The Graphics Magician

By Mark Pelczarski, David Lubar, and Chris Jochumson

penguin software
830 4th Avenue
Geneva, IL 60134
(312) 232-1984

The enclosed product is supplied on a disk that is NOT copy-protected. It is our intent to make this product as useful as possible, and we feel that with applications software the ability to easily make your own backup copies is a great asset. We ask that you not abuse our intentions by making copies for others. The result of such activity only helps promote the unfortunate existing situation of most software being protected, and, in our opinions, less usable. We hope that commercial success of non-protected products such as this one will signal other publishers that copy-protection is not necessary for a product's survival and help reverse the trend in protected applications software.

Although this policy may be more consumer oriented than most software publishers', do not mistake our intentions as meaning that we will not act to protect our copyrights of this manual and the programs on the enclosed disk. We consider this package to be one of the best available, and we will take whatever action necessary to protect our legal rights. We have several mechanisms for tracing copies and we will use them if needed. Please act conscientiously so that we can say that such precautions were not even necessary.

We hope you find many hours of enjoyment and use in this package.

Sincerely,

Mark W Pelczarski

Mark Pelczarski
President, Penguin Software

Part One

The Animation Routines

There are four machine language animation routines included in this package, each giving a slightly different result. There are also three different editors used for the animation routines: a shape editor (no relation to Apple shape tables), a path editor, and an animation editor. The shape editor lets you design the objects that you want to move around the screen, the path editor lets you design the paths in which the objects will move, and the animation editor allows you to put together the animation type with up to 32 objects and associated paths, creating a block of machine language code and data that's ready to be loaded into your own programs. The routines can then be called for performing the animation from any program, with a wealth of information accessible along the way.

Technically, How it works

The design of the animation system is based on the concept of "pre-shifted shapes", which is actually a descendent of character graphics. The hi-res screen corresponds to an area of memory in the computer, and fast animation is the result of being able to store the right thing in the right location at the right time, quickly. Character graphics involve storing a given bit pattern in a set of location in memory. For example, the letter 'C' when it appears on your screen is the result of 8 bytes of information being placed in the proper screen memory locations. Hi-res character generators, such as the one in The Complete Graphics System, or Synergistic Software's Higher Text, do precisely that task. Alphabets are stored as bit patterns, and the hi-res character generator puts those bit patterns in the proper memory locations.

The problem with character graphics is that the basic memory unit for storage is a byte. On the screen a byte shows as a horizontal set of seven dots (seven bits corresponding to points being on or off, and the eighth bit as a color flag--more on that later). Vertical movement is easy...the bytes with the dot pattern of your figure are just moved up or down one screen row. Horizontal movement causes a problem, though. When a byte is moved over one position, the associated figure moves seven dots! To move in smaller increments, the bits within the byte have to actually be shifted over a certain number of places into the next screen location, which is time consuming and messy.

The solution is pre-shifted shapes. For every figure created, seven facsimilies are stored as bit patterns in memory, in your shape table (remember, no relation to the shape tables Apple told you about). The bit patterns are the same, except each one is shifted over one position with respect to the byte boundaries. That way fast character graphics can be performed with totally flexible horizontal movement. There are other speed tricks mentioned later, but this is the basis of the truly fast and smooth animation you see in good, professional programs.

How Programs Use the Animation

There are a few levels used in creating the animation effects. The first is a routine that puts a designated shape on the screen at a specified location. The second is the real animator, a routine that handles up to 32 independent objects and moves them along their appointed paths. Each time the animator is called from your program, all active objects are moved one step along their paths, location and collision information is updated, and a return is made to your program. The simplest program using the animation routines would load the animation information, call the animator, and then loop back up a step to keep calling the animator. To make things interesting before you loop back up, you may want to do such things as check the keyboard or paddles, check the collision and location tables, change paths, or activate and deactivate objects. But that's where your creativity comes in...

The Four Animation Types

There are four animation routines included, each with various advantages and disadvantages. Because of space requirements, you cannot easily mix and match the types, so you should choose the one that best suits your needs. The types are block move, block move with page 2 for background, draw and save background, and XDRAW. This section describes some technical differences and should only be skimmed lightly by the beginner.

The major advantage and reason that you should usually use block moves is that an erase cycle is not needed in that mode. A typical animation loop goes draw-update-erase-draw-update-erase... The block move only uses draw-update-draw-update, and so on. When shapes are created for use with the block moves, a border the size of the maximum move must be left around the shapes. This border should be black in most cases, although colors may be used when the page 2 background option isn't used. The border acts as a self-eraser, removing the old image as the new one is put on the screen. It also eliminates flickering that is associated with the erase cycle.

The regular block move routine will draw on page 1 or 2. The background must be the same color as the border you leave around the shapes when you create them, as the border becomes the background (shapes used with block move leave a trail of their border color). An alternate block move routine lets you load in your background scene on both page 1 and page 2. Animation is done on page 1, with page 2 used for reconstructing the background (technically, the shape is OR'd with page 2 before it is stored on page 1). The border of the shape in this instance must be black with the high bit off. If, when drawing your shape, you refrain totally from plotting points on your border you'll be okay. The disadvantages of using page 2 for background are that 8K more is taken from your available storage, and you have more color considerations to worry about. Speed is virtually the same in both block modes.

The other two modes of animation both require the erase cycle. "Draw with background save" swaps a figure with the background that's on page 1, putting the background in a buffer. Between updates, it restores the background before starting the cycle over. It is, however, a convenient way to save the background without dedicating 8K of storage to it (as in the block/background mode). The background buffers used in the draw/background mode are 256 bytes for each object. In the worst case, using 32 objects would consume the same 8K required as in block/background. Every other case uses less.

The other method is the XDRAW routine. In principle, it works the same as the Applesoft XDRAW command; it reverses the current status of the screen. It uses the pre-shifted shapes as opposed to Applesoft shapes, though, and is much faster. It does require the erase cycle, but it also preserves background with no extra space taken.

An additional advantage of the two routines that use the erase cycle is that a collision table can be kept of objects that hit each other. Because the block method doesn't erase, a shape is constantly colliding with itself, making collisions with other objects impossible to detect. The block/background mode does take advantage of the collision table to flag collisions of an object with the background, but not with other objects. All four routines give you constant access to actual x, y locations of objects and their positions within their defined paths.

Note: with the 'draw with background save' animation, objects that collide with each other will not restore the proper background in normal animation. See the technical section for how this can be averted.

Table of Animation Types

	Speed	Graphics Storage	Collisions	Flicker	Shapes
Block	Fast	Hi-res page 1 or 2 (8K bytes)	Not used	None	Shapes must have border the size of maximum move in background color (usually black).
Block with Background	Fast	Hi-res pages 1 and 2 (16K bytes)	With background only	None	Shapes must have border the size of maximum move in black with high bit off.
Draw with Background Save	Medium	Hi-res page 1 plus buffers = # of objects times 256 bytes (8K bytes + buffers)	With background and other objects	Some	No border needed. Collisions between objects will leave trail unless erased.
XDRAW	Medium	Hi-res page 1 or 2 (8K bytes)	With background and other objects	Some	No restrictions.

The Shape Editor

Creating a Shape - The Easy Way

To enter a shape, from the main choices select to run the shape editor. The hi-res screen will clear, seven orange line segments will appear along with seven flashing dots (four across the top, and three across the middle of the screen), and the text window will display the information shown in figure 1.

The first row of information tells you what single letter commands you can use. The next row tells you whether the high bit is on or off (more on that later), whether you are plotting in even columns, odd columns, or all (again, more later), and your relative x, y position within the shape.

The I, J, K, and M keys control the movement of your cursors (flashing dots) up, left, right, and down, just as they appear on the keyboard. Why seven cursors? Because when you are creating a shape, you are actually creating seven separate pre-shifted versions of the same shape. Each cursor operates on one of the seven shifts. They all move together; when you press 'M', all seven cursors move down one point. But here we come to another of the tricks that can be used in animation. You've all seen games like Super Invasion, Apple Panic, Alien Rain, and Space Quarks, to name a few, where as the shapes move, they animate within themselves. Legs move, eyes shift, antennae swirl, wings flap...By slightly modifying each of the seven shifts of a shape, you can make a man's legs or a bird's wings appear to be moving in the finished product. Objects moving vertically use the same shifted shape and won't animate within themselves, but any horizontal movement causes a different shift to be used, and if the shift was modified somewhat, so will the object be. Information on editing the various shifts appears later. For now, treat them as seven identical shapes.

As you move the cursor around, you can plot points by pressing 'Z', and erase points by pressing 'X'. If there's a large section to be plotted, 'Q' locks plotting. Likewise, 'W' locks erasing. To get out of either locked mode, press 'Z' or 'X', or lock in the other mode.

If you are going to use a block method of animation (skipping the erase cycle), be sure to leave a top and left border around your object. In most cases, your border should be 2 units wide. That would leave the point 3,3 as the upper left corner of your figure.

If this is your first time through, go ahead and create a shape now, not worrying about color or making changes within the various shifts. Leave a top and left border of 2 points around your shape. When you get a figure that looks like anything (a blob will do), move the cursor to the bottom right corner of your shape (try not to leave a trail of dots as you do so). Check the x, y coordinates so you know the width and height of your figure, and add 2 to each to allow for the bottom and right borders. The next step is to compile the shape, so that it's stored in memory in an animatible (is that a word?) form. Type 'D' for 'temporarily done', and you'll get a new list of choices which stand for Edit, Compile, Animate, Save, Load, New Shape, and Menu.

```
I J K M Z X Q W H E O A 1-7 D
HBIT OFF ALL X:3 Y:3
```

Figure 1 - Shape Editor Text Window

Compiling and Saving Your Shape

To compile your shape, type 'C', then enter the width and height as computed above when prompted. Shapes must be compiled before being animated or saved. If you go back and edit the shape, you'll have to recompile it if you want the new version kept.

IMPORTANT NOTE: Yes, there is a maximum size for your shape. The number of bytes taken for each shift must be less than 255. (Shapes that size will take a combined total of 1.75K of storage). To compute the number of bytes, the number of bytes tall is the same as the height (each byte is one unit tall and seven units wide). The width, in bytes, can be computed from this formula, given the width in dots (the number you enter when compiling):

INT((W + 12)/7)

That is, take the width in dots, add twelve, divide by seven, and round it down to the nearest integer. To find the total number of bytes for one shift, multiply the height by the number of bytes wide. This is the number that must be less than 255. From the way it's derived, you may notice that any height added to a shape generally increases the memory requirements much more than additions in width, and any additions to the width reaching the numbers 9, 16, 23, 30, 37, 44, and so on, add an entire new column of bytes. Size, by the way, does play a significant part in speed of animation, as you will learn by experimenting.

After you've compiled your shape, you can try your first test of the animation routines by pressing 'A'. This does a test animation of your shape across the bottom of the screen and back. Paddle zero controls the speed of the object as it moves back and forth. Note that the test animation is done in XDRAW mode, so you'll be able to see firsthand the flicker effect caused by the erase cycle. The reason we didn't dazzle you with the block animation mode here in your first exposure is that we couldn't assume that every shape you ever create will have borders. Alas, the lack of such a border would cause unsightly paths left behind if we didn't use one of the erase cycles. When you get tired of watching your blob go back and forth, press a key to go back to the options. If this is your first attempt, use the 'S' (save) option to save your shape on disk, then press 'M' to go to the menu and skip ahead to the chapter on creating path tables. When you press 'M', it will ask you to verify that you want to lose the current shape. If you saved it to disk, you're safe, so proceed.

Creating Shapes - More Detail

Assuming you've had your first taste of the entire animation system by now, you're probably ready to try ever more sophisticated things with shapes. There is more. Your first shape was probably all white. Here's where it helps to learn the actual layout of Apple's hi-res colors.

A byte of information stores seven dots and a color flag. There are really only four colors (plus black) that appear on the Apple screen, and only two can appear in any given byte, or set of seven horizontal dots. Green and violet go together, and so do orange and blue. When the color flag (high bit) is off, dots within a byte appear green and violet. When the high bit is on, they appear orange and blue. Even columns of dots always show as violet or blue, and odd columns show as green or orange. You get the appearance of white when consecutive dots are set, combining violet and green or blue and orange.

Armed with that bit of information, the answer is yes, you can have as many colors in your shape as you want. You can draw a man with a green shirt, one blue pant leg and one orange pant leg, with his head and hands white, and shoes violet. The only thing that will cause you problems is trying to mix horizontally between the two groups of colors, such as orange next to violet. Vertically, you can do anything you want. Over larger areas, for example, alternating horizontal

lines of orange and green can give the appearance of yellow. By staggering dots in different ways, effects can be created in the same way that all the extended hi-res color generators use, such as the 108-color routine in the picture builder in this package.

The editor has a set of commands that makes the use of color easier. 'H' toggles the high bit; off if it was on, and on if it was off. Every byte in which a point is plotted or erased is assigned the current high bit value. Be careful about setting the high bit in your borders. Depending on which shift of the shape it is, you can create some trailing problems with the block/background mode. By avoiding plotting and erasing on your border, you'll be okay.

'A' sets plotting so that every point that you cross and plot with the 'Z' or 'Q' keys is turned on. 'E' sets plotting so that only even points are turned on, and odd points are turned off if plotted over. This automatically gives blue or violet, depending on the high bit. Likewise, 'O' sets plotting of odd points, turning off the evens and giving orange or green. Experimenting with the shape builder, we've found that creating your figure in white, then going back over the areas to add color works nicely.

	High bit off	High bit on
Even	Violet	Blue
Odd	Green	Orange
All	White	White

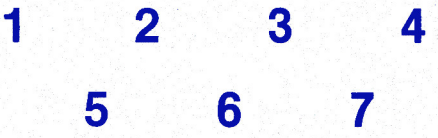
Colors of plotted hi-res points

If you are using the block animation method without page 2 for the background, you may create your shapes so that a background other than black can be used. Create your border area so that, instead of black, it is the color of the background you want to use. If you wanted to animate on a blue screen, for example, make your border area blue.

When using color, it's important to note that your shape can only be drawn in half the available columns if color is to be preserved. If odd screen columns are used, the shape will appear just as you drew it. If you plot your shape on an even column, the colors will be shifted over one, so that orange and blue reverse and green and violet reverse. This may be a nice trick to remember for getting two differently colored shapes out of one shape definition. It is also why the default movement with the path editor is two units, and why we recommend the two-unit border. The path editor will allow you to specify movements of one to three units up, down, left, or right, but horizontal movements of other than two will change colored shapes.

To achieve movement within a shape, it is necessary to make slight modifications in the seven shifted versions of the shape. The keys 1-7 toggle plotting on each of the shifts, 1-4 on the top row, and 5-7 on the middle row. The orange lines above the shapes show whether plotting on that shift is on or off. To create a sequence of a man walking, you might start with having both feet down in the first shift, shifts 2-4 showing the left foot in stages of moving up and down, and shifts 5-7 showing the right foot moving up and down. After 7, the first shift would appear again.

Of course the sequence 1,2,3,4,5,6,7 assumes that the shape will be plotted in that order (or in reverse order). It will, if you use the standard 2 unit movements. (The actual bit shifts corresponding to 1-7 are 0,2,4,6,1,3,5). If you are going to use horizontal increments of 1 instead of 2 (whites and blacks only), use the following shape sequence for your animation: 1,5,2,6,3,7,4. Notice that this corresponds to the order of the actual horizontal locations of the shape on your editing screen (disregarding whether it's in the top or middle row). (You see, the shapes are shown where they are for a reason; not just that four across the top and three in the middle is a nice pattern...). To confuse matters just a little more, if you want to use horizontal movements by 3's, use the sequence 1,6,4,2,7,5,3. (there's no nice way to put them on the screen so that all three sequences are easy to visualize...).



Shape Numbers as they appear on the screen

The Path Editor

Once you have a shape, to animate it on the screen you have to tell it where to go. The final animation editor allows you to set starting points for each shape, but from that starting point the shape must have a path. A path may be as simple as a straight line, or as complex as you like, with as many direction changes as desired. You create a path by plotting it out on the hi-res screen with the path editor, then you save it to disk. The animation editor allows you to integrate shapes, paths, and the animation routines.

Creating a Path

Run the path editor, and you will first be asked to move to the starting point. This starting point is for reference purposes in creating the path only. Paths are always relative to the starting point of an object. In other words, if you wanted 32 objects moving in circles, you would only have to create one path in the shape of a circle, but have each object start at a different location on the screen. An object's starting point, however, is set in the animation editor, not the path editor.

Figure out a good place on the screen to start your path so that you won't be forced off the screen with it (if you're creating a left-to-right path, for example, don't start at the right edge of the screen). Use the UIOJKNM, keys to move your flashing cursor to that point. The keys move the cursor in the directions shown in figure 2, similar to how the keys appear on the keyboard.

When your cursor is at the starting point, press 'S' to start your path. Again, the UIOJKNM, keys control direction of movement. Each time you press one of these keys, your path moves two units up, down, left, or right, or in combinations appropriate with the chosen direction. (For example, 'U' corresponds to 2 units to the left and 2 units up; whereas 'J' is just 2 units to the left.) If you make a mistake, 'D' deletes the last move.

If this is your first time through, trace out the path of your choice on the screen, then press 'S' to save it. You are ready to try your shape and path with the animation editor, so now type 'Q' to quit and go back to the main menu.

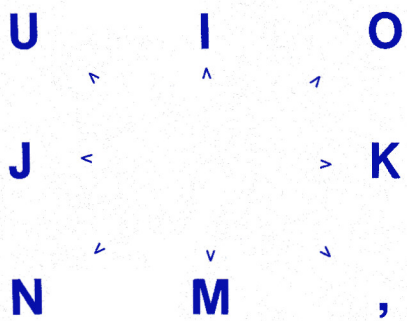


Figure 2 - Key Directions

More Path Commands

Any single move within a path may be up to 3 units in any direction. You may want to use different units for varying speeds of objects or for adding more directions (1 over and 3 up, etc.). Remember that horizontal movements of 1 or 3 will affect color (see the explanation of color in the shape editor section). When the points of the path are drawn on the editing screen, they are either blue or orange. Any point where a change occurs between the two colors tells you that your object will change colors there. If your object is white, however, it will show no effect.

To move in units other than 2's, use the RETURN, <-, ->, and / keys, for up, left, right, and down. These keys move the cursor one unit at a time in the desired direction, but do not add to the path by themselves. When you've moved to the next point along your desired path, press 'Z', which locks in your move and lets you choose the next one. 'Z' also allows you to create a move that goes nowhere, causing a hesitation in the animation when that point is reached. This may be used for timing and speed controls, or for a pause in an object's motion. All of the direction keys may be used as needed. Just remember that the UIOJKNM, keys plot the next point, and the RETURN, <-, ->, and / keys require 'Z' for plotting.

Still More Commands

'F' allows you to switch to full screen graphics and back, if you need to see the bottom of the hi-res page. 'C' clears the current path and allows you to start over.

Space Saving Hints

Each move in a path gobbles up a byte of storage, so if you are creating a lot of different paths for one program, you're going to use a good chunk of memory. Such sacrifices have to be made occasionally for speed's sake, but there are a few built in space saving features in the animation editor. Each object has a path list of up to three paths to follow in sequence, and after the sequence you may choose to repeat the path list,end with the object drawn, or end with object erased.

Having up to three paths gives you one way to save space. You can create several smaller paths and combine them in various ways to make different large paths. The other space saver is using the repeat option wisely. A straight line left to right across the screen, for example, can be as short as one move to the right with a repeat. A spiral movement only need be one arc of the spiral repeated, and so on. The experienced programmer can even switch paths in and out of the path lists, and change the repeat switch as needed. This may not be for the weak of heart and expertise, but as you gain more experience, you'll find all of the animation parameters fairly easy to modify while a program is running.

The Animation Editor

Once you have at least one shape and one path created you can use the animation editor. When you run it, you'll first be asked to choose the type of animation you want to use: block move, block move with page 2 background, draw with background save and replace, or XDRAW. If you choose block move or XDRAW, you will then be asked to choose page 1 or page 2. The appropriate machine language routines will be loaded in, and you are on your way.

You are then presented with a set of options, the most important of which, for the moment, is 'load'. Before you do anything else, you want to load in at least one shape and one path (more, if you want). Besides shapes and paths, you are also allowed to load in hi-res pictures (if needed for a background) and previously designed animation routines (for further editing). After you press 'L' for the load, select which of the four you want loaded, then type the name under which it was saved. (NOTE: If a picture is loaded, the program assumes the extension of '.PIC' in keeping with the standards used in all Penguin Software graphics products. See the appendix for more information on extensions. The picture must have a catalog name of 'name.PIC', although only 'name' should be typed when asked in the program.)

After you've loaded in at least a shape and a path, choose to create a new object (press 'N'). Some terminology should be explained here. 'Shapes' are the actual figures that are created with the shape editor. 'Objects'' consist of a shape, its associated paths and its locations. You may have up to 32 objects at a time with the animation routines. It is possible, however, that all 32 objects may use the same shape (and even the same path). Think of a game, such as 'Invaders', where there may only be 3 or 4 kinds of oncoming creatures, but several of each kind. Only 3 or 4 shapes need to be used, but there may be 7 or 8 objects with identical shapes moving around on the screen. Objects are the things that move around. Shapes are what they look like.

To create a new object, you first select the shape that it should have. Shapes are numbered zero and up, in the order in which you loaded them. One convenience, keeping the shape and path names accessible during use of the animation editor, had to be dropped in the final version you have due to space constraints. Therefore you should keep record of the names of shapes and paths with their associated numbers (order of loading) on the side while you use the program. If you do make a mistake and put the fuzzball where the spider was supposed to be, don't worry; you can go back and edit it correctly.

Next, select the numbers of the paths you want the shape to move in. You have up to three paths to use, although you don't have to use more than one. The last number in your path list has a special significance. If it's 255, it means that the animator should loop back to the beginning of the path list and repeat. 254 means to end the animation of that object, but leave it drawn. 253 means end the animation of that object, but finish by erasing it. As a simple example, suppose path 8 is a square, and you just want that repeated. For the first path in the list (number 0), you'd specify path 8. For the second path in the list for that object (number 1), you'd specify 255, meaning repeat the list over.

To continue with creating an object, you next specify its starting coordinates. Even with one shape and one path, you can create up to 32 independent objects by assigning them each the same shape and path, but different starting locations. The valid range for x is 0 to 1791 and for y is 0 to 255. The visible screen range is 0 to 279 on x and 0 to 191 on y. The extra range for starting locations allows objects to actually continue their animation while off the screen without immediate wrap-around. More discussion of use of this range follows later. If the x value is odd, your shape will appear in the color in which it was drawn. If x is even, the colors will be reversed from those drawn.

The last part of entering an object is to give its starting location within its path list. You may start at any part of any path in the list. First specify the number of the path in which to start (0, 1, or 2; this is not the actual path number, but its sequence number in the path list). Next, specify the step in the path with which to start, for example, the 10th movement in the path. By combining varying starting coordinates of objects with appropriate variations in starting location in a path, you can have objects follow one another in single file in your animation. Chris used this type of arrangement in 'Space Quarks'.

Trying the Animation

Finally, you can see your creation. The object is done, so you can see what it looks like animated to your specifications. Type 'A' from the options, and your commands are now obeyed by the computer. Paddle zero controls the speed of animation. Press the space bar when you are done watching, and you'll be returned to the options.

Saving

If you want to save the animation routine as is, type 'S' from the options and the entire machine language/binary file will be saved to disk in a form that can be used from any program. Give the animation set-up a name for saving. Two files will actually be saved: one with the machine language routines, and one text file with information that will be necessary if you ever want to go back and edit that specific animation set-up. The latter would be loaded along with the machine language routines the next time you run the animation editor and ask to load that animation set-up. If this was your first trial run of the entire animation system, you are ready to either (a) go back and experiment some more, or (b) try hooking the animation routine you just created into a short program of your own. If the latter, skip ahead to that section. If the former, go ahead and read more on any section you choose.

Other Options in the Animation Editor

You may go back and edit any object's parameters by typing 'E' from the options, specifying the number of the object to inspect and/or change, then viewing or changing the specs for that object.

You can obtain a disk catalog from the options by typing 'D'. All those funny extensions you see on your file names are explained in the appendix.

'H' clears the hi-res screens, 'C' clears the entire animation set-up and allows you to start over without re-running the program, and 'M' returns you to the master menu (and also forgets your current set-up, if you haven't saved it to disk).

Animation from your Programs

Once you've used the animation editor to define the parameters for your animation, and have saved an animation file, you may easily use the animation in your own programs. The simplest version of this would be as follows:

```
10 HGR
20 PRINT CHR$(4);"BLOAD name.ANM"
30 CALL 36928
40 GOTO 30
```

The above example assumes using page 1 with a block animation mode. Basically, each call to 36928 updates each object one segment along its path. This allows the programmer the greatest degree of control over the animation, because after each update any commands desired may be inserted. The collision table and locations may be checked, paths may be changed, paddles and keyboard may be read, objects may be activated and de-activated, and so on.

The general program design using animation will be as follows:

1) Set-up
2) Call animation routine
3) Check and change parameters
4) Go back to step 2

Step 3 may be non-existent, as in the sample above, or it may consist of an entire range of commands. The more commands there are involved in step 3, the more reason there is to do that part in machine language, if possible. A long string of BASIC statements in step 3 may slow things down enough to take the smoothness out of the animation.

Necessary Commands

To use the animation, the animation file must first be loaded in your program and the appropriate hi-res screen displayed. If you are using a background picture, that must also be loaded or drawn.

With the XDRAW and 'draw with background' animation types, an initial draw of all the objects is required. This is done with a CALL 37284 (or JSR $91A4 in machine language). This step is done before the loop, not within the loop.

With all the animation types, the statement used within a loop to update all objects is CALL 36928 (JSR $9040 in machine language). That's the extent of necessity in using the animation routine.

You may want to set HIMEM to protect the area of memory taken by the binary animation file. This is especially important if you use any string variables in your program. The first memory location taken by your file is given to you while using the animation editor.

Options and Tables

Specific locations in memory are set aside for providing you with information and a place to make changes during use of the animator. Following is a list of those tables, their formats, and suggestions for their use.

Shape Index $9380-$93FF
(starts at decimal 37760)

This is a table of pointers to the actual locations in memory of the shape information. Each pointer is 2 bytes long, and there is room for 64 pointers. Hence, shapes may be numbered 0-63. $9380,1 holds the pointer to shape 0, $9382-3 holds the pointer to shape 1, and so on. The pointer is an address in lo, hi format. By changing this pointer, you can actually change the shape used for an object in mid-animation, while keeping all the other object information intact.

Object List $9400-$9420
(starts at decimal 37888)

This is a list of shape numbers for up to 32 objects. It is also the list that the animator cycles through as it updates each object. Each shape number takes one byte. If the shape number is 255 ($FF), it signifies the end of the object list and the animator will go no further in the list in updating. If the shape number is any other number above 127 that object is considered inactive and the animator passes it. You can deactivate an object by adding 128 to the shape number in the object list (setting the hi-bit in machine language). To re-activate an object, subtract 128 from the shape number in this list (or clear the hi-bit). When all values in this table up to the first '255' are set greater than 127, no movement will occur. See the following for another way that an object may be de-activated.

Path Locations $9440-$949F
(starts at decimal 37952)

This is a list of path locations for each object. Each entry in this list is 3 bytes long, so object 0's information starts at $9440, object 1's information starts at $9443, and so on. The first byte gives the current path number in the path list (0, 1, or 2; see path lists). The next two bytes are a pointer to the actual memory location of the next step in the path, in lo, hi format. Upon starting, this usually points to the memory location of the beginning of the first path. When a path is finished and there's no repeat used, the first byte of this three byte entry is set greater than 127 (negative, in machine language). This setting causes animation on this object to be skipped, and is another place where an object may be de-activated by adding 128 to the value.

Object Locations $94AO-$94FF
(starts at decimal 38048)

This list also contains three bytes for each object. The first two bytes give the current x location of the object, and the third byte gives the y location. The y location is a value 0 to 255, with 0 to 191 on the screen, and the remainder (192-255) serving as a wraparound area. An object going upward will continue off the screen, disappear off the top for a few moments, then reappear at the bottom. The x location is stored in a byte/bit format. The first value is the byte number across the screen in which the object appears, the second value is the bit number (0-6) in the byte. Since the screen is only 40 bytes wide (7 bits each, giving 280 locations), if the first value of the x location is between 0 and 39, the object will show on the screen. The range 40-255 is the x wraparound area.

This means that if an object is moving left to right continuously, it will disappear off the right edge of the screen, not show for a while, then reappear on the left. It also means that an object moving in such a path would only be in the

screen area about 1/6th of that time. These wraparound areas allow you to use a much larger 'Space' than that actually shown on the screen. You may have dozens of objects animating with only a few in view, depending on which part of this space the objects wander into. Of course you may decrease the size of this space by changing the value of the object location whenever it goes beyond a certain number. Note that if an object is off the screen you may safely change its location. If an object is on the screen and you change its location, you may cause poblems with the erase part of the animation, which is not performed until just before the new location is plotted.

Path Lists $9500-$95DF
(starts at decimal 38144)

There are path lists for each of the 32 possible objects. Each list is 7 bytes long. The first six bytes are pointers to the actual memory locations of the start of the object's path 0, path 1, and path 2, as specified in the animation editor. The seventh byte is only used if three paths are actually used. After the last path, the next byte has a flag with value of 255 for repeat, 254 for end without erasing, and 253 for end with erasing. If there is only one path used, for example, the third byte containes the flag. If two paths are used, the fifth byte has the flag. If all three paths are used, the seventh byte holds the flag. Pointers to the path memory locations are stored in hi,lo format, since the high byte will never contain the possible flag vlaues. Path pointers and the flag may be changed during animation. These pointers are only used between paths. If an object is in middle of a path, the Path Locations table will be used until the end of the path is reached. The animation routine will then jump in and check the path list to see what it is supposed to do next.

Collision Table $95EO- $95FF
(starts at 38368)

The collison table contains one byte for each object. This byte contains the value zero until the corresponding object incurs a collision, at which time a non-zero value will be put in the table. The animation routine does not reset values to zero. Once a collision occurs, a non-zero value stays until you, the programmer, set it back to zero. This is handy in case you don't want to check the collision table every time through. A collision flag will remain until you are ready for it. Remember that in the block mode the collision flag is not used, and that in the block/background mode a collision is flagged only if it's between an object and the background. XDRAW and draw with background will both detect collisions between objects.

memory map for animation

BASIC Programs

$2000-$3FFF
(8192-16383)
Hi-res Page 1

$4000-$5FFF
(16384-24575)
Hi-res Page 2

Free Space

Paths and Shapes
build down from $8F00

$8F00-$95FF
(36608-38399)
Animation Routines
and Tables

Path Tables - Programming Tricks

You may want to have a path of an object tied directly to paddle movement or keystrokes. A simple way to do this is to set up a path one byte long with a repeat. That way every movement is read directly from that byte. Affected tables are the Path Lists and Path Locations. The path list for the affected object should consist of a pointer to the byte in memory being used for the movement, followed by a 255, for repeating. The path location should be a 0 (path number in the path list), followed by a pointer again to the byte used for the movement.

What does a path look like? Each byte signals a movement. The end of a path is flagged with a zero. (So the above mentioned path would consist of a changeable number followed by a zero.) The movements are divided into two bits per direction, as shown in figure 3.

```
0 0 ! 0 0 ! 0 0 ! 0 0
Y-   Y+    X-   X+
```

Figure 3 - Path Bits

Each pair of bits can represent the numbers zero to three, signifying the number to be added or subtracted from x or y. To go up (subtract from y) 2 units and left (add to x) 3 units, the byte would hold the following value:

1 0 0 0 0 0 1 1,

or $83,

or decimal 131.

The fastest way to decode paddles or keyboard may be to set up a lookup table in your program for the movement value, and transfer that value to the path byte between each animation update.

Note that a value of zero ends the path. For a zero movement (pause), set all the bits to 1, giving a value of 255 for the byte. Moving 3 units in each of 4 directions cancels all movement. Various other combinations have the same effect, such as the decimal value 5 (one unit left and one unit right).

More Technical Stuff

Advanced programmers may want to bypass our animation routines totally and just use the plotting routine for drawing a shape at a given location, creating their own animation cycle and movements. Each of the four animation types has a plot routine associated with it. The routine names on disk are BLOCK, BLOCKBACK, BKGND, EOR (block, block with page 2 background, draw with background save, and XDRAW, respectively).

Each plot routine resides at $8F00-8FFF, and each must use the shape index table at $9380-93FF (described above). Each also uses a lookup table P1L for hi-res page1, or P2L for hi-res page 2. The lookup table resides at $9200-9379.

For each of the routines, the x byte is put in $0, the x bit in $1, and y value in $2. The shape number goes in $3. For BKGND, a buffer number for the saved background must also be put in $4. We used our object numbers for the buffer number.

EOR and BKGND return a collision value in location $0B. BLOCKBACK puts its collision value in $8FA1. Zero means no collision, any other value means a collision occurred.

The draw routine for each is called at location $8F00. The erase routine, except in EOR, is called at $8FA2. For EOR, the draw routine performs the erase function, since 2 consecutive calls to it (2 XDRAW's) restore the original background. The erase routine for BKGND restores the background that was saved in the buffer when the shape was drawn. The erase routines in BLOCK and BLOCKBACK are only used when the shapes are to be removed totally from the screen. Otherwise, the borders mentioned much earlier take care of the old image on the screen. Erase in BLOCK clears the shape area to black. Erase in BLOCKBACK restores the background where the shape was.

The only caution that should be heeded on collisions is in the BKGND mode. When two objects collide, for the background to be returned properly they have to be erased in reverse order. (i.e. Draw 1-2-3, erase 3-2-1, draw 1-2-3,...). This is NOT done in the animation routines in this package, but you may wish to use this option if needed in your application.

Part Two

Picture/Object Editor

This part of the Graphics Magician allows you to create pictures and objects in a form so that they can be saved in an amazingly small amount of space. It uses a technique many of you may have seen used in various graphic adventure games. Instead of an actual screen image being stored, which takes over 8000 bytes for each picture, the moves taken in creating a picture are stored. Where a standard picture takes 34 sectors on a disk, pictures created in this manner take 2 or 3 sectors, with the most complex taking perhaps 5. It is this type of savings and storage that allows programmers to store literally hundreds of pictures on a single diskette.

This technique can be applied to anything from adventure games to educational software; any program that would benefit from being able to store screen pictures and shapes in as little space as possible. All that is added to your program is the machine language subroutine provided on this disk that decodes your picture and recreates it in your program. It's as simple as a BLOAD, 2 POKE's, and a CALL.

The concept of saving the moves is one thing. The other is to allow useful moves. We've taken parts of our other graphics packages, The Complete Graphics System and Special Effects, and adapted them as we saw fit toward this application. More or less could have been included, but usefullness has to be balanced with length. We chose to use the line routine from the Applesoft ROM (same as the language card version), an improved 100-color fill routine from The Complete Graphics System, and a slimmed down paintbrush routine from Special Effects (108 colors, but only 8 brushes as opposed to the original 96). Generally, a picture will start as a white background, lines will be added in black defining enclosed areas, the fill routine will be used to add colors to the various enclosed sections, and the paintbrushes will be used to smooth transitions and add details. This sequence is not carved in stone; the only limiting factor is that the fill routine must be used on a white area enclosed by black boundaries or the edge of the screen.

Creating a Picture

As opposed to most picture creation packages, sequence and strategy may be important. Every move you use in creating your picture will be saved. Hence, the fewer moves used, the less space your picture will require. (Don't get too paranoid over every move used. Do several practice pictures without worrying about byte counting to see how much space is taken, then if you find you have to make them more compact decide where you want to save the moves.) The other factor, although it may not be important in your application, is that when a picture is redrawn by the machine language routine (very quickly), it is drawn in the same steps in which you drew it (mistakes and all, if you leave them in). It gives the impression of a super-fast artist laying out the screen.

When you run the picture editor, you'll be shown your options at the bottom of the screen, including your current x,y location, colors, and mode. Your joystick or paddles control the '+' cursor on the screen. Whenever you press button 1, you set a new starting point for a line. Pressing button 0 draws a line, fills the current area, or plots the brush, depending on your mode. Be sure to set a starting point for the first line before you draw it. Otherwise a random point will be used in your final creation, most likely creating a bit of a mess.

When you start, you are in line mode, with fill color 0 (white), and line color 4 (black). Line colors correspond directly to Apple's hi-res colors (see chart below). The fill colors are show on a palette that is available while you are using the editor. Type 'P' to see the palette. An important point is that there are three color groups on the palette, each of which does not mix well side-by-side horizontally with the other two. (For a full explanation of Apple hi-res colors, see the section on shapes in the animation section of this manual.) The groups will not affect one another if placed vertically next to each other (one above the other). We recommend that if you use more than one color group, you set up your screen in horizontal zones (see figure 4). Within each zone, stay with only one of the groups; this will prevent horizontal interference between the groups.

0 Black1	4 Black2
1 Green	5 Orange
2 Violet	6 Blue
3 White1	7 White2

Apples Hi-res Colors

You'll notice 3 whites on the palette. Each is actually different internally, and each marks the beginning of the next color group. Several other colors appear in at least two of the groups, allowing you to give the appearance of a color crossing a zone vertically, while in fact you are using two different constructions of the same color. Color group A consists of colors 0 through 51. Color group B consists of 52 through 76, plus the standard Apple hi-res colors 4-7 (for lines). Color group C has colors 77 through 107, along with Apple hi-res colors 0-3.

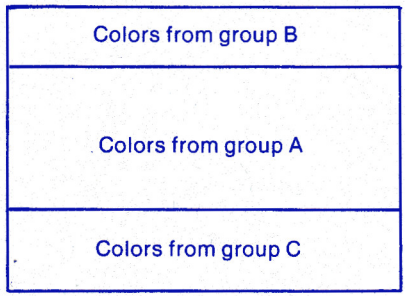


Figure 4 - Example of Using Zones

Following are the drawing commands:
L - Sets line mode. Button 0 draws lines, and button 1 sets new starting points.

F - Sets fill mode. To fill an area with the current color, move the cursor into an enclosed area and press button 0. For most efficient filling, position the cursor so that it is somewhere in a direct line below the highest point in the enclosure. The fill routine uses an averaging method that minimizes the number of tries necessary to fill a complete enclosure, yet still maintains a very good speed.

1 through 8 - Selects the corresponding brush. 1 through 6 are small through large, and 7 and 8 give 'airbrush' affects. Button 0 sets the brush down for one plot. Since every move takes space, you must repeatedly plot the brush; it does not stay down.

C - Selects a new color for automatic filling and brushes. The range for colors is 0 through 107, as shown on the pallete. The palette is first displayed. Decide which color you want, then press any key to get to the text mode and type in the number of your color.

K - Selects a new line color. The range is 0 through 7, the standard Apple hi-res colors. Note that colors other than black will usually not suffice for borders of areas to be filled. Since the other four colors (blue, violet, orange, and green) have only half the resolution, the fill routine may 'leak' through borders of those colors.

P - Displays the color palette for filling and brushes.

J - Reverses the joystick orientation if your joystick does not correspond properly to up/down and left/right. If you are using a joystick, you may want to remove any self-centering springs to make drawing easier.

ESC - Switches display between full screen graphics and mixed text and graphics.

Z - Zeroes in on the area near the cursor, allowing better control in small spaces. 'Z' acts as a toggle between standard and zeroed in modes, and is used to switch to the mode not currently in use.

The above commands are those that allow you to create pictures. The ones below let you edit, reconstruct, and save what you've done:

D -Deletes the last move. This can be used repetitively to delete several moves.

R - Reconstructs the picture at machine language speed, so you can see the end result as it will appear in your programs.

S - Saves the picture as is. You give it a name, and it will be saved as a coded binary file of the moves needed to reconstruct your drawing.

E - Edit mode. This allows you to go in and make changes in any of the steps used in creating your picture. You may add detail in places, change colors, delete steps, and so on. The following section describes edit mode in detail.

Q - Quit. You may choose to start a new picture, or return to the main menu.

Editing a Picture

The edit mode clears the screen and allows you to single-step through the instructions you used to create the picture. At any point, you may delete an instruction, back up an instruction, or insert instructions.

Pressing the space bar causes the next instruction in your picture file to be performed, and displays in words what that instruction is. You may single-step through the entire picture using the space bar.

To backward step, press the left arrow key (<-). This actually clears the screen and re-executes all the instructions up to the one before the current one displayed.

'D' deletes the instruction currently displayed, backing up and reconstructing all the previous ones.

'I' lets you insert instructions immediately after the one displayed. Insert mode looks just like the regular drawing mode, except the reconstruct and save options aren't available, and 'E' returns you to where you were in normal edit mode.

'F' means that you are finished with edit mode, finishes drawing the picture up to the last instruction, and puts you back into normal drawing mode.

Creating an Object

You may want to have pictures that optionally have varying items shown. Again, the classic example is with an adventure game where certain items may or may not appear in a room at any given time. The first thought you may have is to use an Apple shape table for such objects. The problem is that Apple shapes take quite a bit of storage relative to their size and are limited in color. Using the picture/object editor, you can create objects in exactly the same way that you create pictures. The display routine that you attach to your program has two entry points. One erases the screen and draws an entire picture. The other takes x and y coordinates and locates the 'picture' at the given offset on top of an existing picture, giving you what is essentially a multicolored object that can be located anywhere on any picture.

There are two differences you should use when creating something that will be treated like an object instead of a picture. First, the first command in creating the figure MUST be a 'Start Line at' command. This is done by locating the cursor somewhere on the screen and pressing button 1. The dummy 'start line' command is actually interpreted later as a 'start object at' command that gives every other point in the object a relative location. When you later put the object at some x, y location on another picture, the x, y location you give will be where the starting point of your object will go. Keep that in mind, so you can prevent your objects from going off the screen when in use. Note that this is a dummy 'start line at' command. You need a second 'start line' command for any lines you use.

The other difference in creating an object is that you cannot assume that you'll be starting with a white background, since the object usually will be put on top of another picture. That means that the fill command shouldn't be used in an object, unless you can provide an assured white background. That may be done by first laying down a white background with a large, white brush. Usually it will be just as convenient to use colored brushes to draw the object directly. Choose your color usage wisely. Try to arrange your pictures so that objects created from color group A will go on a background zone created from group A, and so on. It is possible to get good results without this consideration, but if you want to be assured of the cleanest, most professional look, take the time to match your colors well.

Using Pictures and Objects in Your Programs

Once you've created a set of pictures and objects, you can use them in your own programs with the machine language program PICDRAW. Your program can be in machine language or BASIC. The picture and object files can be loaded anywhere in memory, with the starting location sent to the PICDRAW routine before it is called. PICDRAW will draw to either page 1 or page 2, whichever is the current hi-res page being used by your program.

(Programming note: location $E6, 230 in decimal, contains the pointer to the current hi-res page. A value of $20, or 32 decimal, points to page 1. $40, or 64 decimal, points to page 2. From BASIC, HGR and HGR2 change this value automatically. To secretly draw on one page while the other is displayed, it's necessary to use a few POKE's. If page 1 is being displayed, use POKE 230,64 to draw on page 2, call the drawing routines, then use POKE -16299,0 to display page 2. If page 2 is being displayed, use POKE 230,32 to draw on page 1, then POKE -16300,0 to display page 1.)

The subroutine PICDRAW uses memory locations $8E00-$95FF (36352-38399) which is the top of available memory in a 48K Apple with DOS loaded. Hi-res page 1 resides at $2000-$3FFF (8192-16383) and page 2 takes $4000-$5FFF (16384-24575). Picture and object files should generally be loaded somewhere above the hi-res page you are using and below the PICDRAW routine. You should use a HIMEM command to protect the top area of memory from any BASIC variables. You may even want to set HIMEM to 8192, if your program is short. It is also a good idea to work out a memory map of the storage used, so you know where everything is located.

To put PICDRAW on your disk, do the following:

1) Put in the Graphics Magician disk and type BLOAD PICDRAW.

2) Put in your disk and type:

```
BSAVE PICDRAW,A$8E00,L$800
```

In your program, if in BASIC, use the command:

```
PRINT CHR$(4);"BLOAD PICDRAW"
```

When you've decided where to load your picture or object file, use the BASIC command:

```
PRINT CHR$(4);"BLOAD name.SPC,Axxxx"
```

where xxxx is the location you chose. (24576, decimal, works fine for starters.)

With either a picture or an object, before calling the PICDRAW routine, you must tell it the location of the picture file. If the location is stored in the variable L, for example, use the commands:

```
POKE 36352,L-INT(L/256)*256
POKE 36353,INT(L/256)
```

From machine language, put the starting location in $8E00-8E01 in lo,hi format.

If your file is a picture, all you do then is use the command:

CALL 36400

If the hi-res page is being displayed, you'll see the picture recreated when this command is executed. From machine language, use JSR $8E30.

If your file is to be treated as an object, you must also tell the routine where to put the object. Give it the screen x,y locations with the following commands:

```
POKE 36354,X-(X>255)*256
POKE 36355,X>255
POKE 36356,Y
```

where x has a range 0-279, and y is 0-191. The object, when drawn, must fall entirely within the screen area. From machine language, the x coordinate goes in $8E02-8E03 in lo,hi format, and the y coordinate goes in $8E04.

To draw the object, use CALL 36361. This skips erasing of the screen, and takes care of adding the offsets. From machine language, use JSR $8E09.

If offsets are inconvenient, and objects will always appear optionally in a certain place on the screen, you may use a trick by creating the object in the location you will always want it. Then, when you go to draw it, CALL 36405 (or JSR $8E35). This is actually the picture draw routine with the erasing skipped, hence it draws the object as if it were a picture on top of the existing picture, and x,y offsets needn't be POKE'd in.

Part Three

Super Shapes

Introduction

Super Shape Tables are an extension of Applesoft shapes, allowing internal control of color and scale and compact storage of large shapes. The major advantage of this technique is the ability to store a large number of shapes (or an entire scene) in RAM, and the ability to rotate or scale each shape. The disadvantage is lower speed due to the way Applesoft handles shape tables.

In general, a shape table is a series of vectors, each of which contains a command to plot or not plot, and a direction to move. Super shape tables interface between you and the Applesoft routines, passing on changes in scale, rotation, and color. There are two programs in this module. The super shape editor allows the user to define shapes, individually or in tables, and view these shapes. The display routine is a short machine-language program that interprets the tables and puts shapes on the screen.

Using the Super Shape Editor

The editor menu offers the following options:

Write on the screen
View a shape
Position a shape from a table
Recall a shape from disk
Append to current shape
Erase screen
Change shape location
Load shape table
Save shape table to disk
Get disk catalog
Delete shape
Quit

A shape is a table entry containing color, scale, plotting, and other information. The shape stores exactly what is drawn on the screen. When a shape is viewed, it is drawn in the same location where it was created. When you desire to see the shape in another screen location, use the Position option. As an example, a scene of a room would probably be drawn in the same place it was originally created. On the other hand, an airplane or rocket would be drawn at various screen locations. Let's step through the menu options.

Write on screen

This is the part where all the action is. When you select this option, you will first be asked : SCALE? This determines the number of points in each line segment you draw. A scale of 1 results in a line of 1 point, a scale of 2 gives you a line of 2 points, and so on. The allowable range is 1 to 255. Proper use of scale results in very compact tables. For instance, suppose you want to draw a small object inside a square border. A square with sides 20 points in length can be drawn using a line of scale 20. Since the data for each line occupies only one byte (plus a few exrtra bytes whenever a new color, scale, or location is selected), this is a massive saving over the 40 bytes an Applesoft shape table would require.

The next question is COLOR? Your input will determine the color of the shape. The range is 0 through 7 (standard Applesoft colors), and the selected color will be used until a change is requested.

Once a color has been selected, a line the size you selected will appear. Note that the selected drawing color will not be seen yet. However, since certain colors can't be drawn at certain locations on the Apple's screen (see the section on colors in part one), the message "basepoint visible" or "basepoint invisible" will appear, depending on whether the bottom point of the line will be visible when drawn in the desired color. Using the paddles, move this line to the desired starting location, then press button 0. Paddle 0 now controls the rotation of the line. The larger the scale, the more possible angles. When you are ready to create the shape, press button 0 again. The line will begin moving in a direction controlled by paddle 0. Pressing the button again puts you back in the previous mode, stopping the motion and allowing a new rotation, color, or scale to be selected. Another press continues the drawing action.

Whenever you stop drawing by pressing button 0, you have these seven additional options: COLOR, SCALE, LOCATION, FINISH, BACKUP, ONESTEP, and VIEWING SCALE. These commands are accessed by pressing their first letter or, in the case of viewing scale, a number from one to nine.

COLOR allows you to change color while maintaining scale and location. Similarly, SCALE changes only that parameter.

LOCATION puts you back in the mode where the line position can be moved with both paddles. Again, you go on from there by pressing button 0.

VIEWING SCALE controls the size at which your shape is being displayed. For instance, when using a small scale, you can press 4 and see the shape at four times its actual size. Pressing 1 causes the display to return to actual size. With values from 2 to 9 the shape is still being created in the original scale, but the display is magnified. Note that the allowable rotations are determined by the actual scale, not the display scale.

BACKUP deletes the most recent move, and can be repeated all the way back to the start of the shape. If the most recent move was the drawing of a segment, that segment will be erased. If the move was a scale or color change, that change will be deleted from the shape. This option gives you the flexibility to experiment with different colors and scales while not making any irreparable errors.

ONESTEP allows accurate control for complex shapes. Each press of button 0 adds a segment to the shape. Between segments, a new rotation can be selected with the paddle, but the segment won't be added until the button is pressed.

FINISH ends the shape. From there, you will be given several options. The first question is SAVE IT? Answering "Y" will result in the shape being placed on disk. Once on disk, it can be appended to an existing shape (more on that

later), or added to a table at any time. The second question is ADD TO TABLE? an answer of "Y" will result in the shape being added to the current table. If the reply is "N" the shape won't be added to the table. Note that contrary to the standard Applesoft shapes, super shapes can exist separate from tables, although they can be put in tables also.

More Menu Options

Getting back to the main menu, RECALL takes a single shape from disk and adds it to the table in memory. APPEND adds to the last shape in the table. This is whatever shape was most recently drawn or recalled. Appending is possible either through loading a shape from disk or by drawing on the screen. A shape appended from disk will appear in the same place it was originally created. When appending to a shape by drawing on the screen, you can use the BACKUP option to get the cursor to the end of the old shape. This allows the appended portion to be stored relative to the starting location of the original shape.

VIEW clears the screen, then draws the desired shape in the location it was created. The user specifies shape number and a speed from 1 to 255. The higher the speed value, the faster the drawing appears. This effect can be used to put a handwritten signature on the screen at pen speed or in any other instance where the user desires the shape to appear at a reduced speed.

POSITION allows the shape to be drawn anywhere on the screen. The user specifies x and y location, shape number, and scale and rotation offsets. The rotation offset is added to the original shape, rotating all elements. If a shape actually starts with an upward move (rot = 0), a rot offset of 8 will rotate it 45 degrees, an offset of 16 will rotate it 90 degrees, and so on. If the entire shape was created without any new location parameters, the whole shape will rotate together. If new locations were used within the shape, or if APPEND was used, each subshape will rotate on its own (as mentioned before, using backup with append avoids this). The scale offset tells how many times the scale of a segment should be added to itself. If a segment has a scale of 3 and a scale offset of 1 is used, that segment will be drawn with a scale of 3 + 3, or 6. An offset of 2 will produce a scale of 3 + 3 + 3, or 9. In other words, an offset of 1 will produce a shape twice the original size, an offset of 2 will produce a shape 3 times the original size, and so on.

LOAD brings a table into memory and allows additional shapes to be added. SAVE places a table on disk. DELETE removes a shape from the table. CHANGE allows the starting location of a shape to be changed. For example, if you want to append a disk shape to an existing shape, but the new shape is in the wrong location, it can be moved, saved to disk, and then appended to the desired shape.

Using the Machine Language Routine

The machine language routine is named SST/ML. To put it on your own disk, first use BLOAD SST/ML with the master disk, insert yours, then type BSAVE SST/ML,A16384,L288 to save it.

The display program has two entry points. Entering at the top results in the shape being drawn exactly as it was created. In this case, the user need only put shape number in location 253 ($FD) and the starting page of the table in location 254 ($FE), then call the start of the routine.

The routine is relocatable, and can be put at any free location. It currently loads at 16384 ($4000), but should be loaded elsewhere if you want to use page 2 of hi-res.

To use the full shape routine, nine parameters are needed. Though all must be set initially, some will maintain their value until changed, and thus needn't be reentered for each draw. The entry point is 16 ($10) bytes beyond the start of the routine. If the code is loaded at 16384 ($4000), the entry would be at 16400 ($4010).

The parameters are:

xlo at 249 ($F9)
xhi at 250 ($FA)
y at 251 ($FB)
*rotation offset at 252 ($FC)
*shape number at 253 ($FD)
*table page at 254 ($FE)
*color switch at 255 ($FF)
*delay at 4 ($04)
*scale offset at 5 ($05)

(Starred locations maintain their value between calls to the machine routine. Others must be set before a call.)

XLO is the lo byte of the x coordinate. This can be found from basic with XLO = X – 256*(X<255). In other words, when the x coordinate is less than 256, the lo byte and the x coordinate have the same value. If x is greater than 255, the low byte is equal to x – 256.

XHI is the hi byte of x. From basic, use XHI = (X>255). If the X coordinate is greater than 255, the hi byte is 1, otherwise, it is 0.

Y is the y coordinate

ROTATION OFFSET is added to the rotation of each shape element. To draw the shape at the original rotation, use an offset of 0. As with Applesoft tables, a rotation of 8 is 45 degrees, 16 is 90 degrees, and so on. If the value goes past 64, it cycles mod 64. thus 65 is equal to 1, 66 is equal to 2, etc.

SHAPE NUMBER is the number of the desired shape in the table

TABLE PAGE is the hi byte of the shape table (the table must be loaded in on a page boundary). Page boundaries are multiples of 256. A table loaded at 1024 (4*256) would have a page number of 4.

COLOR SWITCH allows the shape to be drawn entirely in one color, rather than the color or colors stored in the shape. If the switch is 0, the original colors are used. If the value is 1 (or any other non-zero value), the color will be the one used in the most recent Applesoft HCOLOR command. The most common use for this would be to erase a shape by setting HCOLOR to black (color 0 or 4).

DELAY sets the delay time between segments of the shape. This is used for special effects as mentioned above. The minimum delay is achieved with a value of 1. With values from 2 to 255, the delay increases. 0 gives the maximum delay. Note that while the creation program uses higher numbers for higher speeds, the machine routine accesses the monitor delay routine, and thus higher numbers give longer delays.

SCALE OFFSET adds the scale of each segment in the shape to itself that many times. A setting of 0 uses the original scales. A value of 1 doubles the size of the shape, 2 triples it, and so on.

To increase speed in special applications, options such as delay can be removed from the drawing program. The delay code is located from $40DD to $40E1. To speed up the code, just place $EA in these locations.

Extras

Binary File Transfer Utility

Included on the Graphics Magician disk is a program that lets you easily transfer binary files from one disk to another. Along the way, it also tells you the starting address and length (in decimal and hexadecimal) of each file. You may wish to use this routine just for checking starting addresses and lengths of binary files, as that information isn't easily available normally.

Your choices are (L) load, (S) save, (C) catalog, and (Q) quit. You may load a series of files without saving, just to check the address information. When you save a file, the most recent one loaded is the one saved.

Given the binary length of files such as the sequential pictures (.SPC), you can change their loading addresses with the following commands (the example uses a file named HOUSE.SPC, with a length of 387 bytes, and a desired loading location of 24800):

```
BLOAD HOUSE.PIC,A24800
BSAVE HOUSE.PIC,A24800,L387
```

Subsequent BLOADing of the file will automatically start at 24800.

Demo

A demonstration of various features of the Graphics Magician can be run from the main menu. Since it is designed as a self-operating, repeating demo, you'll have to press RESET to end it. To get back to the main menu without re-booting, type 'RUN N' (or as we do for short, 'RUNN').

Animated Alphabet

Also included as a bonus on the Graphics Magician is an animated alphabet already designed as pre-shifted shapes, ready to use in the animation editor (or modify with the shape editor). Each letter is stored under its own name. CATALOG your disk for a complete file listing.

Appendix

Suffixes on File Names

Throughout our graphics products, we've managed to create quite a few different types of graphic data files. Almost every one of these files is coded in binary, making them impossible to identify in type by looking only at the name given to them. To simplify matters, we've written every program so that a suffix is automatically added to each name you give a file, identifying the type of file. You NEVER use that suffix from one of our programs. The only time you see it is when you CATALOG a disk, and the only time you use it is from a program other than one of ours.

Example: While using The Complete Graphics System, you name a standard picture file 'HOUSE'. On disk, it would appear as 'HOUSE.PIC', but from any other program of ours, you would still refer to it as a picture named 'HOUSE'. When a picture is expected, '.PIC' is assumed automatically.

This greatly simplifies the task of remembering the origin of the files stored on a disk, but it also has caused quite an accumulation of suffixes. Here's a list of each and what it stands for:

.ANM Animation Table
Graphics Magician - animation

.FNT Character Font
Complete Graphics System

.PAK Packed Picture File
Special Effects

.PIC Standard format screen picture
All Penguin Software packages

.PTH Path Table
Graphics Magician - animation

.SHP Apple Shape Table
Complete Graphics System

.SPC Sequential Picture
Graphics Magician - from picture/object editor

.SS SuperShape
Graphics Magician - supershape editor

.SSH Shifted Shape
Graphics Magician - animation

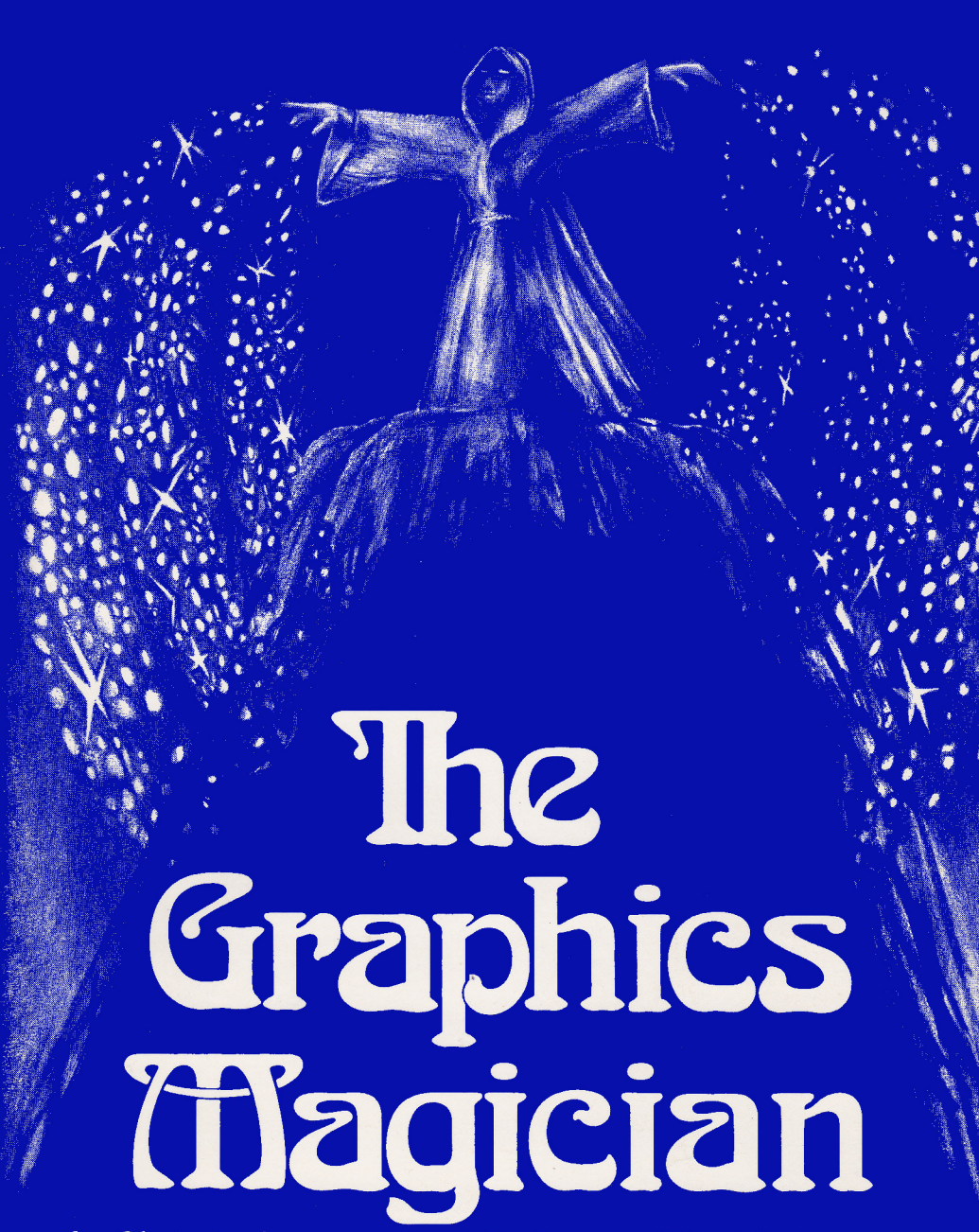
.SST SuperShape Table
Graphics Magician - supershape editor

.TD 3-D File
Complete Graphics System

.TXT Animation Text File
Graphics Magician - animation

The Graphics Magician contains machine language animation routines that use the same techniques as most of the popular Apple arcade games. Three animation editors let you design your figures, their paths, and assemble animations with up to 32 independent objects. Also included is a hi-res picture/object builder that lets you store hundreds of 100-color pictures on a single disk and recall them quickly from your own programs. This capability is useful in designing adventure games, educational software, and other programs requiring a multitude of graphic images to be quickly and easily accessible. Plus, a new shape editor greatly extends the capabilities of Apple shape tables with multiple colors and angles that are preserved on scaling. All design of graphics is done through menu-driven editors; to use in your programs, just attach our machine language routines. The entire package is designed to be easy to use for the beginning programmer, yet flexible enough for the most advanced.

Requires a 48K Apple II with Applesoft and a disk drive.



The Graphics Magician

by Chris Jochumson, David Lubar, and Mark Pelczarski

Now Anyone can put Professional Graphics into their own Programs . . .

penguin software