HOW TO GET THE MOST OUT OF BASIC 8

by

Dave Krohne & Roger Silva

"Professional Programmers Share Their Secrets"

Free Spirit Software Inc.

HOW TO GET THE MOST OUT OF BASIC 8

By

DAVE KROHNE and ROGER SILVA

"Professional Programmers Share Their Secrets"

TABLE OF CONTENTS

PART I: UNDERSTANDING BASIC 8

Chapter 1:Getting Started 1
Chapter 2:Why Basic 8? 3
Chapter 3:Graphics Modes 5
    Your First Program 5
    Modes and Screens 5
Chapter 4:Shapes 11
Chapter 5:Rylander 3D Solids 13
Chapter 6:Buffers & Structures 17
    Graphic effects 20
Chapter 7:User Input 23
    Keyboard Input 25
    Using GETKEY 25
    Filename Input 25
Chapter 8:Utilities 27
    Reading the Directory 27
    Slideshow 28
    Selecting Colors with @Pixel 28
    Using Menus 29
    Using Print Shop Graphics 30
    Using a 128D with a 1581 drive 30
    Function Keys 31
Chapter 9: Additional demonstrations 33
    3D Animator 33
    Potpourri 36

PART II: BASIC 8 ANIMATION

Introduction 39
Chapter 10:Planning your project 41
    Graphic Balance 42
    Text Balance 44
    Screen Aids 44
Chapter 11:The 3D Graphic Environment 49
    Perspective 49
    Rotation 51
    Creating a Cube 53
Chapter 12:Buffers and Screens 61
    Buffers 62
    Screens 65
Chapter 13:Animation Details 69
    @FETCH Animation 69
    @COPY Animation 70
    Pointer Animation 72
Chapter 14:Music Enhancements 73

APPENDICES

Appendix A: Basic 8 Quick Reference List 75

PART I: UNDERSTANDING BASIC 8

CHAPTER ONE
GETTING STARTED

To get the most out of this book, you should have the following equipment:

Basic 8.
A Commodore 128D, or a Commodore 128 with 64K of video RAM installed.
An RGB 80 column monitor.
A Commodore 1351 mouse or equivalent.

Many of the principles discussed in this book will work with the standard Commodore 128 and could be joystick controlled. However, the 64k of video RAM is necessary for the display of multiple screens and full size color screen formats, and the 1351 mouse is a much smoother input device.

You have two disks supplied with "How To Get The Most Out Of Basic 8." The first disk contains many small programs demonstrating the use of various Basic 8 commands. You will be using this disk for the first part of the book. Please note: BEFORE attempting to run any programs, you must first load in the Basic 8 Editor! Without the editor, the programs will not run!

Disk #2 contains the 3D-Animator and the animation demos used in part 2 of this book. Also included is a Basic 8 runtime library that will allow the demos to be run but not listed or edited. To get full use from this book you must have a registered copy of Basic 8.

There are several programs available from Free Spirit Software that allow you to create color screens, graphics, fonts, icons and custom pointers in 80 columns on the Commodore 128. Call your local distributer for information about:

Basic 8 Toolkit
Sketchpad 128
News Maker 128
Spectrum 128
Basic 8 Poster Maker
DigiTalker 128

CHAPTER TWO
WHY USE BASIC 8?

Why use Basic 8? Words such as power, flexibility, and simplicity come to mind. With Basic 8 you can easily access the many features that were probably the reason you bought your Commodore 128. Features such as the 80 column screen, the 1351 mouse, and the RAM expansion unit are all completely supported and easy to access with Basic 8.

Basic 8 is not a replacement for the BASIC 7.0 that is built into your 128, but rather an extension that uses BASIC 7.0 as a base. Basic 8 adds over 50 new commands are added to your 128. These commands add exciting graphics programming capabilities to your Commodore 128. When you begin programming with Basic 8 you will be working with familiar Basic 7 commands, while integrating the new Basic 8 commands.

The purpose of this book is to assist in the development of new, exciting programs for the Commodore 128. We have found Basic 8 to be the perfect base for the development of powerful 128 programs. In addition, the authors of Basic 8 have opened the doors for you by allowing you to distribute YOUR programs using the Basic 8 Run Time Library without asking for any royalties or payments WHATSOEVER! The only legal requirement is that both your program and documentation must state that the program was written using Basic 8. Lou Wallace and David Darus have been more than generous with this offer—it is up to you to put it to good use.

Basic 8 users come in a wide range of types; from the new user who is wondering how to get started with Basic 8, to the user who wants to stretch the limits of the 128. This book has been divided into two parts to help both ends of the range. The first part deals with the "nuts and bolts" aspect of Basic 8. Here you will find assistance on using screens, modes, buffers, shapes and all of the other items that are the foundation of your program. In most cases there are demo programs on the accompanying disk to illustrate the use of each function. The demos themselves range from simple examples to complex program modules. We encourage you to list, print, and experiment with the simple examples as you are starting out. Once you begin to understand Basic 8 more fully, study and experiment with the more complex modules.

The second part of the book deals with many of the advanced features of the Basic 8 language. Concepts such as drawing in a three dimensional environment, animation and special effects are all discussed in the second part.

On-line support for this book and Basic 8 is available on Q-Link in the Free Spirit software support area.

Special Note - Software Pirates, Bunco Artist, Con Men and Swindlers: Thanks to you dubious efforts we have all witnessed a virtual halt in the development of software for the 128. This, in the end, diminishes the usefulness of our Commodore 128 systems—the very systems we have poured our life savings into! Gee thanks! If you are a "Hot Shot" computer expert (and I have no doubt that some of you really are), may I suggest that you prove your technical skills

by developing public domain or commercial software that we could all enjoy? You might end up earning a few bucks, a few friends and a little bit of respect in the Commodore 128 world!

CHAPTER THREE
GRAPHIC MODES

64K of video memory is highly preferred when working with the Basic 8 language. All Commodore 128D's are shipped with this memory installed at the factory. Older 128's can be upgraded either at an authorized service center or with a kit available from mail order houses.

All Basic 8 graphics work with a bitmapped screen. An 80 column monitor will display a standard bitmap of 640 x 200 pixels, or dots. One byte holds 8 bits, so a byte is literally 8 dots across on your screen, and a full monochrome screen will display 80 bytes x 200 scanlines or 16,000 bytes. You can see why more memory is needed if your 128 as only 16K of video RAM!

The 64K of video memory allows the user more memory for additional graphic screens. These additional screens could be used to create a large screen that can be scrolled around for viewing, or as several screens that can be instantly switched from one to the other. Parts of a hidden screen can even be displayed on your main screen. Using several screens and copying portions of them back and forth is the key to achieving great speed with Basic 8. Commercial Basic 8 programs rely on this speed.

In addition to allowing multiple screens and instant screen swapping the 64K of video RAM has another tremendous advantage — all of these screens take no memory away from your program! You have effectively increased your total storage capacity to 176K. I have found that the addition of the video RAM has been the single most important upgrade to my system.

Your First Program

By this time I am sure you are wondering just what you will be able to do with Basic 8. Perhaps the best place to start is at the beginning. Here is the world's shortest Basic 8 program:

```
10 @WALRUS,1
20 @TEXT:END
```

Load the Basic 8 Editor, type in this program and then run it. Pretty amazing! (What did you expect from two lines of code anyway?) All Basic 8 programs MUST start with the @WALRUS command. This command tells Basic 8 whether your computer has 16K or 64k of video ram. It is used only once for each program. In addition to specifying your video RAM, it displays: "BASIC 8 by Walrusoft (c) 1986, 89". Any programs that are distributed using the Basic 8 run time library must display this copyright logo within the program and must state in the written documentation that the program was created with Basic 8.

Modes & Screens

Before you start programming you must decide what type of graphic screens you want to use. If you have a 16K video RAM machine I recommend that you work in monochrome or the 8x8 color cell mode. The additional color takes away from

your screen size, however I find that a 640 x 176 color screen is a good compromise.

If you have a 64K video RAM machine, you have many options. Not only do you need to decide what type of graphic screens you are going to use, but also how many. Most likely you will want to use color in your programs, and most of the examples in this book use the 8 x 2 color mode. The reason is simple; smaller color cells allow you to display more colorful screens. If you just want to have a scrolling landscape or space scene then you may need to use a large monochrome screen.

Basic 8 offers a wide assortment of predefined screens to work with. Custom screens may also be created as your memory allows. These predefined screens are fully explained in your Basic 8 encyclopedia under the @MODE command.

For most of our examples we will use the 8 x 2 color cells. After using the @Walrus,1 command we need to issue an @MODE,1:@SCREEN,2 command. This sets us up in a graphic screen using 8x2 color cells.

A few other commands need to be used at the beginning of our Basic 8 program. We need to set the screen colors, clear the bitmap screen, set the point of origin (used for 3D perspective), and most importantly, tell Basic 8 the direction of growth for lines, boxes and other graphic shapes.

If this sounds complicated, don't worry. Just load in the program called B8.HEADER. All this information is set up for you. Below is an example of a standard program outline. It includes all of the setup commands as well as the commands necessary to return you to the normal text screen at the end of your program. We suggested that you use this outline for all of your Basic 8 program.

```
10 FAST:TRAP 1000
20 @WALRUS,1
30 @MODE,1
40 @SCREEN,2
50 @COLOR,2,8,0
60 @CLEAR,0
70 @DRWMODA,1,0,0,0,0,0,0
80 @DRWMODB,0,0,0
90 @ANGLE,0,0,0,0: @ORIGIN,320,100,100,200,100,200
100 rem ----------start of your program----------
.
.
.
1000 @DRWMODA,1,0,0,0,0,0,0
1010 @DRWMODB,0,0,0
1020 @ANGLE,0,0,0,0
1030 @ORIGIN 320,100,100,320,100,200
1040 @TEXT: PRINT CHR$(14)
1050 END
```

The first step in any Basic 8 program is to define which screens are needed and to display information on these screens. Several Basic 8 commands deal directly with this subject.

@DISPLAY - Loads a picture from disk & displays it on the screen.

Syntax: @DISPLAY,Screen #, Drive #, Drawmode, "Filename"(,X,Y)
Example: @DISPLAY,0,8,0,"pict.art work" (,X,Y)

One of the most basic functions of Basic 8 is the ability to display a graphic screen or brush on your computer screen. To use the @DISPLAY command, you must first create a picture with any Basic 8 paint program. Then it is a simple matter to set up your program with the proper screen format (such as an 8x2 screen) and load your picture with the @DISPLAY command.

There are two @DISPLAY demos on Disk #1. The first one is called B8.DISPLAY-PICT. This short program sets up an 8 x 2 screen and loads a picture for viewing. You may note that the bitmap loads in first, then the color.

If you do not want the user to see your picture as it loads, you can make the background & foreground the same color (black for example.) Another option is to load your pictures to an alternate screen and copy them onto your viewing screen.

Brushes are loaded and displayed in the same manner. Basic 8 brushes and pictures both use the same type of file structure. We differentiate between the two by using PICT. or BRUS. as a prefix (beginning) to the file name. For example, PICT.HOUSE would be a picture and BRUS.HOUSE a brush. Pictures are always full screen size, so the smallest picture would be 640 x 200 pixels. Brushes are anything smaller than this. Pictures and brushes both have identical attributes as far as color cell type and display modes.

When loading a brush you will need to define where you want to place it on your screen by using the optional (X,Y) variables. These two variables specify where the upper left hand corner of your brush will be after loading. Remember that the X location will be rounded down to the nearest byte, which simply means your brush will land on a number which is divisible evenly by 8. Also your brush will be rounded up to the nearest color cell. (More on color cells & bitmap mapping later.)

Run the program B8.DISPLAY-BRUS. The program starts by displaying five small graphics at various places on the screen. Press a key and it will display a plus sign, a circle and a triangle.

The @DISPLAY command can also be modified through the use of four different drawing modes.

Drawmode

0	Erase under	(replace)
1	Merge with	(OR)
2	Common	(AND)
3	Complement	(XOR)

Drawmode 0 is the most often used, however the other modes can be used to create interesting effects. As demonstrated by the B8.DISPLAY-BRUS program, the Merge mode simply allows each graphic to lay on top of another. The Common mode only displays pixels that all shapes have in common with each other (not terribly impressive in this case.) The Complement mode is the most interesting; this creates an interesting graphic with each complementing pixel being turned off/on. The Complement mode may be used to create very impressive graphics.

List the program to see how the @Display command was used in each of the examples. An astute Basic 8 user may note that the disk drive got quite a workout. In a "real" program we could have loaded the graphics in just once and used the @FETCH command to do the same thing, however, this was a @DISPLAY demo - remember?

@COPY - Copies a graphic block from one screen to another.

Syntax: @COPY, Source screen, Start X, Start Y, DX, DY, Destination Screen, EX, EY
Example: @COPY,1,0,0,639,199,2,0,0

The Basic 8 @COPY command is by far the fastest way to display graphics in any Basic 8 application. I have used this command extensively in all of my programs to speed things up. The menus in Spectrum 128 are a typical example; every time you click to access the menus, the current viewing screen (your picture) is copied onto a hidden screen (temporary storage). The menu itself is then copied from yet another screen (menu screen) onto the viewing screen. Once an item is selected the copy command is invoked once again to copy your picture from the temporary storage screen back onto your viewing screen. In Spectrum the @COPY command is also used every time you are using a blinking line (box, circle) to position your shape. Each blink is the copy command updating the viewing screen from the temporary storage screen! This is one technique I used to minimize color bleed. If all of this sounds complex, rest assured it is! For the curious, run the program called B8.MENU for a sample of a quick pull-down menu system.

For the rest of us there are two programs that demonstrate just how fast and useful the @COPY command really is. The first program is called B8.COPY-1. In this demo, a graphic is displayed in the upper left hand corner of the screen. It is then copied several times until the whole screen is filled. All five graphics are displayed in this manner. Note just how fast the screen fills up--the copy technique makes Stash & Fetch look like snails!

The next example of the copy command is B8.COPY-2. This is a simulation of a Basic 8 game. You are commander Drake sitting at the controls of SKYLINK, a multi-billion dollar space facility near Aspen, Colorado. You are in control of a powerful space telescope which is currently orbiting the planet Ashdon (found in the M.33 Tringulum galaxy.) Your mission is to search for life in the far reaches of the galaxy. You may also encounter hazards such as meteorites, black holes and time warps. Good luck Commander Drake!

Unfortunately, the necessary funds have been cut by Congress leaving Commander Drake with a neat game demo, but no game yet! This demo uses two 8 x 2 screens. The first screen is simply a picture created with Spectrum 128 showing computer screens and rows of buttons and levers (just waiting to be pressed!) The second screen is a custom size 8 x 2 screen which is 640 x 346 pixels. When you need a large screen that is not offered by the @MODE and @SCREEN commands you may use the @SCRDEF command to create custom screens. The demo uses this screen format:

```
@SCRDEF,0,0,1,640,346,0,27681
@SCRDEF,1,0,1,640,200,41521,57521
```

The large screen was drawn with Spectrum 128 (640 x 200) in two parts and put together with a simple Basic 8 program (which created a large 8x2 screen, loaded in the pictures and stored it as a single screen). When you run this demo, all you need to do is move the mouse around. The space scene you are viewing on the middle monitor is a portion of screen 0 which is copied onto the center of screen 1. (Of course you knew that already!) To exit this program hold down the RUN/STOP key.

You will be surprised when you list this program: It is very short—only three blocks long! The program itself is mostly setting up and loading predefined screens, and using the @MOUSE command in a loop to keep track of mouse movement and copy a portion of the space scene to your viewing screen.

This is a good example of what you can do with Basic 8. It would not take a genius to figure out that by loading several background scenes you really could develop a new interesting game for your Commodore 128.

@STORE - Stores a screen to disk as a picture file.

Syntax: @STORE, Screen #, Device #, Compression flag, Filename
Example: @STORE,2,8,1,"PICT.65-MUSTANG"

The @STORE command is very simple to use. Whenever you want to save any screen as a Basic 8 picture file use the @STORE command. The example above shows how to store an 8 x 2 screen (screen #2) to drive 8 in a compressed format. The name of the file is PICT.65-MUSTANG.

@TEXT - Returns to the 80 column text screen.

Syntax: @TEXT (no parameters needed)

The @TEXT command takes you away from the Basic 8 graphic environment and back to the Commodore 128's 80 column text screen. Most often this command is used at the end of a Basic 8 program. It also can be used in the middle of a Basic 8 program when you need to ask the user a lot of questions and want to use standard input. Don't forget to issue a new @MODE command or you will end up with garbage when you go back to your graphic screen.

@WINDOWOPEN - Opens a rectangular graphic area as a window.

Syntax: @WINDOWOPEN, X, Y, Window Width, Window Height, Border flag
Example: @WINDOWOPEN,0,0,640,100,1

There are times when you want your graphics and text to appear in certain areas of your screen but not in others. Or, for example, you need to clear only the right side of your screen. Windows are perfect for this. A Basic 8 window defines an area of your screen from which all graphic commands will work. The upper left corner of your window becomes the new 0,0 coordinate.

@WINDOWCLOSE - Sets the graphics parameters to the screen's 0,0 coordinate.

Syntax: @WINDOWCLOSE (no parameters needed)

The @WINDOWCLOSE function deactivates the current window settings. All coordinates are reset to 0,0 on your main screen. @WINDOWCLOSE does not erase the contents of the window nor does it save them to be used again. To save the contents of a windows you need to @STASH it to a buffer. To clear a window issue a @CLEAR command prior to closing it.

CHAPTER 4
SHAPES

Basic 8 offers an assortment of shapes to use in creating your screens. The most used commands are @DOT, @LINE, @BOX, @CIRCLE and @ARC. Each of these commands has a demonstration program on Disk #1.

One point to remember is that none of these shapes will draw unless the @GROW,X,Y,Z command has been executed. I recommend a default setting of: @GROW,0,0,1.

@DOT - Plots a single pixel dot.

Syntax: @DOT, X, Y, Z
Example: @DOT,320,100,0

Run the program B8.DOT. Your screen will display 640 x 200 pixels. A single dot represents one of the 128,000 pixels on the screen. The demonstration program allows you to draw colored dots on an 8x2 screen. To use the demo, simply move the mouse and click the left mouse button to draw a dot. The right mouse button will clear the drawing area. To exit press the Run/Stop & Restore keys.

@LINE - Draws a line between two user defined points.

Syntax: @LINE, X1, Y1, Z1, X2, Y2, Z2, Thickness
Example: @LINE,0,100,0,639,100,0,1

Run the program called B8.LINE. This demo allows you to draw lines with the mouse by simply holding down the left button. Each of the line's drawing points is displayed. The purpose of this demo is to familiarize you with the basic syntax of the @LINE command and show how it relates to a 640 x 200 pixel screen. Advanced users will want to list this program to see how the "rubber band" blinking line is made.

@BOX - Draws a square between user defined points.

Syntax: @BOX, X1, Y1, Z1, X2, Y2, Z2, Shear direction, Shear value, Thickness
Example: @BOX,0,0,0,639,199,0,0,0,1

Run the program called B8.BOX. For this demo boxes are drawn with the mouse by pressing the left mouse button. For a box to be defined you need the upper X,Y,Z corner and the lower X,Y,Z corner. Additional values are shear direction, shear value and thickness. For this demo these values are set to zero.

@CIRCLE - Draws a circle with a defined radius from any defined point.

Syntax: @CIRCLE, Center X, Center Y, Center Z, Radius, Thickness
Example: @CIRCLE,320,100,0,75,1

Run the program called B8.CIRCLE. The first thing you will notice is that the circles are definitely not round! In fact, they are almost twice as tall as they are wide. This is due to the fact that our pixels are not square--they are 60% taller then they are wide.

Basic 8 can use a logical screen of 640 x 512 by using the @SCALE,1 command. This does not give you any more pixels on the screen, however your circles will be drawn much rounder.

When you are using the @SCALE command you must remember that your screen coordinates are also scaled. To place the circle back in the center of your screen (on the Y axis) you need to divide the old Y axis of 100 by .39 (200/512). This gives you the new Y axis of 256. If you do not want to plot all of your shapes on scaled coordinates you can go back to the standard scaling factors of 640 x 200 by using the @SCALE,0 command.

Circles also have thickness. The thickness can be set anywhere from 1 to 65535. The most common thicknesses would range from 1-25. The @GROW command affects the way that your circle is drawn. By growing on the Z axis each rendition of the circle will become smaller and smaller creating a thick round circle. As the demo shows, you may also adjust the growth to be on the X or Y axis. You can set @GROW to negative values and combine different growth in all directions. Write a circle program and experiment with the different possibilities!

@ARC - Used to create various polygons or curved lines.

Syntax: @ARC, Center X, Center Y, Center Z, X radius, Y radius, Starting angle, Ending angle, Increment, Thickness, Subtend flag
Example: @ARC,320,100,0,50,100,0,360,10,1,0

Run the program B8.ARC. Arcs are the most complex shapes. At the simplest they could be considered as ovals. The arc demo shows a variety of ovals made by stretching the X and Y axes. But the arc command can create much more than just ovals! By setting the increment flag to a value between 1 and 180 you can create a variety of shapes including dodecagons, octagons, hexagons, diamonds and triangles. As you view the demo, note that the increment flag is all that has been changed.

The @ARC command can create even more than closed polygons. By setting the beginning and ending angles, you can use the @ARC command to create curves. The demo shows a variety of 45 degree curves ranging from 0 to 360 degrees. The subtend flag may be on (value of 1) or off (set to 0). The subtend will draw lines from each end of an arc to the center. This is very useful when making pie charts.

CHAPTER FIVE
RYLANDER 3D SOLIDS

Rylander solids were created many years ago for the Commodore 64. They are among the most beautiful effects available within the entire Basic 8 system. Spheres are great for planets and columns make nice borders for your Basic 8 creations. For an introduction to the world of Richard Rylander's 3D Solids run the demo called B8.SOLIDS.

This demo starts by showing you several pictures that used these shapes. Note how shapes will appear to be stacked on each other? There is nothing complex happening. If you draw several items like donuts, each successive donut drawn will appear to be on top of the others.

@STYLE - Defines the attributes with which solids will be rendered.

Syntax: @STYLE, Shade, Scale, Lighting
Example: @STYLE,1,1,0

You are given a lot of control over how your finished products will appear. For shading you can select either textured or halftone. Textured offers a rough looking surface made up of random dots, while halftone objects appear to be smoother. The halftone image offers a more ordered pattern of greys to simulate shading. As with circles, spheres will appear to be more egg-shaped than round. Set the scale flag to 1 to correct this problem (and produce a better looking picture.) I always use the scaling set to 1. For lighting your choices are normal or backlight.

The Backlight command will add a ridge of light to the darkest area of your object. In effect, the object is drawn as if it had a strong light in front of it and a dimmer light shining at it from behind. This is also a technique used by artists to enhance the three dimensional effects of their work.

@SCLIP - Clips portions of 3D solids

Syntax: @SCLIP, Left, Right, Up, Down
Example: @SCLIP,50,50,100,75

Ever wanted just half of a donut? The @SCLIP (solids clipping) command lets you have just that. For example, you could define a vertical column and cap it off with half a sphere to create a silo. You will be even more impressed with the solid when you see just how nice the shading blends in with objects clipped in this manner. All of the clipping is defined from the center of the object. For example, to do no clipping set all flags to 127. To not render the lower half of a circle set the down flag to 0.

With a bit of experimentation you will soon be developing incredible three-dimensional pictures that would take many hours to draw with a conventional paint program. (That would be a lot of pixel editing!)

@SPHERE - Renders a shaded three dimensional sphere.

Syntax: @SPHERE,X,Y,Radius
Example: @SPHERE,320,100,50

Run the demo called B8.SPHERE. This demo starts by showing you an unscaled sphere that appears to be more egg-shaped than anything else. When using the solids, the @SCALE commands do not affect the roundness of the sphere. (It will, however, affect the X,Y locations.) To adjust the symmetry you need to set the scaling flag of the @STYLE command to 1 (@STYLE,0,1,0). Spheres cannot be rotated but the shading styles and lighting may be adjusted as shown in the demo.

@SPOOL - Creates a shaded spool shaped object.

Syntax: @SPOOL, X, Y, Inner radius, Outer radius, View
Example: @SPOOL,320,100,50,100,0

Run the demo B8.SPOOL. The spool offers a unique shape for the graphics enthusiast. It is also the hardest one to visualize. The example shows how the spool can be drawn in a vertical or horizontal position. Note how the lighting affects the inside parameters as the shadows wrap around the central cylinder. As with all of the other objects, the texture and lighting may be adjusted.

@CYLNDR - Creates a horizontal or vertical column.

Syntax: @CYLNDR, X, Y, Radius, Halflen, View
Example: @CYLNDR,10,50,10,2,1

Run the demo B8.CYLNDR. Cylinders are very useful objects. With them you can create vertical roman columns and pillars, as well as horizontal tubes and logs. Experiment by making long thin columns or short fat columns for different effects.

@TOROID - Renders a donut shaped object.

Syntax: @TOROID, X, Y, Inside radius, Outside radius, View
Example: @TOROID, 320,100,50,100,0

Run the demo B8.TOROID. Toroids are my favorite shape (I happen to like donuts!) The Toroid is the only solid that allows you to view it from three perspectives. You have two side views, horizontal and vertical, and you also can see the top view. As you will see, the toroid is created from using an inside circle and an outside circle. It works best when the inside is smaller than the outside. As you can see from the demo, the toroid can have a rough texture or a smooth texture. As with all objects I suggest you use it with the scaling flag turned on.

Since each solid takes a while to be created, they are a bit slow to use by themselves in your programs. The best way to use the solids is by creating a scene with Basic 8 and storing it as a picture file. This screen could be used by itself or touched up with a paint program. Items such a spheres could also be clipped out and used as brushes in your program instead of being drawn individually.

CHAPTER SIX
BUFFERS & STRUCTURES

One of the best features of Basic 8 is that the user may easily create a buffer in RAM to hold various fonts, graphics, patterns and logos. Buffers can be created in RAM expansion units to hold up to 512k of information. However, I have found that using an internal buffer of 48k in bank 1 is adequate for almost all of my needs. If users have a RAM Expansion Unit then they could set up large buffers as a RAMdisk to hold backups of pictures, charts, or pages until a final copy is ready to save to a "real" disk.

How do you know how much will fit in a buffer? The rule of thumb I use to estimate the amount of buffer space needed is that a full monochrome screen saved into a buffer requires 16k. Fonts require about 3-4k each. A monochrome menu that fills half of the screen will naturally require about 8k of your buffer space. Any graphic can be saved as a compressed file, saving you even more buffer space.

When you plan your program you need to decide how much information could be stored in your buffers, and how much you can simply load in from your disk when needed. The buffer is always faster then disk access, but remember the user has to sit and wait initially while you are filling up your buffer. Users may become impatient if you load 32 monochrome screens into your 1750 REU for a fantastic slideshow presentation!

Also, remember for graphics to consider if you have enough video RAM left for making a 'menu screen' which would allow menus to be copied instantly onto the users viewing screen. In Sketchpad 128 I use both menu techniques. Users with only 16k of video RAM have menus stored in a buffer. For 128D users the menus are stored on a hidden screen and copied to the user screen. I much prefer the latter as it is very quick.

Basic 8 buffers are easy to set up. Run the program B8.BUFFER. To make a 48k buffer add the following lines to your program:

```
100 rem ——— define a 48k buffer
110 POKE 47,128: POKE 48,191: CLR
120 @BUFFER,1,1024,48000
```

The poke statements move the start of you program variables up 48k to make enough room for your buffer. You must define your buffer BEFORE using any variables in your program because the CLR command clears all variables from memory!

There are five types of structures that can be stored in a buffer to be used later in your program:

Structure 1: Patr.
Structure 2: Logo.
Structure 3: Font.
Structure 4: Brus.

Structure 5: Soun.

Sound files do not come with the Basic 8 package. Sound files are from Lou Wallace's DigiTalker 128 program. Naturally, they are designed for use with your Basic 8 files.

Structures are loaded into your buffer with the @STRUCT and @LSTRUCT commands as in this example:

```
150 rem ------ Load Data Into Buffer
160 @STRUCT,1,1,0,0 : @LSTRUCT,1,8,1,0 ,"PATR.CHECKS": AA = @SEND
170 @STRUCT,2,2,0,AA: @LSTRUCT,2,8,1,AA,"LOGO.MENU"  : BB = @SEND
180 @STRUCT,3,3,0,BB: @LSTRUCT,3,8,1,BB,"FONT.ROMAN" : CC = @SEND
190 @STRUCT,4,4,0,CC: @LSTRUCT,4,8,1,CC,"BRUS.MYPIC" : AA = @SEND
```

The @STRUCT command is used to define a structure type and its starting address in the buffer. The @LSTRUCT command is used to load a structure from disk into a buffer. The @SEND command returns the last buffer address used by the structure loaded. This then serves as the starting address for the next structure.

The commands used to access each of these structure types are as follows:

@PAINT - Fill area with current pattern

Syntax: @PAINT, X, Y, Bank#, Address, Size
Example: @PAINT, 100,100,0,57346,1024

The @PAINT command is used in conjunction with @DRWMODA and the @PATTERN commands. Before trying to fill in an area, set @DRWMODA to pattern mode like this:

```
@DRWMODA,1,0,0,0,1,0,0.
```

We have set the first and fifth values to 1. The first puts us into jaml mode, and the fifth specifies to use the current pattern when drawing. Next you need to tell Basic 8 which structure # is the pattern to be used for drawing. Use the @PATTERN command like this:

```
@PATTERN,1.
```

Then you will be ready to fill an area with the @PAINT command. Before issuing the @PAINT command, be sure that the area you are painting is totally enclosed; otherwise you may end up filling the entire screen with your pattern.

The @PAINT command is unique because to use it you must define a stack area (Address and Size) to be used as a reference for the current fill pattern. In most cases the 1k paint stack suggested by the Basic 8 Manual is adequate. However, this stack could be used up if you are filling around a complex shape on your screen.

@LOGO - Display selected Logo

Syntax: @LOGO, Structure number
Example: @LOGO, 5

Logos can be made easily by using the Logo Maker program supplied on your Basic 8 disk. Logos are very practical for things such as help menus which can pop down on the display. You will find Logo Maker on your Basic 8 disk number two. The program name is B8.Logo Maker.3.

The best way to plan your logo is to first use your favorite paint program to print out several copies of PICT.GRID 8X8 on Disk #1 from this book. This 8x8 grid can be used to create logo messages in the blocks and determine what your finished logos will look like. This layout is essential because Logo Maker will prompt you for the column#, row#, height, width and direction of each string in your logo. You also need to decide which structure to use—either a standard structure such as 254 for upper/lower case or the structure of a font you have loaded into a buffer.

Logos are very practical because they are compact and fast. The alternative is to define each string as a @CHAR structure which uses up precious basic programming space.

Run the program B8.LOGO for a demo of logos being used as a fast, flexible menu system. Since a plain "This is a menu" type demo does not give the user a real "feel" for what a menu system can look like, I developed a sample Basic 8 program called "The Movie Studio". You will be treated to a short animation (made with the Basic 8 3D Animator!) Then the main screen will be displayed. All of the menus (6 of them!) were made with the B8.LOGO MAKER.3. The Fkeys are one logo structure and each of the menus are individual structures. Each structure is only 1 block long and takes very little buffer memory.

To "use" The Movie Studio simply press the function keys F1 - F5. Each of the submenus will be displayed for a few seconds then you will return back to the Main Menu. To exit this demo press the ESCAPE key. As a fun project you are welcome to write the rest of the program, and distribute it commercially!

Several items were used to put this demo together. First I determined the general layout of the screen. Then I used Basic 8 to create a screen layout and saved it as a picture file. Run the demo B8.LAYOUT to see how this worked. All of the color and text was added with Spectrum 128 and the graphic itself came from my personal graphics library, drawn with Sketchpad 128. Sometimes to get the results you want, it is best to use several techniques.

@CHAR - Prints text to the graphic screen using a user defined font.

Syntax: @CHAR, Structure #, Column, Row, Height, Width, Direction,"Char String"
Example: @CHAR,254,0,0,1,1,2,"I Love Basic 8!"
(See Basic 8 manual for special control codes)

Basic 8 offers quite an outstanding selection of fonts to choose from. Your programs can have their own unique style by simply selecting a nice looking font. In addition you can create your very own fonts with the Basic 8 Toolkit! You may use as many fonts as you want in any Basic 8 program.

Fonts may be used two ways with Basic 8. The most obvious is to use the @CHAR command to position a font of any size and color anywhere on your screen. Plus, you can print your statements in any of the 8 directions!

Run the program B8.CHAR for a demo of using fonts with the @CHAR command. This demo will load 8 fonts into a buffer and then display each font. Fonts can also be written in any of the 8 directions, which incidentally may be hard to read! Fonts can be given both foreground & background colors. They may be written upside down, backwards, sideways, any size and placed anywhere on the screen. The listing of this demo is long but the demo itself was kept simple so users may list it to see how it was put together. Also, the entire demo was created with standard Basic 8 fonts.

Most people do not realize that Basic 8 fonts can also be used on the standard 80 column text screen. The command is @FONT which replaces the Commodore font set (upper/lower) with a font in your buffer. There are many reasons you may wish to use the 80 column screen; the most obvious of which is when the user is asked for a lot of keyboard input. Perhaps you need to ask several questions of the user to make your latest Basic 8 adventure game more interesting.

Whatever your reasons are, don't forget that you are welcome to use any of the colors and standard 80 column features with your text screens! Run the program B8.FONT for an example of Basic 8 fonts on your text screen.

@FETCH - Recalls a brush structure stored in a buffer and displays in on the screen

Syntax: @FETCH, Structure #, X, Y, Draw Mode
Example: @FETCH,1,320,100,0

The @FETCH command is used to place a brush you have stored as a structure anywhere on the current screen. The structure number specifies which brush to use from the buffer. The inverse of this command is @STASH, which allows you to clip a part of your screen as a brush.

Graphic Effects

@CBRUSH - Flips a non-compressed brush structure

Syntax: @CBRUSH, Structure #, Reverse, Reflect, Flip
Example: @CBRUSH,1,0,0,1

Run the demo B8.CBRUSH. The CBRUSH command allows you to change a non-compressed brush structure within a buffer. The three options allow you to reverse the colors, make a mirror image or flip the brush upside down. In this

demo we use a full-size screen as our brush and show how it looks when flipped and mirrored. Reverse works much better on monochrome graphics due to the reversal of color cells in color mode.

@ZOOM - Displays an enlarged brush structure

Syntax: @ZOOM, Structure #, Size, Destination X, Destination Y
Example: @ZOOM,1,8,0,0

The B8.ZOOM demo shows two important concepts. First the ZOOM command will enlarge any non-compressed graphic stored in a buffer. The enlargement is always 8x on the X axis and may be adjusted from 1-15x on the Y axis (limited by color cells). For a truly proportional enlargement (such as a pixel editor) you need to clip a brush that is 80 x 25 pixels and expand it 8 times in both directions to make a 640 x 200 screen.

The next important concept is being able to move a blinking box around the screen. In this case (just by chance!) we have a box that is 80 pixels wide and 25 pixels deep. To move the box, set @DRWMODA to complement mode, read the current X/Y position of the mouse and use two @BOX commands like this:

```
@BOX,x,y,0,x+80,y+25,0,0,0,1
@BOX,x,y,0,x+80,y+25,0,0,0,1
```

The first box complements itself with the picture, the next box sets everything right again. List the program to see how this works inside of a mouse reading loop. You will also use this same technique to draw boxes and lines in paint programs.

CHAPTER SEVEN
USER INPUT

Basic 8 uses the @MOUSE command to keep track (internally) of the mouse's current position. This command has two steps; first the mouse reader must be activated and set up for a mouse (port 1) or a joystick (port 2); then the @MOUSE command is used to return the X and Y location of the mouse.

All of the examples in this book assume you are using a 1351 mouse in port 1. Why a mouse? My answer is simple, the Amiga uses a mouse, Macintosh uses a mouse and Nintendo uses a joystick. Need I say more? I have yet to see serious software developed using a joystick - with the exception of games. I use a mouse because it gives a much smoother response and allows you to make full use of both buttons.

You may have a legitimate reason for using a joystick, or you may develop a project that could use either the mouse or the joystick for input. One note: If you have a program that could use either one for input, then you aren't using the mouse to its full capacity. You can test for mouse/or joystick input by using the short example program.

@MOUSE - Read the mouse (joystick) current location.

```
Syntax:   @MOUSE,On/off,Device, X , Y (,Joystick increment)
Examples: @MOUSE,1,0,0,0      -    Mouse on
          X = @MOUSE,2,0           Read X position
          Y = @MOUSE,2,1           Read Y position
          @MOUSE,0                 Mouse off
```

Run the demo on disk #1 called B8.MOUSE. The mouse is set up by using: @MOUSE,1,0,320,100. This command turns on the mouse IRQ reader, sets the device to 0 (1351 mouse in port 1), X location to 320, and Y location to 100. Use this command once early in your program.

To have the pointer move around the screen in response to the mouse you need to make a DO LOOP that reads the mouse X/Y position and places the pointer at that location.

For smooth mouse control, run the loop until a button is pressed and then do your IF THEN statements to find out which button was pressed and to send the program to the correct location.

Here is a sample DO LOOP to read the mouse location:

```
100 DO UNTIL JOY(1) <> 0
110 X=@MOUSE,2,0
120 Y=@MOUSE,2,1
130 @PTR,1,X,Y,0
140 LOOP: @PTR,0
```

```
150:
160 REM check to see which button is pressed
```

The mouse demo will tell you which button was pressed and the current X/Y location of the pointer. Note that the demo does not update the positions until a button is pressed. If it did, smooth mouse control might be compromised. Again, it is best to do all of your screen updates, button checking and such outside of your mouse-reader loop.

In actual practice the DO LOOP as shown above should be just one line of code, scrunched together with no spaces. When programming in Basic every nano second counts! The code should look like this:

```
100 DO UNTIL JOY(1)<>0:X=@MOUSE,2,0:Y=@MOUSE,2,1:@PTR,1,X,Y,0:LOOP:@PTR,0
```

@PTR - Controls location and definition of current pointer (arrow).

```
Syntax:   @PTR, On/Off, X, Y, Definition # (, Height)
Examples: @PTR,0          Pointer OFF
          @PTR,1,0,0,0    Pointer ON
          @PTR,2,0,0,0    Leave trail
```

The @PTR command is used to activate the pointer, place it at the X/Y location, and can be used to display any one of 16 pointers. When Basic 8 is booted, pointer #0 will be the arrow. The above demo uses this arrow as the pointer. Pointers may also be edited to suit your needs by using the Basic 8 toolkit.

The last command in the mouse-reader loop is @PTR,0. This turns the current pointer off. Use this only once after using the pointer. If the command is issued twice you will pick up a glitch on your screen.

Run the demo B8.DRAW-1. This demo uses the same type of mouse reader to control the pointer. The new pointer is customized to be a large dot (pointer #2) and you are working on a monochrome screen for speed. By pressing the left button you can draw with the current pointer, because in this case the @PTR command uses the trail option.

Sketchpad 128 uses this capability to its full potential. The idea for Sketchpad 128 came from a three block program similar to this one!

Run the demo B8.DRAW-2. Draw-2 uses the mouse as a drawing instrument but does not leave a trail of the pointer. Instead, the pointer is used to point to a location and then when the left button is pressed you will begin drawing with a line. (The thickness may be adjusted for special effects.)

When this program is run you can move the arrow pointer around the screen until the left button is pressed. When the button is held down this program will draw a line connecting the last pointer location to the current location. Since these lines are only a pixel or two apart it will appear as a smoothly drawn line.

Keyboard Input

When entering data using a graphic screen you may have noticed a "problem" while using the input statement. As soon as the RETURN key is pressed your screen gets messed up. This is not really a fault of Basic 8: The problem is that a linefeed is issued when the return key is pressed which messes up your graphic screen.

There are several ways to get around the input statement. If data is only needed at the beginning of your program—asking for the player's names, for example—then you can use the @TEXT command to enter the data on a text screen. Use the standard INPUT command for this information, then issue a @MODE and @SCREEN command to start your program.

Using GETKEY

Often you need the user to enter a single number. Input such as the thickness for a line, or a printer variable like a secondary address can be obtained with the GETKEY A$ statement. For this example we would like to enter a single number from 1-9. We will use the VAL function to check the value of the input to see if a number key was pressed. We use the GETKEY A$ because if we tried a GETKEY N and the user entered a letter (or other keypress) the program would crash.

Another reminder about using a GETKEY statement in Basic 8 is to clear the keyboard buffer of any stray characters. This is done by a simple POKE 208,0 statement. The code for this GETKEY subroutine will look like this:

```
10 REM input a number from 1-9, return value of n
20 POKE 208,0: GETKEY A$
30 IF VAL(A$)<1 OR VAL(A$)>9 THEN 20
40 N = VAL(A$)
50 REM at this line n now equals a number from 1-9
```

I suggest that you have this function in a subroutine that could be called with a GOSUB. Your subroutine could also present a command bar asking for a number from 1-9. When the number is pressed the section of screen could be cleared by using a window or by replacing the screen with the stash & fetch commands. With a little planning you will find that simple subroutines can do a lot of work for you, and make your programs more efficient.

Filename Input

In most programs it is necessary to enter a file name. B8.FILENAME shows a simple, yet powerful subroutine that allows the user to type in a name with up to 16 characters. This name can be edited by using the delete or back cursor key. The file name is entered when the Return key is pressed. If the ESCape key is pressed, the routine will abort to the main menu of your program.

Run the program B8.FILENAME. Try entering a filename and using the keys as described above. List the program and you will see how it works. This has proven to be a very useful subroutine and can be used in many types of applications.

CHAPTER EIGHT
UTILITIES

Reading The Directory

Nearly every program needs access to the directory so the user can load files. Basic 8 has the @DIR command, but it is not as simple to use as the other commands. One nice feature of Basic 8 is the file naming convention used. If you prefix your program files with "B8." they are easy to find. Here is a list of Basic 8 prefixes:

B8.	Basic 8 Program file
PICT.	Picture file
FONT.	Basic 8 font file
BRUS.	Brush file (smaller then a full screen)
PATR.	Pattern information
PTRS.	Pointer data
LOGO.	Basic 8 Logo files
P.HC-	Basic 8 printer drivers
PAGE.	Large screen for full page printout
SPRD.	spreadsheet for BASIC CALC
SOUN.	Sound file from DigiTalker 128
ICON.	Basic 8 icon
DATA.	Sequential data file
TEXT.	Sequential text file

The Basic 8 @DIR command can be used to search the disks directory for a specific type of file.

@DIR - Used within a subroutine to read a disks directory

For an example-run the program called B8.DIR READ. This program reads the current directory four times. The first time it reads all of the files on the disk. On the second loop it looks only for Basic 8 program files with the B8. prefix. Next, it looks for all of the PICT. files. Then it will search for all of the BRUS. files.

This directory reader can be accessed as often as desired with a simple GOSUB. To search for a specific type of file all you need is define FTYPE$ as your search string.

In the interest of clarity this demonstration uses no graphic screen or Basic 8 fonts. Reading the directory is one of the critical functions of Basic 8. Please list this program, print it out and take the time needed to understand how it works. Below is an explanation of the "critical" portions of the program:

```
10  REM B8.READ DIR          :Title
50  DIM DE$(296)             :DE$(#) Holds the filenames
60  DE$="                    :The first variable in your program
                              must be a string with 16 blank spaces
```

```
70  DN=8                              :Current drive #
110 FTYPE$=""                         :Search full directory
250 REM READ DIRECTORY
260 FOR I=0 TO DE:DE$(I)="":NEXT:DE=0 :Clear old strings
270 OPEN3,DN,0,"$0:"+FTYPE$+"*"       :Open channel to disk drive
280 L=@DIR$:IF ST<>0THEN 320          :Find length of next file. Check for
                                       I/O errors.
290 F$=LEFT$(DE$,L)                   :f$ = filename
300 DE$(DE)=F$                        :Assign de$(#)
310 dE=DE+1:GOTO 280                  :File counter/loop
320 CLOSE3: DE=DE-1                   :Close I/O channel to disk drive and
                                       Adjust filecount.
360 FOR I=1 TO DE:PRINT DE$(I):NEXT   :Display filenames on a text screen
```

Note: de$(0) will always be the current disk name.

Slideshow

Slideshows are easy to make with Basic 8. At the simplest level you can display one picture after the other. But with a bit of effort you can create a dazzling display with pictures sliding up and down, side-to-side and even diagonally across your screen.

To set up a slideshow you will need several pictures all in the same format (I recommend the 8x2 color cell format.) Set up your program to use mode 2, screens 4 & 5. For our slideshow we will be loading pictures on to screen 5 (hidden from view.) Then we will use the @COPY commands to copy our picture onto screen 4. While that picture is being displayed we will load the next one behind it.

Run the demo B8.SLIDESHOW. The workhorse behind this entire program is the @COPY command. Anyone could figure out how to copy the entire screen with: @COPY,5,0,0,640,200,4,0,0. But as graphics enthusiasts we like to add a bit more to our program. To create the graphic effects, the demo uses the @COPY command to copy parts of the hidden screen onto the viewing screen. For example, you can copy the one side of the picture, pause, then copy the other side. You could cause the picture to appear on the screen in stripes by copying every other column. The picture will appear from the side by copying one column at a time.

These effects are very simple to create and most of them simply use the @COPY command in a loop to copy parts of the screen from the hidden screen to the viewing screen. Our demo only shows one picture and blanks the screen between each copy sequence. Your program should be set in a loop to display several pictures!

Selecting Colors (@PIXEL)

The Basic 8 @PIXEL command serves two functions. First run the demo B8.PIXEL-1. This demo will allow you to point-and-click to select a checkered square. If you pick a black square then you will be told that the pixel you have selected is 'on'. The PIXEL command may be used to check even a single dot! The syntax for the command is:

```
A=@PIXEL,320,100,0
If A = 0 then pixel is off
If A = 1 then pixel is on
```

The second function of the PIXEL command is to return both color values in a given color cell. This is extremely important for example, when you need to select colors for a paint program. Run the demo B8.PIXEL-2. Pressing the first button will allow you to paint with the current color. Pressing the second button will take you up to the color palette where you may select any other color to paint with. (And you thought this was going to be hard!) The syntax to read colors is:

```
C=@PIXEL,320,100,1
Foreground = (C AND 15)
Background = ((C AND 240)/16)
```

Using Menus

There are all types of menu possibilities when using Basic 8. The basic function of a menu is to present the user with a list of options from which to select. Keyboard input could be used (press B for a box, L for a line etc..), however, since this is a graphic environment I would much rather just use the mouse to point at my selections.

Menus can be very simple. Suppose for example I had a list of options listed on the side of my screen like this:

```
Option #1
Option #2
Option #3
```

I would wait for the mouse to be clicked and read the Y location. If y=23 I can divide that by 8 (the height of my font) to get 2.87 or rounded off to an integer of 3! After reading the menu item I could use a GOSUB 100,200,300. I have clicked on the third option and would use the subroutine found at line 300. It can be that simple.

If you are making a drawing program you could make a strip of small boxes across the top of your screen as with Basic Paint. The main point in making menu selections is to know where you are, and keep your boxes the same size for easy division.

Now that you understand the @PIXEL command you can have multicolored options. You could click on something yellow and have your 128 say "YELLOW". You can implement DIGITALKER 128 with your Basic 8 programs. What a neat idea to create educational software!

For an excellent pull-down menu system run the demo called B8.MENU. It will take just a moment to create a menu with 50 options just waiting to be selected! This type of menu is very fast and professional looking. It is used in both News Maker and Spectrum 128. Practice clicking on the header bar, then make a selection. The reason for the speed is the fact that the entire menu is

displayed on a hidden screen. Each portion is copied onto your viewing screen when you select it. Use the copy command generously in your programs—it is very fast!

Using Print Shop Graphics

Run the program B8.PRINTSHOP. Print Shop graphics are special to Basic 8. The reason is that Print Shop graphics are standardized, with pixels drawn on X,Y coordinates of 88 by 52 pixels, and they have no color information to sort out. These graphics are in this format because they are set up for your printer - not your screen. 40 column screen graphics all deal with an 8x8 pixel color card, where each portion of a graphic is stacked 8 bytes deep. Basic 8 graphics, on the other hand, are laid out sequentially using columns 0-79 and rows 0-199 in order.

Print Shop graphics are easy to use within a Basic 8 environment. They may be loaded directly from a Print Shop disk using the 3 block graphics for non-Commodore printers. Run the program B8.PRINTSHOP for an example.

Now for a short explanation. Print Shop graphics are vary similar to Basic 8 brush files, however, they contain graphic data only and the files are not proceeded with the 11 bytes of data that define a brush structure to Basic 8. What this demo is doing is first defining a buffer and then stashing a BLANK area of the screen the size of the Print Shop graphic. The graphic itself is BLOADed on top of the structure. Then to display the graphic, the structure is fetched from memory.

When BLOADing your graphic you must add the buffers address (which is usually 1024), plus the structure's address (zero in this case), plus 11 bytes to skip over the brush data. For example, to load your Print Shop graphic in structure 0 you would BLOAD it to 1035, as in the example.

Using a 128D With a 1581 Drive

128D users who have purchased a 1581 drive for its increased speed and storage capabilities are faced with a problem: Their internal drive is set to be device #8. This means that they need to set the device number on the 1581 to be device #9. What happens when they want to boot a program from drive #9? NOTHING! Most commercial programs must load several modules, and the program expects these to be on drive #8. Also, a typical way to fix this problem is via software, you know the type - "Please turn off drive 8, now turn it back on...". This just doesn't cut it for a 128D owner with no dip switch, much less a power switch!

Basic 8 programs happen to run best on a 1581 drive! You need the speed to load in all of the graphics and fonts, and you can certainly use the storage (I like all of the fonts etc. to be on my workdisk!) So I created a small program that will allow a software swap of device numbers 8 and 9. This program is on your disk and is called SWAP#8/9. You may now back up any non-copy protected software (Spectrum 128 for example) onto your 1581 drive. You may even create an autoboot sector for it! I would even suggest that you put the swap program on your small disk so it is handy.

Swap will swap any 8/9 disk drive combination, however we will consider the following scenario for this example: You have a 128D with a built-in 1571 drive which is device #8. You would really like to be using your nifty 1581 drive, but it is set (by the dip switches) to be device #9. It is a good place to keep your data, but it would sure be nice to run software from it.

Enter SWAP#8/9. Run the program and a message will appear telling you that your device numbers are about to be swapped. You will receive a prompt asking if this is what you want to do. Press "Y" for yes. At this point the program changes your 1571 drive to be device #10 (Yes, device 10 this is not a misprint!) Then it changes your 1581 drive to be device #8. Then it looks back at the 1571 drive (device #10 remember) and sets it to be device #9. Lastly, the program initializes both drives to prevent any lingering I/O errors.

List the program and you will see that it is a very small program. What amazes me is the fact that this has been a problem since the 128D's were introduced and nobody has addressed this issue. I find Swap to be a very useful utility which is also quite simple. As you can see by the listing, there is simply no room for a bug or virus to cause any problem whatsoever. It simply goes in and swaps device numbers.

Function Keys

Function keys make life a lot more interesting when using a computer. However, there are times when a user may inadvertently press a function key while in the midst of your Basic 8 program. This is not good, considering the words @TEXT could appear on your screen instead of the user's intended input. Naturally, they will assume the program is defective since you wrote it!

It is a good idea to turn off the function keys at the start of your programs (or to define them to your own needs as in B8.LOGO demo!) Please remember to set your function keys back to normal when your program ends.

To make the Function Keys Null:

```
100 REM ---- Null function keys
110 KEY 1, ""
120 KEY 2, ""
130 KEY 3, ""
140 KEY 4, ""
150 KEY 5, ""
160 KEY 6, ""
170 KEY 7, ""
180 KEY 8, ""
```

Add the following line to restore function keys to their normal (Basic 8) definitions:

```
500 REM ---- Restore function keys
510 KEY 1, "FAST:@TEXT" + CHR$(13)
520 KEY 2, "DLOAD" + CHR$(34)
530 KEY 3, "DIRECTORY" + CHR$(13)
```

```
540 KEY 4, ""
550 KEY 5, "DSAVE" + CHR$(34)
560 KEY 6, "RUN" + CHR$(13)
570 KEY 7, "LIST" + CHR$(13)
580 KEY 8, ""
```

When I am programming I prefer to reset keys F1 and F7 as:

```
KEY 1, "@TEXT"+CHR$(14)+CHR$(13)
KEY 7, "LIST "
```

F1 will now restore you to the graphic screen and also put you in upper/lower case text mode. Graphics can be a little hard to read! F7 will print "list " to your screen and you may type in which line numbers you want listed. You may change the program FKEYS (on Disk #1) to suit your own needs.

For the examples with this book the function keys are left active. This is because this book is being read by intelligent, programmer type people who would never press extra keys when not asked to!

CHAPTER NINE
ADDITIONAL DEMONSTRATIONS

Basic 8 is such a versatile language it would be impossible to demonstrate all of its potential. The following demonstrations have been selected because they show the commands we have just studied, and how they can be used in a variety of situations. These demos range from animations, to paint programs, to business applications. I have also included some demonstrations that use very unusual techniques to achieve "impossible" results. Can you put a Basic 8 graphic on a text screen? It can be done, but don't expect to find the answer in your Basic 8 manual!

3D Animator

On Disk #2 is a program called B8.3D ANIMATOR. If there was ever a program that takes advantage of you 128 and peripherals this is it. With the animator you may use a mouse or a joystick for input and works with two drives. It runs on a standard 16k machine but can create smoother animation by using the full 64k video RAM capabilities. Also, if you have an REU you can the view the animations full screen size! (With a little help from the @ZOOM command.)

This program also takes advantage of the Basic 8 commands we have just studied, and it shows how everything fits together in a large program. Each section of the program is fully documented by remark statements. You are encouraged to list it out and see how it was put together.

Since the program is so large and this is a tutorial of how to get the most from Basic 8, we decided to include the Basic 8 Run Time Library to show how it could be used to run your own programs. Note: One small change has been added to the startup program—it now offers a menu selection that will allow you to exit with the Run Time system installed. With the RTL installed you may run any of the demos on either disk without booting your editor disk.

Special thanks to Kathy Wright for allowing us to use her beautiful Print Shop graphics. These animals look so nice when animated—much better then stick men I would have drawn!

You may load the Run Time Library and select the 3D Animator from the menu, Or load in B8.3D ANIMATOR itself. The first prompt will ask if you want to use the 16k or 64k version (the best demos use the 64k VDC!), and it will ask if a Ram Expansion unit is available. You may select the joystick or the mouse by clicking a button. The title screen will be displayed while the program is loading in the menus. In a moment the Main Menu will be presented.

MAIN MENU

All menu selections can be made by simply pointing at the desired selection and clicking the Left mouse button (or fire button).

Information

By selecting Information, you will be presented with a screen showing what mode you are currently using and how many pictures your final animation will consist of. Also, there is a short demo of what rotation on the X/Y axis will look like. The Z axis makes objects appear larger or smaller. Once you have read the information file you may 'click' your mouse to exit.

Create 3D File

By selecting this option you will be presented with the CREATE 3D MENU. This menu will allow you to adjust rotations and set the vanish option from 0 - 100%. This menu will be fully explained in a few moments.

Display 3D File

There are several 3D files supplied with this demo. You will be given a requester to select a filename. Use the up/down arrow keys to select a file or press the escape key to exit. If you are in the 16k video RAM mode, you will be viewing an animation on a single monochrome screen.

Drive #

Click on this selection to change your data drive to 8/9. Your data drive may be a Print Shop Graphics disk. The 3D Animator will save the animations to drive #8.

Background Color
Foreground Color

Clicking on these options will cycle through the sixteen colors. Usually a black background with light colored object gives a more dramatic effect. (Black objects on a white background works well with everything!)

Exit

One click here and you are out of the program!

CREATE 3D MENU

-Rotation-
X Axis On/Off #

You have two places to click for these settings. First you may select the On/Off option. Off resets the number to zero. On will allow you to increment the rotations by ten degrees by clicking on the number. When using the X axis remember that the top will come toward you while the bottom rotates away.

Y Axis On/Off #

Objects spun on the Y axis will rotate as if they are on a barber pole. The right side will move away from you, while the left side moves toward you.

Vanish On/Off #

The Vanish feature make very interesting animations. If you set an object to vanish 100% it become a small dot in the distance! Other settings will have your graphics zooming in and out at you! All of these settings may be combined with any other settings to produce fantastic results.

-Load Graphic-

The Basic 8 3D Animator uses two types of graphics. The first are Basic 8 brush files, and the second type are Print Shop graphics (3 block non-Commodore printer types). These may be loaded directly from Print Shop disks. No conversion is needed because the Animator takes care of it. There is only one limitation on your brush files—they can be no larger then 88 x 52 pixels. Use you favorite drawing program such as Sketchpad 128 and load in Pict.Template-3D to use for a guide. Monochrome brush files work best, but if you load in a color graphic remember that only the bitmap will be used.

Print Shop

You will be given a standard requester to load in a Print Shop graphic. Since Print Shop files have no standard header such as PICT. or BRUS. you will be presented with a full directory to select from. Loading in a non-Print Shop file may cause your system to crash! Take note: If you attempt to load a whole picture into a buffer waiting for a three block graphic, it is not going to fit! To assist you in making the correct choice the sample pictures on Disk #2 are prefixed with "PS." meaning it is a Print Shop graphic.

Brush File

As mentioned, this is any Basic 8 brush file which is no larger than 88 x 52 pixels. Please practice by using the 16K mode. Line art will render much faster then highly detailed shaded drawings. When using the full 64k mode you will need to give it some time. Remember that it must create 40 pictures pixel by pixel. I have had pictures take as little as 12 minutes to as long as an hour!

Main Menu

This selection will take you back to the MAIN MENU.

Ok, so you are impressed with the spinning artwork. How is this done? Most of the subroutines have been explained in this book. List the program to see how the directory is read, the files are selected, and how the screens work together. Once a graphic is on your screen each pixel is read. Your computer will take some time to count to 4576. Each pixel that is turned on will be stored in an X/Y array. From this point on only these pixels need to be drawn. Line art works fast because only a fraction of the 4576 pixels need to be rendered.

The @ORIGIN command is then incremented by the setting you have chosen previously. Because graphics are flat, a 180-degree spin can look like 360 when played forwards and backwards. These graphics are drawn on a screen dot by dot. The final product is a picture file that may be viewed with Basic Paint.

The animation is played by with a simple loop using the @COPY command. Animation techniques will be explained in detail by Roger Silva in part two of this book.

B8.PLAYER16K
B8.PLAYER64K

These animations may be created and used in your own programs. These two player files are the guts of the system. You may run them to simply play back an animation. List them and see how the loops work within each other. The original concept for this idea came from Lou Wallace's B8.Fasthouse64k demo. This subroutine is used with permission.

As with the other routines in this book you are encouraged to experiment with these programs. You could adjust the animator to create different size files, or the player could repeat certain sequences fast, then slow. Make a feature length movie with your VCR. Basic 8 is what you make of it!

Potpourri

Here are several additional bonus programs that show just how powerful Basic 8 can be. Enjoy!

B8.Paint 128

You have heard a lot about how the 128 can paint using 128 colors. This is because each of the 16 colors can be mixed with the others using dithering. When painting with these colors you may not have the freedom to paint with lines and such due to the size of the color cells. The smallest paintbrush you have access to is the size of the 8x2 color cell. The advantage of this is that you will have no color bleed since cells will not overlap each other. I came upon this unusual effect when I was experimenting with multicolored windows using @CLEAR,170 which causes windows to be cleared with thin vertical stripes of background / foreground colors. The smallest window I could make was an 8x2 color cell. It was obvious that I could then use the mouse and paint by opening very small windows. (You would be amazed at what is obvious when you exist so close to the edge of reality.) Later I used this same technique to create Spectrum 128.

B8.ET Demo

This is an interesting animated demo using brush animation and multiple screens. The animation looks smooth because the technique I used is not the obvious way to do animation. There are four monochrome screens used in this demo. The logic behind the program works like this: Screen 0 is what the user is

viewing. The background scene is loaded onto screen 3, a hidden screen. The background (screen 3) is then copied onto screen 2 (also a hidden screen). The bike graphic is then overlaid on the scene. Once the scene is compiled it is copied to the viewing screen. Think of the scenes as being created as a single frame in a movie. With this demo the user never sees the graphic as it "drops down" onto the screen, and the stars and moon are visible through the spokes of the bike tire!

B8.Graph

This sales graph was created using three dimensional colored bars. The techniques will be discussed in the second part of this book.

B8.Bounce

A simple bouncing ball demo using a sphere and the scroll command.

B8.Dice

These dice use pointers to create an animated effect. A subroutine like this could be used in many types of computer games where the roll of the dice is required.

B8.Donut-Spin

The light source on a toroid can be changed by cutting the object out as a brush and flipping it with the @CBRUSH command. Using the @COPY command for speed gives us a wobbling donut effect.

B8.4 Donuts

If you enjoyed the last demo, you will like this one four times as much! Four donuts, all wobbling!

B8.Spin Sphere

The same technique can be used on any of the Rylander solids. This demo uses a spinning sphere.

B8.Graph/Text

Here is a challenge for even the most advanced Basic 8 user. Can you create a Basic 8 graphic and display it on the standard text screen? Sure you can!

I have included this (and the next) demo to show that not everything in Basic 8 is obvious. I will not explain this one in detail but for the curious it involves reading byte data from a graphic (stored in a buffer!) and transferring that to a character set also in a buffer. (Remember Graphics have bytes stacked end to end, and a character set has 8 bytes stacked on top of each other!). Once this is done it is a simple matter to use the @FONT command to display your new 80 column text graphic! For fun rename font-sphere to font.sphere and look at it with any text program.

B8.Expand it

One unusual technique deserves another. Again this is a demo to keep enthusiastic Basic 8 users thinking. Since a graphic can be converted to a character set, and the character set can be displayed with proportional settings on the X and Y axes, it stands to reason that you could enlarge any graphic proportionally on both the X and Y axis. This is nice because the @ZOOM command is only proportional when a graphic is enlarged 8 times. Run this demo and you will see just how nice it will make an enlargement. I was so impressed that I used this technique to create Poster Maker 128. It is the only picture enlarger for 80 column graphics.

HOW TO GET THE MOST OUT OF BASIC 8
Part II: ANIMATION

INTRODUCTION

This portion of the book will deal primarily with animation using the Basic 8 language extension from Walrusoft. It has been divided into five chapters which will address the various aspects of producing an animated sequence. A demo disk is also included with this book that will more or less follow the activities described in the text. You are encouraged to first read the book, and then to follow along with the demo.

Chapter 10 will deal with the importance of proper planning and organization of an animation project. In this section I will discuss how to achieve balance and symmetry with your graphics. You will discover how to find out where you are on the screen, and to know where you are within the program. The final topic of this section will deal with developing aids to assist you with screen presentation.

Chapter 11 will discuss the wonders of the Basic 8 3-D graphic environment. You will see how this can produce a feeling of depth and solidity within your graphics. The major 3-D commands will be explored: @ANGLE, @ORIGIN, and @VIEW. The difference between parallel and perspective mode using @DRWMODB will be explained, with several examples of this provided on disk.

Chapter 12 illustrates how to handle the numerous BRUS. and PICT. files that are a large part of many animation sequences. Screen and buffer layout used in the demo is detailed, and includes advanced techniques for managing large amounts of data.

Chapter 13 describes the various methods of animating a graphic. The merits of @COPY and @FETCH will be discussed, with examples provided on the demo disk. We will also explore the possibilities of animation using some of the other Basic 8 commands. These techniques will all be brought together in the main demo program on the disk.

Finally, Chapter 14 will explore other presentation aspects; most notably music and sound enhancements. Although this is not a Basic 8 subject, I believe you will see how this helps to flesh-out your program and compliment its graphic animations.

It is assumed that you have at least some knowledge of programming, understand nested loops, and that you have read and understood the first section of this book. I believe you will be amazed at the versatility and detail available with the aid of Basic 8. Perhaps this will encourage you to try your hand at producing animated graphics. All you need is a little imagination!

CHAPTER TEN
PLANNING YOUR PROJECT

Have you experienced this scenario before: You have this wonderful idea for an animation sequence. You sit down to the task at hand, and find, after many attempts, that there are still bugs in the program. Perhaps your animation is jerky; or you can't seem to place the graphics where you want them. Do you find yourself asking, "What the heck is going on?", the colors change, garbage appears on the screen, or some other unknown malady strikes your program. Maybe you have a good animation sequence, but its presentation could be better. If you experience these types of problems then this book is for you! All you need is a little planning and organization to help you know where you are in the program and how your graphics will be manipulated and displayed on the screen.

We will begin with a quick review of the 80 column screen. You should know that the screen is 640 pixels by 200 pixels. (Although with 64K VDC RAM you may have a virtual screen up to 1280x409, the monitor can only show 640x200 at a time.) Each pixel is asymmetrical; that is, it is taller than it is wide. The color resolution available in Basic 8 allows one foreground and one background color per "cell". These colors can be combined to produce 128 shades from the 16 primary colors available. The color cell sizes (in pixels) are: 8x2, 8x4, 8x8, 8x16, and monochrome.

One of the most important things to remember is that many of the BASIC 8 commands will index your graphics to the nearest color cell. Regardless of where you want to place the graphic, unless it happens to directly fall on that intersection, the location is rounded up or down to the nearest cell. This action does not apply to monochrome! That's one of its advantages. It allows for very smooth animation and scrolling sequences. Also, because there is no color to manipulate, the execution time is accelerated within the program. Unfortunately, you are only allowed one foreground and one background color for the entire screen.

This leads to the logical conclusion that if this "grid" is utilized, things will become much simpler. Because all color cell sizes have the same number of pixels across (8), there are only 80 locations that you can place an object in the x direction. (In the case of a wider virtual screen you should only display in increments of 8 pixels). The same kind of relationship exists in the y direction.

Depending on the cell size, the y axis locations will be in increments of 16, 8, 4, or 2 pixels. So if your location is NOT evenly divisible by 8 in the x direction, or your location is NOT evenly divisible by the color cell size in the y direction, it's a safe bet that you may experience problems. If they are evenly divisible, then you are well on your way to easier screen layout. Obviously there is an advantage to the 8x2 cell size—you can display four foreground and four background colors in the same space as an 8x8 cell area (which can only hold one foreground and one background color.) This size allows for smooth scrolling in the y direction; plus you have 8000 locations where you can place your graphics! (640/8 pixels in x direction, times 200/2 pixels in the y direction = 8000). This "color resolution" of 8x2 pixels is becoming somewhat

of a standard in the BASIC 8 community, so, it will be used for our discussions, and on the demo disk provided with this book.

The only negative aspect of the 8x2 color cell is the minor fact that your maximum screen size is limited to 640 x 546! I believe you will find this adequate for most animation projects. Of course if you do require the additional screen memory, use one of the other cell sizes. Just remember; there will be a reduction in the number of screen locations for your graphics, vertical scrolling will be less smooth, and you will also experience a reduction in color density.

Graphic Balance

Now that you are aware of the importance of screen location based on the starting x and y coordinate of the color cell, it is time to discuss the need for balance and uniformity in screen presentation. Presenting the user with a mishmash of text and graphics will never make an impression. Even if your animation sequence is excellent, its presentation will be lost as the viewer tries to take it all in, not knowing how it relates or where to begin.

There are several ways to balance the screen. The first is also the most widely used; Centering! Placing an object in the middle of the screen looks much better than the upper left hand corner of the monitor. Whenever you look at the monitor, screen center is usually the first place you focus.

This works fine for an individual graphic or animation sequence, but what about the full screen? In this case you should look at the screen as consisting of four sections: Upper left,upper right, lower left,lower right. Take the time to place your text and graphics around the center of the screen. Text or graphic areas on one side of the screen should be offset by similarly proportioned areas of text and/or graphics on the other side. If the area is small in the Y direction, then it should also be centered top to bottom on the screen.

Of course not all graphics or text areas will be capable of centering. If this is the case then I have found that offsetting them towards the top of the screen or to the left will provide the best looking display. After a while you will be able to judge for yourself. If it "feels" out of place, then try moving it to another location. Eventually you will see where it fits best on the screen. Just keep in mind to adjust to the next color cell location. Remember; its the little details which add up to a professional look!

A little restraint is sometimes called for. It is not necessary that every pixel be turned on. Also, it actually detracts from the screen if you try to cram in all 128 possible shades. Oh, it can be done! But it is the rare occurrence when this looks acceptable. Avoid color clashes. When you are deciding on your color choices for your program, try using a simple color wheel. This handy item can usually be found in any art or paint store. It displays the primary colors as well as the complimentary ones.

128 shades! I know many of you are not aware of this capability with BASIC 8. If you believed that you were limited to the 16 colors available on the 40 column screen then this is a wonderful surprise. By combining the complimentary

colors within the standard sixteen, you can produce ONE HUNDRED AND TWENTY EIGHT colors or hues! The simple process that makes this possible is called dithering. Because of the high resolution, you do not see the individual background or foreground color but a blending of the two. It is the bitmap which controls this deception. In its finest state the bitmap is composed of a matrix of off and on pixels. This can be represented as a series of 0's and 1's (where 0=off and 1=on). In this case an 8x4 pixel area would look like this:

01010101
10101010
01010101
10101010

It is hard to see how this is accomplished within this book. By using @ZOOM within a paint program written for BASIC 8, You will easily see the individual pixels. It is a great way to study the shadings and detail in a picture. You can learn a lot about the style and technique used by the artist.

Coarser textures can be achieved by using larger matrices such as 00110011, but you will find that as the matrix increases, the colors blend less and you begin to see the individual foreground and background colors in what I call the checkerboard effect.

Dithered areas can be saved and stored on disk as patterns using the PATR.filename format. In fact, anything that is placed on the BASIC 8 screen can be used as a pattern! Developing a library of various file types will make it much easier to produce a graphic presentation. This is also true of program subroutines that perform the different types of animations. There is no need to repeat the task of figuring out a routine, or drawing a dithered pattern every time you wish to use it.

Sometimes, when your co-ordinates overlap, there is a color change on a part of the bitmap. The last color selected or changed, alters a screen color cell near the new graphic. This is known as color bleed!

It is important to remember that a color cell has TWO colors active at one time! Think of the background color as your canvas. If you have not turned any pixels "ON", then this is the color you will see. If a pixel is ON, this color can be thought of as laying over or on top of the canvas. Even though no pixels are turned on in a color cell, the foreground color is very much present! The easiest way to think of this is by saying that the foreground color is transparent. You will not see it unless a pixel is turned on. It will however still affect any other cell it may overlay when you draw or fetch a brush to screen.

Why do you sometimes lose the pointer in a drawing program? The pointer is not a sprite! It is a simple bitmap image that is placed on the screen as an interrupt-driven graphic. The pointer color is derived from the foreground color of the current cell it occupies. When it encounters a section of the screen where the foreground color covers the entire area, you loose the pointer. If your paint program has a "compliment" mode then you can use this. For your own program application you would activate the complement element of the @DRWMODA

command. In this mode pixels are toggled; pixels on are turned off, pixels off are turned on. In this fashion, you will always be able to see the pointer. Because it is interrupt driven, it will always leave the bitmap intact as it is moved about. You should deactivate the pointer when you plan to fetch a BRUS.file as it may corrupt your bitmap when the BRUS is placed over a pointer location. The affected area will appear reversed.

By following the information in this section, you should be able to conquer what can only be described as some of the most annoying occurrences within a picture or program.

Text Balance

The restraint you show with the colors also applies to the text shown on the screen. Not only did you receive a diverse number of FONT. files with BASIC 8 but, you are capable of obtaining many more with the font converter located on the editor disk. You can display any number of fonts on the screen at one time. You also have the wonderful ability to display them in different sizes. A character can be the regular eighty column size, expand it to forty column on up to sixteen columns per character. You can also expand in the Y direction, up to 16 rows deep! On the other end there is even a 160 column character set that is quite readable.

It is advisable not to mix to many font types on the screen at the same time. Also, do not try to achieve a different color or size for every text area. This tends to confuse the user and things usually end up looking jumbled. The next time you pick up a magazine, take a look at the text layout and picture/graphics positioning on the page. You will see that they follow the same principles discussed here.

Notice how they use columns to display the text. Column layout not only looks better, but it is more readable. A two-column layout is optimum for the screen but you can also have three columns at 25 characters per column and still maintain a quality "look". I also find that justifying both edges of the column leads to a very clean and professional look. Justification means aligning each end of a line with the respective ends of all other lines in the column. This usually means that you will have to add spaces between some of the words in order to align the right edge. A good rule of thumb is to not exceed two spaces between each word in the line. If you have to many spaces then try re-wording your sentence, or moving words up from the next line.

When you do mix text and graphics, the column layout will automatically partition the screen for locating your graphics. You can also center them in the screen and wrap the text around them. This method is good when you have a single graphic that does not fit within a column width.

I hope you will begin to realize just how important it is to plan out your project, and to stay organized. The next few topics we will discuss, to wind up this section, will deal with developing aids to assist you in screen layout and organization.

Screen Aids

One of the easiest and most versatile screen aids to have can easily be produced and printed in a matter of minutes. It is simply a grid that is drawn on the screen and output to your printer. Probably the best size of grid to use is one that is 8x8 pixels. The apparent versatility becomes evident when you use this grid to lay out text on the screen. Because the Basic 8 fonts are expandable in multiples of 8x8 pixels, it simply becomes a matter of transferring the paper layout to the screen while still maintaining a balanced look in the presentation. I keep a huge file of xeroxed copies on hand for instant use. A BASIC 8 screen printed out sideways on my MPS802 emulated printer produces a copy that is very close to the actual screen size! This is invaluable when I use it for sketching a picture and maintaining the proportions and symmetry between the paper grid and the screen.

Another grid I use only shows the 80 columns. I use this grid to lay out the drawing I want to display as a graphic on the screen. Because I use the 8x2 color cell size, if I am off in the Y direction, then it is only a matter of moving the graphic up or down one pixel row.

One of the handy application programs on Disk #2 is called B8.MAKE GRID. You can use this program to draw every conceivable type of grid. Experiment with it and you are sure to find a grid that fits your purpose. Save the grid as a picture file and you will always have easy access to it. It can be printed out later when needed. The file PICT.GRID 8X8 on Disk #1 is pre-created to give you a fast 8X8 grid to print out.

Now before you start drawing all over this grid; think about how you intend to use it. The simplest function of the grid is to provide you with the X and Y coordinates to load your graphics to. You can also use pieces of paper that approximate the text and graphic areas you desire when you lay out the screen. These can be arranged and rearranged many times faster than by hit or miss placement on the screen. I tend to use different colored pieces of paper to represent the text and graphics areas. In this way, if I have similar sized areas, I don't have to worry about mixing up the layout.

It also becomes easier to locate the center of a circle, or the X1-Y1,X2-Y2 coordinates of boxes and lines. Remember to follow the grid when determining the colored areas of the screen.

A grid can also be used when you want to draw freeform shapes. The easiest way to do this is to place a plastic sheet over the grid. I suggest using the type sold for use with an overhead projector. Don't forget to get a couple of the special pens they use on the plastic. The ink can be wiped off with a damp sponge allowing the plastic to be reused.

Now its a simple matter of placing the plastic (which you may have to cut to fit) against the glass of the monitor's screen. It can stay there usually by static electricity alone, but I like to use the sticky strip (less paper) from one of those little sticky note pads. You can reuse these strips many times with minimal sticky residue. Just remember to index the drawing with the screen columns.

With this task completed you can now use the various drawing options of your favorite paint program to copy, trace or whatever you want to call it, to the screen. You can transfer complex pictures to the screen using this method. Actually, almost any picture you place under the plastic can be traced and transferred to the screen. This method is used by both amateurs and professionals alike. It does not mean that it is the only method available. For instance; as you get used to using a mouse, you will be able to freehand draw your graphic many times faster than the trace method. There are certain artists who have become very proficient at this and can render a drawing faster than you can imagine.

If you do draw freehand, then you can and should check your rendering against a column grid. This will show you where there will be potential conflict with other color cells, and you can make the necessary changes. Just remember, it is easier to make changes to your outline sketch, than it is to change a nearly completed picture.

There are additional screen aids that you can use to help in locating your graphics. If you are using a black border with a black screen background, then locating the screen corners can seem very difficult. When I first became interested in drawing graphics, I found it very useful to develop a library of PICT.STARTDRAW files. Several of these files were of the grids we spoke of earlier. One that I used quite often consisted of nothing but a dot in each corner and center of the screen. I also had dots at top and bottom of the screen area that located the space between the text columns. The dots are relatively easy to edit out, and help in maintaining the visual layout of the screen.

Did you ever want to know exactly where the pointer was on the screen? Wouldn't it be useful to point somewhere on the screen and know exactly what the X and Y values were?

It's as easy as entering a line into the mouse/joystick pointer subroutine of a program. As a mater of fact, there is already such a line in the BASIC PAINT program that was supplied with the BASIC 8 package. It is buried in a REM statement in line 210! By removing the REM from the line, the mouse coordinates are displayed at the bottom left of the screen whenever the icon strip is visible.

Within your own programs you should display the numbers where it won't interfere with the graphic image. To display to the lower left just include this line in the mouse/pointer subroutine:

```
@CHAR,254,0,192,1,1,2,STR$(MX)+"  ":STR$(MY)
```

It is assumed that you are using MX and MY as the variables that hold the x and y coordinates of the pointer.

As you become familiar with Basic 8, you will probably develop even more useful methods to lay out and organize your graphics. The process of rendering graphics will come more and more easily to you. Just keep developing new aids as the need arises. Save and record these aids for future projects. It saves a lot of time and decreases the frustration level if you can call upon previous work

to assist you. You have the added benefit of knowing exactly what to expect from past experiences. The key is in maintaining the fun and feeling of creative progress when you program and/or render graphic images.

For the purposes of this book, I will discuss how to produce animated sequences for a typical space type adventure. I chose a program of this type because of its wide appeal and unlimited graphic possibilities. The demo on Disk #2, titled "B8.MAINANIMATION" incorporates most of the subjects discussed in this section of the book.

I will not try and explain how to come up with a story line or how to produce a game; that would require another book! You will see however, that a well presented set of sequences can appear to follow a story line where none exists. With the demo disk you will see the elements of a program come together with a display of simulated space travel.

The screen presentation in the demo is divided into three sections. The top 5 rows will display the banner graphic containing the demo's title. The remaining section of the screen is divided into two halves, left and right. The left half will be used to present the graphics, while the right hand side will display text. Other graphics or text will be placed in other locations following the principles of centering graphics for balance and unity.

The demo will display a banner graphic with the word "animation" spinning into the picture at screen center. It will then scroll to the top of the screen. The lower screen will fill with text, and an opening song will be played. You will then be presented with the Main Menu where you can access any section you desire.

The graphics used in the demos are a product of imagination! You are encouraged to produce your own pictures based on whatever concept you have. The graphics in the demo were drawn with the aforementioned principles and techniques.

As you read the other sections you will see how the animation was accomplished. In the next section we will discuss the three dimensional aspect of Basic 8. You will discover how to use it to develop animation sequences of your own.

Please keep in mind that the information contained in the following sections should be accounted for when planning your project. They are the nuts and bolts of animation on the 128. Each section will require its own special attention to layout and organization.

CHAPTER ELEVEN
THE 3D GRAPHIC ENVIRONMENT

One of the wonderful things about BASIC 8 is the ability to draw three dimensional graphics. How can you do this on a two dimensional screen? Basic 8 introduces the element of perspective into the drawing to give you a distinct impression of distance.

For a simple review, the elements that make up the three dimensions on the screen are:

Length - How long or wide an item appears
Height - How tall or high an item appears
Depth - How thick or deep an item appears

You are familiar with the X (length across) and Y (height up and down) coordinates of the screen. The third dimension is referred to as the Z (depth in-out of the screen) coordinate. The effect is that of a picture shrinking into, or expanding beyond the screen boundaries.

Perspective

The main ingredient of the third dimension is the vanishing point. As an object is drawn closer to this point it becomes smaller and smaller until it is but a single point. Conversely, as it is drawn further away it gets larger and can be made to appear as if you are traveling right through it! This can allow for some interesting animations. A line drawn towards the vanishing point will follow the Z axis.

In Basic 8 the location of the vanishing point can be set by the programmer. This point literally becomes the center of your graphic universe. The points you specify can even be off the screen! The one thing to keep in mind is that when Z=0 that point is considered to be on the same plane as the monitor screen. Negative coordinates lead out of the screen while positive numbers sink in.

The wondrous command that makes this all possible is @ORIGIN. The syntax for the @ORIGIN command is:

@ORIGIN, Center X,Center Y,Center Z,Vanish X,Vanish Y,Vanish Z

Lets take a look at some examples.

```
@ORIGIN,320,100,0,320,100,0
```

Structured this way, the command sets the center of rotation and the vanishing point to the center of the screen. An object drawn and rotated around these coordinates will remain stationary on the screen as it rotates without distortion from the Z axis. If the object is rendered with its center drawn outside of these coordinates, the object, when rotated, will travel around the specified center. Depending on the coordinates supplied, the object may be drawn

off the screen out of view. However, if well planned out, you could obtain an effect such as that of a moon or spaceship orbiting a planet.

@ORIGIN,320,100,0,640,0,640

As structured, the center of rotation is set at X=320, Y=100, Z=0. The vanishing point is set at the upper right hand corner of the screen at X=640, Y=0, and Z=640 pixels "into" the screen. When you draw graphics, the nearer to this number, the smaller your object will appear. If the object is rotated, it will still rotate around its center but will appear offset towards the vanishing point.

For added depth you can even place the center of rotation and the vanishing point off the screen! Just remember that if you want to rotate your object, it will rotate around that center, and your object will eventually disappear from view. Unless this is the desired result, once the program begins you will have to abort it (or wait until the drawing ends). By using the TRAP command within your program, you would be able to exit this type of problem by pressing the Run/Stop key. Trapping errors is a valuable aid when programming animated objects. You will find that all of the demos have this handy feature at the beginning of the program.

What about using a straight drawing command like @LINE? A good example of perspective can be shown by drawing just two lines! On the demo disk there is a program called B8.HEADER. This is a program module that sets up the Basic 8 environment and was discussed back in Chapter 2. I use a file such as this when I begin to write all of my Basic 8 programs. It really saves time; there is no need to retype the same "setup" lines you will be using in every Basic 8 program.

Load B8.HEADER and enter these lines:

```
110 @ORIGIN,320,200,0,320,0,1000: rem new rotate/vanish points
120 @LINE,300,200,0,300,0,1000,1: rem draw left line
130 @LINE,340,200,0,340,0,1000,1: rem draw right line
140 SLEEP5: rem wait 5 seconds before ending the program
```

In Basic 7.0 this would only draw parallel lines from the bottom to the top of the screen (and only on a 40 column screen). But, because of the Z coordinate and the ability to set a vanish point, the lines appear to converge upon one another; much like the view you would get looking down a long, straight stretch of railroad tracks. That is how perspective works. Long lines in real life, such as roads, railroad tracks, and tall or long buildings, all appear to get smaller and seem to head for a point as they continue into the distance.

A special note: If you have the earlier release of Basic 8 (one purchased before December 1988) then you will see parallel lines when you run this program. The original release had a bug in the Z axis plotting function. This bug also affected other 3D commands as well. The upgraded version of Basic 8, available from Free Spirit Software, has fixed this bug. Small fixes such as this, along with several companion programs, makes this upgrade well worth purchasing.

If you were wondering about the @ORIGIN command in line 90, the last command given will supersede the previous one. As a matter of fact, you can change this command as often as you like. This is useful if you are drawing something like a city street scene. For example, you could produce a perspective picture which would appear like you are looking down two streets from the corner of a building.

Now lets try this again using the @BOX command. Just add these lines to the program:

```
150 @CLEAR,0: rem clear the screen for the new activity
160 @BOX,120,0,0,420,100,0,0,0,1: rem draw box where Z=0
170 @BOX,120,0,500,420,100,500,0,0,1:rem draw box where Z=500
180 @BOX,120,0,900,420,100,500,0,0,1: rem draw box where Z=900
190 sleep5: rem wait five seconds
```

As you can see, the only values that have been changed are the Z values in each line. The X and Y values remain the same. In 2D this would produce identical boxes; one on top of the other. With the 3D environment however, the boxes are placed along the Z axis, and as we discussed, the closer to the vanishing point, the smaller the object appears. All this is made possible with @ORIGIN.

@ORIGIN is active when the Zview element of the @DRWMODB command is set to 0. To see the change you would experience in parallel mode, set the Zview element of @DRWMODB in line 90 to 1.

Now run the program. You will note that, because there is no vanishing point, the lines remain parallel. However they do not seem to travel along the expected route at all but run off to the right hand side of the screen. This is because, when in parallel mode, the drawings are offset in the Z axis. The boxes remain the same size and also run off the screen. More on this will be discussed later when we review the differences between perspective and parallel drawing.

You should save this short program for future use. It is a visual aid that will help you become more aware of the effect @ORIGIN has on some of the Basic 8 3D commands such as @BOX, @LINE, @CIRCLE and @DOT. I can not stress enough the importance of hands-on experience when it comes to using, and becoming familiar with BASIC 8. Actively typing in a program, and modifying its commands will teach you more than any book. It is an important part of the learning process.

Rotation

Now that you have a grasp of perspective mode, it is time to discuss the additional manipulation of your graphics with the @ANGLE command! This command is used to rotate your drawing. Superb animations can be produced in this way.

With the @ANGLE command you can rotate your drawing on any one or a combination of all three axes; X, Y, and Z. If this wasn't enough, you can also rotate in one of 6 sequences: XYZ, XZY, YXZ, YZX, ZXY, ZYX. Isn't that amazing!

The sequence of rotation that you use will produce a much different result, even if you are using the same data. You are allowed to rotate by degrees (called angles in the Basic 8 manual), from -360 to 360. When you specify negative degrees, the result is that of your object rotating clockwise. Positive numbers produce a rotation counter-clockwise.

Some people have a difficult time knowing how a rotation will look. When you specify a rotation along the X axis the object will appear to flip towards you until it is upside down. When you specify Y axis rotation your object will seem to spin around and to the right like a top. The Z axis is a little different. The rendered graphic is rotated counter-clockwise. The presentation of the graphic does not change. The effect is similar to the one you would receive if you placed your graphic on the center of a record player turntable, and then (looking down on it) rotated the turntable counter-clockwise.

It is important to realize that you are drawing just one increment of a rotation each time the command is used. This individual rendering is sometimes called a "frame". To produce an animated sequence you will have to save each one of these renderings and combine them into a "movie" of the action.

Within the B8.MAINANIMATION demo program you will get to see examples of displaying an object that makes use of the vanishing point and the @ANGLE command. The exact method used to produce the animation is explained in Chapter 13.

There is also a Bonus program included on the demo disks that makes good use of these commands. Look for the program B8.3D-ANIMATOR. With this handy application you can rotate and/or reduce Print Shop graphics or any 88 x 52 size BRUS. structure. It even produces a little movie of the animation that occurs and saves the results in the form of a PICT. 3D-screen for later recall.

For now we will concentrate on the mechanics of producing an individual frame.

Two dimensional objects can be animated in three dimensional space with remarkable results. In "B8.MAINANIMATION" demo on disk, you will see several examples of this. In the beginning of the demo you see the word "ANIMATION" spinning into view. Later in the demo a satellite, and a planet grow in size as you appear to approach them.

In the case of the spinning word, not only was it drawn farther and farther away from the vanishing point in each frame, but the Y axis angle was incremented also, giving it the spin you see. The command that produced this spin was of course @ANGLE.

In the manual you received with Basic 8 there is a very nice demo that you can type in that shows the rotation of a line around the Z axis. I would encourage you to type in all the programs in the manual; not only is this a good learning experience but you can also modify the programs to see the results of changes. For the rotating line program, just move the variable AN to the X and then Y element of the command. Try applying the same variable using the various combinations of X, Y, and Z . Making simple changes to programs such as these

will increase your awareness of how altering various commands will effect the outcome of the rendered graphic. As your knowledge base increases you will be able to produce more complex and detailed graphics and animation.

Creating a Cube

Lets look at the rotation of a three dimensional rendering. Perhaps the simplest one is the cube. This solid has only eight points to keep track of. It is also easy to visualize. I determined the eight points of the example cube by laying out the values on graph paper. I also used this drawing to locate the center of the cube. This is necessary to keep the object in the same place on the screen as each frame is drawn.

If you do experience difficulty visualizing a cube, then perhaps you should make use of a model. A model is very useful, especially when you wish to produce a complex object; one that has numerous angles and planes. Having a visual reference such as this can be an enormous help, both in plotting, and as a check against the rendered drawing.

For a model of our cube, you could use a die from any board game. If you do not have access to a pair of dice, then you could make use of any rectangular object. The only difference is that a cube has sides of the same length. Any geometry book should also have examples of cubes that you can reference for this discussion or for your own projects.

Notice that the cube (or rectangle) has only eight corners where the edges intersect. These corners are the points you must locate on the screen in order to plot the cube. Also notice that there are 12 edges that connect the corners and make up the six sides. These elements all come together to give the cube its shape. Complex solids as well as curved structures are plotted in a similar manner.

Our cube will be drawn on paper so that it will be 80 units long, 80 units tall, and 80 units deep. These units can be in any scale on the paper; 1/8- and 1/4-inch increments will work fairly well. With a 1/8-inch per unit scale you would draw an object 10 inches square. On the screen the units we will use are the pixels themselves.

We will draw a cube that is 80 pixels across in the X direction. Because of the asymmetrical pixel, you must adjust the Y direction accordingly. Probably the easiest way is to evoke the @SCALE command. There are two scale sizes available to you with Basic 8. For complex structures this is the optimum way to go. However, I wish to show the effects when no scale is used.

You will have to make the appropriate conversions to obtain an initial symmetrical looking rendering. The formula to accomplish this is relatively simple. I find that making the conversions as I lay the object out, and noting them on the paper simplifies the task, and gives me ready access should I wish to make adjustments or corrections. As you will see, this is fine as long as you don't want to rotate the data.

To find the correct distance in the Y axis multiply the distance you plotted on paper by .39. For our cube the Y distance is the same as the X distance on paper. Thus our screen distance becomes 80 times .39 or 31.2 pixels long. If this number is a fraction it should always be rounded up to the next whole number (in this case 32). You will need to make the same calculation to determine the Z axis or depth of the cube. The result is also 32.

When this type of drawing is rotated you will see the benefit of using @SCALE when rendering a 3D solid. The conversion is discussed here for those who have wondered how to produce a symmetrical rendering of an object without the aid of @SCALE.

For this exercise, you should have a fresh copy of the B8.HEADER program in memory. You will be adding several lines that will result in a demo that you can save for your collection.

Our first task is to locate the center of the cube using @ORIGIN. We will be placing the cube in the upper left portion of the screen. Because we want the cube to rotate on its own axis and remain in the same spot on the screen without a visible offset along the Z axis, the vanishing point will have the same coordinates as the center of rotation.

With the "B8.HEADER" program in memory, type in the following line:

```
110 @ORIGIN,132,44,0,132,44,0
```

This places the center of the cube at X=132 pixels, Y=44 pixels and Z=0. Remember that when Z=0 it is considered on the same plane as the screen. Because we are using a cube, it is fairly simple to place the cube around this center. We will need to find a total of six coordinates in order to plot our cube. They consist of the left and right X coordinates, the top and bottom Y coordinates and the in and out Z coordinates.

Why so few? Because Basic 8 rotates the data for you. You should always endeavor to plot an object as simply as you can. Look back at the drawing you made for this cube. In its simplest form it looks like a two dimensional object; a square. The X and Y coordinates of the "front" and "back" of the cube are the same. It is the Z axis that will give your object depth!

We know that the cube is 80 pixels wide, so the X coordinates will be located 40 pixels on each side of the center X coordinate (80/2=40). Therefore, because our center X is 132, the left X coordinate will be 92 (132-40=92) and the right X coordinate will be 172 (132+40=172).

In the same fashion you can derive the upper and lower Y coordinates. The center Y of the @ORIGIN command is 44. The calculated Y height is 32. We know that the upper Y coordinate will be 28 (44-32/2=28) and the lower Y location will be at 60 (44+32/2=60).

The Z coordinate will produce some negative numbers. The depth of our cube is 32 pixels. We will therefor be plotting 16 pixels out of, and 16 pixels into the screen. Because we have set the center Z axis at the screen plane of 0, our

"out" Z coordinate will be -16 (0-16=-16). The "in" Z coordinate will be 16 (0+16=16).

We now have all the coordinates we need to draw our cube! The eight X,Y,Z data points are:

92,28,-16 - front (out) upper left corner
92,28,16 - back (in) upper left corner
172,28,-16 - front upper right corner
172,28,16 - back upper right corner
92,60,-16 - front lower left corner
92,60,16 - back lower left corner
172,60,-16 - front lower right corner
172,60,16 - back lower left corner

Now it is a simple mater of using the @LINE command to connect the points that form the 12 edges of the cube.

Add these lines to B8.HEADER:

```
150 @LINE,92,28,-16,172,28,-16,1
160 @LINE,92,60,-16,172,60,-16,1
170 @LINE,92,28,-16,92,60,-16,1
180 @LINE,172,28,-16,172,60,-16,1
190 @LINE,92,28,16,172,28,16,1
200 @LINE,92,60,16,172,60,16,1
210 @LINE,92,28,16,92,60,16,1
220 @LINE,172,28,16,172,60,16,1
230 @LINE,92,28,-16,92,28,16,1
240 @LINE,172,28,-16,172,28,16,1
250 @LINE,92,60,-16,92,60,16,1
260 @LINE,172,60,-16,172,60,16,1
```

Please use the line numbers stated. Additional lines will be added between them later in our discussion.

In order to view the rendered drawing enter this line in the program:

```
900 GET A$:IF A$="" THEN900
```

This line will wait for a key to be pressed before the program ends. It gives you as much time as you want to view the rendered drawings.

Now go ahead and run the program. You will see a box drawn in the upper left of the screen. But wait! It does not look like a cube, just a square. This is because it is a head on view. If we were to rotate the cube we would see the other sides. Lets rotate the object on both the X and Y axis. We will rotate 60 degrees on the X axis and 165 degrees on the Y axis. To do so enter this line in the program:

```
130 @ANGLE,60,165,0,0
```

Try running it again. We now have a cube! Actually we always had one; its the rotation of the rendering that produces the view you see. You can enter a variety of values to the @ANGLE command and see the results.

In Chapter 10 I stated that a drawn object looks much better if it is located in the center of the screen. I will show you how easily this can be done using the same data points and @LINE commands that you have already entered.

Just enter this line in the program:

```
120 @WINDOWOPEN,190,50,200,100,0
```

When a window is opened in BASIC 8, all drawing commands translate to this window. The X (190) and Y (50) coordinates becomes pixel coordinates X (0), Y (0). You will now see the cube drawn in the center of the screen! As a matter of fact, as we continue, you will see how you can use the window command to place your renderings in numerous locations of the screen.

As you experiment with these commands, you will begin to develop an understanding of the complexities of the Basic 8 language extension, and the three dimensional environment. Save the program you have just typed in as B8.SHORT DEMO. We will be using it again.

You have seen how perspective affects your renderings. Now we will look at parallel mode. As I said earlier you can activate parallel mode buy changing the Zview element of @DRWMODB to 1.

Parallel Mode

When you select parallel mode, @ORIGIN will no longer affect your data. There is no vanishing point, but you will still have to enter the X or Z coordinates if you want to have an offset view of your rendering. Parallel mode allows for a simple rotation in the Y axis only. You will not be able to accomplish multi-directional rotations as you can with perspective mode. Parallel mode is a simple rotation around the Y axis. If you do wish to rotate the graphic, you will be confined to the original location on the screen; the ability to rotate around a definable point is lost.

Do you recall the @LINE demo we discussed earlier? At that time I had you draw a pair of lines from bottom to top of the screen. In perspective mode the two lines came to a point at the top of the screen. When you were instructed to activate parallel mode with @DRWMODB the lines remained parallel, but were drawn off to the right of the screen. This is essentially the default direction of the Z axis. Because the ending Z coordinate was 1000, the lines were offset towards this point.

Did you also try the @BOX commands at the same time? If you did, then you noticed that the boxes remained the same size but were drawn off to the right as the Z increment increased. These examples are perhaps the best way to see how the Z axis affects a two dimensional rendering in parallel mode.

@ANGLE will also affect the rendering of a graphic in parallel mode. The effect is similar to a perspective drawing. When you rotate a cube in perspective mode the original angles you set with @ANGLE will remain at the position set. The simple rotation will occur along the Y axis only.

How can you rotate in parallel? Basic 8 has a simple command that allows you to do this. By using the @VIEW command, you can, in effect give the rendering the appearance of spinning on the Y axis. To see a demonstration of this you should load in the program you typed in earlier called B8.SHORT DEMO. You will be adding the following lines:

```
120 :
140 GOTO300
270 RETURN
300 @CHAR,254,10,160,1,1,2,"perspective"
310 @WINDOWOPEN,0,50,200,100,0:rem open new window
320 FORI=0 TO 160 STEP16:rem increment x 16
330 @CLEAR,0:rem clear the window
340 @ANGLE,60,I,0,0:rem increment y by the value of i
350 GOSUB150:rem draw cube
360 SLEEP1:NEXT:rem wait 1 a second and draw next increment
370 @WINDOWCLOSE:SLEEP3:rem close window and wait 3 seconds
380 :
390 @DRWMODB,1,0,0
400 @CHAR,254,55,160,1,1,2,"parallel"
410 @WINDOWOPEN,320,50,200,100,0:rem open new window
420 FORI=0 TO 160 STEP16: rem use same increment
430 @CLEAR,0:rem clear the window
440 @VIEW,I:rem increment view angle by value of i
450 GOSUB150:rem draw cube
460 SLEEP1:NEXT:rem wait 1 second draw next increment
470 @WINDOWCLOSE:rem close window
480 @DRWMODB,0,0,0:rem reset command
910  IFA$="r"ORIFA$="R"THEN@CLEAR,0:GOTO100:rem if r is pressed the program will
be repeated
```

This program should be saved as B8.CUBE DEMO.

Lines 300-370 draw a cube in rotational increments of 16 degrees per frame. Lines 390-470 show the rotation when parallel mode is used. In each case the Y axis is the rotational axis. In both subroutines, a loop is used to increment the rotational values of the various commands.

As each graphic is rendered you will notice a distortion in the symmetry of the cube. This is because we are not using @SCALE. As the graphic is rotated, the Z element appears shorter as it is translated to the two dimensional screen. This distortion is less apparent in parallel mode due to the way it is presented. It is a change based on how the object looks as you move around the origin, rather than how the object looks as it rotates around its origin.

If you were to deactivate parallel mode in line 390, you would produce identical renderings of the cube. @VIEW will only work when parallel mode is activated.

Try substituting the variable I for the X or Z elements in the @ANGLE command in line 340. Notice how these changes affect the rotation. Rotation along the X and Z axis is not possible in parallel mode. If you wish to present an X or Z view it must be done using @ANGLE before you render it with @VIEW.

I encourage you to modify this program in as many ways as you can. The best way to understand how these commands work is by studying the numerous variations possible. Eventually you will know which mode of rendering 3D graphics will work for your particular situation. I cannot stress enough, the importance of hands-on experience in developing your understanding of the Basic 8 3D graphics system.

Suppose you want to produce an accurately proportioned rotation of the cube. This is possible by the use of @SCALE. For most applications @SCALE,1 is sufficient. You will have to replot your data based on the new scale values. With @SCALE,1 the 640X200 pixel screen is recalibrated to represent a logical screen that is 640X512! Using this scale, you do not experience the distortion caused by the asymmetrical pixel as an object is rendered.

You don't have to make allowances for the pixels before you generate your data. Basic 8 takes care of this for you. If you wish to produce a cube with 80 units to a side, that is what you use. Length X=80 and height Y=80 will produce a box that is proportional. If Z=80 then you have a cube.

To render the cube in the center of the screen you would use:

```
@ORIGIN,320,256,0,320,256,0.
```

If you had set your origin to:

```
@ORIGIN,320,100,0,320,100,0
```

before @SCALE,1 was used, don't worry. Basic 8 will make the necessary adjustments for you. You must,however, remember to place your data around the new origins if you wish to keep the object in the center of the screen.

There is an anomaly that I must report. When using @SCALE,1 I have experienced difficulty in keeping an object stationary on the screen as it is rotated. When I set the origin to the center of the screen and then plot the data around the coordinates, all seems fine until I try to rotate the object on the X or Z axis; the Y axis does not seem to be affected. When I attempt to rotate on the X axis, the cube is rendered closer and closer to the top of the screen. The same type of anomaly occurs with the Z axis.

The error seems to occur only when @SCALE is used. It may not occur with your copy of the BASIC 8 operating system. If it does occur, Try making this adjustment: I have found that you must set the Y origin and the Y vanishing point to 640 (@SCALE,1)! Just remember that you must also plot around the scaled

origin and vanish point (Y=256), if you want to keep the cube renderings in the center of the screen. I have included a demo program on disk called, B8.SCALED CUBE. When you list the program, you will see how I plotted the demo cube around the center of the logical screen (where x=320; y=256; z=0), but the @ORIGIN command shows the modified Y:

```
ORIGIN,320,640,0,320,640,0.
```

Do not get discouraged by this anomaly. It merely proves that the authors of Basic 8 are truly pushing the Commodore 128 to its limits!

To see the different rotations available with a scaled cube you are encouraged to change the values in line 340. In lines 341 and 342 there are some suggested values to use. Try these, and any other combination you may think of. Remember: Patience, practice, and program experimentation are your keys to understanding the complexities of this marvelous graphic system.

In Chapter 13, Animation Details, you will see how to "capture" each rendered frame to use in producing a "movie" of the animation!

CHAPTER TWELVE
BUFFERS AND SCREENS

This chapter addresses the important tasks of data placement within the various RAM locations within your computer. You may have already experienced the frustration of setting up what you believe to be an organized program, only to find that you are not seeing the display you desired, but strange symbols and garbage on the screen. In the next few pages I will explain how to organize the memory and screen layouts so you can minimize this occurrence. We will deal primarily with the RAM within the computer itself. Those of you who own an REU (Ram Expansion Unit) will be able to circumvent many of the problems experienced by placing your structure files in this device. The topics discussed within this section will address the organization you need when using the computer's RAM and no REU.

Basic 8 is able to address as many as 10 "banks" of RAM to store its data structures. These banks are referred to as buffers. Within the computer itself there are two buffers available; 0 and 1. The additional buffers can only be accessed if you have an REU installed. The 1700 REU with 128K of additional memory, can hold two additional buffers (#2 and #3) as RAM. If you were lucky enough to purchase the 1750 REU than you have 512K of additional memory. This provides you with 8 additional buffers of 64K RAM (#2-#9).

When you use internal banks 0 and 1 you must remember that these locations are also used by your program and the Basic 8 operating system! bank 0 is used to hold your program and the Basic 8 operating system itself. bank 1 is used to hold any variables and strings generated by your program. This reduces the available RAM considerably. When you first turn on the computer you are presented with a message that says in part, "122365 bytes free". But, if you have booted up Basic 8, there are only 100861 bytes free. 21504 bytes have been used to house the Basic 8 operating system alone! As you can see, having the external RAM expansion installed gives you access to free, undisturbed RAM. You will not even have to touch the resident RAM.

To further see the actual bytes available, just type these lines in direct mode (after the Basic 8 editor is installed):

```
PRINT FRE(0) (press return)
PRINT FRE(1) (press return)
```

These two lines are basic functions that return the number of bytes free within the two main banks of RAM. When you type FRE(0) the number returned should be 36605. This means that there is approximately 35k left to hold your program in bank 0. The FRE(1) returns 64256 or approximately 63k of variable storage in bank 1. Because the average program does not require this much storage, it is the optimum location to declare as your buffer area.

Using the FRE(n) BASIC function with the programs you are developing will keep you informed as to the memory remaining and the memory size of your program. I feel it is a much under-used command function. We all know what its

like to get an "out of memory error" on the screen. This function can help make you aware of the problem before it occurs.

Buffers

The Basic 8, @BUFFER command allows you to declare a buffer to start at any location within a specified bank. You can also declare the size of the buffer. It is unwise to declare a buffer size larger than what is available. You will probably lock up the computer, or at the very least, end up with nothing but garbage on the screen.

Try to stay within buffer 1. The only exception I make is when I wish to use @PAINT to color my graphics. Because @Paint requires a stack area to perform its task, I declare a buffer in bank 0 that is 1K in size for the work area. The syntax for this command is:

```
@BUFFER,0,57343,1024
```

It is the same one used in the Basic 8 manual within the @PAINT command definition. Using this buffer, all @PAINT commands will use this format:

```
@PAINT,X,Y,0,57343,1024
```

@PAINT also works with scaled coordinates! In this way you can easily paint your rendered 3D graphics.

As we saw earlier, bank 1 has 64256 bytes available for storage of your structures. You can't use all of this memory or there won't be room for the program strings and variables. However, it is possible to use as much as 48K for your structure storage. Obviously, if you are using large or numerous arrays, you will be further restricted, but, you can still do a lot with 32K, or even 16K of memory. 16K is the amount of memory required to store a 640X200 monochrome screen.

To estimate the size of the buffer needed you can either calculate the number of bytes in the structure, or convert the number of disk blocks the structure uses. Just remember that these are rough calculations; the actual memory requirements can be affected by other things such as a BRUS. structure that was saved as a compressed file.

Lets look at roughly calculating the size of a BRUS. or PICT. structure. Unless it is in monochrome, you will have to calculate both the bitmap and the color of the structure. For our example the brush is 24 pixels wide and 200 pixels tall. The formula for calculating the bitmap is:

```
(X/8)*Y
```

For our example the bitmap is (24/8)*200 = 600 bytes. If this is a monochrome structure then you can continue to add in the other structures you wish to store in memory. To calculate the color requirements the formula is:

```
((X/8)*y)/sizecode
```

Sizecode is the color resolution the graphic was rendered with. Monochrome screens do not need a sizecode. The code numbers for the other resolutions are:

8 X 16 - Sizecode = 16
8 X 8 - Sizecode = 8
8 X 4 - Sizecode = 4
8 X 2 - Sizecode = 2

As you can see, they are easy to remember. The sizecode is the same as the color cell Y size.

In our example the color cell is 8X2 (therefore the sizecode is also 2) which means that we need ((24/8)*200)/2 or 300 bytes for the color. Now, to determine the total number of bytes needed, add the bitmap number (600) to the color number (300) to find the answer of 900 bytes (600+300=900.)

When you estimate the amount of memory required by a structure file on disk you can estimate that 1K of memory will be used for every 4 disk blocks. Just add up the total number of blocks and divide by 4.

You can easily check your estimations later when you are developing a program. I usually place this line in the loop that loads the various structures:

```
@CHAR,254,2,96,2,3,2,str$(ad)+" "
```

Because the variable AD contains the next load address in your buffer, it will print this value on the screen. I usually erase the line when it is no longer needed. With it you can find the memory requirements of any structure you load into memory.

Once you have determined the size of your buffer. It is easily set with the @BUFFER command. For our example we will use a 16K buffer and the command will look like this:

```
@BUFFER,1,1024,16000
```

This sets aside 16K for our use.

Why start at 1024? Due to the architecture of the 128, the area of memory below 1024 is accessible by all configurations of ROM/RAM banking. It contains such things as the various pointers and stack areas that are used by the BASIC operating system. If you were to attempt to use this area your data would be corrupted and the computer would probably lock up.

You must protect this buffer area from being overwritten by the accumulation of variables the program generates. Because BASIC will return to the start of variables when it reaches capacity at the top, your structure will probably be corrupted. To prevent this from happening, you need to move the start of variables pointer to a value above the last address of your buffer.

There are two pointers that together contain the address for the start of variables. They are contained in locations 47 and 48 within the computer's memory. Did you notice that these pointers lie within the area below 1024 that was discussed earlier?

The normal value for register 47 is 0, and for register 48 it is 4. This produces the hexadecimal address of $0400. Register 47 is the low byte and 48 is the high byte of the address. This hex address ($0400) has a decimal value of 1024! Through the use of pokes, you can change the values of these registers and move the start of variables wherever you wish. The pokes should be used before you activate the Basic 8 graphics system with the @WALRUS,n command. To obtain a 16K buffer you would enter this line:

```
POKE47,0:POKE48,68:CLR
```

For a 32K buffer you would use:

```
POKE47,0:POKE48,132:CLR
```

And for a 48K buffer it would be:

```
POKE47,128:POKE48,191:CLR
```

In the case of the 48K buffer, the start of variables moves up to address 49024! Register 47 holds the lo bite value of 128 (which equals hex $80). Register 48 holds the high bite of 191 (which equals hex $BF). Together these produce the address of hex $BF80. Converting this address to decimal produces the number 49024. Remember to subtract the value of the untouchable area ending at 1024. This gives a total usable buffer of (49024-1024) = 48000 bytes.

Once you have protected the buffer in memory, you can use it to store your structures. I recommend using the structure numbering system outlined in the Basic 8 manual. Load in the FONTS first, then the LOGO., PATR., and BRUS. structures in that order. This buffer can also be used to @STASH a structure generated in a program.

Advanced users can further utilize the buffers by splitting them into several addressable locations. This becomes useful when you want to overwrite sections of the buffer as you @STASH, @LSTRUCT, or otherwise process various structures through the buffer. This becomes most useful when you are saving BRUS. areas of the screen to memory for temporary storage. Without partitioning the buffer, you will quickly run out of room.

This partitioning is accomplished with the variable AD. This is the variable Basic 8 uses to locate the next address in memory with the @SEND command. If you dimension an array with this variable you can produce as many elements as needed. For instance: DIM AD(3) will produce three areas where you can store structures. AD(1) should be used for FONT structures. The initial value of AD(1) would be 1024. AD(2) could store PATR. or BRUS. structures. The initial address of AD(2) will have to be calculated based on the last address needed for FONT structures. AD(3) could be used to hold transient BRUS. or other structures.

How do you get these addresses? Remember this command we discussed earlier?

@CHAR,254,2,96,2,3,2,STR$(AD)+" "

When you develop programs you always need tools to help you along. This simple line is one of my favorites. I use it to determine all the addresses for every structure in the buffer. In this way, after all the FONT. structures are loaded, I have the address for the next partition on the screen ready to record. When I no longer need the information, the line is deleted from the program. (Actually I usually leave it in a REM statement should I wish to upgrade or make other changes to the program).

If you do have an REU, most of these problems are gone. You do not have to worry about the program variables or overwriting memory. If you use LOGO. structures you will still need to declare a buffer in bank 1. The LOGO. structure in Basic 8 gets its string data from bank 1, via the @CHAR command. The LOGO. structure is a method of calling up text on the screen that is often used (such as for menus). It requires much less memory than a BRUS. structure containing the same character information. Be sure and run the B8.LOGO demo to see a fine example of its use.

Now that you have an understanding of buffer layout within the computers memory, lets look at an entirely separate area of RAM.

Screens

Within every 128D there is 64K of RAM dedicated for use by the 8463 VDC (Video Display Chip). This is usually referred to as screen memory. You can use a part of this memory to hold PICT. or BRUS. structures for viewing. It can be used for one giant 1280 X 400 monochrome screen that you can view with help from the @SCROLL command.

The older "flat" 128 had only 16K of video RAM and so could only display a limited size of color screen. Unless you have the upgraded RAM chips installed, you will not be able to take advantage of the full graphics power of Basic 8.

Basic 8 allows you to access this RAM in several ways. You can use one of four groups of predefined screens. These are accessed with the @MODE command. A convenient listing of the available screens is provided within the Basic 8 manual. This is perhaps the easiest method of using the available RAM. Various combinations of view and drawing screens can be called up within each of the 4 modes. The draw and view screen is specified with the @SCREEN command. If no number is specified for the view screen then the draw screen also becomes the view screen. This ability to draw on one screen while viewing another is the basis for double buffered displays. This display mode will be further explained in Chapter 13, "Animation details".

If the screens available with @MODE do not fit your requirements, you can easily program your own with the @SCRDEF command. Issuing this command will override any @MODE command that may have been called earlier.

With @SCRDEF you can define a screen or several screens to meet your needs. You can have a screen as wide as 2040 pixels or as long as 819. The only other requirements is that it fit within the 64K of RAM. Also, any screen that exceeds 640 pixels across can only be a monochrome screen. Screens of different color resolutions will not work together on the view screen. An 8X4 color BRUS. placed on an 8X8 screen will produce color distortion. The bitmap image remains the same, it is the colors that don't fit in. Monochrome graphics will work when displayed to a screen of any color resolution. Because a monochrome screen is basically a bitmap image with no color information, it has nothing to distort the screen. Just remember to use structures of the same resolution as the screen that you are viewing.

The method of calculating the amount of video RAM the screen will require is well documented in the manual. It is almost identical to the way we determined the amount of buffer memory discussed earlier.

It is important that you carefully plan the various screens you may require. Your first consideration is the size and resolution of your main view screen. I tend to opt for an 8x2 color resolution for my main screen. 8X4 or 8X8 are also viable alternatives. They require much less memory than the 8X2. The B8.MAINANIMATION demo on Disk #2 has an 8X2 view screen. The major benefits of this resolution have been outlined in Chapter 10. There are several ways you can conserve memory when using the higher resolutions.

I have found that a monochrome screen can be copied to a color view screen with excellent results. By first opening a window where the monochrome screen will be copied to, and using @CLEAR,0,bc,fc, you can specify the colors that the monochrome screen will take on when it is placed on the view screen. A good example of this use might be for office building corridors within an adventure game. The corridor could be up to 2040 pixels long (255 columns). By declaring the color of the foreground in your window to be blue, any monochrome bitmap copied to this area will be blue. Then, by using the @WINDOWOPEN:@CLEAR,0,bc,fc combination, you could change the color as often as you wish. In this way you could represent the various levels of the game by giving the corridor a different color depending on which "floor" you were on. The use of monochrome screens can represent a considerable savings of VDC memory.

Another trick you can use is to make your view screen longer than 200 pixels. If you define a screen of 640 X 300 pixels, you have only the top 640X200 pixels actually within view. The other 640 X 100 pixel area makes a great work area for your program. It is out of sight and, because you can only @FETCH to the view screen, I find this a great place to put BRUS. structures containing strips of animation "frames". These can then be copied from the buffer onto the screen work area as required by the program.

By moving color BRUS. structures from the buffer to the work area, you can save the remaining screen memory for the more important graphics. This becomes particularly evident when your program requires several animations which are only used once during its execution. They can be retrieved as needed and placed on the work area for copying.

Why don't we @FETCH directly to the final location? Well, for some animation sequences this is exactly what you do. For smooth action scenes however, nothing beats the fluid movement you get when you use @COPY. This is especially true of color structures. When a BRUS. structure is @FETCHed to the screen, first the bitmap, and then the color information is displayed at the specified location. This is a rather slow process that becomes more apparent the bigger the BRUS. structure. When you use @COPY, even large areas of graphics can be displayed instantaneously. There is no wavyness or jerky movement. This makes @COPY the best method of animating graphics.

By thorough planning, and organization of your screen and buffer layout, you can store an amazing quantity of graphic structures for use in your program. I always try and plan the amount of disk access I will need within a program. For small programs this is not a problem. Sometime the amount of graphics in a complex program demands storage on disk. Unless you want to wait forever while a large program loads, you should try and access the various disk files intermittently throughout your program. It is easier to take the wait for disk access from within a program than to stare at a blank screen. Some of the initial boredom can be cured if you have a nice TITLE screen to look at while the other files load during the initial startup of the program.

By combining numerous BRUS. files of animation frames into a single PICT. file, you can save as much as 50% on disk access time over loading the files separately.

You can see how all these elements we have talked about, come together in the production of an animated sequence of simulated game play in the B8.MAINANIMATION demo on Disk #2.

CHAPTER THIRTEEN
ANIMATION DETAILS

In Chapter 11 you saw how to produce a single rotation of a cube. In this section we will discuss how to combine a sequence of single rotations into a "movie" of animated action.

As we mentioned earlier, each one of the graphic renderings can be thought of as a frame. It will take a number of these frames, displayed in rapid succession, to produce the appearance of motion. Sometimes this is referred to as page flipping. It is similar to the page flip books that were available when I was a child. They usually had a short animated scene of a rabbit hopping about or some other such action.

The type of animation you use will have a profound effect on the way the motion is perceived. It can be slow or fast paced. The number of frames you use may affect the appearance of the animation. Will your animation sequence remain in the same spot on the screen, or will it move about? Conversely, does your animation consist of a single graphic entity, or is it a larger graphic that is animated an area at a time? In this section we will discuss the details of these various types of animations.

In the demo that was prepared with this book, you will see several types of animation. There will be examples of animation using @FETCH to display computer-type lights on the screen. @COPY will be used in several ways. One example will be the animation of various frames of a satellite as it orbits a planet. The other is an example of scrolling over a larger graphic which gives the feeling of actually orbiting the earth from a space ship. There is also a simple animated rendering of an energy bomb using pointer animation! All of these methods can be used to produce fantastic animation in your own program. Its only limited by your imagination.

Lets look at one of the easiest forms of animation; double buffered displays. With double buffered animation, a graphic is rendered on the draw screen. You then @COPY this object to the view screen you defined earlier in your program.

This type of animation is quite slow. Sometimes it may take several minutes to render a complex graphic on the draw screen. Its main value is to show graphic relationships within a text program. As you read the descriptive text, the time can be used to render the graphic image. You could also render graphics on different locations of the draw screen and display them to the view screen as needed.

For more rapid action you will have to utilize one of the other forms of animation available with Basic 8.

@FETCH Animation

Lets look at animation using @FETCH. For this exercise you should have a listing of the B8.SCALED CUBE program printed out so you can follow along. To

get a program listing, make sure your printer is on and the program is in memory. In direct mode type in this line and press return:

```
OPEN4,4,7:CMD4:LIST
```

With most printers this line will work. You should check the manual that came with your printer to be sure.

The B8.SCALED CUBE program uses @FETCH to animate the rendered frames of a spinning cube. This is accomplished with the use of two separate loops.

Lines 420 through 490 render 40 frames of a scaled cube in increments of 9 degrees for a full rotation of 360 degrees. Each frame drawn is saved to buffer 1 with the @STASH command in line 460. Line 470 finds the next buffer address before rendering the next frame.

The @ANGLE command in line 440 can be modified by the user to produce different results. Lines 441 and 442 include two possible combinations for you to try. You can also try these combinations:

```
@ANGLE,15,I,60,0
@ANGLE,I,I,I,0
```

There are literally millions of rotational possibilities the cube can attain with the @ANGLE command.

Lines 550 through 580 set up a display loop for viewing the 40 brush structures created by the previous loop. Each BRUS. structure contains one frame of the incremented cube animation.

As you can see in the demo, the animation is not exactly smooth. This is the one drawback of @FETCH animation. It is most apparent when you animate a three dimensional object. With a two dimensional object however, it becomes a nice way to display changes in an area of the screen. The main animation demo included with this book uses @FETCH to produce the effect of changing lights on a space ship console. Small BRUS. structures can be @FETCHed to screen almost as fast as @COPY. You will have to experiment with your own BRUS. files to see if this will work for you.

The various BRUS. files that make up the changing lights you see within the B8.MAINANIMATION demo can be displayed with any paint program written in Basic 8. What you will see are the different color variations of the lights themselves. As each different BRUS. is @FETCHed to the screen, the lights appear to be switching on and off. Others appear to change color. This gives a very convincing effect of a busy control panel. I use this type of animation as peripheral action that gives the user a feeling that something is always going on. It is quick and simple. When placed in a subroutine, it can be called upon at any time within the program. This saves your screen memory for the larger animations.

@COPY Animation

For the fastest (and most versatile) animation available with Basic 8, you will be using @COPY to display the various frames of animation to the screen. These frames must reside in screen memory. The manner of producing these frames can be just like the process used to create the various BRUS. structures that were stored in memory in the B8.SCALED CUBE demo. The only difference is that you @COPY the rendered graphic to a separately defined screen. You then end up with row upon row of the renderings. These are usually saved as a PICT. file for later use in your program. The graphics should be spaced evenly so they may be accessed within an incremented loop subroutine.

This method of storing PICT. files of the rendered graphics is also utilized by the B8.3D ANIMATOR program on your demo disk. When you create a new animation with this program you can actually see the renderings of each rotation drawn to the screen. This screen is saved for later use with the program. By listing the program, you can not only see how the graphics are rendered, but how they are placed and retrieved from the screen. This bonus program makes a great utility when used to produce animation screens for you to load and @COPY within your own programs.

You will see several examples of animation that are created exactly like those created with B8.3D ANIMATOR. I drew a picture of a satellite and also a planet, and reduced them by having them drawn closer to the vanishing point. This produces a wonderful sense of depth of movement within the program. With B8.3D ANIMATOR you can now produce animation such as this for your own program.

Another type of animation using @COPY is to view only a portion of the copy screen at a time on the view screen. You can adapt mouse or joystick control to allow moving about on the copy screen. Because you can only see this small area being @COPYed you can get a sense of moving within a larger area. The demo program B8.COPY-2 shows an example of this type of scrolling @COPY. In the main animation demo, you will also see this type of animation (the program selects the scroll direction).

This type of animation is great for scrolling around a play screen within a game program. You also have the option of scrolling a series of animation frames around the view screen. In this manner you could have a monster, or other such creature, walking around the view screen. An animation effect need not appear in only one place on the screen. You could even combine the effects; @COPY around a view screen while you also use @COPY to animate a creature within the new screen area. This can be done with consecutive @COPY commands accessed from within the same loop. The action is so fast that, unless you have many large areas, you will get the distinct impression of smooth movement in both the character and background.

Again; experiment; use your imagination. With a little practice you will soon be creating your own animation demos. When you decide to scroll around an animation on the screen, you will have to take in to consideration the grid factor. You do not want to leave a piece of the last copied frame behind as your graphics move about. To prevent this, you should place a buffer of empty space around the graphic and within the frame you are copying. This will always be 8 pixels on either side of the frame in the X direction. The number of pixels you allow on either side in the Y direction is determined by how large the scroll

will be. The smallest empty space will be the same as the color cell size. For example, an 8X2 frame would have an empty area 2 scanlines above and below the actual graphic. You can then @COPY up or down two scanlines at a time without leaving anything behind. If you were to attempt a scroll of 4 scanlines, you would again have the problem of leaving a piece of the last copied frame behind. The solution of course is to increase the buffer accordingly. This same detail should be considered when producing the PICT. files of animation frames - don't cram them so close together that you loose this buffer area.

As with all the supplied demos, you are encouraged to not only list the programs, but to use a paint program to view the various BRUS., and PICT. files that went into the making of these demos. Both will teach you a great deal.

Pointer Animation

One type of animation that is available with Basic 8, but not readily evident is pointer animation. The pointer is the sprite-like arrow you see in many Basic 8 applications. This is the object you use to point at selected areas on the screen. The amazing thing is that Basic 8 allows you to define as many as eight pointers. This becomes useful when you think of each pointer definition as being a frame of animation. You are limited to a size of 16x8 pixels, but these can be altered also! You can specify a 16x16 pixel area. This will cause a loss of 4 of the available pointers. You can have decent animation sequences with four frames, but, you will also find that eight will produce more fluid movement. In the main animation demo, the pointers were defined to represent a ball of energy. This ball is fired from the satellite. As it travels into the distance it becomes smaller in size. Using pointers for this animation has its advantage when you realize that it does not disturb the bitmap or color image of the view screen.

Think of the possibilities when you combine the various animation techniques available with Basic 8!

CHAPTER FOURTEEN
MUSIC ENHANCEMENTS

Although this really has nothing to do with Basic 8, I feel that a chapter of this book should detail the important roles of music and sound in a program.

Probably the easiest way to see the difference that sound and music can make within a program is to turn the volume down (or off if your monitor permits) until you can no longer hear anything. Try running any program that incorporates music, first with and then without sound. Isn't it amazing how empty the program "feels" without it. Did you notice how it tends to set the mood of the graphics.

The music and sound routines used in the demo are all written in BASIC. Unfortunately, you can not have your favorite tune playing while an animated sequence is in progress. It is possible but, this requires the use of interrupts and is well beyond the intent of this book. In this section we will deal with the use of sound and music in a non-ML (machine language) environment.

I will not attempt to discuss how to write the music. You should learn to develop this on your own. Its best to start with the manual that came with your computer. Then if you wish, you could purchase a Programmers' Reference Guide to further your knowledge base. You will however, see how to use this music within your own program.

Because we are restricted by BASIC, you should place the music routines in the program whenever there is a lull in the program execution. As you will see in the B8.MAINANIMATION demo, most of the music is "played" when you are presented with a lot of text. I usually limit the length of play to the time it takes me to read the text out loud. This is just a ball park judgement—some people read much faster than others. Because of this, I almost always require the user to press a key before the program will continue.

There are two places that you should always access a music subroutine. This is at the beginning and just before, or at the end of the program. As I said earlier, it will help to set the mood of the program. Also, it does not have to be the same song. A variety of similar songs can really help to maintaining user interest. Also, it is best if the music fits together.

Of course if you have lulls in the middle of the program, be sure to include additional music, or perhaps a few measures from an already played song. This will help maintain a feeling of continuity to the program.

Many BASIC 7.0 music programs can be found at your local user group, in magazines and also on almost any Commodore BBS. The manual that came with your computer also lists several type-in programs that you can adapt for your own programs.

When you use sound commands within a program, you open up additional enhancement possibilities. The beauty of sound lies in the ability to program the duration, or length of time it will occur. Used intelligently, it can be

called just before the action occurs, and can be made to last until the animation routine ends!

This ability is due to the SID (Sound Interface Device) chip. Once it receives a command, the chip acts on its own while the program moves elsewhere.

As you can see, timing is very important when using the sound command. It takes a lot of experimentation to get it just right. You can overcome some of the tedium by using a stop watch or the second hand on a wristwatch to determine the amount of time a particular animated sequence takes. Then, you can check the adjustments to the sound command in direct mode, without having to run the program over and over. When you think you have it, check it out within the program. This saves valuable programming time.

The variety of sounds is infinite. Try these variations of the sound command:

```
SOUND1,62000,375,2,55000,1700,1,1      - sensor scan
SOUND1,6000,280,0,300,20,3,20          - spaceship take-off
SOUND1,49152,450,1,0,100,3             - spaceship landing
SOUND1,60000,240,0,32768,3000,0,2600   - ray gun
```

With a little practice, you can develop just the right sound for your program. Remember, you have three separate voices to use, alone or together!

Another enhancement that is new to the market is DigiTalker by Free Spirit Software. This program allows you to add digitized voices to your Basic 8 programs. It is not a speech synthesizer. It uses digitized blocks of letters, words and phrases. Because it was produced by the authors of Basic 8, these clip-sounds can easily be incorporated into your program. There are both masculine and feminine voices recorded. They are the digitized voices of real people, not just computerized sounds. Just think of the possibilities this program addition has to offer! Sentences such as "Press any key when ready" can greet the user of your program with this wonderful addition.

In conclusion, I hope you will gain the confidence to try your hand at producing animation with Basic 8. The information contained within this book is the result of many hours of experimentation with the Basic 8 graphic development package.

I would like to thank David Darus and Lou Wallace of Walrusoft for producing this revolutionary product. They have allowed the average user to unleash the graphics power of the 128; an ability that was not supposed to be possible! I challenge you to produce a Basic 8 animation program. The only limit is your imagination!

APPENDIX - A
BASIC 8 QUICK REFERENCE LIST

@ANGLE, X angle, Y angle, Z angle
```
@ANGLE,0,0,0
```
@ARC, Center X, Center Y, Center Z, X radius, Y radius,
Starting angle,Ending angle,Increment,Thickness,Subtend flag
```
@ARC,320,100,0,50,100,0,360,10,1,0
```
@BOX, X1, Y1, Z1, X2, Y2, Z2,Shear direction, Shear value, Thickness
```
@BOX,0,0,0,639,199,0,0,0,1
```
@BRUSHPATRN, Brush Structure #, Pattern Structure #, Buffer #, Address
```
@BRUSHPATRN, 2,1,1,0
```
@BUFFER, Buffer #, Beginning Address, Size
```
@BUFFER,1,1024,48000
```
@CBRUSH, Structure #, Reverse, Reflect, Flip
```
@CBRUSH,10,0,0,1
```
@CHAR, Structure #, Column, Row, Height, Width, Direction, "Char String"
```
@CHAR,254,0,0,1,1,2,"I Love Basic 8!"
```
@CIRCLE, Center X, Center Y, Center Z, Radius, Thickness
```
@CIRCLE,320,100,0,75,1
```
@CLEAR, Bitmap fill value
```
@CLEAR,0
```
@COLOR, Background color, Foreground color, Outline color (border)
```
@COLOR,2,15,0
```
@COPY, Source screen, Start X,Start Y, DX, DY, Destination Screen, EX, EY
```
@COPY,1,0,0,639,199,2,0,0
```
@CYLNDR, X, Y, Radius, Halflen, View
```
@CYLNDR,10,50,1
```
@DISPLAY, Screen #, Device #, Drawmode, "Filename" (,X ,Y)
```
@DISPLAY,1,8,0,"PICT.SAILBOAT"
```
@DOT, X, Y, Z
```
@DOT,320,100,0
```
@DRWMODA, Jam, Inverse video, Complement, Undraw, Pattern, Merge, Clip
```
@DRWMODA,1,0,0,0,0,0,0
```
@DRWMODB, Zview, Unplotlast, Unplotvertex
```
@DRWMODB,0,1,1
```
@FETCH, Structure #, X, Y, Draw Mode
```
@FETCH,1,320,100,0
```
@FLASH, X, Y, DX, DY, Number of flashes (,Fast)
```
@FLASH,0,0,639,25,50,1
```
@FONT, Charset #, Structure #
```
@FONT,1,0
```
@GROW, X Step, Y Step, Z Step
```
@GROW,0,0,1
```
@HCOPY, Sec Address, Height, Density, Rotation
```
@HCOPY,5,2,1,1
```
@LINE, X1, Y1, Z1, X2, Y2, Z2, Thickness
```
@LINE,0,100,0,639,100,0,1
```
@LOGO, Structure number
@LSTRUCT, Struct #, Device #, Buffer #, Buffer address, Filename
```
@LSTRUCT,1,8,1,0,"PICT.TITLE": @SEND
```

```
@MODE, Mode# (,Interlace flag)
@MOUSE, On/Off, Device, X, Y (,Joystick increment)
     @MOUSE,1,0,0,0 / @MOUSE,0          Mouse on / Mouse off
     X = @MOUSE,2,0 / Y = @MOUSE,2,1    Read X position / Read Y position
@ORIGIN, Center X, Center Y, Center Z, Vanish X, Vanish Y, Vanish Z
     @ORIGIN,320,100,100,200,100,200
@PAINT, X, Y, Bank#, Address, Size
     @PAINT, 100,100,0,57346,1024
@PATTERN, Structure number
     @PATTERN, 1
@PIXEL, X, Y, Mode
     A = @PIXEL, 50,50,0            A = 0 :Pixel is OFF, A = 1 :Pixel is ON
@PTR, On/Off, X, Y, Definition # (, Height)
     @PTR,0 / @PTR,1,0,0,0          Pointer OFF / Pointer ON
     @PTR,2,0,0,0                   Leave trail
@SCALE, Scale #
     @SCALE,0                       No Scaling
     @SCALE,1 / @SCALE,2            640 x 512 / 1280 x 1024 Logical screen
@SCLIP, Left , Right, Up, Down
     @SCLIP,50,50,100,75
@SCRDEF, Screen #, Display type, Color size, Size X, Size Y,
         Bitmap beg addr, Colormap addr
     @SCRDEF,0,0,1,640,346,0,27681
@SCREEN, Draw screen, View screen
@SCROLL, Direction, Number of units, Speed
     @SCROLL, 2, 10, 50
@SDAT, Parameters/ optional data
@SEND
@SPHERE, X, Y, Radius
     @SPHERE, 320,100,50
@SPOOL, X, Y, Inner radius, Outer radius, View
     @SPOOL,320,100,50,100,0
@SSTRUCT, Struct #, Device #, Filename
     @SSTRUCT,5,5,"BRUS.MYCLIP"
@STASH, Structure #, Buffer #, Buffer Address, X, Y, DX, DY, Compression
     @STASH,1,1,0,0,0,320,100,1
@STORE, Screen #, Device #, Compression flag, Filename
     @STORE,2,8,1,"PICT.65-MUSTANG"
@STRUCT, Structure #, Type, Buffer, Beginning address
     @STRUCT,1,4,1,0
@STYLE, Shade, Scale, Lighting
     @STYLE,1,1,0
@TEXT
@TOROID, X, Y, Inside radius, Outside radius, View
     @TOROID, 320,100,50,100,0
@VIEW, Angle
@WALRUS, VDC RAM TYPE
@WINDOWOPEN, X, Y, Window width, Window height, Border flag
     @WINDOWOPEN,0,0,640,100,1
@WINDOWCLOSE
@ZOOM, Structure #, Size, DestX, DestY
     @ZOOM,1,8,0,0
```

APPENDIX B
COMMON VARIABLE ASSIGNMENTS AS USED IN THIS BOOK

With each example of a Basic 8 command given in this book, several common variables are used. Since this book is designed as a tutorial, we will assume the variables are stated as they commonly are used. For example: @DOT, X, Y, Z - we will define X as a point on the horizontal axis from 0-639 since this relates to a common screen size. Since screen sizes are variable, technically the X could be a number up to 65535 (maximum size of a virtual screen). For specific information regarding maximum sizes and and specific uses for each command, please refer to your Basic 8 manual. It is assumed that every purchaser of this book is a registered Basic 8 user. There would be little point to rewrite the 200 page manual as part of this book.

X = Point on horizontal axis usually from 0-639 depending on screen size.

Y = Point on vertical axis usually from 0-199 depending on screen size.

Z = Point on Z axis (in & out) usually set to 0. Used for 3D plotting.

Angle = Number of degrees from -360 to 360

X1,Y1,Z1 = X,Y,Z location to start line.

X2,Y2,Z2 = X,Y,Z location to end line.

Center X,Center Y,Center Z = define the center of circle on the X,Y,Z axis.

DX = Distance on X axis. Use the number of columns x 8 to figure distance.

DY = Distance on Y axis. Use number of rows x color cell size to figure position.

DestX,DestY = Destination X,Y locations, used in @COPY and @ZOOM.

Radius = 1/2 the width of a circle

APPENDIX C
ABOUT THE AUTHORS

Dave Krohne:

Dave Krohne (aka "Whiz Kid") resides in Southern California where he is a computer science teacher for the local elementary school. Dave Krohne is also an avid Commodore 128 supporter who has authored several top quality 128 specific programs including Spectrum 128, News Maker 128 and Sketchpad 128. Each of these programs take full advantage of your Commodore 128 in its 80 column graphic mode.

By writing this book Mr. Krohne hopes to inspire other Basic 8 enthusiast to write additional public domain and commercial software. He believes that as long as users are willing to support the 128, the 128 will prove itself to be a practical computer for years to come.

Roger Silva:

Roger Silva resides in Charlotte, Vermont with his wife Terry and 7 year old son Jared. He is interested in Biology, Astronomy, and anything else that involves nature and the outdoors. As far as an occupation goes, he is a cabinetmaker and a woodworker.

Mr. Silva's first computer purchase was in 1985 when he bought a 128. Since the arrival of Basic 8 he has endeavoured to produce high quality art and animation with this wonderful development package. There are several public domain pictures and animation demos available on Q-LINK where he hosts the graphics chat room (known as the Starving Artist Cafe) every Sunday night. Roger's screen name is Mr Silly!

INDEX

1581..... 30,31
address..... 18,25,30,31,39,61,63-65,70,75,76
@ANGLE..... 6,39,51,52,55-58,70
@ARC..... 11,12
background..... 7,9,20,34,36,37,41,43,46,71
bank..... 17,61-63,65
bitmap..... 5,6,7,35,43,44,62,63,66,67,72
@BOX..... 11,21,51,56
BRUS..... 27,35,39,44,52,62,64-67,70-72
brush..... 7,15,20,21,30,35,36,37,43,62,70
@BUFFER..... 17,61,62,63
buffer..... 17,18,20,21,30,61-67,69,70-72
@CBRUSH..... 20,37
@CHAR..... 19,20,46,57,63,65
@CIRCLE..... 11,12,51
@CLEAR..... 6,10,36,51,57,66
@COLOR..... 6
@COPY..... 8,28,36,37,39,67,69-72
@CYLNDR..... 14
DigiTalker..... 1,18,27,29,74
@DISPLAY..... 7,8
dithering..... 36,43
@DOT..... 11,51
Drawmode..... 7,8,75
@DRWMODA..... 6,18,21,43
@DRWMODB..... 6,39,51,56,57
@FETCH..... 8,20,39,66,67,69,70
@FONT..... 20,38
FONT..... 18,20,27,44,64,65
Fonts..... 17,20
foreground..... 7,20,36,41,43,66
@GROW..... 11,12
joystick..... 23,33,46,71
@LINE..... 11,50,51,55,56
logo..... 5,17,19
@LOGO..... 19
LOGO..... 18,27,64,65
@LSTRUCT..... 18,64
@MODE..... 5,6,9,25,65,75
mouse..... 9,11,21,23,24,29,33,34,36,46,71
@MOUSE..... 9,23,24
music..... 39,73
origin..... 6,36,50,56,57,58,59
@ORIGIN..... 6,35,36,39,49-51,54,56,58,59
@PAINT..... 18,46,62
parallel..... 39,50,51,56,57,58
PATR..... 18,27,43,64,75

pattern..... 13,17,18,43
@PATTERN..... 18
perspective..... 6,14,39,49,50,51,56,57
PICT..... 7,9,19,27,35,39,45,46,52,62,65,67,71,72
@PIXEL..... 28,29
pointer..... 1,23,24,37,43,44,46,63,64,69,72
Print Shop..... 30,33,34,35,52
@PTR..... 23,24,27
RAM..... 5,6,17,33,34,41,61,63,65,66,76
resolution..... 41,43,63,66
rotation..... 34,49-59,69,70,71
Rylander..... 12,13,37
@SCALE..... 12,14,53,54,57-59,69,70,71
scale..... 12,13,14,53,58,59,62,70
scanlines..... 5,72
@SCLIP..... 13
@SCRDEF..... 9,65,66
@SCREEN..... 6,9,25,61,65
screens..... 5-7,9,11,17,20,28,35,36,63,65,66,71
scroll..... 5,6,37,41,42,47,69,71,72
@SCROLL..... 65
@SEND..... 18,64
solids..... 12-15,37,53
SOUN..... 27,74
sound..... 6,8,39,73,74
@SPHERE..... 14
@SPOOL..... 14
stack..... 13,18,30,38,62,63
@STASH..... 10,20,64,706
@STORE..... 9
@STRUCT..... 17,18,64
structure..... 7,17-21,30,50,52,53,61-67,70,71
@STYLE..... 13,14
@TEXT..... 5,6,9,25,31,32
@TOROID..... 14
vanishing point..... 49,50-52,54,56,58,71
video RAM..... 1,5,6,17,33,34,65,66
@VIEW..... 39,57,58
@WALRUS..... 5,6,64
window..... 10,25,36,56,57,66
@WINDOWCLOSE..... 10,57
@WINDOWOPEN..... 10,56,57,66
@ZOOM..... 21,33,38,43

How To Get The Most Out Of Basic 8

Basic 8 is the most powerful hi-res graphics development system ever designed for the Commodore 128. "How To Get The Most Out Of Basic 8" was written to show you how to access the many powerful features of Basic 8.

"How to Get the Most Out of Basic 8", written by Dave Krohne, aka "Whiz Kid", and Roger Silva, aka "Mr. Silly", provides in-depth explanations of many Basic 8 concepts along with helpful examples and demos. Chapters cover such important topics as graphics modes, Rylander 3D solids, user input and utilities. A multi-chapter portion of the book is dedicated to creating animations with Basic 8. This section provides the basics of good animation technique, and then goes on to give examples of how to create an animation from scratch.

Two disks are included which are filled with examples, demos and utilities that are described in the text. You can load the programs and follow along with the text, modify the programs with a little guidance from the authors and then experiment on your own.

Basic 8, an 80-column monitor, and either a Commodore 128 with 64K of video RAM or a 128D are required.