# GraFORTH

## Animation Guide

# Overview and Turtlegraphics

Overview and Turtlegraphics

If you, like many of us, purchased your Apple to do graphics, you were undoubtedly pleased with its many graphics modes, color capability and documentation that begged you to lay down your money and carry it home. Your new Apple even came with a fairly powerful BASIC with commands that let you define your own shapes and draw them anywhere on the screen. Hi-Res graphics, you thought, here I come!

But after the first week or so with shape tables, you wanted more. How could you generate the types of graphics you saw in the arcades? So after another trip to your local computer store, you came home armed with a Hi-Res character generator that let you redefine the Apple's character set. Later, maybe you purchased a 3-D graphics utility and now, now you were finally on your way.

After several hours of trying to write your own game, you probably came to the painful conclusion that you just couldn't make those dwarves, orcs and elves scamper around the screen like you knew they should. BASIC, it seemed, just wasn't fast enough for really smooth animation graphics. You may have wandered back into the computer store and noticed that all the fast-action games were written in assembly language. Unless you were completely fearless, you probably walked back home thinking you'll just have to wait until somebody makes animation graphics easy and fast enough for the non-machine language programmer.

Enter GraFORTH. The purpose of this new language is to provide a fast, structured programming language and all the graphics tools needed to produce real-time animation in both two and three dimensions - as one complete package. GraFORTH contains all the favorite graphics tools: line, plot and fill commands; character graphics; turtlegraphics; 3-D graphics; and even a music synthesizer for producing notes in several voices, and sound effects. The language of GraFORTH is similar to FORTH, which is known for its speed and flexibility and for its somewhat novel approach to programming. But in many ways, GraFORTH is different as it is a language created specifically for graphics on the Apple. You won't have to start over; many commands are taken from Applesoft to make GraFORTH easy to learn. HOME still clears the screen, HTAB and VTAB still position the cursor, and PLOT and LINE are used in the elementary graphics modes. GraFORTH operates under DOS 3.3 and your completed programs are compiled



to machine language and stored as standard binary files. It is faster than Integer BASIC, Applesoft (even compiled), and Pascal. For comparison, counting to 32,000 in Applesoft requires 35 seconds, Integer Basic takes an amazing 40 seconds, and Apple Pascal requires 20 seconds. GraFORTH, however, takes only 3 seconds. The graphics are also easy to use. Character shapes consisting of several redefined images may be drawn to the screen with one command. Three dimensional graphics include color, perspective and shapes of almost unlimited complexity. And turtlegraphics works at speeds that allow its use in game programming.

In this series, we'll explore graphics on the Apple with emphasis on fast, smooth two and three dimensional color animation. We'll leave you with examples, hints and create several sample programs to allow you to effectively use GraFORTH and your Apple to create impressive animations.

For background, the ability to program in some high level language such as one of the Apple supplied BASICs (Integer or Applesoft) is helpful. You should be comfortable with the Apple Hi-Res screen and graphics in general. Of course any knowledge of assembly language or a structured language such as Pascal won't hurt, but won't be required. The only hardware required is an Apple ][ or ][ Plus with one disk drive (DOS 3.3) and either 48 or 64K of Memory. A color monitor is also a plus, as much of what we will be doing is in color. And, of course, you will need GraFORTH.

As you have probably noticed, GraFORTH is quite different from BASIC or PASCAL. Like PASCAL it is a structured language, and also like PASCAL it is compiled, not to a pseudo 'P-code' but directly to machine language. Like BASIC, GraFORTH is immediate - there is no separate compiler or linker to be invoked, making GraFORTH fast and easy to use.

Where GraFORTH is different from BASIC or PASCAL is in its use of stacks. PASCAL and BASIC also use stacks internally, but they hide them from the user. This is done so you can type something like:

```
LET A = B + C
```

The BASIC interpreter is smart enough to translate this line into the proper sequence of machine language routines and calls to get the job done. In the interest of making BASIC easy to learn, this type of algebraic notation is used. The price you pay for it is speed. Generally, the more work a compiler must do to

translate your program into the language of the machine, the more it will depend upon very general purpose routines. These routines are not particularly optimized for the task at hand, therefore the program - your program - slows down. Of course, very smart (and large) compilers can be written that do optimize code, but they run slowly and take up far more memory than is available on small computers such as the Apple ][.

The trade-off, then, has been simple. Use machine language when you want speed, and a high-level language when you want understandable, easy-to-develop programs.

Is this the only alternative? Fortunately, it is not. A high level language can be built that operates in much more harmony with the machine for which it is generating instructions. This is the idea behind GraFORTH and FORTH-like languages in general. By learning to use something that computers can use efficiently (a stack), you can keep all the features of high level languages you like and still write programs that easily run 10 times faster than they would written in BASIC.

Understanding stacks is not difficult. In fact, if you have used an H.P. calculator, you already know what is called R.P.N. (Reverse Polish Notation). This system is often used in mathematics as it eliminates the need for parentheses. R.P.N. is implemented using a stack in machines such as computers and calculators. With this system, all operands precede their associated operators. What this means is that if you want to add the numbers 3 and 4 together, you would write this as 3 4 + rather than 3 + 4. This is most easily demonstrated using GraFORTH's turtlegraphics.

Turtlegraphics is a vector graphics system, similar in many ways to Applesoft's shape tables. What we mean by vector graphics is that the shape is drawn by a series of relative moves and draws. This allows us to design an image and draw it at any position or angle on the screen. Let's tie up some loose ends with examples of both R.P.N. and turtlegraphics.

If you haven't done so already, boot your GraFORTH disk. When you see the 'Demonstration (Y/N)', type an 'N'. You now see GraFORTH's Ready prompt. To get the turtle commands from disk, type:

```
READ " TURTLE "
```

Make sure to leave spaces on either side of the quotation marks. The quote is a GraFORTH command that says "this is some literal



text". Commands in GraFORTH are always separated by spaces. For this reason, they are often called 'words'. The text "TURTLE" is the name of the disk file containing the turtlegraphics words.

When the Ready prompt reappears, we're ready to start. Type:

```
TURTLE
```

Imagine that you now have a turtle with ink on its tail in the center of your screen. "TURTLE" is a GraFORTH word that erases the screen and sets a text window along the bottom four lines. It also selects the color white and positions our imaginary turtle in the center of the screen, facing the top. We can tell it to TURN and MOVE (relative motion) or we can tell it to TURNTO or MOVETO an absolute position. We can also tell it to lift its tail so that no line will be drawn (PENUP) or put it down again (PENDOWN). Let's try this out. Type:

```
50 MOVE
```

Notice the distance (50) is specified before the command (MOVE). Now let's command our turtle to turn to the right and walk another 50 units.

```
90 TURN 50 MOVE
```

Multiple commands can be placed on one line, as GraFORTH uses spaces to separate them. Let's complete a square with the following:

```
90 TURN 50 MOVE
90 TURN 50 MOVE
```

You can see the advantage of turtlegraphics over standard line drawing already. With standard line commands, we would have had to calculate the actual X,Y coordinates of each corner of the square. Using turtlegraphics, we only have to know how to make a square to plot one.

In GraFORTH, every time you type a number, it is placed on the 'data stack'. You can see this graphically by typing "STACK" from the Ready prompt. Now type the following:

```
4 5 6
```

Don't forget the spaces between each number. When you press "RETURN", you will see a list of these numbers inside square

brackets. This is a picture of the data stack. The stack display can be turned off again simply by typing "STACK". How do GraFORTH words use the stack? When we typed the number "50", it was placed on the data stack. The turtlegraphics word 'MOVE' then removed this number and used it as a distance to move the turtle. Most words in GraFORTH will either use the stack for parameters, or affect the stack in some way.

No language would be complete without looping structures, and GraFORTH has a large variety. One of the easiest and most useful is the 'DO...LOOP' construct. Very similar to the Applesoft FOR...NEXT loop, DO...LOOPs can be used to repeat a group of words a pre-defined number of times. Here is a GraFORTH program that will draw our square using a DO...LOOP.

```
TURTLE 4 0 DO 50 MOVE 90 TURN LOOP
```

Again, notice the 4 and 0 precede the DO. This loop will simply count to 4, each time executing the body of the loop which draws one side of the square. (Actually, it will count from 0 to 3 as the loop limit is not included in the loop - a FORTH convention.)

If you typed the above line into your Apple ][, you immediately saw a square drawn on the screen. GraFORTH compiled and ran your short program, and then promptly forgot all about it. Just as you can type immediate commands in Applesoft by leaving the line numbers off, GraFORTH has both an immediate and a deferred execution mode.

In Applesoft, you write a program, and while it is necessary to be within the language to run it, the language and your program are entirely separate entities. GraFORTH is quite different in this manner. The distinction between your program and the language itself is much less clear. Your GraFORTH program actually extends the language. In other words, what you write is an addition to the language that specializes it for a given task. That task could be a 3-D space simulation, a maze game or anything else you might like to write.

GraFORTH comes complete with a long list of words. These are the general purpose commands that are useful in nearly any program. Some are also specific graphics commands for two and three dimensional graphics, others for music and sound effects. You can see this list at any time by typing "LIST". Pressing ConTRoL-C will stop the list, any other key will continue. If you have been staying with us so far, the list should look like this:



```
Ready LIST

TURTLE
TURN
TURNTO
MOVE
MOVETO
PENDOWN
PENUP
```

These are the commands that you can use to write your programs. Your programs actually extend this list, each word you create is appended to the top. In fact, we have already extended it for turtlegraphics when we typed:

```
READ " TURTLE "
```

The turtlegraphics commands were read from the disk file called TURTLE and added to our list of commands. This list is often called a 'DICTIONARY' or 'WORD LIBRARY' where each word performs a certain task (like a subroutine in BASIC or PASCAL), and the entire list is your program.

Armed with this knowledge, let's create our own word. Earlier, we wrote a simple GraFORTH program to draw a square. Unfortunately, it was immediately run and then forgotten. We can create programs that are added to the word library by defining a new word. Here is the new GraFORTH word 'SQUARE'.

```
: SQUARE
  TURTLE
  4 0 DO
      50 MOVE
      90 TURN
  LOOP ;
```

Notice that except for the first and last lines, it is identical to our earlier program. We indented this one for clarity since a GraFORTH word definition may extend over several lines. The first line is made up of two parts, a colon and the text "SQUARE". The colon is a GraFORTH word that means "define a new word". The text "SQUARE" is the name of the new word being defined, and is the name that will appear in the word libary. Everything following the name "SQUARE" is the string of words that define what the new word will do. Notice it simply uses words that have already been defined to create the new word. The semicolon (;) at the end says "end of definition". At this

point, the commands are compiled and the new word is added to the word library. We can see it now by typing "LIST":

```
Ready LIST

SQUARE
TURTLE
TURN
TURNTO
```

and we can run it by typing "RUN", which runs the top (last defined) word in the word library, or by typing the name of the word itself.

```
Ready RUN

Ready SQUARE
```

Both words cause our square word to be executed, and we are returned to the "Ready" prompt.

We've covered quite a bit this month. Before we leave, here is a new "TURTLE" file you can type into the GraFORTH editor and save over the existing file (make sure you do this to a copy of your GraFORTH disk, not the original!). With this version, animation is much easier as you can use "PENUP" to move without drawing a line, "PENDOWN" to draw, and "UNPEN" to remove an existing line. This gives you the ability to erase entire shapes similar to the "XDRAW" capability of Applesoft shape tables.

```
VARIABLE TURTLE.X
VARIABLE TURTLE.Y
VARIABLE TURTLE.ANG
VARIABLE TURTLE.PEN

: TURTLE.WALK
  OVER OVER -> TURTLE.Y -> TURTLE.X
  128 / SWAP 128 / SWAP
  TURTLE.PEN DUP 1 =
  IF   DROP LINE
  ELSE -1 =
       IF   UNLINE
       ELSE POSN
       THEN
  THEN ;

: PENUP   0 -> TURTLE.PEN ;

: PENDOWN 1 -> TURTLE.PEN ;

: UNPEN  -1 -> TURTLE.PEN ;

: MOVETO
  128 * 64 + SWAP
  128 * 64 + SWAP
  TURTLE.WALK ;

: MOVE
  TURTLE.ANG 16 * 45 / PUSH
  DUP I SIN * TURTLE.X + SWAP
  I 32 + SIN * TURTLE.Y SWAP -
  POP TURTLE.WALK ;

: TURNTO -> TURTLE.ANG ;

: TURN
  TURTLE.ANG +
  360 + 360 MOD TURNTO ;

: TURTLE
  GR ERASE
  0 40 20 24 WINDOW
  3 COLOR PENUP
  128 96 MOVETO
  0 TURNTO
  PENDOWN ;
```

Here is a GraFORTH program to show off your new turtlegraphics words. You can either type it in directly or use the GraFORTH editor to enter, edit, and save it as a disk file. After it is entered, just type "RUN" to watch the show.

```
: RESET
  PENUP
  128 96 MOVETO
  0 TURNTO ;

: SQUARE
  4 0 DO
    DUP MOVE
    90  TURN
  LOOP DROP ;
```

```
: SQUARE.CIRCLE
  36 1 DO
    DUP SQUARE
    10  TURN
  LOOP DROP ;

: SQUARE.SPIRAL
  65 1 DO
    I  SQUARE
    10 TURN
  LOOP ;

: COLORS 3 , 6 , 2 , 1 , 5 , ;

: MANY.SPIRALS
  5 0 DO
  TURTLE
  I ' COLORS + PEEK COLOR
  SQUARE.SPIRAL
  RESET UNPEN
  SQUARE.SPIRAL
  LOOP ;

: MANY.CIRCLES
  5 0 DO
  TURTLE
  I ' COLORS + PEEK COLOR
  I 2 + 10 * DUP
  SQUARE.CIRCLE
  RESET UNPEN
  SQUARE.CIRCLE
  LOOP;

: TURTLE.SHOW
  MANY.SPIRALS
  MANY.CIRCLES
  ABORT ;
```

Next month, we'll talk some more about the language of GraFORTH and discuss different types of loops (IF...ELSE...THEN, BEGIN...UNTIL, BEGIN...WHILE...REPEAT). For the main event, we'll introduce animation with GraFORTH's character graphics and also show you how to save your programs to disk. In future columns, we'll expose the more advanced capabilities of GraFORTH. A space shuttle simulation program will demonstrate the use of three dimensional graphics and the music section will include a library of general purpose sound effects. We'll also discuss



programming techniques for computer graphics and above all, leave you with the knowledge you need to do it on your own.

See you next month!

Note: These are the sample programs used for the above timing comparisons. This is not meant to be a complete benchmark for the languages, as it only tests the simple FOR..NEXT (DO...LOOP) construct.

```
Applesoft:     FOR I = 1 TO 32000: NEXT

Integer:     FOR I = 1 TO 32000: NEXT I

PASCAL:     FOR I:= 1 TO 32000 DO;

GraFORTH:     32000 0 DO LOOP
```

# Introduction to Character Graphics

Introduction to Character Graphics

Last month, we introduced GraFORTH and discussed a number of its special language and graphics features. We also promised to show how to create animations using character graphics. Before we start, let's take a look at some of the more advanced conditional and looping commands available in GraFORTH.

The branching and looping capabilities of GraFORTH are much more varied than those of Basic. There is only one branching command found in Basic that GraFORTH does not have a parallel to: GOTO. However, GraFORTH has a number of constructs not found in Basic that not only make the GOTO unnecessary, but increase a program's readability as well.

The simplest is the IF - THEN construct, which is similar to Basic's. In either language, a test is made, and if the test evaluates as true, the code following the test is executed. The format for GraFORTH's IF - THEN is somewhat different, though: A test is done before the word IF, leaving a number on the stack. If the number is nonzero (meaning "true"), the words between the IF and THEN are executed. If the number is zero ("false"), then the words are skipped and execution continues after the THEN. Here is an example of IF - THEN, along with a similar statement in Basic:

```
GraFORTH:   X 5 = IF
               L 1 + -> L
            THEN

Applesoft:  40 IF X=5 THEN L=L+1
```

The first extension to Basic-like capabilities comes with IF - ELSE - THEN. This works like IF - THEN, except that if the test is true, the words between the IF and ELSE are executed; otherwise the words between ELSE and THEN are executed. This allows for a simple choice between two options and clearly defines the program statements for each.



```
GraFORTH:   X 5 = IF
               L 1 + -> L
            ELSE
               L 1 - -> L
            THEN

Applesoft:  40 IF X=5 THEN L=L+1 : GOTO 60
            50 L=L-1
            60 ....
```

Last month we discussed DO - LOOPs (similar to Basic's FOR - NEXT loops) last month. GraFORTH provides two looping constructs to simplify programming when the number of repetitions is not known ahead of time. The first of these is BEGIN - UNTIL. In a program, the words between the BEGIN and UNTIL are executed, then UNTIL removes a value from the stack (placed there by the previous words). If the number is zero ("false"), execution loops back to the words following the BEGIN. This loop continues until the top stack value when the UNTIL is executed is nonzero ("true").

```
GraFORTH:   BEGIN
               L 1 + -> L
               X L * -> X
               X 1000 >
            UNTIL

Applesoft:  40 L=L+1
            50 X=X*L
            60 IF X<=1000 THEN 40
```

Another looping construct, BEGIN - WHILE - REPEAT, allows the test to occur before a group of words are executed. Control is first given to the words between BEGIN and WHILE. WHILE removes a number from the stack. If the number is nonzero, the words between the WHILE and REPEAT are executed, then the loop repeats back to the BEGIN. If the number is zero, then the program hops past the REPEAT and continues from there. (It's easiest to remember that the second group of words are executed while the test is true.)

```
GraFORTH:   BEGIN
               X L * -> X
               X 1000 <=
            WHILE
               L 1 + -> L
               M L + -> M
            REPEAT

Applesoft:  40 X=X*L
            50 IF X>1000 THEN 90
            60 L=L+1
            70 M=M+L
            80 GOTO 50
            90 ....
```

The last test construct, similar to Basic's ON GOSUB, is CASE: - THEN. Here, a number of separate words appear between the CASE: and THEN. The word CASE: removes a number from the stack to select and execute one of the words. A 0 selects the first word, a 1 selects the next, etc. The program continues following the word THEN. Each word in the list can, of course, be a complicated routine itself, calling many other words.

```
GraFORTH:   CASE:
               COLOR.IT
               DRAW.IT
               UNDRAW.IT
            THEN

Applesoft:  40 ON X GOSUB 100, 150, 200
```

As we discuss various aspects of animation in future columns, we'll be using these branching and looping words quite a bit. In introducing character graphics animation, however, we're going to rely on the good ol' DO - LOOP.

In the GraFORTH demonstration program ("Graphics of the Second Kind"), you can see an animation of a small man walking across the screen. The man (actually the collection of images making up the man) is called "Maxwell" and originated from Apple Computer, Inc., and is an example of character graphics. We'll discuss the basics of animation with character graphics and present a program showing Maxwell walk across the screen.

When GraFORTH first boots up, it loads its standard character set, which is simply a table containing the shapes of all of the printable characters. Whenever a character is to be printed on the graphics screen, GraFORTH "looks up" the shape of the



character in the character set and "draws" the character at the appropriate position on the screen. The GraFORTH disk contains a number of character sets, with different shapes for each character. These shapes can be either various lettering styles, other symbols or images, or parts of larger images which are printed as a rectangular block of characters.

Maxwell is an example of this last type of character shape. The character set CHR.MAXWELL contains three separate pictures of Maxwell, showing three phases of a single walking step. Each picture, or block, is 2 characters wide by 3 tall, using 6 characters in the character set. Each individual character makes up one sixth of the block.

The GraFORTH manual discusses how to create character shapes, and plot individual character blocks on the screen. The plotting process can be broken into a few simple steps: 1) Load and select the desired character set, 2) set the size of the block to be printed with BLKSIZE, 3) position the cursor with HTAB and VTAB, and 4) print the desired block by calling PUTBLK.

Animation with character graphics, as in cartoons, is created simply by rapidly displaying a number of still images one after another, creating the illusion of smooth motion. This is precisely what happens when Maxwell walks across the screen. The three images are repeatedly displayed in order and moved to the left across the screen.

Before we do any programming, let's have a look at what we're about to animate. Start by loading the GraFORTH character editor into memory:

```
READ " CHAREDITOR "
```

The disk will whir as the character editor is both read and compiled into memory. Since the program does not automatically clear the screen (a feature we'll find very useful in future columns), do so now and run the character editor program:

```
HOME RUN
```

Get ("G") the character set named "CHR.MAXWELL" and type "D" for Display. At the bottom of the screen (in inverse) you will see many standard characters, and a number of different shapes that are the images that make up our Maxwell character. The shapes have been redefined to look like a walking man, rather than alphanumeric characters.

Type "B" and select a Blocksize of 2 characters wide by 3 tall. Now type R, then the number 64, to Read the 2 by 3 character block that begins with character number 64. The first of the three Maxwell images will appear in the upper left corner of the screen.

Now read from character number 70, then from number 76. This will display the other two positions of Maxwell. Note that each of them shows him in different stages of walking, moving from right to left within the character block. Read each of the three images in turn back to the screen again, one right after another. You have just performed a simple animation: Maxwell has taken a step. (That's one small step for Maxwell, one giant leap.... ) A GraFORTH program that shows Maxwell taking one step (not very exciting yet), would simply display each of the three images in turn.

The next step is to convince Maxwell to walk all the way across the screen. That's not at all difficult, but it does require a bit of planning. As Maxwell took his one step in the character editor, note that his upper body moved smoothly from right to left, nearly an entire character width. If we start the three-part animation over again, one character position to the left, his upper body will continue its smooth leftward motion as his legs maintain their stride.

There is one more thing to consider: When we showed Maxwell taking only one step in the character editor, each character block was cleanly drawn over the top of the last one, completely erasing it. But if we start shifting the blocks one position to the left, then the right side of the previous two-character wide block will still be visible after the new block is drawn. It's up to us to erase it. Fortunately, GraFORTH has a command to do that. It's called UNBLK.

Let's write an actual program to animate Maxwell. We'll build it in four steps, using separate word definitions for each step. You can type the entire program into the editor, compile it, and watch it run; or you can enter each word definition one at a time and see how each word builds on the last. If you choose the latter, we do have one suggestion, which is best demonstrated: Type "Q" to Quit the characters editor, then type:

```
2816 CHRADR
```

2816 is the default address used by the character editor, and thus is the current address of the Maxwell character set, still in memory. CHRADR selects this as the character set to use when



doing character graphics or displaying text. The "Ready" prompt will appear as a combination of letters and pieces of Maxwell. You can see that the Maxwell character set was never intended for normal text display! You can recover by typing either "CHRSET CHRADR" or "ABORT" (the cheaters can press RESET).

The moral of the story is this: After using character graphics, remember to return to the system character set if you want to make sense of things. The following short word definitions can help: "IN" clears the screen, sets the character set (in this case, the one at memory location 2816), and tabs to the top of the screen. "OUT" puts us back into the system character set, and tabs down so that the graphics won't be overwritten by text. IN and OUT can be helpful when used before and after any "immediate-mode" character graphics work.

```
: IN  ERASE  2816 CHRADR  0 VTAB ;

: OUT  CHRSET CHRADR  15 VTAB ;
```

If you didn't follow us through the character editor example, you'll need to load the character set into memory and select the blocksize before continuing:

```
CR 132 PUTC PRINT " BLOAD CHR.MAXWELL,A2816 " CR

2 3 BLKSIZE
```

Now onto the example itself. GraFORTH is fast enough to draw many large character blocks on the screen quickly, so with a single small block like this, a delay loop is needed each time the block is drawn. Without it, Maxwell would skitter across the screen like a paper doll in a hurricane. The animation starts with the following word definition:

```
: ONE.FRAME
  PUTBLK
  1000 0 DO LOOP ;
```

The word ONE.FRAME removes a number from the stack, which is used by PUTBLK to select and draw the appropriate character block, then it waits for a moment. (A loop to 1000 in GraFORTH takes about a tenth of a second.) To see ONE.FRAME work by itself, you can type:

```
IN 64 ONE.FRAME OUT
```

This selects our Maxwell character set and draws the first of the

Maxwell images. It then returns us to the normal set and stops. ONE.FRAME can be called three times for each of the three blocks that make up one walking step:

```
: STEP
  64 ONE.FRAME
  70 ONE.FRAME
  76 ONE.FRAME ;
```

If you want to see this much in action, type:

```
IN STEP OUT
```

Here's another way to do STEP. This version simply uses a loop to do the same sequence:

```
: STEP
  77 64 DO
     I ONE.FRAME
  6 +LOOP ;
```

Last is the word definition that runs the show:

```
: WALK
  -1 37 DO        ( Start a loop to move left across the screen )

     2 3 BLKSIZE ( Set the 2 by 3 block for Maxwell )
     I HTAB      ( Tab to the position set by the loop )
     STEP        ( Have Maxwell take one step )

     1 3 BLKSIZE ( Set the blocksize to erase half of the block )
     I 1 + HTAB  ( Tab to the right half of the block )
     UNBLK       ( Erase it )

  -1 +LOOP ;     ( Loop back and repeat )
```

The walking sequence is in a loop, counting backwards from 37 to 0. (Remember that GraFORTH DO-LOOPs stop one short of the final value.) This loop provides the horizontal positioning as Maxwell walks from right to left. Inside the loop, we first set the blocksize. The loop value is recalled and used by HTAB to position the block, then STEP is called and Maxwell takes one step.

Now we have to erase the current block before the next block appears one space to the left. Actually, the next block will overwrite the left half of the current one, so we only need to



erase the right half of the block. That makes a block 1 character wide by 3 tall, and we set the blocksize accordingly with BLKSIZE. We then position one more space over by HTABbing to the loop value plus 1, and call UNBLK to erase it.

The loop repeats and Maxwell takes one step after another. Animation! The sequence is started by typing:

```
IN WALK OUT
```

One last complication appears at the end of the walk: The last image drawn is partially erased with the UNBLK at the end of the loop, and half of a Maxwell remains on the left side of the screen. In most animation applications, you will want to erase this leftover image. Let's clean it up with one more UNBLK and combine the entire animation in one word definition:

```
: MAX.WALK
  IN
  WALK
  0 HTAB UNBLK
  OUT ;
```

The animation shown here is fairly simple, but the same basic concepts can be used for designing much more complicated character graphics, with more movement and more shapes. Insoft recently released two new Apple games, Spider Raid and Zargs. Both of them are written in GraFORTH and use animated character graphics extensively. Here is a quick overview for producing any sort of animated character graphics:

Animation is simply a series of still pictures rapidly displayed one after another. Each still picture, or block, should be planned with regards to its relationship to previous or subsequent pictures. Choose the blocksize(s), and design the shapes with the character editor, storing them into one or more character sets. To display the animation, simply draw each block in turn, with a short delay created by either a time-wasting loop, or graphics being drawn elsewhere on the screen. If the picture doesn't move about on the screen, each block can simply be drawn over the top of the previous one. If movement is required, then all or part of the block will have to be erased before moving. For maximum speed, erase only what will not be drawn over.

Next month, we'll describe more advanced character graphics and introduce the use of color and different character sizes. We'll conclude with greater detail on how GraFORTH manipulates characters on the screen, giving you the tools to create complex animations with character graphics.



# More Advanced Character Graphics

More Advanced Character Graphics

In last month's column, we discussed the concepts behind creating character graphics animations with GraFORTH and included a sample program in which the Maxwell figure walked across the screen. We'll expand on those concepts this month and look at some additional graphics features to see how they can help us refine the process. We'll discuss smoothness of animation and the use of color.

The Maxwell animation shown last month can be broken down into a few simple steps: Three separate images are displayed, each followed by a short pause, showing Maxwell take one step. The area which will not be overwritten by the next step is erased, and the block is positioned one character space to the left. This process is repeated and Maxwell takes one step after another, moving to the left.

Two aspects of this animation are worth exploring: the length of the pause, and the number of images displayed before repeating the cycle. Both of these contribute to the overall smoothness of the animation.

The question of how long to pause between images is of course very subjective. In most cases, some kind of pause will be needed to view the image being displayed before it is erased or overwritten. A 'ghost' effect can be created by printing and immediately erasing a character block. Too long a pause can also cause problems. It allows people to notice the individual steps in the animation, changing it from a movie-like motion to a rapid slide show. The pause in the Maxwell animation is about a tenth of a second. Depending on the application, a "good" pause between images will usually range from a twenty-fifth to a quarter of a second. The pause can be created by drawing other graphics elsewhere on the screen, using a sound effect, or simply wasting time with an empty loop.

The second, and more important, contributor to smooth animation is how far to move the object before redisplaying it. (We'll call this the 'step size'.) Obviously an image that moves a third of the way across the screen with each frame will not be very smooth. The most convenient step size to use is one character width (assuming we're moving horizontally). Here the process is simple: (1) Draw the block, (2) wait, (3) erase any portions of the image that will not be replaced by the next

block, (4) reposition one character space over, and (5) repeat. Only one block image is needed, the program is short, and the animation is reasonably smooth.

The quality of the animation can be improved by using a step size smaller than one character width, however. Since character blocks can only be printed on character boundaries, two or more separate images are used, each subsequent image shifted a portion of a character width.

This is what is done with the Maxwell demonstration. Three separate blocks are displayed on each character position, and each image shows Maxwell about a third of a character position farther to the left. Therefore, the step size is a third of a character width. You may want to have another look at the Maxwell images from the character editor. Type:

```
READ " CHAREDITOR "

HOME RUN
```

Get the file CHR.MAXWELL, select a Blocksize of 2 by 3, then Read character numbers 64, 70, and 76, one after another. You can see that Maxwell moves to the left with each image while the block itself is printed in the same place.

Since there are 7 pixels in the width of one character, the limit to step size is seven images per character position. With this step size, a new image could be drawn starting on every pixel. (When using smaller step sizes, the wait pauses can usually be of a shorter duration, since each image will not be very different or far away from the previous one.)

Step sizes need to be considered before creating the character sets. Suppose you want to create an image that fills a 3 by 2 character block and moves 3 steps per character width. Three separate blocks are needed to show the image in each of the three positions. If you draw your 3x2 image in the character editor using a 3 by 2 block, there will be no room within the block to shift the image from side to side. The solution is to define a block that is 4 characters wide, rather than 3. The first image will leave some free space on the left, the next will be centered, and the last will leave space on the right.

If you wish to create duplicate images in the character set that are offset by a few pixels, redrawing each image can be time consuming. Fortunately, the combination of a new routine and a feature built into the character editor can make the task much



easier.

When an image created in the character editor is saved into a character set, the editor reads the image directly off of the high-resolution screen. This means that anything that can be drawn in the upper left corner of the screen can be written into a character set by the character editor. This is exactly why the editor does not automatically erase the screen as it starts up. It allows you to place an image on the screen, run the editor, and save the image as a character block. The screen area used starts at "1 VTAB 1 HTAB", which is the point (X=7,Y=8), and extends to the right and downward according to the blocksize selected in the editor.

If we had a way of plotting character blocks starting on any pixel (instead of just on character boundaries), then we could save the shifted image back into the character set. Below is a routine to do just that. It reads a given character block from memory and plots it point-by-point at any position on the screen. Enter the GraFORTH text editor, type in the word definitions, save them to disk, then compile them into the word library. (Typing the comments is of course optional.)

```
VARIABLE BX             ( Horizontal block size )
VARIABLE BY             ( Vertical block size )
VARIABLE CPX            ( Horizontal pixel start for character )
VARIABLE CPY            ( Vertical pixel start for character )
VARIABLE BPX            ( Horizontal pixel start for block )
VARIABLE BPY            ( Vertical pixel start for block )
VARIABLE CN             ( Character number )
VARIABLE ADR            ( Address of character set )

: PIXELCHAR
8 0 DO                          ( 8 lines / char )
   DUP I + PEEK                 ( Read char byte )
   DUP 128 AND                  ( Check & )
   IF 7 ELSE 3 THEN COLOR       ( set color bit )
   7 0 DO                       ( 7 pixels / line )
      DUP 2 MOD IF              ( If bit is on, plot it )
         CPX I +  CPY J +  PLOT
      THEN
      2 /                       ( Next bit, please )
   LOOP
   DROP
LOOP
DROP ;
```

```
: PIXELBLK
BY 0 DO                          ( Vertical loop )
   BX 0 DO                       ( Horizontal loop )
      I 7 * BPX + -> CPX         ( Set char X position )
      J 8 * BPY + -> CPY         ( Set char Y position )
      J BX * CN + I +            ( Find char # )
      8 * ADR +                  ( Find char address )
      PIXELCHAR                  ( Draw the char )
   LOOP
LOOP ;
```

Two routines are included here. PIXELCHAR plots a single character starting at any pixel, and PIXELBLK calls PIXELCHAR to plot a block of characters. Single characters can also be plotted with PIXELBLK by selecting a blocksize of 1 by 1.

To run PIXELBLK:
1. Load the character set.
2. Set:
   BX to the horizontal block size,
   BY to the vertical block size,
   BPX to the desired starting X coordinate for the block,
   BPY to the desired starting Y coordinate,
   ADR to the character set address,
   CN to the starting character number.
3. Call PIXELBLK. The block will be drawn on the screen.

The variables you need to set are not changed when PIXELBLK is run, so they do not need to be reset every time.

Let's run the whole pixel-shift-and-save procedure through, assuming GraFORTH has just been booted. Since Maxwell is already a "shifted" set, we'll instead shift one of the helicopter images from the CHR.STUFF character set (discussed on page 7-9 of the GraFORTH manual), so that it starts midway between two character positions:

1. READ the pixel-shifter routines onto the word library:

```
READ " <filename> "    ( Use the filename you saved the
pixel-shifter with. )
```

2. READ the character editor in above the pixel-shifter:

```
READ " CHAREDITOR "
```



3. Load the character set:

```
CR 132 PUTC PRINT " BLOAD CHR.STUFF,A2816 " CR
```

4. Set the appropriate variables:

```
5 -> BX  3 -> BY  ( 5x3 character block )
8 -> BPY          ( Start at correct vertical coordinate for
character editor )
11 -> BPX         ( Start at horizontal coordinate + 4 more to
shift it 4 pixels (7+4=11) )
2816 -> ADR       ( Set character set address )
33 -> CN          ( Set character number for first helicopter )
```

5. In one line, erase the screen, draw the block, and enter the character editor:

```
ERASE PIXELBLK MAIN    ( MAIN is the word that runs the character
editor. )
```

6. Set a blocksize wide enough to fit the new image. Press "B":

```
Enter Block Horizontal Size :6
Enter Block Vertical Size :3
```

7. Write the image into the character set. (We're going to overwrite the three 'happy-face' images here.) Press "W":

```
Enter character number
to be written : 78
```

From here you can either edit the image further, or save the character set to disk. Pixel-shifting can also come in handy for changing the colors in character shapes, as we'll see in a few moments.

GraFORTH can also display characters in 8 different sizes, using the word CHRSIZE. CHRSIZE removes a number from the stack to determine what size subsequent characters will be displayed in. A character size of zero selects the normal text display. Character sizes 1 through 8 use a different method for displaying characters: Each dot, or pixel, of the character is plotted as a small rectangle, similar to the rectangles created with the FILL command. "1 CHRSIZE" is the same size as the normal display; "8 CHRSIZE" draws characters 8 times larger.

The rectangle-type character plotting allows some additional

capabilities. The color of the characters can be selected with the word COLOR, while the normal text display will only display color if the characters were created with color. The larger character sizes can also be used in GraFORTH's exclusive-or mode (EXMODE), so that characters can be plotted over graphics, and then erased without disturbing the underlying graphics! The price paid for these features is speed: The normal display can print character much faster. The larger sizes don't have the speed necessary for smooth animated character graphics. They are best used for displaying assorted messages or still character images.

Let's have a look at how GraFORTH keeps track of the characters on the screen, and how this affect character display. When your Apple is in 'text mode' (no graphics), all of the characters on the screen are stored as ASCII values in a reserved area of memory. The hardware continuously reads these values and creates a character video display from them.

The high-resolution graphics mode uses one of two other areas in memory for its display. Here, each dot on the screen is stored as a bit in memory. Changing the bits changes the display.

When GraFORTH displays a character on the hi-res screen, it stores the ASCII value of the character in the unseen text memory space. This allows GraFORTH to keep track of what characters are where without having to read through a lot of hi-res memory. It then 'looks up' the shape of the character in the current character set and writes that shape into the hi-res space, which causes that character to appear on the screen.

As mentioned on pages 7-14 and 7-15 of the GraFORTH manual, before GraFORTH decides to print a character, it first checks the text area to see if that character is already on the screen. If it is, then the character is not reprinted. This speeds hi-res scrolling considerably. However, character set changes and UNBLK commands do not affect the contents of the text page, and can produce discrepancies between what you see and what you get. This is best clarified with an example. Enter the following lines:

```
CR 132 PUTC PRINT " BLOAD CHR.SLANT,A2816 " CR
2816 CHRADR
HOME PRINT " ONE LINE " CR
CHRSET CHRADR
0 VTAB PRINT " TWO LINES " CR
```

We first printed the line "ONE LINE" using the "slant" character



set, then printed the line "TWO LINES" over the top of it using the standard character set. Note that the "LINE" in "LINES" is still in the slant set. GraFORTH checked the text screen, and since it found the characters already there, it did not reprint them in the new character set.

One solution is to clear the text memory. If this is done, GraFORTH will not find any identical characters and will always reprint. The command -936 CALL will clear the text window to spaces without affecting the graphics screen:

```
2816 CHRADR
HOME PRINT " ONE LINE " CR
CHRSET CHRADR
-936 CALL 0 VTAB PRINT " TWO LINES " CR
```

The call cleared the text window and allowed the entire line to be reprinted in the new character set.

If you ever find characters not printing when you think they should, then their ASCII values are probably already in the text memory, preventing them from being reprinted. This is especially true if you want to print a space character (ASCII 160, or character number 0) that has been redefined to be a visible graphics image, since there are many space characters already lurking in the text page of a screen that is mostly blank.

The large size characters also use the text screen. This means that (for example) if you print a normal size character at 2 VTAB 1 HTAB, then print a character in 8 CHRSIZE also at 2 VTAB 1 HTAB, GraFORTH will lose track of the first character even though the characters occupy different portions of the hi-res screen. This won't cause any problems unless you try to scroll or reprint the character.

An amusing and enlightening effect can be created by going into TEXT mode before running a program that uses character graphics. All character blocks will be printed simply as groups of letters on the text screen. The following example runs the character graphics portion of the GraFORTH demonstration program from text mode. (If you've been following the above examples, you'll first need to clear the character editor from memory with "FORGET X" to prevent some word names from being duplicated.)

```
TEXT
READ " GRAPHICS2 "
```

(Press reset to exit, as the demo will otherwise continue.)

Another aspect of animated character graphics worth exploring is color. The larger character sizes allow you to select color as characters are printed, but, as mentioned above, the slower speed can get in the way. The best method is to use the normal character display and design the color right into the character sets. This also means that intricate color patterns can be used, rather than a single color. Using the GraFORTH character editor, you can select colors as you're creating the image, subject to the usual color limitations of the Apple.

Pages 19 and 20 of the Apple Reference Manual discuss the basics behind the Apple color limitations, and the Softalk column "Assembly Lines" explored the subject in depth recently. Let's take some time out here to look more closely at how the Apple stores pixels and keeps track of color in memory. Then we'll see how it affect character display in GraFORTH.

The pixels in each horizontal row of the Apple screen are stored in memory in groups of 7, one group per byte of memory. With each group is a single bit which determines what colors the pixels in the group can have. A byte can contain either green and violet pixels, or orange and blue pixels. If for some reason the color bit is changed, then all of the pixels in that byte will change color. If any two adjacent pixels are plotted, their colors will combine to form white. A true white is always two pixels wide.

Here's how to determine the color of an individual pixel: If the color bit is set to 1 (orange and blue pixels), then a pixel will be orange if it is in an odd numbered column, and blue if in an even numbered column. If the color bit is cleared to 0 (green and violet pixels), then a pixel in an odd column will be green, and a pixel in an even column will be violet. Thus the actual color of an individual pixel is determined by (1) the setting of the color bit for the byte in memory the pixel occupies, and (2) the column the pixel is plotted in.

For most kinds of plotting, GraFORTH takes care of all of this for you. If you plot, for example, a green dot, GraFORTH will clear the color bit to zero, then check what column the dot is being plotted in. If the column is odd, GraFORTH will simply plot the pixel. If the column is even, GraFORTH will automatically move the pixel one column to the right, since green dots cannot lie in even columns. In general, when a color is specified, GraFORTH shifts the dots if necessary; if the color is set to white, the dots are always left in place.



Let's try a few examples. If you haven't specified a color, GraFORTH will use white (3 COLOR, which has its high bit cleared to zero). Enter:

```
0 40 18 24 WINDOW ERASE

0 10 PLOT
```

A violet dot appears, since it was plotted in an even column.

```
1 10 PLOT
```

Another dot plotted adjacent to it changes it to white.

```
1 15 PLOT
```

A single dot in this column is green.

```
1 COLOR

0 20 PLOT
```

We're forcing this dot to be green. Since a green dot could not be plotted in an even column, GraFORTH moved it to the right into column 1.

```
5 COLOR

5 20 PLOT
```

Surprise! Plotting an orange dot near the green dot changed the green dot to orange, too. Here's why: Since orange was specified, GraFORTH set the color bit for that byte to 1. But the green dot occupies the same byte (the same group of 7 pixels) and required the color bit to be zero to keep it green. When the color bit changed, the pixel then satisfied all the requirements of being orange, namely being in an odd column with the color bit set to 1.

Let's pull back from this digression on bytes and color, and apply it to character graphics.

The first revelation can be found by remembering that character shapes in GraFORTH are 7 pixels wide by 8 pixels tall. The character width of 7 pixels happens to coincide with the 7 pixels per byte (with one color bit) discussed above. Therefore, each hi-res character is one byte 'wide' and 8 bytes 'tall' and each row of the character has its own color bit.

Suppose that you've created and printed at the left a single character shape that is a solid green block. This means that the color bits for the character are zero and all of the odd columns (1, 3, 5) contain pixels. Suppose that the same shape is then printed one character space to the right. That means it will be offset by 7 pixels from the original. The odd pixels (1, 3, 5) will now fall into even (1+7=8, 3+7=10, 5+7=12) columns, changing the green block to violet! Second revelation: Colored character blocks change color when moved between even and odd columns. (Actually, character blocks do not have a 'true' color while in the character set. Color and color changes only become apparent when the blocks are plotted on the Apple screen.)

This brings us to the realization that if we want to keep the colors constant while moving horizontally across the screen, two sets of character shapes will be needed. They will be identical, except that one will be offset by 1 pixel. One set will be used in even columns, the other in odd. As the blocks are moved one character space, or 7 pixels, at a time, the 1 pixel offset in every other block will actually make the distance either 6 pixels or 8 pixels. Even columns will stay even, and odd will stay odd.

The pixel-shifter routine shown above can be used to easily create the 'other' color images. Two things should be kept in mind: First, since the character editor edits the images starting on an odd character column (1 HTAB), then the colors will coincide with the editor when the block is printed on any odd column. Printing a colored character block on an even column will show colors reversed from those in the character editor. Second, when we created a shifted character image above, we needed a wider block to save it. The same is true for shifts of one pixel. The actual image must be at least one pixel narrower than the block it is saved in, to leave room for the one-pixel shift.

Designing character sets and animating character blocks is of course a very new art. Questions of 'what looks best' and 'how best to do this' are always subjective, and depend on the particular application being written. We've hopefully given you the tools you need for working confidently with GraFORTH's character graphics, and some ideas on smooth animation.

Next month, we'll turn to the three-dimensional capabilities of GraFORTH, showing how to create, save, and manipulate 3-D shapes. See you then.



# *Introduction to 3-D Graphics*

Introduction to 3-D Graphics

This month we'll turn our attention to the three-dimensional graphics capabilities of GraFORTH, and how they can be used for animation.

In any type of computerized 3-D graphics system, you start by creating a set of points, lines, and shapes in 3-D. Every point has a relationship to every other point: It can be higher or lower, closer or farther away, and more to the left or right. And of course, this relationship depends on your point of view. The three different direction aspects of a point are represented using three numbers, or coordinates, labeled X, Y, and Z. It's the computer's job to convert your set of points according to some formula into points on a two-dimensional screen, using only X and Y coordinates. Then the points are connected with the appropriate lines, just as the 3-D points were connected with lines.

There are two different philosophies used in creating 3-D graphics. For the first, imagine a universe in which all of the 3-D objects exist. You describe the objects and and tell the computer where they are in the universe. You then decide where your eye is, and which direction you're looking. The computer figures out which objects lie in that direction, converts them into a single two-dimensional image, and draws that image on the screen. This concept makes it fairly easy to represent complex scenes, but manipulating individual objects within that scene can be more time-consuming.

Another philosophy is to treat each 3-D object separately on the screen. You describe each 3-D object, then tell the computer where the objects should appear on the two-dimensional screen (or if they should appear at all), what size to draw them, and how they should be oriented. Each object is converted from three dimensions to two, independent of every other object. This means complex scenes can require more programming to produce, but manipulating each individual object is faster and easier. This is the technique used by GraFORTH.

GraFORTH allows you to manipulate 3-D objects through direct high-level commands. For example, the GraFORTH word SCALE sets the displayed size of a 3-D object, XROT rotates the object about the X-axis, and YPOS sets the vertical position of the object on the screen. These straightforward commands provide an

easy-to-follow method of generating 3-D graphics.

The 3-D process can be divided into two parts: First, the image is created using the Image Editor supplied on the GraFORTH system disk. Then, the GraFORTH commands are used to read the image and draw the object on the screen with the appropriate rotation, scale, etc. The image may reside in any free area of memory and is not changed by the drawing commands.

Let's define a couple of words for this discussion: An "image" is a set of 3-D points and lines as stored in memory. An "object" is a picture of the 3-D image as it is manipulated and actually displayed on the screen. Images can reside in memory without being assigned as objects and drawn; and two objects, though positioned and oriented differently on the screen, can both use the same 3-D image in memory. (For example, two rotating cubes on the screen can use the same set of 3-D lines.)

For each image, the X, Y, and Z coordinates can range from -128 to 127, giving a possible 256 positions along each of the three axes, which is plenty for most applications. The actual number of lines in an image is limited only by the amount of available memory. (Each endpoint or line entry in the image uses four bytes of memory.)

Up to 16 different objects can be manipulated at one time in GraFORTH. They are numbered 0 through 15, and referenced with the GraFORTH word "OBJECT". After giving an OBJECT command, the 3-D commands will manipulate that object until another OBJECT command is given. For example, if you type:

```
 3 OBJECT
30 XROT
10 SCALE
```

then object 3 will be rotated 30 units around the X-axis and scaled to a size of 10. To manipulate a number of objects, you select each object in turn with OBJECT, then give the appropriate commands for that object.

Here is a quick summary of the individual 3-D commands, their effects, and the appropriate ranges of numbers to use:

XPOS, YPOS - These set the X and Y position on the screen of the 3-D point (0,0,0) for the object, and and are used for positioning the object in the appropriate place on the screen. XPOS can range from 0 to 255 and YPOS can range from 0 to 191. At the extremes, however, the object may overlap the edge of the



screen, causing wrap-around.

SCALX, SCALY, SCALE - These commands determine the size of the object on the screen. SCALX sets the width and SCALY sets the heighth. The word SCALE simply sets both width and heighth to the same number simultaneously. The range is -31 to 31. A scale of 0 produces a displayed object with no thickness, and negative numbers create a mirror-image effect. Since two objects can use the same image in memory, symmetrical objects, such as bird wings, can be created using two objects side-by-side, with positive and negative scale numbers. This is the technique used for the two wings of the flying bat in the "Die Fledermaus" portion of the demonstration program.

SCALZ - This determines the amount of perspective used. Perspective is what causes the front of an object to appear larger than the back. A large perspective number makes the front a good deal larger, and negative numbers provide "reverse perspective", with the back of the object larger than the front. Zero perspective means the front and the back will be the same size. The range, as above, is -31 to 31.

XROT, YROT, ZROT - These commands rotate the current object around each of the three 3-D axes. A complete rotation is divided up into units from 0 to 256. Zero is no rotation, 64 is a right angle, 128 is the same as 180 degrees, and 192 is three-quarters around the circle. Values greater than 256 or less than 0 can also be used for rotating more than once around. For example, a rotation to 258 units is the same as to 2 units. Note: The actual rotation of the object changes for every other rotation value. This means that if you rotate an object in steps of 1 unit per DRAW, the view of the object will change every other DRAW, making the animation appear slower. It's best to increment rotation values in steps of 2.

XTRAN, YTRAN, ZTRAN - These commands translate, or "slide", the object in each of the three directions in space. The object can be shifted as long as none of its points falls out of the -128 to 127 position range. If this happens, a wrap-around effect will occur. Therefore, translation works best with small images, having room to move.

OBJCOLOR - This determines what the object's color will be when it is drawn if color was not specified when the image was created. If color was specified, then OBJCOLOR is ignored. The standard GraFORTH color numbers (1, 2, 3, 5, 6, 7) are used. Note that OBJCOLOR also sets the normal COLOR command, so be sure to reset COLOR to the desired value after using OBJCOLOR.

Here is a table of the 3-D parameters and the range of values they use:

| Parameter | Range | In steps of |
|---|---|---|
| XPOS | 0 to 255 | 1 |
| YPOS | 0 to 191 | 1 |
| SCALX | -31 to 31 | 1 |
| SCALY | -31 to 31 | 1 |
| SCALE | -31 to 31 | 1 |
| SCALZ | -31 to 31 | 1 |
| XROT | 0 to 255 | 2 |
| YROT | 0 to 255 | 2 |
| ZROT | 0 to 255 | 2 |
| XTRAN | -128 to 127 | 1 |
| YTRAN | -128 to 127 | 1 |
| ZTRAN | -128 to 127 | 1 |
| OBJCOLOR | 1,2,3,5,6,7 | |

Let's try some examples. First, we need a 3-D object to work with:

```
0 40 18 24 WINDOW ERASE
CR 132 PUTC PRINT " BLOAD TETRA,A2816 " CR
OBJERASE                    ( Clear 3-D variables )
0 OBJECT  2816 OBJADR       ( Set parameters for object 0 )
80 YPOS  10 SCALE
20 XROT  40 YROT
DRAW
```

As we present word definitions, you'll probably want to use the editor to enter the definitions, then compile them into the word library using MEMRD. Then you can experiment with the definitions by changing some of the parameters from the editor and recompiling. Of course, you can also type the word definitions directly into GraFORTH from the keyboard.

Creating animations with GraFORTH's 3-D graphics is easy and straightforward. As we mentioned in an earlier column, animation is simply a series of still pictures displayed rapidly one after another, providing the effect of movement. One fast way to generate this movement is with a DO-LOOP:



```
257 0 DO I YROT DRAW 4 +LOOP
```

This example rotates the object a full circle around the Y-axis. Since the loop is in steps of two, it repeats 128 times, producing 128 separate DRAWs, one after another. For each DRAW, the rotation around the Y-axis is set to the loop value, incrementing from 0 to 256.

This type of animation is straightforward, but for most applications a number of parameters need to be manipulated at once. Let's look at how to do more complicated manipulations with a few examples.

When using a DO-LOOP, usually one DRAW will be performed each time through the loop. The size of the loop then determines how many times the object will be drawn. To change the parameters, two approaches are possible: The loop value can be used to generate the desired parameter values, or separate variables can be used to keep track of each parameter.

In the first method, the conversion from loop value to parameter value is done with short formulas. For example, if you want the tetrahedron to rotate around the Y-axis three times for each rotation around the X-axis, you can use this routine:

```
: THREE.ROT
  257 0 DO
     I XROT
     I 3 * YROT
     DRAW
  2 +LOOP ;
```

After entering THREE.ROT into the editor and compiling (or entering it directly from the keyboard), it can be run by simply typing:

```
THREE.ROT
```

The trick is to find the right formula for the desired motion. Suppose, with the above example, you also wanted to make the tetrahedron grow in size from 12 SCALE to 20 SCALE. The change from 0 to 256 in the loop must be translated to a change from 12 to 20. Note that the difference between the start and end loop values is 256, and the difference in the scales is 8. If we divide the loop value by 32, we get a range of 0 to 8. If we then add 12, we get the desired range of 12 to 20:

```
Loop value  0   / 32 = 0 ...  0 + 12 = 12  Scale value
Loop value  256 / 32 = 8 ...  8 + 12 = 20  Scale value
```

The new routine looks like this:

```
: ROT&SCALE
  257 0 DO
     I YROT
     I 3 * XROT
     I 32 / 12 + SCALE
     DRAW
  LOOP ;
```

Below is a program adapted from the "rolling tetrahedron" display in the GraFORTH demonstration program. The tetrahedron moves down and to the right, rotates end over end, and grows and shrinks, giving the appearance of "rolling" closer then farther away. You can use this routine with any image in memory.

```
: ROLL.OBJECT
  37 0 DO
     I 3 * XROT
     I 5 * YROT
     I 6 * 25 + XPOS
     I 3 * 35 + YPOS
     I 18 - ABS CHS 18 + SCALE
     DRAW
  LOOP ;
```

None of these formulas are "magic". As the routine was written, we "tweaked" each formula until we got the desired display. Here are the numbers that come out:

```
Loop value: 0 to 36
XROT: 0 to 108
YROT: 0 to 180
XPOS: 25 to 241
YPOS: 35 to 143
SCALE: 0 to 18, then back to 0
```

The scaling formula deserves more comment. The desired effect was to have the object grow and then shrink. We could have used two scaling loops one after another: the first increasing and the next decreasing. But then we would have had to keep all the other parameters moving smoothly through the transition from one loop to the next, without a skip in values. For simplicity, we decided to use a single loop.



With the loop value moving from 0 to 36, we wanted the scaling function to slide from 0 to 18 and back to 0. This can be shown in Figure 1. Figure 2 shows the steps we used to achive the effect.

Sometimes a more complicated animation cannot be performed inside a simple DO-LOOP. This is especially true if the user is interacting with the program through a joystick or keyboard, and the program must make decisions. In this case, it's often best use separate variables to keep track of each parameter. The parameters can then be updated at any time from the running program. The following program duplicates the ROLL.TETRA routine using this technique.

```
VARIABLE XR   ( X rotation )
VARIABLE YR   ( Y rotation )
VARIABLE XP   ( X position )
VARIABLE YP   ( Y position )
VARIABLE SC   ( Scale )
VARIABLE DIR  ( Scale direction larger or smaller? )

: UPDATE.TETRA
  XR 3 + DUP -> XR  XROT   ( Increase X rotation by 3 )
  YR 5 + DUP -> YR  YROT   ( Increase Y rotation by 5 )
  XP 6 + DUP -> XP  XPOS   ( Increase X position by 6 )
  YP 3 + DUP -> YP  YPOS   ( Increase Y position by 3 )
  DIR IF                   ( If scale is increasing: )
     SC 1 + DUP -> SC  SCALE      ( Increase scale by 1 )
     SC 18 = IF 0 -> DR THEN      ( change direction? )
  ELSE
     SC 1 - DUP -> SC  SCALE    ( Decrease scale by 1 )
  THEN ;

: ROLL.TETRA1
  0 -> XR  0 -> YR      ( Initia'ize variables )
  25 -> XP  35 -> YP
  0 -> SC
  1 -> DIR              ( Set scale direction )
  DRAW                  ( Draw first object )
  36 0 DO               ( Start loop )
     UPDATE.TETRA          ( Set new parameters )
     DRAW                  ( Draw object )
  LOOP ;                ( Loop back )
```

We used a DO-LOOP to run the animation since no branching decisions were needed for this program. If they were required, the current value of any 3-D parameter would always be available.

For smooth animation, the GraFORTH 3-D graphics routines automatically take advantage of both high-resolution screen pages in the Apple memory. During 3-D animations, one screen area is displayed while the other is being invisibly updated. This way, the lines are not shown being erased and redrawn. This is only true for 3-D graphics. GraFORTH text printing, line drawing, and character graphics always draw to both screens simultaneously. In this way, the screen-flipping 3-D graphics can be mixed with other kinds of graphics without causing lines and characters to repeatedly appear and disappear.

The sequence GraFORTH uses in putting a 3-D object on the screen is a four-step process: Whenever the word DRAW is executed, the drawing routines are first directed to the graphics screen that is not currently being displayed. Then the previous 3-D objects are individually erased line-by-line by following the parameters that were originally used to draw them. Next, the new objects are drawn on the screen using the current parameters. Lastly, the display is switched to this screen, so that the new objects can be seen.

To increase speed, the word DRAW only works with the objects that have been referenced since the last DRAW command. This reference can be made by giving the object one or more new parameters, or by simply calling it again with OBJECT. This means that objects that don't need to be changed can be left on the screen as they are, and will not slow the drawing of objects still in motion.

Suppose you're manipulating two 3-D objects (call them objects 1 and 2) simultaneously. First, both of them are in motion, and the animation toggles between the two graphics screens with each DRAW command. Then you decide to stop the motion of object 1, while continuing object 2. To do this, you simply stop giving object 1 any new commands. Since object 1 was just in motion, the picture of the object on the two graphics screens is different. As the animation contimues with object 2, the display will switch back and forth between the two screens. The two pictures of object 1 will alternate back and forth, rather than remaining still.

The solution to this problem is simple: When you don't need to move an object any more, give it one extra OBJECT command, without any new parameters:

```
1 OBJECT
```

This will cause the same picture of the object to be drawn on the second graphics screen. The two pictures of the object will then



be identical, and the object will remain still while other objects are manipulated.

Here then is a quick overview for doing 3-D graphics with examples to follow:

1. Load the image(s) into memory:

```
CR 132 PUTC PRINT " BLOAD CUBE,A2816 " CR
CR 132 PUTC PRINT " BLOAD HOUSE,A3000 " CR
```

2. Initialize GraFORTH's 3-D variables:

```
OBJERASE
```

3. Select object numbers and the image addresses for those objects:

```
0 OBJECT  2816 OBJADR
1 OBJECT  2816 OBJADR
2 OBJECT  3000 OBJADR
```

4. Initialize the position and orientation for each object. (This could be combined with providing the image address.):

```
0  OBJECT 5 SCALE
20 XROT  20 YROT
50 XPOS  40 YPOS

1  OBJECT  180 XPOS
10 SCALE     6 SCALZ

2  OBJECT
15 SCALZ  10 SCALE
50 XPOS  110 YPOS
```

5. For each picture to be drawn, execute a DRAW command:

```
DRAW
```

6. To continue animation of all objects, again select each object in turn, provide any new parameters, and call DRAW:

```
0 OBJECT  25 YROT
1 OBJECT   8 SCALE
2 OBJECT  65 XPOS
DRAW
```

7. If you want to stop the motion of one object while continuing to change others, call the object one more time (without any new parameters) to draw it again, to prevent residual motion:

```
1 OBJECT
```

Moving faster: With a little extra planning, the speed of 3-D graphics can often be increased considerably. The line-by-line undrawing of each 3-D object uses as much time as drawing the new object. A faster method to remove old images is to simply erase the area of the screen the object lies in, and then not bother doing a line-by-line erase.

The GraFORTH word UNDRAW is designed for doing just this. UNDRAW erases a portion of the screen just as UNBLK does, on a character-size basis. However, UNDRAW also sets a flag telling GraFORTH not to do a line-by-line erase of the 3-D object. After setting the blocksize and the position appropriately, you can erase the object yourself, so that the 3-D routines don't have to erase it. This method requires that you know what rectangular area of the screen is used by the object, and that no other graphics lie in this area, since they would also be erased.

Here is an example of using UNDRAW. Starting from scratch, let's first get an object onto the screen:

```
0 40 18 24 WINDOW ERASE    ( Optional )
CR 132 PUTC PRINT " BLOAD CUBE,A2816 " CR
OBJERASE
0  OBJECT 5 SCALE
20 XROT  20 YROT
DRAW
```

An easy way to determine the blocksize and placement to use with UNDRAW is to fill the screen with characters, then draw the object over them:

```
0 VTAB  1000 0 DO I 10 MOD . LOOP  0 OBJECT DRAW
```

By simply counting down and across, you can see that the cube fills a block 9 characters wide by 8 characters tall, starting at 8 VTAB 14 HTAB. The UNDRAW command can be used to erase this block during a 3-D animation:

```
ERASE
9 8 BLKSIZE
```



Now type this entire line, then press the return key:

```
8 VTAB  14 HTAB  257 0 DO I YROT UNDRAW DRAW 4 +LOOP  18 VTAB
```

This sets the character position for the block and rotates the object while erasing the block with UNDRAW. Compare it with the same loop without UNDRAW:

```
257 0 DO I YROT DRAW 4 +LOOP
```

The difference in speed is quite noticeable.

Next month, we'll continue with 3-D graphics, describing GraFORTH's internal 3-D object table and the format for 3-D images in memory. We'll also include a space shuttle simulation program, with a complete discussion of how it works. See you then.

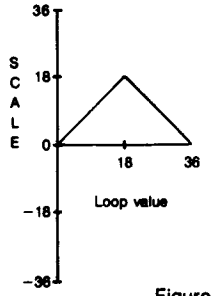
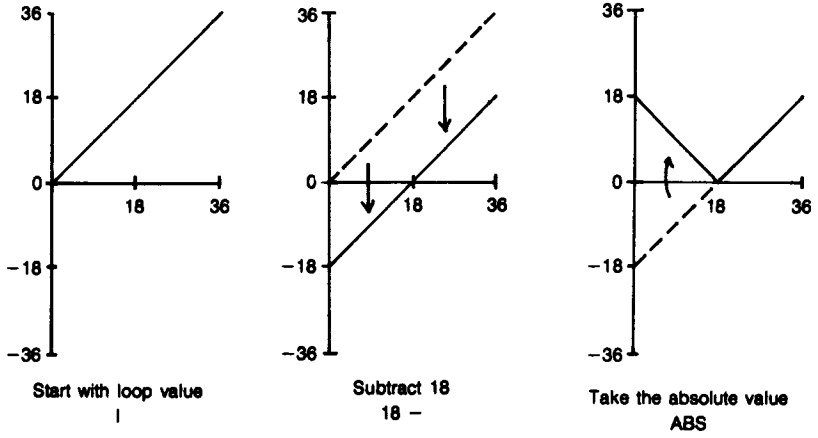



Figure 1.



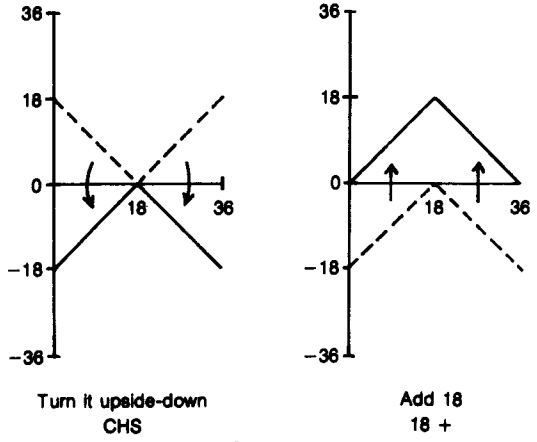




Figure 2

# More Advanced 3-D Graphics



Last month's introduction to GraFORTH's three dimensional graphics features showed some straightforward methods for manipulating 3-D graphics. With a better understanding of how GraFORTH handles things internally, we can create some interesting new effects.

Remember that using 3-D graphics is a two-step process: First the Image Editor is used to create a 3-D 'image' in memory. Then GraFORTH's 3-D commands are used to display the image as an 'object' on the screen, with a given position, size, and orientation. The object commands do not affect the image in memory; they only change the way it is displayed.

For some 3-D animations, you may want to have several images in memory at one time. In order to keep track of what free areas of memory are available, it's handy to know how the images are stored.

A 3-D image is made up of a number of line entries, one for each Move or Draw in the image. Each line entry uses four bytes of memory. The format for the entry is described fully on page B-6 of the GraFORTH manual. Briefly, the first byte determines color and whether the entry is a Move or a Draw, and the next three bytes specify the X, Y, and Z positions for the point in space. The end of a 3-D image is marked with a 255 (hex $FF) in memory immediately after the last entry. The length in bytes of a 3-D image is then 4 bytes times the number of entries, plus 1 more byte for the end-of-image marker.

From the Image Editor, the List and Enter options show the address in memory for the beginning of each line entry. An image starting at address 2816 will have line entries at 2816, 2820, 2824, etc. The last address shown in the listing is for the beginning of the last line entry. The end-of-image marker will be 4 bytes after this. The byte immediately after the end-of-image marker (5 bytes after the last listed address) is the next byte free for use.

Loading the Image Editor every time you want to find the length of a 3-D image can be cumbersome. The following short word definitions will also do the job. With a 3-D image already in memory, END.3D can be used to find the next free location after the image, and LENGTH.3D will return the length in bytes of the image:

```
: END.3D
  BEGIN
     DUP
     PEEK 255 <>
  WHILE
     4 +
  REPEAT
  1 + ;

: LENGTH.3D
  DUP END.3D
  SWAP - ;
```

To use END.3D, simply place the starting address of the image on the stack, then call END.3D. The routine scans through the list of 4-byte line entries until it finds the 255 end-of-image marker. It then adds 1 to this address, leaving the next free address on the stack.

LENGTH.3D is called the same way. It uses END.3D to find the next free address, then subtracts the original starting address from this value to find the length.

Here is an example which uses END.3D and LENGTH.3D to find the length of the XYZ image stored on the GraFORTH system disk. First, load the XYZ image into memory starting at location 2816:

```
CR 132 PUTC PRINT " BLOAD XYZ,A2816 " CR
```

To find the next free address after the XYZ image, enter:

```
2816 END.3D .
2877
```

2877 is the address of the next free area of memory. This means that the XYZ end-of-image (255) marker must be one location before that:

```
2876 PEEK .
255
```



The length of the XYZ image can be found by typing:

```
2816 LENGTH.3D .
61
```

The Keep command in the Image Editor allows you to save 3-D images to disk, one image per binary file. If you're working with a graphics program that uses several images, it's often easier to combine all of the images into one disk file. By keeping track of image length information, you can BLOAD several images into memory one immediately after another, then BSAVE the images back to disk as a single binary file. This can save a lot of space on a crowded disk, and make it easier to load several images at one time. The 3-D bat used in "Die Fledermaus" in the demonstration program is an example of this. Three separate images making up the bat are stored together in the binary file BAT.

Another technique is to combine two 3-D images together into one larger image. We can create a "double" image by concatenating the XYZ image with the CUBE image, also on the GraFORTH disk. First, let's get the XYZ shape displayed on the screen:

```
OBJERASE
0 OBJECT 2816 OBJADR
0 40 20 24 WINDOW ERASE
12 SCALE 4 SCALZ 80 YPOS
10 XROT 20 YROT DRAW
```

We can BLOAD the CUBE image right over the end of XYZ, so that the first line entry in the cube overwrites the end-of-image marker for XYZ. This leaves a single image with a set of line entries for XYZ and a set for the cube, with one end-of-image marker left at the end.

We know that the XYZ end-of-image marker is at location 2876, so this is where we load the cube:

```
CR 132 PUTC PRINT " BLOAD CUBE,A2876 " CR
0 OBJECT DRAW
```

Other unusual effects are possible by changing line entries from a running program, moving end-of-image markers, etc. Keep in mind, however, that the image in memory is also used by GraFORTH when erasing old objects from the screen before redrawing. If you modify images while displaying them, you might affect what lines from old objects are erased and what lines aren't.

Another internal aspect of 3-D graphics worth looking at is the Image Data Map (described on page B-4 of the GraFORTH manual), which stores the parameters used to orient and draw objects on the screen. (A more accurate name would be "Object Data Map", but we'll stick to the name used in the manual.) The map keeps track of all of the 3-D parameters for 16 objects, for three different functions: erasing the old picture, keeping track of the picture on the other graphics screen, and drawing the new picture. The manual calls these data sets Undraw, Interim, and Draw.

Last month's column included a program which used variables to keep track of 3-D parameters. However, the Image Data Map already stores this same information, and can be used the same way. The technique is to determine the address for the desired parameter of a given object, then take a PEEK at that address to find the current value of the parameter.

The Image Data Map is first broken into the three data sets, with 16 object tables for each data set, and 16 bytes for each object table. Here is the format for object tables:

| Function | Relative Byte |
|---|---|
| Flag (draw, nodraw) | 0 |
| XROT | 1 |
| YROT | 2 |
| ZROT | 3 |
| XTRAN | 4 |
| YTRAN | 5 |
| ZTRAN | 6 |
| XPOS | 7 |
| YPOS | 8 |
| SCALX | 9 |
| SCALY | 10 |
| SCALZ | 11 |
| OBJCOLOR | 12 |
| Image Address | 13 and 14 |
| (unused) | 15 |

The starting addresses for the data sets are:

| | | |
|---|---|---|
| 5888 | $1700 | Undraw |
| 6144 | $1800 | Interim |
| 6400 | $1900 | Draw |

Here is an example to show how it all works: Suppose you want to



find the current X position (XPOS) for object 5. Start with the base address for the Draw data set at 6400. Multiply the object number by 16 and add this to the base address:

6400 + (5 * 16) = 6400 + 80 = 6480

Now add the Relative Byte for the XPOS command, which is 7:

6480 + 7 = 6487

6487 is the address which stores the X position for object 5. You can find the current value by PEEKing at this address.

One application of this is to define words which update a 3-D parameter by PEEKing its value, adding some offset, then resetting the parameter. Here is a word which will increase the X position for object 5 by 2 pixels:

```
: INCX
  5 OBJECT
  6487 PEEK 2 + XPOS ;
```

The following is another version of the ROLL.TETRA program shown last month. It uses the Image Data Map to update parameters, so that separate variables are not needed:

```
0 VARIABLE OBJ  ( Object number )
  VARIABLE DIR  ( Scale direction larger or smaller? )

: FIND       ( Finds address of parameter given relative byte: )
  OBJ 16 * +     ( Object number * 16 + relative byte )
  6400 + PEEK ;  ( Add base address and PEEK current value )

: UPDATE.TETRA
  1 FIND 3 +  XROT   ( Increase X rotation by 3 )
  2 FIND 5 +  YROT   ( Increase Y rotation by 5 )
  7 FIND 6 +  XPOS   ( Increase X position by 6 )
  8 FIND 3 +  YPOS   ( Increase Y position by 3 )
  DIR IF                    ( If scale is increasing: )
     9 FIND 1 +  SCALE      ( Increase scale by 1 )
     9 FIND 18 = IF 0 -> DIR THEN      ( change direction? )
  ELSE
     9 FIND 1 -  SCALE    ( Decrease scale by 1 )
  THEN ;
```

```
: ROLL.TETRA2
  OBJ OBJECT            ( Set object number )
  0 XROT  0 YROT        ( Initialize parameters )
  25 XPOS  35 YPOS
  0 SCALE
  1 -> DIR              ( Set scale direction )
  36 0 DO               ( Start loop )
     UPDATE.TETRA          ( Set new parameters )
     DRAW                  ( Draw object )
  LOOP ;                ( Loop back )
```

Special provision is made to work with any object, regardless of what object number is used. The object number is kept in the variable OBJ. Note the new word FIND. This takes the relative byte for a 3-D parameter from the stack, finds the address for that parameter, and PEEKs the current value.

To run the program, first BLOAD the tetrahedron image into memory:

```
CR 132 PUTC PRINT " BLOAD TETRA,A2816 "
OBJERASE
0 OBJECT 2816 OBJADR
```

Now store the object number into OBJ and call ROLL.TETRA:

```
0 -> OBJ
ROLL.TETRA
```

If you want to use other images with ROLL.TETRA, you may need to adjust the scale so that the object fits on the screen.

Another aspect of 3-D graphics to consider is the way in which images are transformed into objects on the screen. The Three-Dimensional Mathematical Method on page B-5 of the manual describes some of the technical wizardry performed on the points using matrices. Understanding the details of the math isn't really important. What is important is the order in which the operations are performed.

The 3-D transformations are performed in this order:

1. X, Y, and Z translation
2. X, Y, and Z scaling
3. X rotation
4. Y rotation
5. Z rotation



This means that the 3-D points in an object are first translated, then the translated points are scaled. These new points are rotated around the X, Y, and Z axes in order. (Actually, GraFORTH does steps 2 through 5 simultaneously, though the mathematics used treats the objects as if the transformations were done one at a time.)

The most noticeable effect of this order is with rotations. Consider an object that has been set to 30 XROT 20 YROT. The X rotation is performed first. Remember that the X-axis passes through the object from left to right. A rotation around this axis tips the top of the object forward, and the bottom back. The Y rotation is done next. But since the X rotation occurred first, the Y-axis has been tipped forward, along with the points in the object. The Y rotation will actually be done on this new tipped axis. Similarly, any Z rotation that follows will rotate around an axis that has already been tipped by both X and Y rotations.

Suppose you want to create an animation of a 3-D spaceship that continually banks or rolls as it flies across the screen. This rolling should be the same regardless of which way the ship is facing. Before designing the ship with the Image Editor, you need to know which axis (X, Y, or Z) to use for the rotation. The Z-axis is the proper choice for this kind of situation. The rotations around the X and Y axes will first set the ship (and the Z-axis) facing the desired direction, then the Z-axis rotation will perform the rolling in that direction.

This is best demonstrated using the program PLAY. Compile PLAY into memory:

```
READ " PLAY "
```

Select the 3-D shape XYZ from the disk. When the object appears, rotate it a little so that you can see all three arrows: Press 2, right-arrow, F, 1, right-arrow, F. The white arrow pointing from back to front corresponds to the Z-axis. Start the object rotating around this axis at a good rate by pressing 3 and the right-arrow key about 6 times. Notice that the rotation does occur around the white arrow.

Now press either 1 or 2 and an arrow-key, wait a bit, then press F. This changes the rotation around the X or Y axis. Note that even though the white arrow is now facing a new direction, the Z rotation is still around the white arrow. This is because the X and Y rotations tip the Z-axis before the Z rotation is done.

Because of these "rotation gymnastics", it is occasionally handy to have the ability to "transpose" a 3-D image in memory, changing X coordinates into Y coordinates, Y to Z, Z to X, etc. This has the effect of turning an image around so that it lies in a new direction along the three axes.

The following word definition does just that. It reads each line entry of an image in memory, plucks out the values for the (X,Y,Z) point, and places them back into the image in a new order:

```
VARIABLE ADDR
VARIABLE FIRST
VARIABLE SECOND
VARIABLE THIRD
VARIABLE X
VARIABLE Y
VARIABLE Z

: TRANSPOSE
  -> THIRD -> SECOND -> FIRST  ( Save new order )
  -> ADDR                      ( Save starting address )
  BEGIN
     ADDR PEEK 255 <>          ( While not end-of-image: )
  WHILE
     ADDR 1 + PEEK -> X        ( PEEK X, Y, and Z values from
                                 image )
     ADDR 2 + PEEK -> Y
     ADDR 3 + PEEK -> Z
     X  ADDR FIRST +  POKE     ( POKE values back into image in
                                 new order )
     Y  ADDR SECOND +  POKE
     Z  ADDR THIRD +  POKE
     ADDR 4 + -> ADDR          ( Increment address to next line
                                 entry )
  REPEAT ;
```

Before calling TRANSPOSE, four values should be on the stack:

<image address> <new X place> <new Y place> <new Z place>
TRANSPOSE

The numbers for the places should be 1, 2, and 3, in the desired order: 1 for X, 2 for Y, and 3 for Z. For example, to trade the X and Y coordinates in an image at location 2816, you would enter:

2816 2 1 3 TRANSPOSE



To run TRANSPOSE without exchanging coordinates (a do-nothing operation), you would type:

2816 1 2 3 TRANSPOSE

The numbers 1, 2, and 3 must be on the stack in some order, or the image will be destroyed. For example, typing:

2816 2 2 2 TRANSPOSE

will copy all three coordinates in turn into the Y coordinate position, with the Z coordinate copied last, losing the old Y value.

The concepts we've shown are more than isolated ideas. They can be combined to produce interesting new animations. Below is a space shuttle simulation program which makes extensive use of 3-D graphics. The program shows the shuttle flying into view over the Earth, performing some rotations, releasing a satellite, and flying away. The simulation makes use of a number of concepts, including multiple 3-D objects, UNDRAW, GraFORTH's Image Data Table, and Z-axis rotations.

The first step is to create the shuttle using the Image Editor. Below are the line entries that make up the shuttle. READ the Image Editor onto the word library, RUN the program, and type "Z" to zero any image (or garbage) that may be in memory. You can set a viewing angle before entering points. Enter a scale of 14, X-rotation of 20, and Y-rotation of 32. Now press "E" to enter the points. The general shape of the shuttle should become apparent after the first 15 or 20 line entries. When asked for color at each line entry, just press return.

Note that the shape is somewhat complicated. This gives the shuttle greater detail, but requires more time to draw. For speed, you may later want to design a simpler shuttle.

Body

| | | X | Y | Z |
|---|---|---|---|---|
| 1. | Move | -12 | -20 | 64 |
| 2. | Draw | -12 | -20 | 80 |
| 3. | Draw | -12 | -10 | 90 |
| 4. | Draw | -4 | 0 | 127 |
| 5. | Draw | 4 | 8 | 127 |
| 6. | Draw | 20 | 16 | 100 |
| 7. | Draw | 20 | 16 | -120 |
| 8. | Draw | -20 | 16 | -120 |

```
 9. Draw -20  16  100
10. Draw  -4   8  127
11. Draw   4   0  127
12. Draw  12 -10   90
13. Draw  12 -20   80
14. Draw  12 -20   64
15. Move  20  16 -120
16. Draw  20   0 -120
17. Draw  17 -12 -120
18. Draw   8 -20 -120
19. Draw  -8 -20 -120
20. Draw -17 -12 -120
21. Draw -20   0 -120
22. Draw -20  16 -120
23. Move   8 -20 -120
24. Draw   8 -20  -64
25. Move  -8 -20 -120
26. Draw  -8 -20  -64
27. Move  20  16  100
28. Draw -20  16  100

Wings

29. Move -20  16 -112
30. Draw -88  16 -112
31. Draw -88  16  -84
32. Draw -36  16  -10
33. Draw -20  16   80
34. Move  20  16 -112
35. Draw  88  16 -112
36. Draw  88  16  -84
37. Draw  36  16  -10
38. Draw  20  16   80

Cargo opening

39. Move  20   0  -64
40. Draw  17 -12  -64
41. Draw   8 -20  -64
42. Draw  -8 -20  -64
43. Draw -17 -12  -64
44. Draw -20 -12  -64
45. Draw -20   0  -64
46. Draw -20   0   64
47. Draw -17 -12   64
48. Draw  -8 -20   64
49. Draw   8 -20   64
50. Draw  17 -12   64
```



```
51. Draw  20   0   64
52. Draw  20   0  -64

Fin

53. Move   0 -20 -120
54. Draw   0 -60 -128
55. Draw   0 -60 -112
56. Draw   0 -20  -84

Open doors

57. Move  20   0  -64
58. Draw  33  -3  -64
59. Draw  40 -12  -64
60. Draw  40 -20  -64
61. Draw  40 -20   64
62. Draw  40 -12   64
63. Draw  33  -3   64
64. Draw  20   0   64
65. Move -20   0  -64
66. Draw -33  -3  -64
67. Draw -40 -12  -64
68. Draw -40 -20  -64
69. Draw -40 -20   64
70. Draw -40 -12   64
71. Draw -33  -3   64
72. Draw -20   0   64
```

After creating the shuttle, press "K" to save the image to disk. Use the filename "SHUTTLE". At this point, you might want to use the PLAY program to check the shuttle image more closely. Remember to forget the Image Editor program ("FORGET X") before typing READ " PLAY ". From PLAY, you can easily see the shuttle from all angles.

The next step is to create the satellite. This is a simple shape created with the PROFILE program. (PROFILE is described in greater detail on pages 8-18 to 8-21 in the GraFORTH manual.) Use FORGET to remove any other programs from the word library, then type:

```
READ " PROFILE "
RUN
```

Answer the questions as follows:

```
Enter number of polygon sides : 6
```

```
Enter Object File Address : 3500
Data from [K]eyboard or [D]isk ? K
Enter X,Y pair (end = "E") : 0,20
Enter X,Y pair (end = "E") : 20,0
Enter X,Y pair (end = "E") : 0,-20
Enter X,Y pair (end = "E") : E
```

The program will generate a small 6-sided diamond-like shape. This is the satellite. As prompted, save the image to disk with the name SATELLITE.

We now have the 3-D images needed for the simulation. The program must be entered next. Clear the word library, then type:

```
TEXT EDIT
```

to enter the text editor. (Calling TEXT switches GraFORTH out of graphics mode, so that scrolling in the editor will be much faster.) Type "E" to erase the editor memory, then enter the following word definitions. As usual, the comments are optional:

```
: SETUP
  CR 132 PUTC PRINT " BLOAD SHUTTLE,A2816 "
  CR 132 PUTC PRINT " BLOAD SATELLITE,A3500 " CR
  OBJERASE
  0 OBJECT 2816 OBAJDR
  1 OBJECT 3500 OBJADR 0 SCALE
  39 19 BLKSIZE ;

: PARAM
  SWAP 16 * 6400 + +          ( Retrieve address of parameter )
  PEEK ;                      ( & PEEK current value there )
```



```
: FLY.IN
  ERASE  0 VTAB 0 HTAB          ( Set position for UNDRAW )
  PENUP
  0 191 MOVETO  60 TURNTO       ( Initialize Turtle & )
  PENDOWN
  63 0 DO  4 MOVE  1 TURN  LOOP ( Draw outline of Earth )
  0 OBJECT
  -20 XROT  32 YROT  128 ZROT   ( Set initial position of
                                  shuttle )
  18 XPOS  132 YPOS  0 SCALE
  21 0 DO                       ( Fly shuttle into view: )
     0 7 PARAM 5 + XPOS         ( Move to the right )
     0 8 PARAM 3 - YPOS         ( Move upward )
     0 9 PARAM 1 + SCALE        ( Increase in size )
     UNDRAW DRAW
  LOOP ;

: ROTATE1
  -1 124 DO
     I ZROT                     ( Roll to upright position )
     UNDRAW DRAW
  -4 +LOOP
  21 -18 DO
     I XROT                     ( Tip down for better view )
     UNDRAW DRAW
  2 +LOOP ;

: RELEASE
  1 OBJECT 128 XPOS 66 YPOS     ( Select and position satellite )
  16 SCALE 20 XROT 32 YROT
  10 0 DO
     0 OBJECT
     0 8 PARAM 2 + YPOS         ( Move shuttle down )
     1 OBJECT
     1 8 PARAM 2 - YPOS         ( Simultaneously move satellite
                                  up )
     DRAW
  LOOP ;
```

```
: DRIFT.AWAY
  0 OBJECT                      ( Select to redraw shuttle )
  1 OBJECT                      ( Select satellite )
  -1 15 DO                      ( Quickly drift away: )
     I SCALE                    ( Decrease in size )
     1 7 PARAM 3 + XPOS         ( Move to the right )
     1 8 PARAM 3 - YPOS         ( Move up )
     DRAW
  -1 +LOOP ;

: ROTATE2
  0 OBJECT                      ( Reselect shuttle )
  105 32 DO
     I YROT                     ( Pivot around )
     UNDRAW DRAW
  4 +LOOP
  -21 18 DO
     I XROT                     ( Tip down )
     UNDRAW DRAW
  -4 +LOOP ;

: FLY.OUT
  21 0 DO                       ( Fly away: )
     0 7 PARAM 5 + XPOS         ( Move to the right )
     0 8 PARAM 3 + YPOS         ( Move down )
     0 9 PARAM 1 - SCALE        ( Decrease in size )
     UNDRAW DRAW
  LOOP ;

: FLY.SHUTTLE
  FLY.IN  ROTATE1               ( Call each part one at a time )
  RELEASE  DRIFT.AWAY
  ROTATE2 FLY.OUT ;
```

Save the program to disk using the filename "FLY.SHUTTLE". Note that this is also the name of the last word in the program.

All of the "tools" needed for the simulation are now on disk. Since the Turtlegraphics capabilities are used to draw the outline of the Earth, the TURTLE file must be read into memory. Type:

```
READ " TURTLE "
READ " FLY.SHUTTLE "
SETUP
FLY.SHUTTLE
```

The shuttle flies! To run the program again, you only need to



type "FLY.SHUTTLE".

Let's take a closer look at how each of the words work:

SETUP simply loads the images into memory and selects them as 3-D objects. Notice that the satellite object is also set to 0 SCALE. Since the satellite has been selected, this will prevent it from being drawn at the first frame of the animation.

PARAM is called by later words. It reads a parameter value from the GraFORTH Image Data Map much like the word FIND (discussed above) did. However, PARAM removes two numbers from the stack, one to select the object number, and the other to select the relative byte for the desired parameter.

FLY.IN begins the simulation. It first clears the screen, then uses Turtlegraphics to draw an arc along the bottom of the screen. This is an outline of the Earth. It sets the initial orientation for the shuttle, then uses a DO - LOOP to repeatedly draw the shuttle while changing its scale and position. Each parameter is updated by using PARAM to read its current value from GraFORTH's Image Data Map, adding an offset, then resetting the parameter.

Note that UNDRAW is being used to speed up drawing. An interesting trade-off occurs here: The shuttle is a complicated shape and is being drawn with a fairly large size. Both of these contribute to slower drawing times. UNDRAW can be used to speed up the animation, but a large blocksize is needed to cover the object. Erasing a large area also takes time. Even though the blocksize covers nearly the entire screen (39 19 BLKSIZE), UNDRAW is still faster.

ROTATE1 performs two rotations on the shuttle, one after another. The first rotation rolls the ship around the Z-axis to an upright position. The shuttle was designed with its body along the Z-axis so that this rotation could be done regardless of which way the shuttle was facing. The second rotation simply tips the shuttle down for a better view.

RELEASE causes the satellite, which is 1 OBJECT, to (magically!) appear in the shuttle's cargo hold. The satellite then moves up 20 pixels while the shuttle moves down, lifting the satellite away from the shuttle. The drawing time is a little slower here because UNDRAW is not used. If it were used, it would have to be called for both 0 OBJECT and 1 OBJECT, since calling UNDRAW prevents the automatic erasing of only the currently select object.

DRIFT.AWAY causes the satellite to quickly drift into the distance and disappear. Note that the shuttle (0 OBJECT) is referenced once before the satellite is moved. This puts the same view of the shuttle on both graphics screens, preventing residual motion. UNDRAW is not used here, because the satellite shape is simple, small, and therefore fast.

ROTATE2 rotates the shuttle into position for FLY.OUT.

FLY.OUT again adjusts SCALE, XPOS, and YPOS to fly the shuttle away and out of sight.

FLY.SHUTTLE performs the entire animation by simply calling each of the segments in turn.

This shuttle simulation program was designed to give you a better idea of how more complicated three-dimensional animations are implemented. We hope this removes some of the "magic" behind creating 3-D graphics displays.

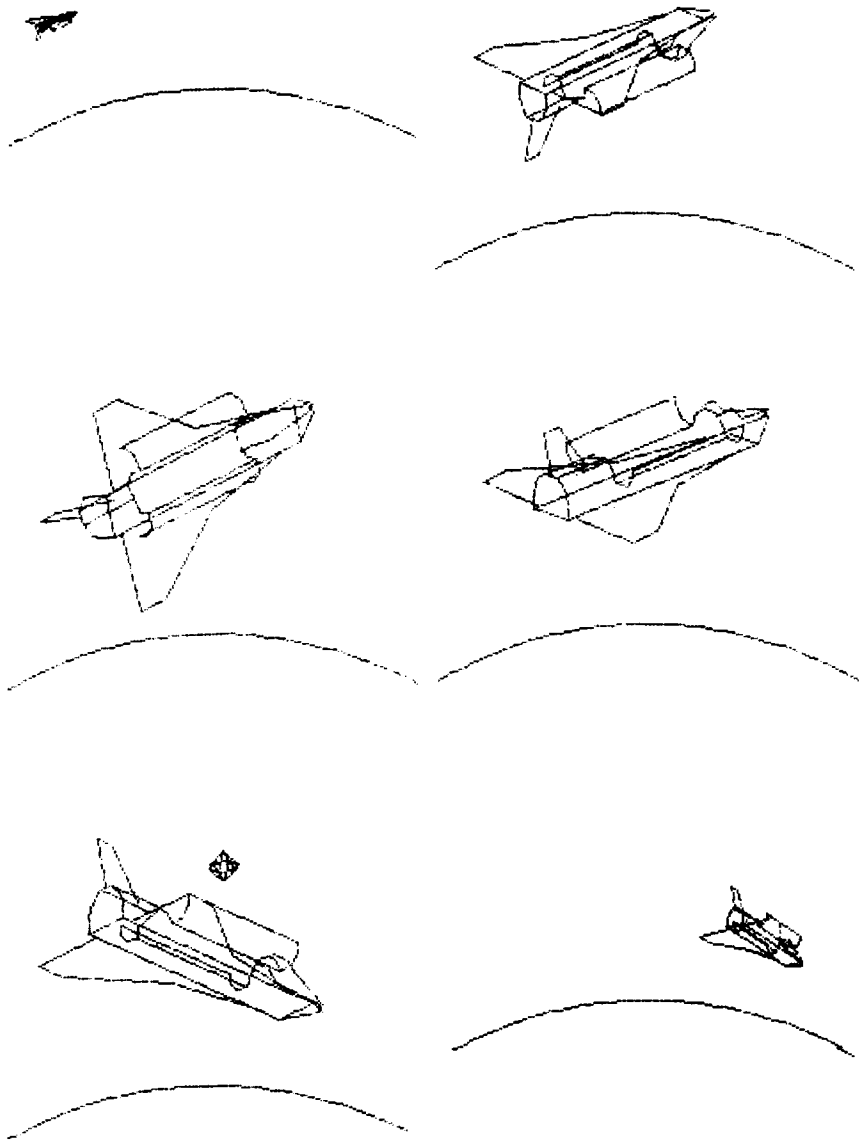


**Figure 1.**

# Music and Sound Effects

Music and Sound Effects

As a graphics language, GraFORTH's usual emphasis lies in creating fast high-resolution animations. One feature that is frequently overlooked is GraFORTH's ability to produce music and sound effects using a built-in synthesizer. In this column we'll explore GraFORTH's sound capabilities more closely, with examples of sound effects and a song generating program.

The music synthesizer is controlled by two GraFORTH words, NOTE and VOICE. NOTE actually plays the notes and VOICE determines the tone quality of the notes played.

NOTE removes two numbers from the stack, for pitch and duration. Greater pitch numbers produce lower notes, and greater duration values increase the time the note plays. Both pitch and duration numbers can range from 2 to 255. For example,

```
255 255 NOTE
```

plays a low note with a long duration.

```
50 2 NOTE
```

plays a very short, medium-pitched note. In fact, the note is almost too short to be recognized as more than a "click". However, curious effects can be obtained by repeating short notes:

```
25 0 DO 50 2 NOTE LOOP
```

By experimenting and choosing appropriate values, music can be played. The following word definition plays the first phrase from "Twinkle Twinkle Little Star":

```
: TWINKLE
104 100 NOTE
104 100 NOTE
69 100 NOTE
69 100 NOTE
62 100 NOTE
62 100 NOTE
69 150 NOTE ;
```

Notes played using the NOTE command can be given any one of a



number of tone qualities, or voices, with the GraFORTH command VOICE. VOICE removes a number from the stack to select the tone quality which will be used for subsequent NOTE commands. Valid voice numbers range from -6 to 2:

-6 VOICE to -1 VOICE produce notes with constant volume. Each voice has a different volume and tone quality. -1 VOICE produces the loudest note, with the "flattest" sound. (VOICE values less than -2 sometimes produce inconsistent volumes for notes of different pitches, and are more suitable for sound effects than music.)

0 VOICE through 2 VOICE produce notes with varying volumes. A note played with 0 VOICE begins loudly, then dies away. 1 VOICE causes a note to increase in volume, then decrease again. 2 VOICE plays a note with an increasing volume.

When GraFORTH first starts up, 0 VOICE is automatically set. You might want to try the TWINKLE routine after setting various voices:

```
-1 VOICE TWINKLE
-3 VOICE TWINKLE
2 VOICE TWINKLE
```

On page 9-4 of the GraFORTH manual is a table of musical notes and their corresponding pitch numbers. This table was used to determine the pitch numbers for the above TWINKLE example. Note that the numbers 104, 69, and 62 refer to the notes C, G, and A respectively.

For short musical phrases, manually looking each pitch up in a table is satisfactory. For longer tunes, however, this can quickly become tedious. The COMPUTE.NOTES example in the manual solves this problem by creating a pitch table in the Apple's memory. The table has 48 entries spanning 4 octaves. By indexing into the table, the notes can now be represented with the numbers 0 through 47.

The examples on page 9-5 in the manual are inconsistent. Note that the definition of GETPITCH and the first example have GETPITCH print the pitch value on the screen. Subsequent examples do not print the value, but leave it on the stack to be used by the NOTE command. The definition and the first example should be changed to agree with the other examples. Here is the correct definition for GETPITCH:

```
: GETPITCH  PITCH PEEK ;
```

(PITCH and COMPUTE.NOTES must have already been compiled and COMPUTE.NOTES executed for these examples to work.) GETPITCH can be used to convert a note number into a pitch value to be used by the NOTE routine. This example (correcting the example in the manual) prints the pitch value for note number 3, which is a C in the first octave:

```
Ready 3 GETPITCH .
209
```

Here the pitch value is used to actually play the note:

```
Ready 3 GETPITCH 128 NOTE
```

(The note sounds.)

The following word definition is equivalent to the above TWINKLE routine, except that GETPITCH is used to retrieve the pitch value for each note. The actual note names are included in the comments:

```
: TWINKLE2
15 GETPITCH 100 NOTE     ( C, octave 2 )
15 GETPITCH 100 NOTE
22 GETPITCH 100 NOTE     ( G, octave 2 )
22 GETPITCH 100 NOTE
24 GETPITCH 100 NOTE     ( A, octave 3 )
24 GETPITCH 100 NOTE
22 GETPITCH 150 NOTE ;   ( G, octave 2 )
```

The NOTE command is designed for playing notes, but has no built-in capacity for playing rests of a similar duration. The following word, REST, removes a duration number from the stack, and simply waits for this amount of time. The durations used are nearly identical to those used by NOTE:

```
: REST
45 * 0 DO LOOP ;
```

This example will play two quarter notes of "C" in the first octave, separated by a quarter note rest:

```
3 GETPITCH 64 NOTE  64 REST  3 GETPITCH 64 NOTE
```

Music Playing Programs. As you can guess by the TWINKLE2 example, playing entire songs by entering a long string of NOTE commands can take a lot of time and use a lot of memory. What is



needed is a program that allows you to enter songs in a more convenient form, optionally save them to disk, and play them at any time. The easiest way to enter notes would be to actually type in the note names as text, but this text must at some time be converted into numeric pitch and duration values for the NOTE command. The numeric values also use less memory.

The programs in Listings 1 and 2 use a two-step approach for generating and playing music. Using the first program, COMPILE.SONG, you can enter note names and durations from the keyboard, or have the program read the commands from a textfile on disk. COMPILE.SONG converts these lines into a list of note numbers in memory. This song list can be saved back to disk as a binary file. The second program, PLAY.SONG, reads the values from the list and calls NOTE repeatedly to play each note in turn. In addition to playing notes, COMPILE.SONG and PLAY.SONG also allow you to play rests and change voices during the song.

The two-step technique used by these programs is similar to the method used by the 3-D PROFILE program. PROFILE converts a set of X,Y points in a text format into a list of 3-D image values in memory. This image is then used by GraFORTH's 3-D routines to actually draw a 3-D object. This similarity should help clarify the way these music programs work.

Using the GraFORTH text editor (or another DOS compatible text editor, if you have one), enter the program in Listing 1 into memory, and save it to disk with the name "COMPILE.SONG". (The COMPILE.SONG file is somewhat long. If you are using the GraFORTH text editor without a language card or RAMcard, you will need to adjust the editor "program position" to allot enough memory for the file. Enter the editor, type "P" to select Program Position, type "Y", then enter a new position of 34000. This will provide enough room for the COMPILE.SONG file.)

After COMPILE.SONG is saved to disk, enter the program in Listing 2, saving it as "PLAY.SONG".

Here are instructions for using these programs:

To create a new song file, first load COMPILE.SONG into memory and run it:

```
READ " COMPILE.SONG "
RUN
```

On the screen will appear:

GraFORTH Song Compiler

Enter Song File Address : 2816

This question determines where the numeric song list will appear in memory. Press Return to accept the address of 2816, or enter a new address if desired. Next you will see:

[K]eyboard or [D]isk?

You can enter the formatted music lines from the keyboard, or have the program read the lines from a textfile on disk. For this example, press "K" for keyboard entry.

Now you are asked to enter a music command:

Music Command (E=end) :

You can enter commands to play a note or rest, change voices, or end the song. Here is the format for a note command:

1. (optional spaces)
2. <octave number>
3. (optional spaces)
4. <note name>
5. <at least one space>
6. <duration value>

The octave number corresponds to the octave numbers in the table on page 9-4, ranging from 1 to 4. The note name is a letter from A to G. You can also add "#" to the letter for a sharp, or "/" for a flat. The duration value is simply the number used by NOTE for duration. Here are some sample note commands you can enter. Notice how each command follows the above format. (Entering these example entries will produce the "Shave and a haircut" tune.):

```
Music Command (E=end) : 2C 120
Music Command (E=end) : 1G 40
Music Command (E=end) : 1F# 40
Music Command (E=end) : 1G 40
Music Command (E=end) : 1A/ 120
Music Command (E=end) : 1G 120
```

Rests are entered by typing an "R", followed by the desired rest value. Here is the format, with the next entry for the example:



R <duration value>

Music Command (E=end) : R 120

The voice used can be changed by typing a "V" and the new voice number. Here is the format and example entry:

V <voice number>

Music Command (E=end) : V-1

The tune is finished with the following entries:

```
Music Command (E=end) : 2B 60
Music Command (E=end) : R 60
Music Command (E=end) : 2C 60
```

The command to end the song (as you might have guessed from the prompt) is the letter "E". This is required as the last entry of the song.

Music Command (E=end) : E

The program will inform you that you entered 12 lines, which have been converted into a number list 24 bytes long. You will then be asked if you want to save the list to disk. For this example, press "Y". You will be prompted for a filename. Type "SHAVE" and press Return. This disk will whir as the list is saved as a binary file named SHAVE, and the program will end.

COMPILE.SONG can also read music commands from a textfile on disk. To compile songs using this method, first use the GraFORTH text editor to create a list of music commands, then save them as a textfile. When running COMPILE.SONG, select the [D]isk option, then enter the filename of the textfile. The program will read the commands from the file and compile them as if they were entered at the keyboard. The advantage to this technique is that a music textfile can be modified or corrected using the text editor, then recompiled into a numeric song list. Keyboard entries must be reentered every time.

The program also includes error checking. If it can't interpret a line as a valid music command, or if a number is out of range, an error message will be printed. This message includes the line number where the error occurred and a display of the illegal line. The program stops compiling lines when it finds an error. The program will also exit if you press CTRL-C, or CTRL-C and

Return, for an input.

Assuming no errors occurred, a sample song list is now in memory beginning at location 2816, and also on disk. To play the song, first load PLAY.SONG into memory:

```
READ " PLAY.SONG "
```

Place the starting location of the song list (2816) on the stack, then execute PLAY.SONG:

```
2816 PLAY.SONG
```

The "Shave and a haircut" tune should play. (Whenever a song is playing, pressing any key will immediately stop the song.)

With the song list saved to disk, the song can be loaded into memory and played at any time. Simply BLOAD the song list into a free area of memory, read PLAY.SONG onto the word library, and call it with the starting address of the song on the stack.

A list of COMPILE.SONG music commands for the final part of "Stars and Stripes Forever" can be found in Listing 3. This provides an example of a longer song generated with these music programs.

Song List Format. In last month's column we described the format for three-dimensional images in memory. In the interests of equal time, we'll discuss the song list format here.

Each note entry in the list is stored as a pair of bytes in memory. The PLAY.SONG program reads each byte pair in turn and determines whether each byte is zero or nonzero. The function performed (Note, Rest, Voice, or End) is determined by this test as follows:

| 1st byte | 2nd byte | Function |
|---|---|---|
| nonzero | nonzero | Note |
| zero | nonzero | Rest |
| nonzero | zero | Voice |
| zero | zero | End song |

For the Note function, the first byte contains the pitch value, and the second byte is the duration. The nonzero byte for Rest determines the duration of the rest. For the Voice function, the nonzero value is 7 greater than the voice to be selected. Adding 7 guarantees that the voice number in memory will be nonzero.



The PLAY.SONG program subtracts 7 to convert it back to a valid voice number. A pair of zeros flags the end of the song.

Watching the Keyboard. There is one aspect of the GraFORTH NOTE command that should be mentioned here. While playing a note, the word NOTE also looks at the keyboard. If a key is pressed while a note is playing, the note will be cut short. Try executing:

```
100 255 NOTE
```

and press a key before the note would normally finish. The note will end abruptly. This feature was included to give users the ability to stop the sound without having to wait for the note to end. There is one minor drawback, however: If a key is pressed before the note begins, the NOTE routine will still sometimes click the speaker for a short moment. For a demonstration of this effect, enter the following line and press a key while the first note is still playing:

```
10 0 DO 100 255 NOTE LOOP
```

The "flutter" noise is caused by the subsequent 9 notes all clicking the speaker in turn. While this type of effect can sometimes be desirable (it is used purposely in one of the sound effects described below), it is usually unwanted. Two solutions are possible: Either stop playing notes if a key has been pressed, or clear the keyboard after every note. This second approach continues the string of notes, cutting only one note short at each keypress. Here are examples of both techniques. Try pressing a key while running each:

```
10 0 DO GETKEY 128 < IF 100 255 NOTE THEN LOOP
```

```
10 0 DO CLRKEY 100 255 NOTE LOOP
```

The NOTE routine itself can be directly modified so that it either ignores keypresses altogether, or always acts as if a key has been pressed. In game applications with a lot of sound effects, ignoring keypresses can make the sound "cleaner". (Once this modification is made, you can make the change permanent if you want by saving the GraFORTH system back to disk with SAVEPRG.) To force NOTE to ignore keypresses, type:

```
24686 24687 POKEW
```

To cause NOTE to always behave as if a key has been pressed, type:

```
24688 24687 POKEW
```

NOTE can be returned to normal by entering:

```
-16384 24687 POKEW
```

Sound Effects. The GraFORTH synthesizer provides an interesting and versatile tool for creating sound effects. On one hand, it cannot be expected to duplicate the all of the sounds generated by custom-written machine language routines. (In other words, it may be difficult to create a sound reminiscent of an opera singer in bed with a cold on a foggy London evening....) To be more specific, each NOTE command plays a note with a constant pitch for a given amount of time, with a preset voice quality. On the other hand, this format still allows a great deal of flexibility, and the GraFORTH language helps make the synthesizer very easy to use. With a little experimentation, a wide variety of sounds can be readily produced.

Below are some sample sound effects that can created with GraFORTH. This list is not at all comprehensive, but should be considered as a starting point for further experimenting. Notice that DO - LOOPs are used extensively. By substituting a few values, the sounds can change tremendously. A couple of the routines use "illegal" pitch and duration values. By changing the voice numbers, these routines can be made to behave very strangely. The word names we used are arbitrary; you may prefer other names for these sounds. (Our thanks to Max McKee of Ashland, Oregon for sound effects ideas.)

This is a short laser shot:

```
: SHOT
0 -2 DO
   I VOICE
   24 10 DO
      I 3 NOTE
   LOOP
LOOP ;
```

This effect (and others) can be shortened simply by stripping off the outside DO - LOOP:



```
: ORBIT
4 0 DO
   -6 0 DO
      I VOICE
      30 80 DO
         I 2 NOTE
      -1 +LOOP
   -1 +LOOP
LOOP ;

: RADIATION
4 0 DO
   0 -6 DO
      I VOICE
      38 24 DO
         I 2 NOTE
      LOOP
   LOOP
LOOP ;

: ROBOTISH
4 0 DO
   -3 0 DO
      I VOICE
      60 50 DO
         I 3 NOTE
      LOOP
   -1 +LOOP
LOOP ;

: FLIGHT ( THRU SPACE )
4 0 DO
   3 -6 DO
      I VOICE
      55 44 DO
         I 5 NOTE
      LOOP
   LOOP
LOOP ;

: SPEAK      ( ROBOT SPEAKING )
3 -6 DO
   I VOICE
   55 44 DO
      J 5 NOTE
   LOOP
LOOP ;
```

```
: PHONE.BELL
2 VOICE
30 0 DO
   2 7 NOTE
LOOP ;

: GLISS
75 100 DO
   I 4 NOTE
   I 19 - 4 NOTE
-2 +LOOP ;

: CHIRPING.BIRD
9 0 DO
   0 -2 DO
      I VOICE
      14 8 DO
         I 3 NOTE
     LOOP
   LOOP
LOOP ;

: WARNING.SIREN
9 0 DO
   0 -2 DO
      I VOICE
      34 20 DO
         I 4 NOTE
      LOOP
   LOOP
LOOP ;
```

Try pressing a key while running this next routine. The sound is rather obnoxious! You can duplicate the sound without pressing a key by first modifying the NOTE routine as described above to act as is a key has already been pressed.

```
: LOUD.RACKET
10 4 DO
   255 230 DO
      J 2 NOTE
      255 2 NOTE
      30 10 DO
         I 2 NOTE
      LOOP
   LOOP
LOOP ;
```



This routine is similar to the inside loop of the above routine. You can press a key or modify the NOTE routine.

```
: GUN
255 230 DO
   4 2 NOTE
   255 4 NOTE
   30 10 DO
      I 4 NOTE
   LOOP
LOOP ;

: SPACE.SHOT
0 VOICE
5 0 DO
   75 5 DO
      I 1 NOTE
   LOOP
LOOP ;

: STATION    ( MOVING SPACE STATION )
30 10 DO
   30 10 DO
      I 1 NOTE
      J 1 NOTE
   LOOP
LOOP ;

: FLUTTER
20 0 DO
   240 2 NOTE
   220 3 NOTE
LOOP ;

: ZIP.UP
4 40 DO
   I 6 NOTE
-2 +LOOP ;
```

Listing 1 - COMPILE.SONG

```
2816 VARIABLE FILE
VARIABLE POINT
VARIABLE COUNT
VARIABLE DISK
VARIABLE OCTAVE
VARIABLE PITCH
VARIABLE DURATION
VARIABLE AT.END
VARIABLE GOT.ERROR

50 STRING PITCHES
50 STRING NAMES

: SET.UP
24870
48 0 DO
   DUP 100 / I PITCHES POKE
   DUP 18 / -
   DUP 1655 / -
LOOP DROP
0 NAMES ASSIGN " A A#B C C#D D#E F F#G G#A B/B C D/D E/E F G/G A/
" ;

: GETPITCH
PITCHES PEEK ;

: CTRL-C?
131 = IF ABORT THEN ;

: PUT.END
CLOSE
0 POINT POKEW
1 -> AT.END ;

: ERROR
AT.END 0 = IF
   1 -> GOT.ERROR
   PUT.END
   CR PRINT " ERROR - LINE  "
   COUNT . CR
   PAD WRITELN CR
THEN ;
```



```
: BEFORE
0 -> AT.END  0 -> COUNT
0 -> GOT.ERROR
HOME NORMAL DECIMAL CR
PRINT " GRAFORTH SONG COMPILER "
CR CR
PRINT " ENTER SONG FILE ADDRESS :  " FILE .
26 HTAB PAD READLN PAD PEEK CTRL-C?
PAD GETNUM
VALID IF -> FILE  ELSE DROP THEN
3 VTAB 26 HTAB FILE . CLEOL CR
CR PRINT " [K]EYBOARD OR [D]ISK?  "
GETC DUP CTRL-C? DUP PUTC CR CR
196 = IF
   1 -> DISK
   CR PRINT " FILENAME :  "
   PAD READLN PAD PEEK CTRL-C?
   CR 132 PUTC PRINT " OPEN "
   PAD WRITELN
   CR 132 PUTC PRINT " READ "
   PAD WRITELN CR
ELSE 0 -> DISK
THEN
FILE -> POINT ;

: SKIP.SPACES
BEGIN
   DUP PEEK 160 =
WHILE
   1 +
REPEAT ;

: PUT.REST
1 + SKIP.SPACES GETNUM
VALID IF
   DUP DUP 1 > SWAP 256 < AND IF
      0 POINT POKE
      POINT 1 + POKE
   ELSE DROP ERROR
   THEN
ELSE DROP ERROR
THEN ;
```

```
: PUT.VOICE
1 + SKIP.SPACES GETNUM
VALID IF
   DUP DUP -7 > SWAP 3 < AND IF
      7 + POINT POKE
      0 POINT 1 + POKE
   ELSE DROP ERROR
   THEN
ELSE DROP ERROR
THEN ;

: PUT.NOTE
DUP GETNUM -> OCTAVE
VALID 0 = IF ERROR THEN
OCTAVE DUP 1 < SWAP 4 > OR IF ERROR THEN
1 + SKIP.SPACES DUP PEEKW
PUSH 0
BEGIN
   DUP DUP NAMES PEEKW I <>
   SWAP 48 < AND
WHILE
   2 +
REPEAT
POP
DUP 48 = IF ERROR THEN
DUP 24 >= IF 24 - THEN
2 / DUP 0 < IF ERROR THEN
OCTAVE 1 - 12 * +
GETPITCH -> PITCH
2 + SKIP.SPACES GETNUM -> DURATION
VALID 0 = IF ERROR THEN
DURATION DUP 2 < SWAP 255 > OR IF ERROR THEN
PITCH POINT POKE
DURATION POINT 1 + POKE ;
```



```
: DURING
BEGIN
   COUNT 1 + -> COUNT
   DISK 0 = IF
      PRINT " MUSIC COMMAND (E=END) :  "
   THEN
   PAD READLN
   PAD PEEK CTRL-C?
   PAD SKIP.SPACES
   DUP PEEK 210 = IF PUT.REST
   ELSE DUP PEEK 214 = IF PUT.VOICE
   ELSE DUP PEEK 197 = IF PUT.END
   ELSE PUT.NOTE
   THEN THEN THEN
   POINT 2 + -> POINT
   AT.END
UNTIL ;

: AFTER
CR COUNT . PRINT "  LINES,  "
POINT FILE - . PRINT "  BYTES "
CR CR PRINT " SAVE TO DISK (Y/N) ?  "
GETC DUP PUTC 217 = IF
   CR CR PRINT " FILENAME :  "
   PAD READLN PAD PEEK CTRL-C?
   CR 132 PUTC PRINT " BSAVE "
   PAD WRITELN
   PRINT " ,A " FILE .
   PRINT " ,L " POINT FILE - . CR
THEN
CR CR ;

: COMPILE.SONG
SET.UP
BEFORE DURING
GOT.ERROR 0 = IF
   AFTER
THEN ;
```

Listing 2 - PLAY.SONG

```
VARIABLE END

: SET.END
1 -> END
DROP DROP ;

: REST
45 * 0 DO LOOP
DROP ;

: SET.VOICE
DROP
7 - VOICE ;

: PLAY.SONG
0 -> END
BEGIN
   GETKEY 128 <
   IF
      DUP PEEK
      OVER 1 + PEEK
      OVER SGN 2 * OVER SGN +
      CASE:
         SET.END
         REST
         SET.VOICE
         NOTE  *
      THEN
      2 +
   ELSE SET.END
   THEN
   END
UNTIL
DROP ;
```



Listing 3 - Stars and Stripes Forever

```
V0
2F 80
2E 60
2F 20
2D 20
R 20
2F 80
2G 40
2G# 40
3A 40
3B/ 40
3B 40
3C 40
R 40
V -2
3C 80
3C 80
3B/ 40
3A 40
3A 80
2G# 40
3A 40
3A 200
R 40
2G# 40
3A 40
3A 80
2G# 40
3A 40
3C 80
3A 40
3C 40
3B/ 160
2G 80
R 40
V-1
2G 40
2G 80
2F# 40
2G 40
2G 80
2F# 40
2G 40
3B/ 240
3A 40
```

```
2G 40
3A 40
3C 80
3C 40
3D 80
3D 80
2G 160
R80
V1
3C 80
3C 80
3B/ 40
3A 40
3A 80
2G# 40
3A 40
3A 200
R 40
2G# 40
3A 40
3A 80
2G# 40
3A 40
3B/ 40
3A 40
2G 40
2E 40
2G 160
2F 80
R 40
V 2
2F 40
2F 80
2E 40
2F 40
2A/ 80
2G 40
2F 40
3F 200
R 40
V -4
2F 40
V -3
2G 40
V -2
3A 40
V -1
3C 20
```



```
R 20
V -4
2F 40
V -3
2G 40
V -2
3A 40
V -1
3C 20
R 20
V 0
2C 40
2D 40
3A 40
2G 160
2F 40
R 40
2F 40
E
```