 7E 00 00 39 4C 67 00
9028- 00 3F 7F 7F 00 00 0F 7F
9030- 7C 00 00 03 7F 70 00 00
9038- 00 7F 40 00 00 00 1E 00
9040- 00 00 00 00 00 00
```

can change direction at any time, even when we've partly shifted through a byte going in the opposite direction.

Here's how our program works . . .

## LISTING 2: SHIFTUP AND SHIFTDOWN DEMO

```
]LIST

5  REM  REQUIRES BLOCK ROUTINES $90AA AND BLOCK SHA
    PE #144
10  HGR2 : CALL 37799: POKE 251,144
20  POKE 252,1: POKE 253,12: POKE 254,3: POKE 255,1
    : CALL 37679
25  CALL 37085: CALL 37143: REM  ADD 1 EMPTY ROW AB
    OVE AND BELOW
30  FOR X = 1 TO 190: CALL 37034: NEXT : REM  MOVE
    SHAPE DOWN
50  FOR X = 1 TO 190: CALL 37088: NEXT : REM  MOVE
    SHAPE UP
60  GOTO 30
```

## LISTING 4: PADDLE MOVEMENT

```
]LIST

4  REM  REQUIRES BLOCK ROUTINES $90AA
5  REM  REQUIRES BLOCK +SHAPE+ #144
10  HGR2 : CALL 37799: POKE 251,144
20  POKE 252,89: POKE 253,102: POKE 254,22: POKE 25
    5,18: CALL 37679:NS = 7: REM   DRAW STARTING S
    HAPE
35 P0 =  PDL (0):P1 =  PDL (1): REM  READ THE PADDL
    ES
45  IF P0 < 150 AND P0 >  = 100 THEN 70: REM  NO VE
    RTICAL MOVE
50  IF P0 < 100 THEN  CALL 37088: GOTO 70: REM  MOV
    E UP
60  CALL 37034: REM  MOVE DOWN
70  IF P1 >  = 100 AND P1 < 150 THEN 35: REM  NO HO
    RIZONTAL MOVE
75  IF ( PEEK (255) = 0 AND P1 < 100) OR ( PEEK (25
    4) = 39 AND P1 >  = 150) THEN 35: REM  STAY ON
     THE SCREEN
80  IF P1 >  = 150 THEN  CALL 37390:NS = NS + 1: GOTO
    110: REM   MOVE RIGHT
90  CALL 37301:NS = NS - 1: REM  MOVE LEFT
100  IF NS = 0 THEN  CALL 37281:NS = 7: REM  DECREM
     ENT HR-HL/RESET COUNTER
105  GOTO 35
110  IF NS = 14 THEN  CALL 37296:NS = 7: REM  INCRE
     MENT HR-HL/RESET COUNTER
120  GOTO 35: REM  MAKE NEXT MOVE
```

**Lines 10-20** get our shape on the screen, and set the starting value for NS.

**Line 35** reads the game paddle setting PO=PDL(0) and P1=PDL(1).

**Line 45** tests to see if we want any vertical move. Paddle readings between 101 and 149 will indicate that no movement is desired.

**Line 50** If PDL(0) is less than 100, then we use SHFTUP to move up one dot.

**Line 60** If PDL(0) is greater than 100, (actually it's 150), then we use SHFTDN to move down one dot.

**Line 70** tests to see if we want to make any horizontal move.

**Line 75** is used to prevent our shape from going off the right or left side of the screen. It first checks to see if HL (PEEK(255))=0 and our paddle is set to move left. If it is, then it prevents any further movement left. Next it checks to see if HR (PEEK(254))=39 and our paddle is set to move right, again preventing further rightward movement. Remember that our horizontal bytes are numbered 0-39.

**Line 80** If PDL(1) is greater than 150, then we move right using SHFTR and INCrement NS.

**Line 90** If PDL(1) is less than 150 (actually it's 100), we use SHFTL to move left, and DECrement NS.

**Line 100** checks to see if NS=0. If it does, then HR/HL are DECremented and NS is reset to 7.

**Line 110** checks to see if NS=14. If it does, then HR/HL are INCremented and NS is reset to 7.

The problem with our program, of course, is that it's awfully slow. All the testing and going through the Applesoft Interpreter is slowing us down. I remember when I first brought my Apple about four years ago, and thought it did things fast. It seemed that I'd never complain about it being too slow; but that was then. For just about any application other than graphics, it's got all the speed you'll need; but for graphics programs, Applesoft really doesn't have what it takes.

In all of our prior tests, we've looked at how our program performed when translated to machine language. We'll also do that here, but rather than just looking at a hex dump, we'll look at a documented translation (see Listing 5) to give you some ideas not only regarding the speed differences, but also some very basic ideas on how to translate from Applesoft to machine language. We won't go through the translation in great detail, but we'll try to cover the high points.

The first thing that you should note in the center column are the instructions, preceded by numbers such as L35 and L80. Those numbers represent the Applesoft line numbers that are being dealt with; i.e., L35 means line 35 and L80 means line 80. The statements on the right are simply remarks that describe what the instruction is doing.

The next thing that you should note is that the instructions that we CALLed in Applesoft are JSRed in machine language, and that our Applesoft GOTO is JMPed in machine code.

The way to deal with variables in machine code is to select specific memory addresses where you'll store the particular value you're dealing with, and always manipulate that specific address. This is also much faster, as your Apple doesn't need to search a Variable Table. In this case, I defined P0, P1, and NS as memory addresses $19, $1A, and $1B respectively.

The next thing you may want to look at is how we read and store the values of the game paddles. First, we need to load the X-Register (a special byte in the 6502) with the number of the paddle we want to read, then JSR (GOSUB) the Apple's PREAD routine at $FB1E. PREAD stores the paddle value in the

Y-Register (another special byte in the 6502), which we then store in the address where we keep our variable.

Another area you may want to explore is how we do our tests for branching. The commands most often used for this purpose are BCC (branch if less than), BCS (branch if greater/equal to), BNE (branch if not equal to), and BEQ (branch if equal to). The BNE and BEQ commands usually refer to whether something equals zero, unless you specifically CMP (compare) to some other value.

If you think of the way Applesoft IF...THEN statements work, you'll remember that if they FAIL a test, execution drops through to the next line. What this means is that when we translate our Applesoft tests to machine code, we're better off looking for tests that FAIL and cause a branch to the instructions from the next line. For example, while the first test in Applesoft line 45 tests to see if PO < 150, in our machine language version lines 1310-1330, we test to see if PO >= 150, failure of which causes us to fall through to line 50.

In Applesoft, if any part of a test fails, the entire test fails. Therefore, knowing how Applesoft would treat the line should help us know how to write the machine code translation. While this isn't going to teach you how to write assembly language programs, it should help give you a better idea of what's happening, and make some of our driver routines a bit easier to understand.

To run the machine language version of our paddle-shift program, you'll need to have the driver and BLOCK +SHAPE+ #144 in memory, along with our machine language translation which starts at $800. Then enter HGR2:CALL 2048 to get it going. I'm sure that you'll agree there's quite a big difference in execution time. As I've said before, your biggest roadblock to rapid animation is the Applesoft interpreter.

## SHFTUP/SHFTDN — AN ALTERNATE APPROACH

You've seen just how smoothly our up-and-down shift routines work, as well as how easy they are to work with. They really don't need much improvement, but it would be nice if we could move our shapes a little faster. As they stand now, a shape at the top edge of the screen, with a VB of 18, would require 173 shifts to get to the bottom of the screen. If we could cause our shape to move two or three bytes per move, that same shape would only need 86 or 57 shifts, respectively, to move the same distance.

## MACHINE LANGUAGE TRANSLATION LISTING 5:

```
                 1000  .OR $800
920E-            1010 SHFTR .EQ $920E
91B5-            1020 SHFTL .EQ $91B5
90AA-            1030 SHFTDN .EQ $90AA
90E0-            1040 SHFTUP .EQ $90E0
91A1-            1050 MOVEL1 .EQ $91A1
91B0-            1060 MOVER1 .EQ $91B0
0019-            1070 P0 .EQ $19
001A-            1080 P1 .EQ $1A
001B-            1090 NS .EQ $1B
932F-            1100 DRAW .EQ $932F
0800- 20 A7 93   1110 L10 JSR $93A7      ** CALL 37799
0803- A9 90      1120  LDA #144
0805- 85 FB      1130  STA $FB           ** POKE 251,144
0807- A9 59      1140 L20 LDA #89
0809- 85 FC      1150  STA $FC           ** POKE 252,89
080B- A9 66      1160  LDA #102
080D- 85 FD      1170  STA $FD           ** POKE 253,102
080F- A9 16      1180  LDA #22
0811- 85 FE      1190  STA $FE           ** POKE 254,22
0813- A9 12      1200  LDA #18
0815- 85 FF      1210  STA $FF           ** POKE 255,18
0817- 20 2F 93   1220  JSR DRAW          ** CALL 37679
081A- A9 07      1230  LDA #7
081C- 85 1B      1240  STA NS            ** NS=7
081E- A2 00      1250 L35 LDX #0         ** SELECT PDL(0)
0820- 20 1E FB   1260  JSR $FB1E         ** READ PDL(0)
0823- 84 19      1270  STY P0            ** P0=PDL(0)
0825- A2 01      1280  LDX #1            ** SELECT PDL(1)
0827- 20 1E FB   1290  JSR $FB1E         ** READ PDL(1)
082A- 84 1A      1300  STY P1            ** P1=PDL(1)
082C- A5 19      1310 L45 LDA P0         ** GET P0
082E- C9 96      1320  CMP #150          ** COMPARE TO 150
0830- B0 07      1330  BCS L50           ** IF GREATER-FALL THRU
0832- C9 64      1340  CMP #100          ** COMPARE TO 100
0834- 90 03      1350  BCC L50           ** IF LESS-FALL THRU
0836- 4C 44 08   1360  JMP L70           ** JUMP-PASSED BOTH TESTS
0839- B0 06      1370 L50 BCS L60        ** IF P0>100 FALL THRU
083B- 20 E0 90   1380  JSR SHFTUP        ** CALL 37088
083E- 4C 44 08   1390  JMP L70           ** GOTO 70
0841- 20 AA 90   1400 L60 JSR SHFTDN     ** CALL 37034
0844- A5 1A      1410 L70 LDA P1         ** GET P1
0846- C9 64      1420  CMP #100          ** COMPARE TO 100
0848- 90 07      1430  BCC L75           ** IF LESS-FALL THRU
084A- C9 96      1440  CMP #150          ** COMPARE TO 150
084C- B0 03      1450  BCS L75           ** IF GREATER-FALL THRU
084E- 4C 1E 08   1460  JMP L35           ** JUMP-PASSED BOTH TESTS
0851- A5 FF      1470 L75 LDA $FF        ** DOES PEEK(255)=0 ?
0853- D0 06      1480  BNE J1            ** NO-NEXT TEST
0855- A5 1A      1490  LDA P1            ** GET P1
0857- C9 64      1495  CMP #100          ** IS P1<100 ?
0859- 90 C3      1500  BCC L35           ** YES-GOTO 35
085B- A5 FE      1510 J1 LDA $FE         ** GET PEEK(254)
085D- C9 27      1520  CMP #39           ** IS IT 39 ?
085F- D0 06      1530  BNE L80           ** NO-GOTO 80
0861- A5 1A      1540  LDA P1            ** GET P1
0863- C9 96      1550  CMP #150          ** IS P1>=150 ?
0865- B0 B7      1552  BCS L35           ** YES-GOTO 35
0867- A5 1A      1560 L80 LDA P1         ** GET P1
0869- C9 96      1562  CMP #150          ** IS P1>=150 ?
086B- 90 08      1564  BCC L90           ** NO-FALL THRU
086D- 20 0E 92   1570  JSR SHFTR         ** CALL 37390
0870- E6 1B      1580  INC NS            ** NS=NS+1
0872- 4C 86 08   1590  JMP L110          ** GOTO 110
0875- 20 B5 91   1600 L90 JSR SHFTL      ** CALL 37301
0878- C6 1B      1610  DEC NS            ** NS=NS-1
087A- D0 A2      1620 L100 BNE L35       ** IF NS<>0 GOTO 35
087C- 20 A1 91   1630  JSR MOVEL1        ** CALL 37281
087F- A9 07      1640  LDA #7
0881- 85 1B      1650  STA NS            ** NS=7
0883- 4C 1E 08   1660 L105 JMP L35       ** GOTO 35
0886- A5 1B      1670 L110 LDA NS        ** GET NS
0888- C9 0E      1680  CMP #14           ** DOES NS=14 ?
088A- D0 92      1690  BNE L35           ** NO-GOTO 35
088C- 20 B0 91   1700  JSR MOVER1        ** CALL 37296
088F- A9 07      1710  LDA #7
0891- 85 1B      1720  STA NS            ** NS=7
0893- 4C 1E 08   1730 L120 JMP L35       ** GOTO 35
```

The larger the distance that we move our shape, the more we tend to lose some smoothness of movement; so there are limits to how large our steps can be.

To give us this new capability, we will now look at a set of **ALTERNATE** SHFTDN and SHFTUP routines (see Listing 6). By "alternate", we mean that we'll load these routines in the same memory area as the SHFTDN/SHFTUP routines that we just looked at. You can use one of the two methods, but not both at the same time.

Listing 6 includes the alternate SHFTDN and SHFTUP routines, along with two new routines called YINCRD and YINCRU. They reside at the same addresses as our prior vertical shift routines, and begin at $908B. A good way to deal with these is to save them as a separate disk file, and when you want to use them, just load the main driver, then load these in right on top of the old vertical shift routines.

Here's how they work . . .

Since we can now move up/down more than just one byte (dot) per move, we'll need to tell the routines how many dots per move we want by POKEing YINCR to set the vertical INCrement. As we've done before, we'll do this in memory location 227. **POKE 227, YINCRement.**

You may specify any increment that you'd like and the routines will keep your shape safely on the screen.

The next thing that you'll need to know (a pleasant surprise) is that you won't need to carry any extra bytes with you, as the routines have their own built-in erase.

The alternate routines are exactly the same as the routines that we began with, except for the following differences.

1. In our first approach, we used the extra row of bytes to erase the last row of bytes in our shape. In this version, we erase the byte (make it black) where we got our shape byte, **after** making the move.
2. Rather than jumping directly to INCRY or DECRY to get the address of the byte directly above or below, we now jump to YINCRD or YINCRU, which simply loops through INCRY or DECRY, YINCR times to find the address of the destination byte.
3. Rather than simply INCrementing or DECrementing VT/VB at the end of the routine, we will now use the GOUP or GODOWN routines (from Part II) to reset VT/VB up or down YINCR bytes at the end of each routine, in readiness for the next move. Now let's try moving a shape up and down to see what we need to do differently with these vertical shift routines.

To run this test, you'll need the driver with our new vertical shift routines in memory, along with block shape #144 (the one with no extra bytes), and Listing 7. Try running it several times, changing the value of YINCR in line 5 to different increments. Your first reaction to what you see might not be very positive, because at the speed that we're going, you will see some of the erase actions that are taking place. For a change, we have a routine that actually moves our shape **too fast**, and that is best suited for use with lots of other activity and testing going on. We can move a large distance fast, and any delay between moves will work to our benefit.

## LISTING 6: SHFTDN ROUTINE (ALTERNATE)

```
                1000  .OR $908B
                1010  .TA $800
00FC-           1020 VT .EQ $FC
00FD-           1030 VB .EQ $FD
00FE-           1040 HR .EQ $FE
00FF-           1050 HL .EQ $FF
00E3-           1060 YINCR .EQ $E3
0006-           1070 YO .EQ $6
00F9-           1080 HLDR .EQ $F9
925E-           1090 GOUP .EQ $925E
926D-           1100 GODOWN .EQ $926D
9391-           1110 YADDR .EQ $9391
0026-           1120 HBASL .EQ $26
F504-           1130 INCRY .EQ $F504
F4D5-           1140 DECRY .EQ $F4D5
001D-           1150 VBLIMT .EQ $1D
908B- A5 FD     1200 SHFTDN LDA VB        ** CALL 37003 TO ENTER
908D- 18        1210  CLC
908E- 65 E3     1220  ADC YINCR           ** ADD YINCR
9090- C9 BD     1230  CMP #189            ** IS VB+YINCR>=189 ?
9092- B0 34     1250  BCS EXIT            ** YES-EXIT ROUTINE
9094- A5 FD     1260  LDA VB              ** CALL 37012 TO ENTER
9096- 85 06     1270  STA YO              ** STORE IN $6 FOR USE BY YADDR
9098- 20 91 93  1280 L1 JSR YADDR         ** GET ADDRESS OF BYTE 0
909B- A4 FE     1290  LDY HR              ** POINT TO RIGHTMOST BYTE
909D- B1 26     1300 L2 LDA (HBASL),Y     ** GET SHAPE BYTE FROM SCREEN
909F- 85 F9     1310  STA HLDR            ** PUT IN HOLDER
90A1- 20 08 91  1320  JSR YINCRD          ** MOVE DOWN YINCR BYTES
90A4- A5 F9     1330  LDA HLDR            ** RETRIEVE SHAPE BYTE
90A6- 91 26     1340  STA (HBASL),Y       ** PUT IN ON SCREEN
90A8- 20 11 91  1350  JSR YINCRU           ** MOVE UP YINCR BYTES
90AB- A9 00     1352  LDA #0               ** GET SET TO ERASE
90AD- 91 26     1355  STA (HBASL),Y        ** ERASE OLD BYTE
90AF- 88        1360  DEY                  ** DONE-GOTO NEXT BYTE <--
90B0- 18        1370  CLC
90B1- C0 FF     1380  CPY #$FF             ** HAS Y-REGISTER PASSED 0 ?
90B3- F0 04     1390  BEQ NXTLN            ** YES-GOTO NEXT LINE
90B5- C4 FF     1400  CPY HL               ** HAVE WE REACHED HL ?
90B7- B0 E4     1410  BCS L2               ** NO-GOTO NEXT BYTE
90B9- C6 06     1420 NXTLN DEC YO          ** MOVE UP TO NEXT LINE
90BB- A5 06     1430  LDA YO               ** GET Y-COORDINATE
90BD- C9 FF     1440  CMP #$FF             ** HAVE WE DONE 0 ?
90BF- F0 04     1450  BEQ J1               ** YES-WE'RE DONE
90C1- C5 FC     1460  CMP VT               ** HAVE WE DONE VT ?
90C3- B0 D3     1470  BCS L1               ** NO-GOTO NEXT LINE
90C5- 20 6D 92  1480 J1 JSR GODOWN         ** RESET VT/VB-DOWN YINCR
90C8- 60        1490 EXIT RTS              ** DONE EXIT ROUTINE
90C9- A5 FC     1500 SHFTUP LDA VT         ** CALL 37065 TO ENTER
90CB- C5 E3     1530  CMP YINCR            ** ARE THERE YINCR LINES LEFT ?
90CD- 90 38     1540  BCC EXIT2            ** NO-EXIT ROUTINE
90CF- A5 FC     1550  LDA VT               ** GET TOP Y-COORDINATE
90D1- E6 FD     1560  INC VB               ** ADD 1 MORE LINE BELOW
90D3- 85 06     1570  STA YO               ** STORE VT FOR USE BY YADDR
90D5- 20 91 93  1580 L1A JSR YADDR         ** GET ADDRESS OF BYTE 0
90D8- A4 FE     1590  LDY HR               ** POINT TO RIGHTMOST BYTE
90DA- B1 26     1600 L2A LDA (HBASL),Y     ** GET SHAPE BYTE FROM SCREEN
90DC- 85 F9     1610  STA HLDR             ** PUT IN HOLDER
90DE- 20 11 91  1620  JSR YINCRU           ** MOVE UP YINCR BYTES
90E1- A5 F9     1630  LDA HLDR             ** RETRIEVE HOLDER
90E3- 91 26     1640  STA (HBASL),Y        ** PUT IT ON SCREEN
90E5- 20 08 91  1650  JSR YINCRD           ** MOVE DOWN YINCR BYTES
90E8- A9 00     1652  LDA #0               ** GET SET TO ERASE
90EA- 91 26     1655  STA (HBASL),Y        ** ERASE OLD BYTE
90EC- 88        1660  DEY                  ** DONE-GOTO NEXT BYTE
90ED- 18        1670  CLC
90EE- C0 FF     1680  CPY #$FF             ** HAS Y-REGISTER PASSED 0 ?
90F0- F0 04     1690  BEQ NXTLN2           ** YES-GOTO NEXT LINE
90F2- C4 FF     1700  CPY HL               ** HAVE WE REACHED HL ?
90F4- B0 E4     1710  BCS L2A              ** NO-GOTO NEXT BYTE
90F6- E6 06     1720 NXTLN2 INC YO         ** MOVE DOWN TO NEXT LINE
90F8- A5 06     1730  LDA YO               ** GET Y-COORDINATE
90FA- C9 BE     1740  CMP #190             ** HAVE WE DONE 190 ?
90FC- F0 04     1750  BEQ J2               ** YES-WE'RE DONE
90FE- C5 FD     1760  CMP VB               ** HAVE WE REACHED VB ?
9100- 90 D3     1770  BCC L1A              ** NO-GOTO NEXT LINE
9102- 20 5E 92  1780 J2 JSR GOUP           ** RESET VT/VB-UP YINCR
9105- C6 FD     1790  DEC VB               ** REMOVE EXTRA LINE BELOW
9107- 60        1800 EXIT2 RTS             ** DONE-EXIT ROUTINE
9108- A6 E3     1810 YINCRD LDX YINCR      ** CALL 37128 TO ENTER
910A- 20 04 F5  1820 L3 JSR INCRY          ** POINT TO NEXT LOWER BYTE
910D- CA        1830  DEX                  ** BUMP LOOP FLAG
910E- D0 FA     1840  BNE L3               ** LOOP YINCR TIMES
9110- 60        1850  RTS                  ** DONE-EXIT ROUTINE
9111- A6 E3     1860 YINCRU LDX YINCR      ** CALL 37137 TO ENTER
9113- 20 D5 F4  1870 L4 JSR DECRY          ** POINT TO NEXT HIGHER BYTE
9116- CA        1880  DEX                  ** BUMP LOOP FLAG
9117- D0 FA     1890  BNE L4               ** LOOP YINCR TIMES
9119- 60        1900  RTS                  ** DONE-EXIT ROUTINE
```

There's probably no need to go into a long explanation of how it works. The only real difference between Listings 2 and 7 is that Listing 7 doesn't need to add any extra rows of bytes, and has the addition of POKEing the YINCR into location 227.

Now let's see how we might go about moving a fleet of eight alien spaceships up and down the screen. This is where the big improvement will be seen over the first set of vertical shift routines. If we were going to move eight shapes from top to bottom, starting at VB=17, we'd need to do 1352 (191-17)*8 shifts with our original method. Being able to select an increment of two or greater will allow us to reduce the moves needed by one-half or more. Bear in mind that we're not in any way putting down the original one-byte shifts, as both methods will be helpful in different graphics situations.

Listing 8 isn't nearly as complex as it first looks; most of what you see are REMs that tell what's happening. To run it, you'll again need the driver and BLOCK SHAPE #144 in memory, along with Listing 4. When you RUN it you'll see a fleet of eight spaceships move up and down the screen, each moving two dots per move. You should also notice that the apparent jumpiness of the shapes that you saw in Listing 8 is now gone. The delay while the other seven shapes are being drawn is about all we needed. Try running it a few times, changing the value of YINCR in line 5 to different values. What you're seeing isn't really arcade game quality smoothness, but it's similar to the way the shapes moved in the once popular Apple Invaders.

Now let's see how it works.

**Line 5** sets YINCR and the location of our leftmost shape.

**Lines 15-25** DRAW our original eight spaceships on the screen.

**Line 50** sets the number of moves that we'll make going down.

**Line 55** resets the HR/HL to the leftmost position and starts our move loop.

**Line 60** moves the shape down; however, since SHFTDN also moved VT/VB down, we need to use GOUP (CALL 37470) to restore VT/VB for the next shape to use.

**Line 65** changes HR/HL to point to the next shape to the right. After moving all eight shapes, we then use GODOWN (CALL 37485) to move VT/VB down in readiness for moving all eight shapes down again.

**Lines 75-90** repeat the process followed in lines 50-65, except that this time we're moving up.

The program certainly isn't the most efficient one in the world, but by doing all the individual steps, you should get a fairly good idea of just what's happening.

Finally, you may want to see how this little test would run in an all machine language version. To do so, just enter the hex dump shown in Listing 9, then enter HGR2, and CALL 2048. It will run the same as Listing 8, moving two dots per move. Because of the greater speed, the fact that each shape is being individually moved won't be so obvious; in fact, it will pretty much look as though the entire row of ships is moving at the same time. On the minus side, the ships won't (I think) look as good as they did in the Applesoft version, as we're again moving too fast. Depending on your program, you may find that our original routines are best when used with machine language programs, and that the alternate routines work best with Applesoft.

In the next issue, we'll take what we've learned so far and apply it to a real life Hi-Res game program.

**Summary of New Driver Entry Points**

For primary SHFTDN and SHFTUP:

| Routine Name | CALL Address | Hex Address | Routine Function |
|---|---|---|---|
| SHFTDN | 37034 | $90AA | Shift entire shape down one dot. |
| SHFTUP | 37088 | $90E0 | Shift entire shape up one dot. |
| | 37085 | $90DD | INCrement VB to add an extra row below the shape. |
| | 37143 | $9117 | DECrement VT to add an extra row above the shape. |

Special POKEs to use with the driver:

| | |
|---|---|
| POKE 37036,157 | Change bottom edge protectors. |
| POKE 37132,158 | to keep shape on HGR page 1. |
| POKE 37036,200 | Remove protectors to allow shape to go off the bottom of the screen. |
| POKE 37132,200 | These may need to be changed, depending on the height of shape. |

For alternate SHFTDN and SHFTUP:

| | | | |
|---|---|---|---|
| SHFTDN | 37003 | $908B | Shift the entire shape down YINCR dots. |
| SHFTUP | 37065 | $90C9 | Shift the entire shape up YINCR dots. |
| YINCRD | 37128 | $9108 | Loop through INCRY-YINCR times. |
| YINCRU | 37137 | $9111 | Loop through DECRY-YINCR times. |

Special POKEs to use with the driver:

| | |
|---|---|
| POKE 227,YINCR | Set the number of dots to move each shift. |
| POKE 37009,157 | Set SHFTDN to keep shape on HGR page 1. |
| POKE 37009,200 | Set SHFTDN to allow shape to run off the bottom edge of the screen. |

## LISTING 7: DRIVER FOR NEW SHFTUP/DN

```
]LIST

1  REM  REQUIRES BLOCK ROUTINES $908B AND BLOCK SHA
   PE #144
5 YINCR = 2
10  HGR2 : CALL 37799: POKE 251,144
20  POKE 252,6: POKE 253,17: POKE 254,3: POKE 255,1
   : CALL 37679
25  POKE 227,YINCR: REM  SET YINCR
40  FOR X = 1 TO 170: CALL 37003: NEXT : REM  MOVE
   SHAPE DOWN
50  FOR X = 1 TO 170: CALL 37065: NEXT : REM  MOVE
   SHAPE UP
60  GOTO 40
```

## LISTING 8: EIGHT ALIEN SHIPS

```
]LIST

1  REM  REQUIRES BLOCK ROUTINES $908B AND BLOCK SHA
   PE #144
5 YINCR = 2:HR = 3:HL = 1: REM  SET YINCR/SET LEFT
   MOST SHAPE
10  HGR2 : CALL 37799: POKE 251,144
15  POKE 252,6: POKE 253,17: FOR X = 1 TO 8: REM  S
   ET VT/VB-START LOOP
20  POKE 254,HR: POKE 255,HL: CALL 37679: REM  SET
   HR/HL-DRAW A SHAPE
25 HR = HR + 5:HL = HL + 5: NEXT : REM  MOVE OVER F
   OR NEXT SHAPE
35  POKE 227,YINCR: REM  SET YINCR
50  FOR Y = 1 TO 170 / YINCR: REM  SET # OF MOVES D
   OWN
55 HR = 3:HL = 1: POKE 254,HR: POKE 255,HL: FOR X =
   1 TO 8: REM  RESET HR/HL-START MOVE LOOP
60  CALL 37003: CALL 37470: REM  MOVE SHAPE DOWN-RE
   STORE VT/VB UP
65 HR = HR + 5:HL = HL + 5: POKE 254,HR: POKE 255,H
   L: NEXT X: CALL 37485: NEXT Y: REM  MOVE TO NE
   XT SHAPE-RESET VT/VB FOR NEXT SHAPE
75  FOR Y = 1 TO 170 / YINCR: REM  SET # OF MOVES U
   P
80 HR = 3:HL = 1: POKE 254,HR: POKE 255,HL: FOR X =
   1 TO 8: REM  RESET HR/HL-START MOVE LOOP
85  CALL 37065: CALL 37485: REM  MOVE SHAPE UP-REST
   ORE VT/VB-DOWN
90 HR = HR + 5:HL = HL + 5: POKE 254,HR: POKE 255,H
   L: NEXT X: CALL 37470: NEXT Y: REM  MOVE TO NE
   XT SHAPE-RESET VT/VB FOR NEXT SHAPE
95  GOTO 50: REM  START ALL OVER
```

## LISTING 9: M/L VERSION OF LISTING 8

```
*800.898

0800- A9 02 85 E3 A9 03 85 FE
0808- A9 01 85 FF 20 A7 93 A9
0810- 90 85 FB A9 06 85 FC A9
0818- 11 85 FD A9 08 85 1B 20
0820- 2F 93 18 A5 FE 69 05 85
0828- FE 18 A5 FF 69 05 85 FF
0830- C6 1B D0 EB C6 FC E6 FD
0838- A9 55 85 1C A9 03 85 FE
0840- A9 01 85 FF A9 08 85 1B
0848- 20 8B 90 20 5E 92 18 A5
0850- FE 69 05 85 FE 18 A5 FF
0858- 69 05 85 FF C6 1B D0 E8
0860- 20 6D 92 C6 1C D0 D5 A9
0868- 55 85 1C A9 03 85 FE A9
0870- 01 85 FF A9 08 85 1B 20
0878- C9 90 20 6D 92 18 A5 FE
0880- 69 05 85 FE 18 A5 FF 69
0888- 05 85 FF C6 1B D0 E8 20
0890- 5E 92 C6 1C D0 D5 4C 38
0898- 08
```