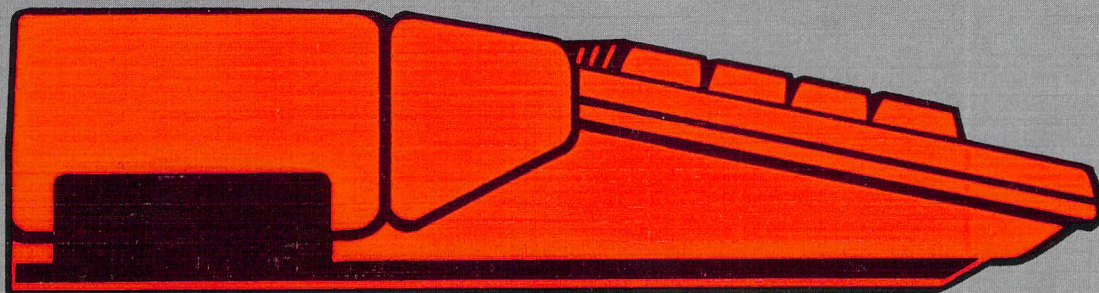
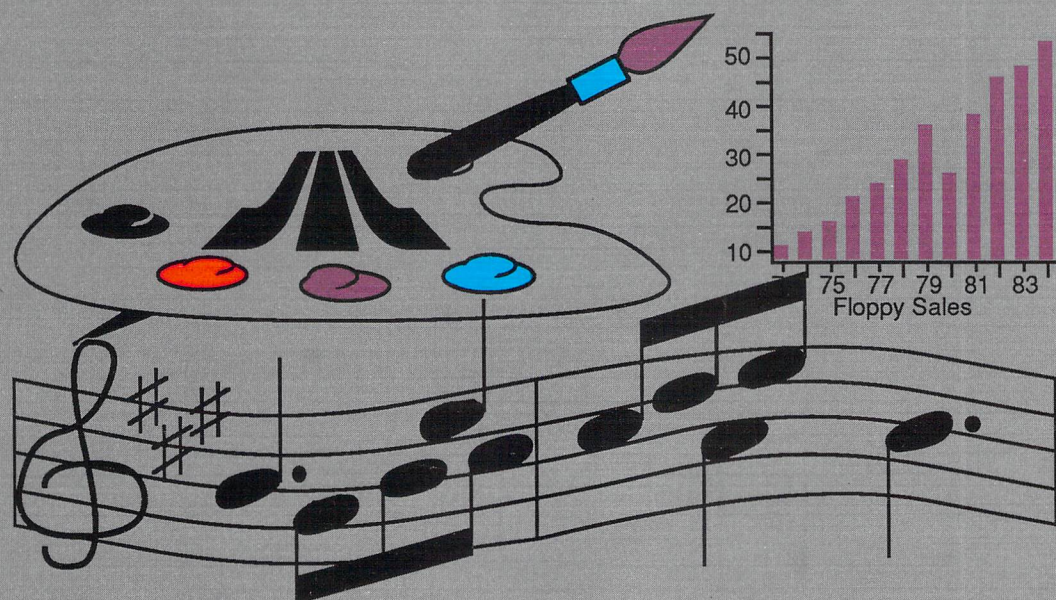


ATARI[®] ST[™]

GRAPHICS & SOUND

Master the graphics and sound capabilities of the Atari ST

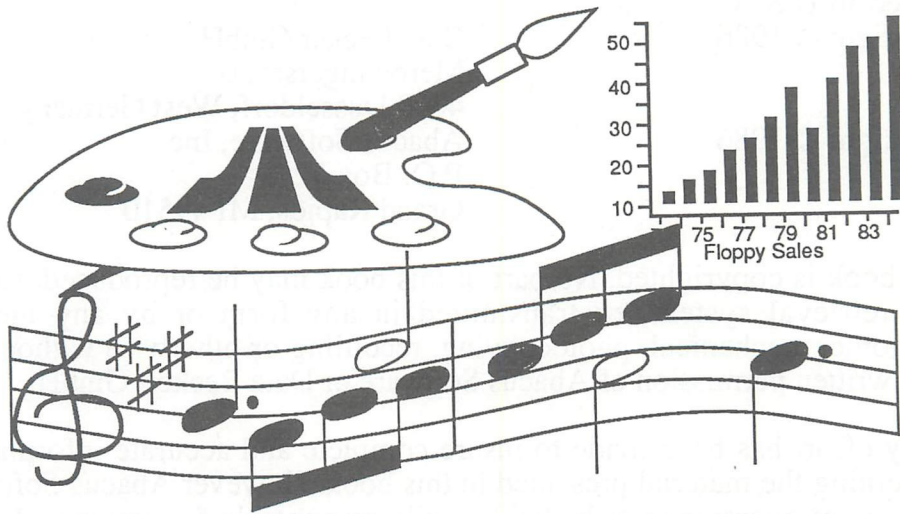


A Data Becker book published by

You Can Count On  Software

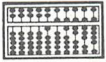
GRAPHICS & SOUND

J. Walkowiak



Data Becker Book

Published by

Abacus  Software

First Printing, May 1986
Printed in U.S.A.
Copyright © 1986

Copyright © 1986

Data Becker GmbH
Merowingerstr.30
4000 Dusseldorf, West Germany
Abacus Software, Inc.
P.O. Box 7219
Grand Rapids, MI 49510

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus Software or Data Becker, GmbH.

Every effort has been made to insure complete and accurate information concerning the material presented in this book. However Abacus Software can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors will always appreciate receiving notice of subsequent mistakes.

ATARI, 520ST, ST, TOS, ST BASIC, ST LOGO and 1st Word are trademarks or registered trademarks of Atari Corp.

GEM, GEM Draw and GEM Write are trademarks or registered trademarks of Digital Research Inc.

IBM is a registered trademark of International Business Machines.

ISBN 0-916439-33-X

Table of Contents

Preface

Chapter 1 Graphics and Sound Fundamentals 1

1.1	Introduction	3
1.2	Hardware - the limiting factor	7
1.2.1	Raster Graphics	8
1.2.2	Vector Graphics	9
1.2.3	The ST's Video RAM	9
1.3	Software - choosing the Media	15
1.3.1	The tools...	15
1.3.2	...and the computer Artists Canvas	16

Chapter 2 From Points to Pictures 17

2.1	A word about the Program Listings	19
2.2	Getting Started: Graphics with LOGO	20
2.2.1	LOGO and the Turtle	20
2.2.2	Graphics as Art	23
2.2.2.1	Moire, Symmetry, and Recursion	28
2.2.2.2	LOGO Graphic demos	37
2.3	Graphics under GEM	48
2.3.1	GEM	49
2.3.2	Using GEM Routines	52
2.3.2.1	Example 1: Initializing Open-Workstation	53
2.3.2.2	Example 2: Graphics/Text Output	55
2.3.2.3	GEM calls in BASIC	58

Chapter 3 Two-dimensional Graphics 61

3.1	Points, lines and circles	63
3.1.1	Plot point	63
3.1.2	Lines	64
3.1.3	Circles	67
3.1.4	Ellipses	72
3.2	Applications	74
3.2.1	Pie charts	74
3.2.2	Bar graphs	81
3.3	Manipulation of 2-D graphics	91
3.3.1	Shapes	91
3.3.2	Coordinate conversion	96

3.3.2.1	The coordinate system	97
3.3.2.2	Parallel shifting	100
3.3.2.3	Rotation about the origin	103
3.3.3	Matrices	104
3.3.3.1	Changing size	108
3.3.3.2	Reflections	109
3.3.3.3	Oblique graphics	111
3.3.3.4	Matrix rotation	112
3.3.3.5	Applications	114
3.4	Functions	119
3.4.1	Plotting 2-D functions	120
3.4.2	A function plotter	120
Chapter 4	3-D Graphics	131
4.1	The third dimension	133
4.2	Perspective Drawings	134
4.3	A 3-D function plotter	135
4.3.1	Eliminating hidden lines	136
4.4	3-D representation of real objects	140
4.4.1	CAD Application	146
4.4.2	Using the mouse as an input device	150
4.4.3	Real stereo graphics	155
Chapter 5	Tricks, Tips... and still more Graphics	163
5.1	Tips & Tricks	165
5.1.1	Determining the resolution	165
5.1.2	Low-, Medium- and High-Resolution	166
5.1.3	Inputting colors	166
5.1.4	Gray levels on the monochrome monitor	167
5.1.5	Storing screens	168
5.1.6	The write-mode	169
5.1.7	Filling the screen with patterns	170
5.2	Still more Graphics	172
5.2.1	Animated Graphics	172
5.2.2	Fractals	179

Chapter 6 Sound	199
6.1 Sound	201
6.2 The tone makes the music	202
6.3 The envelope	203
6.4 The synthesizer	205
6.5 The chip	206
6.5.1 The Register Array	208
6.5.1.1 The register	211
6.6 Tone generation using registers	214
6.7 A practical application	221
6.8 Sound manipulation using the envelope	223
6.9 Programming Music from BASIC	229
6.10 The sound chip under LOGO	236
6.11 What is MIDI?	238
Appendix	241
Appendix A: Program: Compute Periodic Values/Tone Generators	243
Appendix B: Program Listing: Table of Periodic Values	245
Appendix C: MIDI Resources	247
Index	251

Chapter 1

Graphics and Sound Fundamentals

1.1 Introduction

Computer graphics is the hot issue in today's hardware and software markets.

One of the prime selling points of any new computer is its graphics capabilities. Whether it's a large or small system, very few buyers are satisfied with a system that can only crunch numbers or process text.

So it's not surprising to see the hardware manufacturers in fierce competition to introduce computers with more memory, faster processing speed, higher resolution and more colors. Each new advance makes the "state of the art" computer obsolete. Today, a user-friendly keyboard is in itself not enough to sell a computer. New concepts, such as the integration of a mouse into the system for graphic-oriented operation, are paving the way.

What can all this new technology do for us?

In the profession of electronic data processing, there are many uses for computer graphics. In fact, applications undreamed of just a few years ago have been literally created by these technical innovations.

With the appropriate software, a mainframe computer can "replace" an entire airplane. A flight simulator using a computer can generate every aspect of winged flight realistically.

Flight simulator training is already so advanced that it is equivalent to training in a real (i.e. crashable) airplane. As a result, many airlines are using simulators to train their pilots thereby reducing their training costs considerably.

In the areas of natural and physical science, computers are irreplaceable. They manage the vast amounts of data needed by scientists in an incredibly short time, and provide data bases that can be accessed around the clock by colleagues. Research in fields such as plasma-physics or genetics would be simply impossible without the aid of computer systems, data bases and complex programs to process the myriad of numbers into comprehensible information.

The human senses are just not sufficient anymore to facilitate the work of today's researchers. Whereas scientists in the past were able to use their minds to visualize the concepts of their calculations, they would find it quite difficult to imagine complex gene patterns or molecules without some form of assistance.

A graphic terminal on which molecules rotate in a simulation, and can be viewed from any angle, at any size, is a required tool for modern researchers.

Of course, such graphics aren't limited to a molecular model, which resides mathematically in the memory of the computer and is reproduced on the screen. It could just as well be a car, a house, or an entire skyscraper. With a plotter available as a peripheral, an architect can concentrate on creating designs, instead of spending his time calculating, measuring, and drawing. The computer can do these tasks while the architect goes to lunch or gets a good night's sleep.

With the aid of his computer, the architect can accomplish tasks that previously took weeks to finish in a matter of hours. This is called CAD, for Computer Aided Design. As an example, look at figure 1. This drawing could have very well originated on a drawing board instead of the computer. It was drawn with a CAD program which facilitates scale drawings on the screen and then outputs the design to a plotter or printer.

In manufacturing, data generated through CAD can be fed into lathes, presses, robot sprayers, etc. equipped for automation. The products conceived by an engineer with his computer can be manufactured almost completely automatically. This whole process is referred to as Computer Aided Manufacturing (or CAD-CAM).

Thanks to their amazing graphic capabilities, these CAD computers are also being applied to more artistic applications than industry and research.

Computer graphic studios have sprung up in the U.S. and Japan. These studios can create graphics with such incredible resolution and color that the first science fiction film wholly created with computers, titled **The Works**, is nearing completion at the New York Institute of Technology.

Movies such as **Tron**, produced by Disney Studios, integrated actual film and computer graphics, whereas **The Works** will be purely digital. It would seem as though the much-used "blue screen" technique is obsolete already. Today's overpaid stars might be replaced by creative computers someday ...

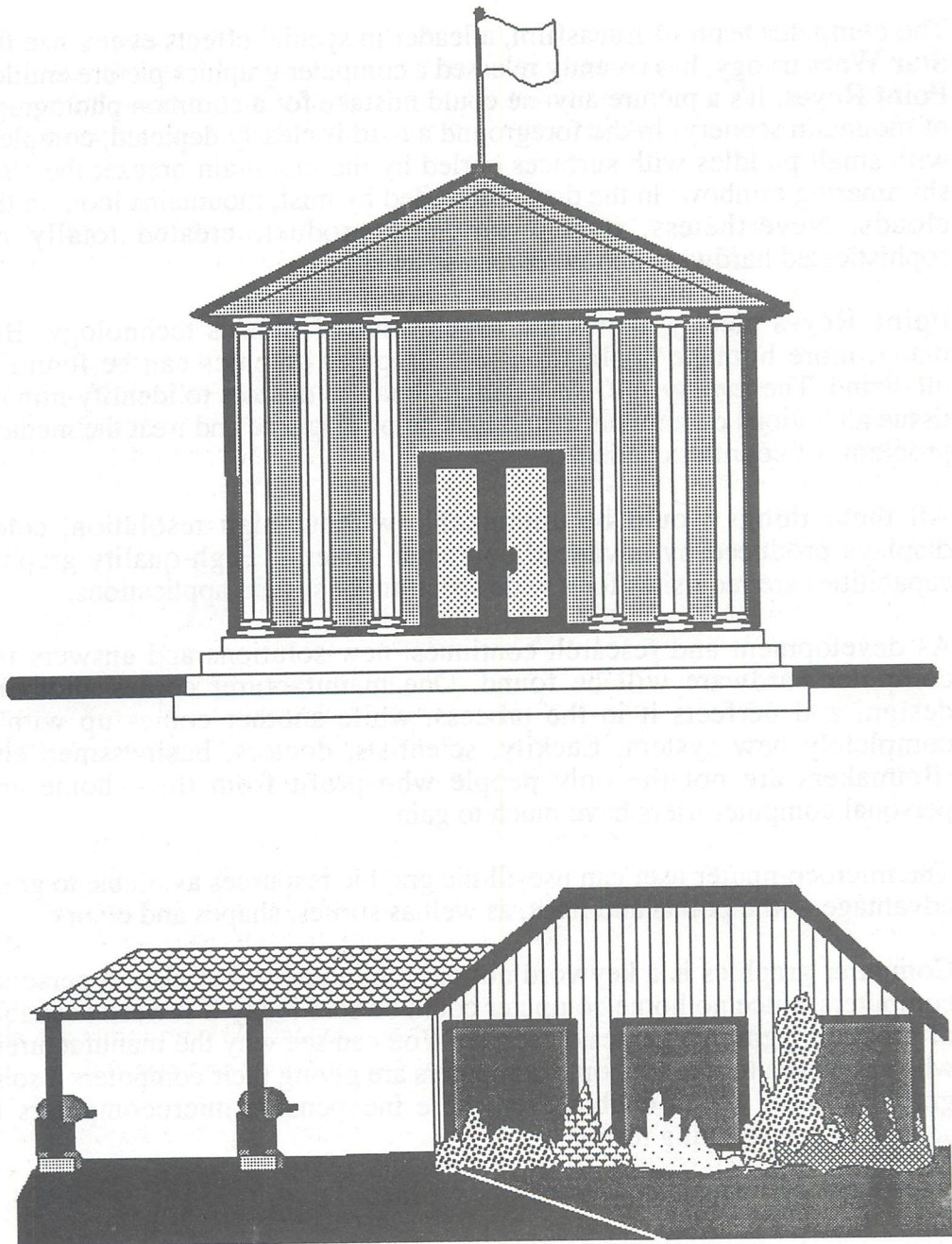


Figure 1: Sample printout of CAD designs

This example demonstrates the detail and discrimination that current computer graphics systems can provide.

The computer team of Lucasfilm, a leader in special effects ever since the *Star Wars* trilogy, has recently released a computer graphics picture entitled **Point Reyes**. It's a picture anyone could mistake for a common photograph of mountain scenery. In the foreground a road is clearly depicted, complete with small puddles with surfaces curled by the mountain breeze; there's a shimmering rainbow; in the distance, veiled by mist, mountains loom in the clouds. Nevertheless, it is a synthetic product, created totally by sophisticated hardware and ingenious software.

Point Reyes clearly shows the capabilities of today's technology. But many, more humane applications of computer graphics can be found in medicine. The *tomograph*, for example, enables doctors to identify minute tissue alterations on a monitor, and thus helps diagnose and treat the medical problems of countless patients.

All these things would be unthinkable without high resolution, color displays produced by advanced computer systems. High-quality graphic capabilities are requisites for a wide spectrum of system applications.

As development and research continues, new solutions and answers for computer hardware will be found. One manufacturer copies another's design, and perfects it in the process, while another comes up with a completely new system. Luckily, scientists, doctors, businessmen and filmmakers are not the only people who profit from this—home and personal computer users have much to gain.

The microcomputer user can use all the graphic resources available to great advantage—the points and lines, as well as sprites, shapes and colors.

Computer graphics is a keyword not only in industry, but also in personal computers, since no home computer could be sold today if it wasn't capable of producing graphic games on screen. You can see why the manufacturers who've chosen to market home computers are giving their computers a solid graphics core, which enable even these inexpensive microcomputers to achieve some stunning results.

1.2 Hardware—the limiting factor

Even though the technology incorporated in the Atari ST is not quite capable of producing realistic movies, it is possible to achieve good animations and useful, quality applications.

Good graphics require a high resolution display and a large number of colors. The Atari ST (priced at under \$1000) is obviously a far cry from the multi-million dollar Cray, which is capable of displaying one million dots on a graphics terminal. Nevertheless, the ST can print 256,000 dots (called *pixels*), which is better than most current home computers (Apple, Commodore, and IBM).

The computer has to "remember" every pixel it needs to light up, which requires a large amount of memory. And if the ST has to remember a specific color for every pixel, even more *video-RAM* is required. Of course, not everything depends on the size of video RAM, as you will soon see. The Atari ST's video RAM requires only approximately 32K of the ST's 512K (or 1024K) of memory. The designers could have accommodated even more than the 640x400 pixels available in monochrome.

If you take a closer look at different characteristics of current personal/home computers, you'll notice that manufacturers have made compromises that divide their machines into four groups:

1. The computer is capable of high resolution, but can use very few colors (usually two) in hi-res mode.
2. A multi-color picture is possible only at a lower resolution.
3. Colors may be finely shaded at high resolution.
4. Choice of colors is virtually limitless at the highest resolution.

Characteristics 1 and 2 apply to the Atari ST. It's a good compromise, making the 640x400 resolution possible in monochrome, and giving the 320x200 resolution sixteen colors. And keep in mind that the 16 colors can be chosen from a palette of 512 colors.

Even though characteristics 3 and 4 are possible in a home computer, the hardware—and consequently the price—would be completely different.

1.2.1 Raster graphics

The system we'll use most frequently is a *raster system*. To visualize raster graphics, imagine placing a fine screen over a picture. If you used a piece of mosquito netting, the picture created would be low resolution. If you use a pair of woman's nylons, the picture created would be in high resolution. If you closely examine the picture produced by the woman's nylon, you would see that it's made up out of countless tiny rectangles, each of which can be individually specified. Similarly, the ST's screen is composed of 400 horizontal lines, each containing 640 points.

Only two colors (black and white) are used in the high resolution mode. Each point of our raster is assigned a *luminance* (black or white.) In total, 256,000 points have to be defined. We designate black pixels with a 1 and white pixels with a 0, and can represent the whole picture in this manner.

This is exactly the way a picture is stored in a computer that uses raster graphics. The difference is that the computer deals not with single points, but with *bytes* (a group of 8 bits.) To store the picture described above requires exactly 32,000 bytes, or 31.25 Kbytes.

To store a picture of still higher resolution, say 1024x1024 pixels, 130 Kbytes is required. This is not a problem today—megabyte chips are going into mass production, and their prices are falling rapidly.

Instead, the problem is the amount of time it would take to draw such a picture on the screen. The *raster monitor* builds the picture line by line. It has to rapidly switch back and forth 640 times to construct a complete picture. To prevent the flickering of this picture, this process has to be repeated at least 25 to 30 times per second (the ST monochrome monitor repeats it 70 times). This ensures that a clear, steady picture is produced.

This means that, at the minimum, the computer has to read the values of 130,000 memory locations, 30 times a second. Also, the processor keeps checking the video RAM during the execution of a program, since the display must be controlled at all times. Go ahead and calculate how many read operations would have to be executed if the screen consisted of 4096x4096 pixels...

1.2.2 Vector graphics

Vector graphics are a completely different method of creating pictures. The electron beam is not repeatedly scanned horizontally across the screen, but rather is guided only where it is needed to illuminate the screen. The electron beam does not have to reach those areas on screen that don't require any changes. This saves an enormous amount of time.

1.2.3 The ST's video RAM

The Atari ST uses raster graphics. Your next question might be, "Which area of memory is reserved for screen graphics?" Answer: As long as the video RAM is in its normal address—not in a memory expansion—you can find the uses of various memory locations by writing to specific ones.

In the 520 ST, the address of the start of video RAM is at &H78000 (decimal: 491519). In the 1040 ST the start address of video RAM is &HF8000 (decimal: 1015808). Enter the following in the Command window of ST BASIC:

```
VSTART =491520: REM 105808 ON 1040 ST
POKE VSTART+9600,65535
POKE VSTART+9600,0
```

The first POKE draws a line about a third of the way down the screen. The second POKE clears the line.

Those of you who've previously worked with bit-mapped graphics might wonder why 65535 corresponds to the 8th bit of a byte (whereas it's usually 255, or &HFF in hex.) The answer to this question lies in the MC68000 processor. The processor can address bytes, although a POKE is a 16-bit command.

One line of the screen consists of 80 bytes (80 bytes x 8 bits = 640 pixels). As a result, 40 POKES fill the line (since 16 bits are set with each):

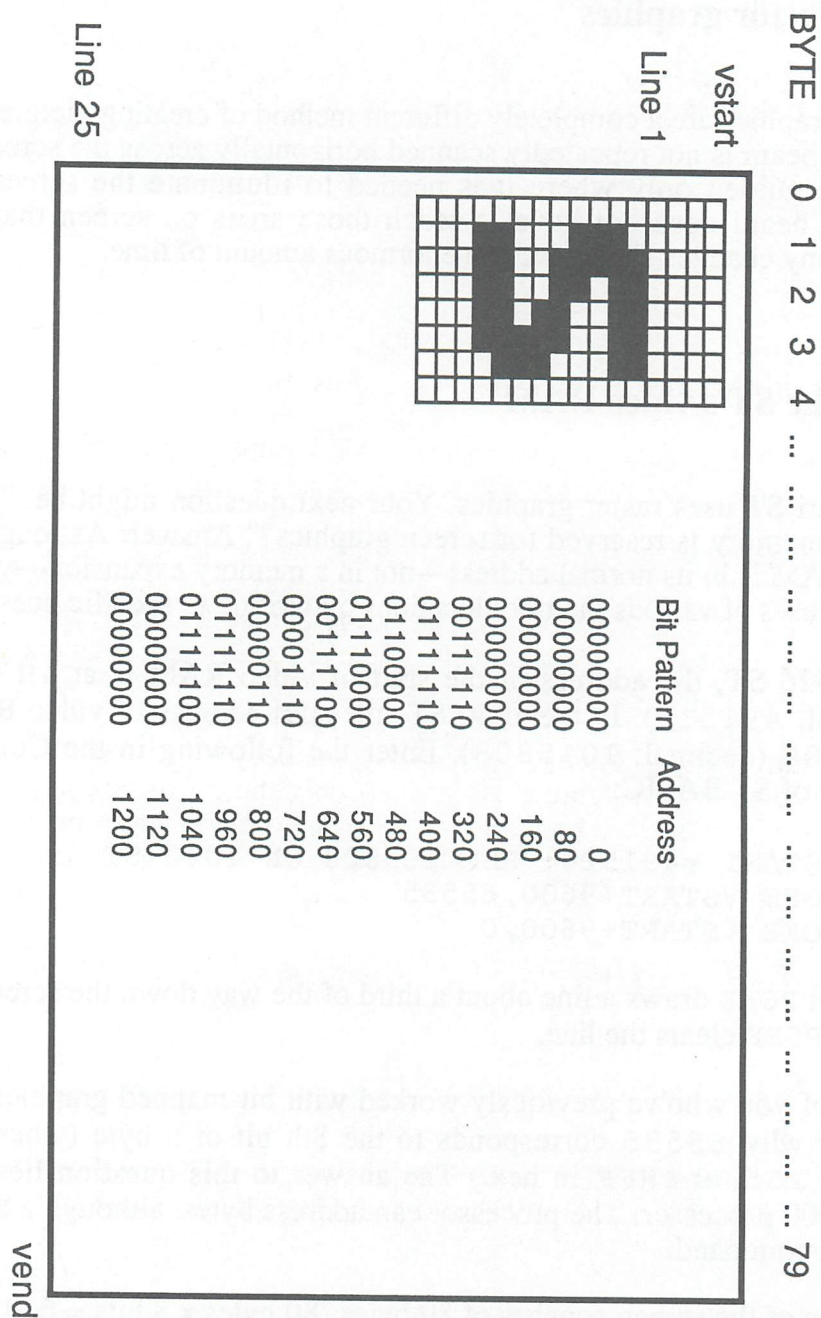


Figure 2: Screen organization at HiRes

This short program proves our point:

```
5 REM Numbers in [] are for 1040 ST
10 VSTART = 491520 +(4*640): REM [1015808]
20 FOR I=VSTART TO VSTART + 79 STEP 2
30 POKE I,65535
40 NEXT I
```

Mathematically, the video RAM ranges to the address 523519 (1047806). A quick test will prove this: POKE 523519,0 clears the point at the extreme right bottom corner of the screen. Figure 2 clarifies this relationship between points and their addresses.

The illustration also shows how the ST forms its characters. For each character, 16 of the 640 rows and 8 columns are used.

The beginning of the individual raster lines are separated by 80 bytes. The addresses where the bit-patterns are written (the patterns that create the individual lines of the letter "s") therefore can be found as specified. (See figure 2). With this method, we can make any character, at any size, appear at any position on the screen.

Be certain that you always address two memory locations. This is a good reason to get in the habit of addressing only the even-numbered memory locations. It's a great habit if you're ever going to program in machine language. Once you get used to this, you'll be able to refer to any desired pixel position.

The following large numbers might be a little difficult to get used to:

[illegible]

The values between 1 and &HFFFF have to be POKED in order to set the desired bit combination within a word.

It should be mentioned that the ST's video RAM does not necessarily have to be positioned at these memory locations. Each time the system is booted, the computer ensures that the last 32 Kbytes of free memory are always reserved for the video RAM. (The exception is the few bytes reserved by the MMU in the area between FF 0000 to FF FFFF. The MMU uses them as an I/O area to control the peripherals.)

When a color monitor is used, things aren't quite as simple—but still quite logical and consistent. The color mode lacks the "monochrome monitor detect" signal sent out by a monochrome monitor, which causes the operating system to switch to the color mode. The resolution is now restricted to 640x200 pixels (or rather 320x200). The slides in the desktop accessory's color control panel are also activated.

Color adjustment is quite simple. From the two bottom lines of the color control panel, you can choose a pen, and then mix the desired color from red, green, and blue.

The ST gives a choice of sixteen pens in low resolution. This lets each point on the screen take on sixteen different conditions. In this mode, 2^4 (two to the fourth power) bits of a byte are lost per point in the video RAM. Since the size of the video RAM must remain constant at 32 Kbytes, only 64,000 bits remain to form individual points on the screen.

Now one byte contains not eight points in one color, but two points in sixteen colors (2x4 bit). Figures 3 and 4 illustrate how the ST organizes its video RAM in either of the two color modes.

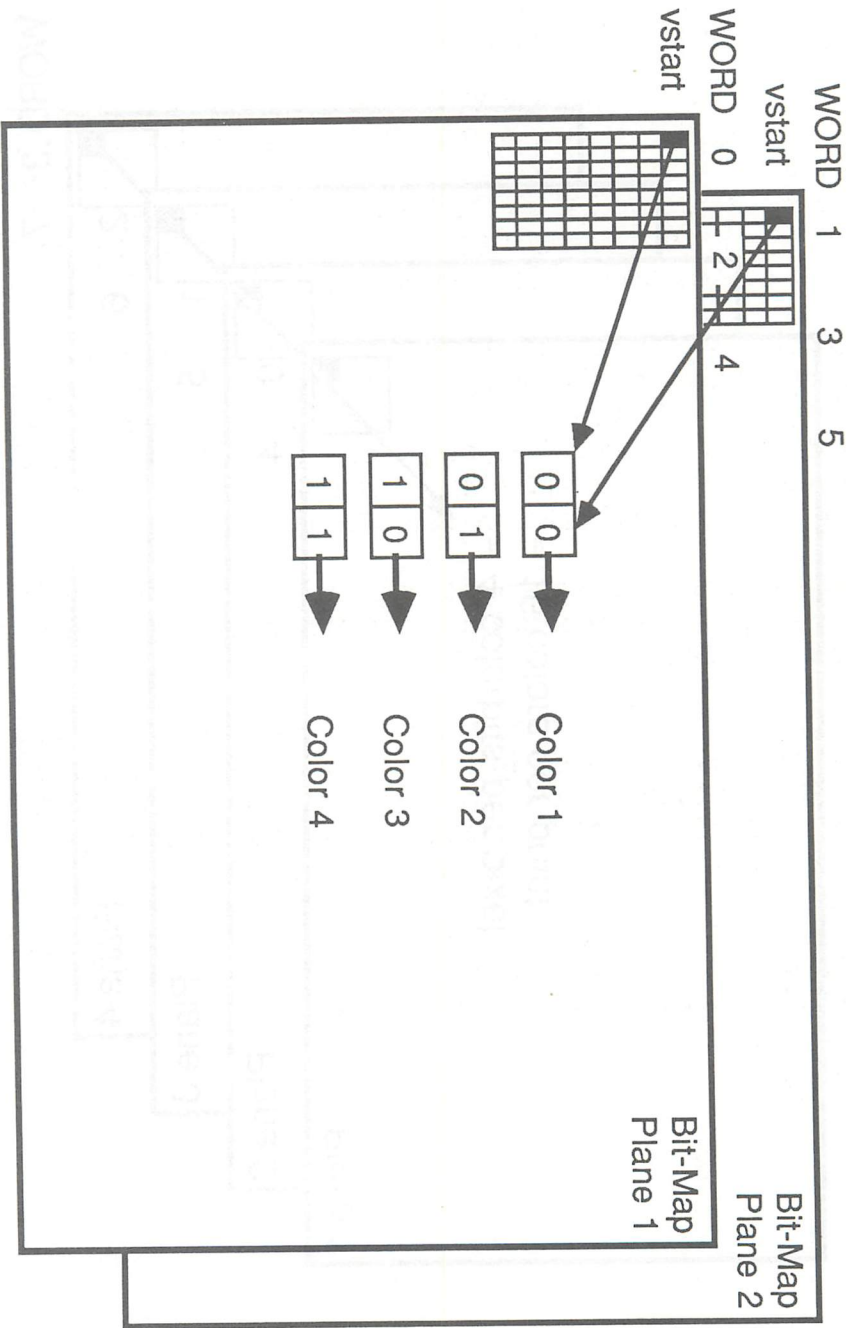


Figure 3: Screen organization at Medium Res

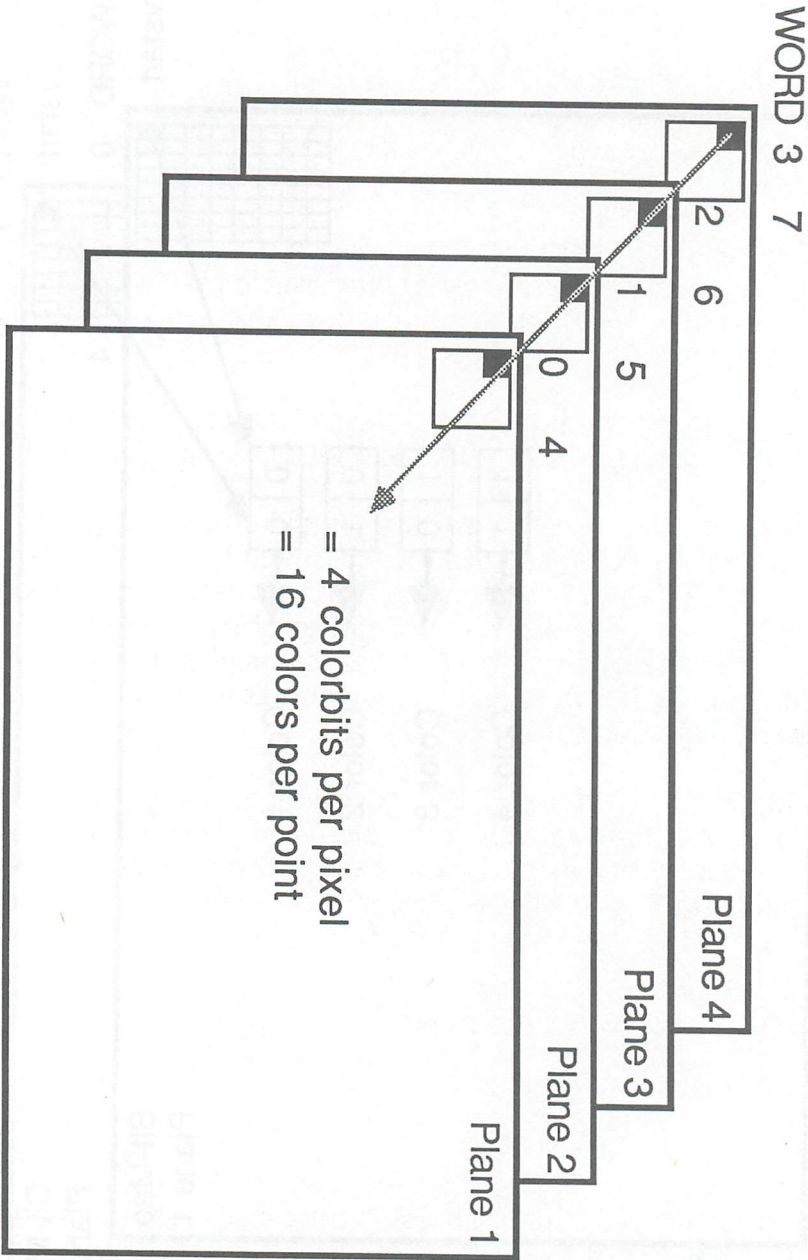


Figure 4: Screen organization at LowRes

1.3 Software—choosing the media

Most of us have tried to express ourselves as artists, whether in kindergarten with fingerpaints and watercolors, at school with pen and ink, or at home with oil paints and canvas.

All these graphic art forms have one thing in common: we first choose a media, the appropriate tools, chose a comfortable easel or workplace, and select our favorite colors from the choices available. Only after this does the actual work begin.

The Atari ST gives us the roughly the same choices. For instance, which tools (language) should I use—BASIC, LOGO, C, or assembler?

Each of these languages has its advantages and disadvantages. BASIC has been a standard for years, and has been mastered by more people than any other language. Graphic programs in LOGO are easily understood, thanks to the turtle. C is geared for speed and is often used by professional software writers, partly because its code is so complex it serves as a form of protection. This applies to assembler programs as well, but these are quite a bit faster and require less memory.

The hardware used with the ST is important, too. If you plan to do a lot of word processing, use spreadsheets, etc., you'll probably want a high resolution monochrome monitor. If you are creating computer art or want to play arcade games, you'll probably want to use a color monitor for best effect.

1.3.1 The tools...

The computer lets you set single points on the screen, as we have explained earlier in the chapter. However, if you wanted to construct a more complex picture on the screen of the ST from scratch, the work would be so tedious and painstaking that you'd probably return to your pencil and paper.

This is why software designers have developed techniques that allow you to create lines, circles, and whole shapes in one easy step. In addition, most programs incorporate simple coloring and shading routines. Thus our ST has commands such as `PEN` and `COLOR`. In Chapter 5, **Tricks, tips... and more graphics**, you'll find many examples of these kinds of commands.

1.3.2 ...and the computer artist's canvas

The computer equivalent of canvas can take two different forms. The first approach is to make the drawing area any desired size, with only a portion of the area displayed on the screen (e.g. `LOGO`). The second approach is to define the drawing area size and access this area using a coordinate system (e.g. `BASIC`). A coordinate system can be incorporated into the first system as well, but because of the absence of limiting values, this method is more difficult. Such systems very seldom operate with coordinates. Vectors are often used instead. Vectors are specified by distance and direction in reference to a given point.

This reference point corresponds to the position of the turtle (in `LOGO`), which is usually visible on the screen at all times. This method is quite simple, since no coordinates have to be specified or transformed. Simple inputs are sufficient, like `go 40 units forward`, or `turn 90 degrees to the left`.

Chapter 2

From points to pictures

2.1 A word about the program listings

The program listings in this book were all transferred from the ST directly to the computer this book was printed with using KERMIT. This method was used to minimize the possible errors in the publishing process, and assure that the book contains accurate program listings.

Note that some of the routines listed in this book appear in the text several times. It's not necessary to re-enter these routines over and over again. However, do not use the BASIC RENUMBER function. Instead, use the MERGE function to combine the various modules of the program.

If you're programming in Modula-2 or C, we highly recommend using the **ST TextPro** wordprocessor, available from Abacus. It has a C-source language option that automatically formats the structure of the program. **ST TextPro** also displays the program text's line number, which may help in the debugging of your C programs.

2.2 Getting started: graphics with LOGO

Before we start working with computer graphics and deal with problems like rotations, reflections, or scaling, we want to mention some fundamental computer graphics concepts and terms. Even if the following material is familiar to you, you should read through it for review.

We'll be moving quite rapidly from simple programs to more complex ones. As a result, we emphasize some details now which are vital for later chapters.

2.2.1 LOGO and the turtle

In the mid-Seventies, LOGO designer Seymour Pappert attracted much attention with his success in teaching children to use computers. His assumption was that the electronic representation of a tangible object would greatly simplify our interaction with computers.

During his ambitious experiments, Pappert placed a child in the middle of a room of children. The other children would give verbal directions to this child asking him/her to "draw" mathematical figures with his/her footsteps. The directions for a square might "sound" like this:

```
go five paces forward
turn 90 degrees right
go five paces forward
turn 90 degrees right
go five paces forward
turn 90 degrees right
go five paces forward
```

The child marked his path with a piece of chalk, and the children could discuss the results and speculate what other "games" might be possible. The children were learning geometry in a completely new manner.

The next step was to replace the room with a computer's video screen, and the child with a graphically displayed turtle.

In this way, children were able to simulate processes from their real, tangible world on the computer. Children naturally like to draw, so they looked on the computer as a convenient drawing device.

Every child in the experiment was able to draw on the computer the picture of the square walked out by their friend:

```
TO SQUARE  
FORWARD 50  
RIGHT 90  
FORWARD 50  
RIGHT 90  
FORWARD 50  
RIGHT 90  
FORWARD 50  
RIGHT 90  
END
```

Enter this simple LOGO program on your ST. This routine alone can be the foundation for complex graphics. To simplify things, you can use the abbreviations `fd` for forward and `rt` for right.

Now select the `Save as...` option from the File menu, and name the program `SQUARE`. To run it, just type `SQUARE` in the LOGO DIALOGUE window. As you'll see, the turtle is at its home position. This home position can be reached at any time with the command `HOME`.

To rotate the square a specific number of degrees, the turtle has to be instructed accordingly—for example, `rt 10`. Entering `SQUARE` then executes the command.

Once you've done this a few times, you'll have created your very first computer graphics.

We don't have to draw these out by hand each time, but instead write a short program (which calls up `SQUARE` as a sub-program):

```
TO GRAPHIC 1
SQUARE
RT 5
GRAPHIC 1
END
```

Try writing the equivalent of this program in BASIC. You'll see that it is much more difficult to accomplish the same thing in BASIC, whereas a few lines do the job here. Simply changing the length of the sides or the rotation angle will produce a totally different picture. Try the same thing with different figures (triangles, or letters and numbers).

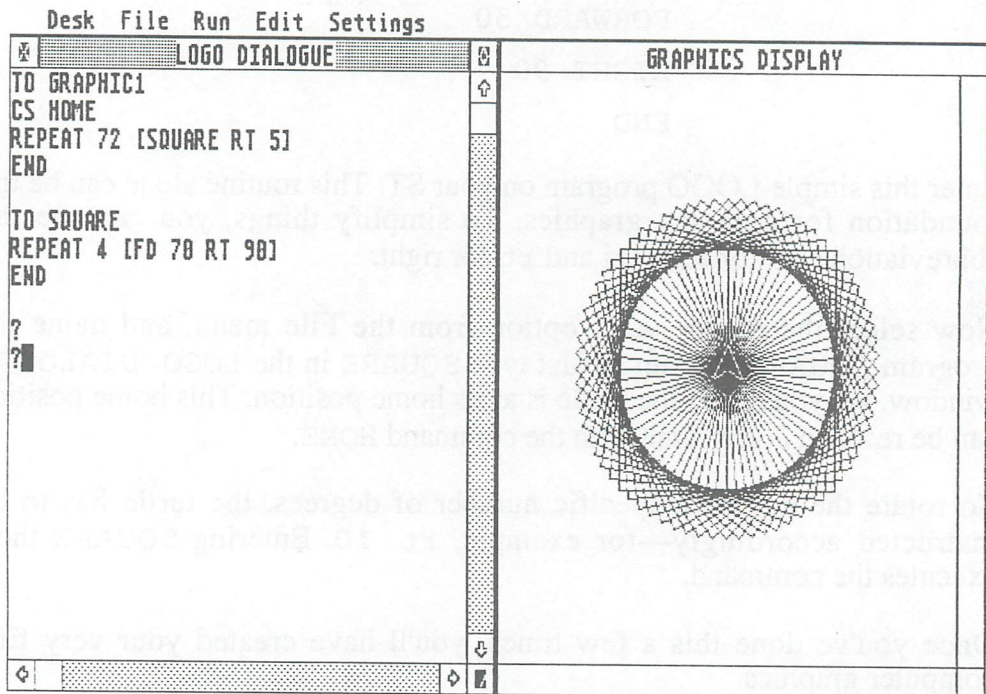


Figure 5: Graphics programs in LOGO—short, but effective

2.2.2 Graphics as art

This section deals with an area of computer graphics known as video art. You'll learn a few things about computer graphics that can be used to make exciting, creative graphics.

One method of creating computer art is to enlarge or condense a single shape, and use the variations in size to create the whole picture. You can create bewildering graphics that can be interpreted any number of ways.

Two people looking at Figure 6, for example, might disagree on whether the square is receding or protruding from the screen.

When both of these techniques are combined, the illusion of depth can be created. A careful choice of parameters can produce some very realistic pictures. For example, take a look at the GRAPHIC 2 program, Figure 7.

The program GRAPHIC 3 simply prints a row of squares. Nonetheless, the end effect is quite different from the original picture.

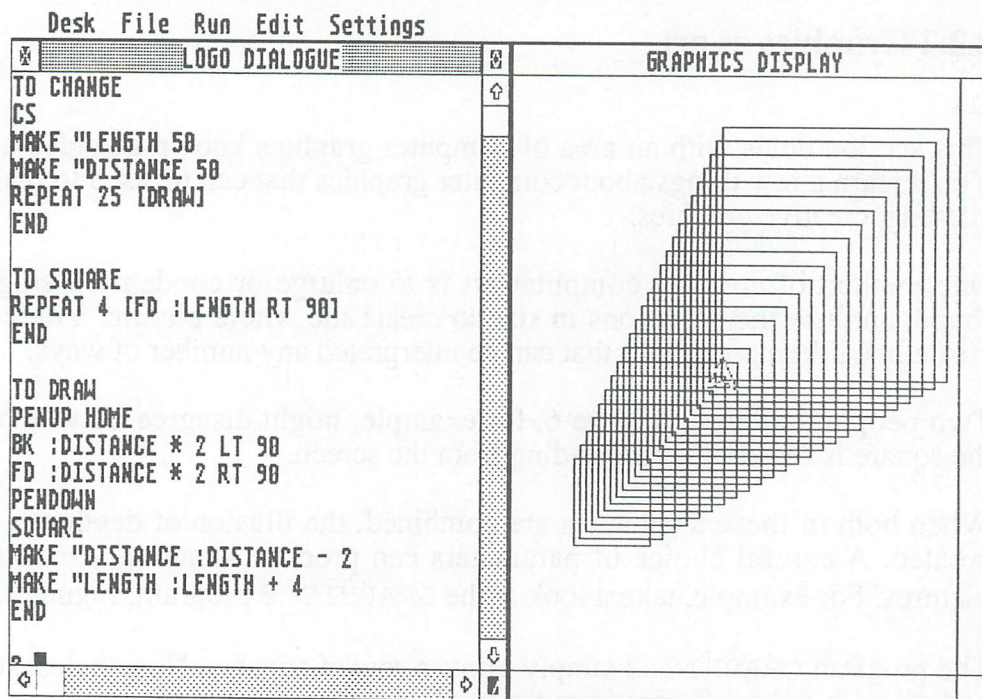


Figure 6: LOGO program ILLUSION

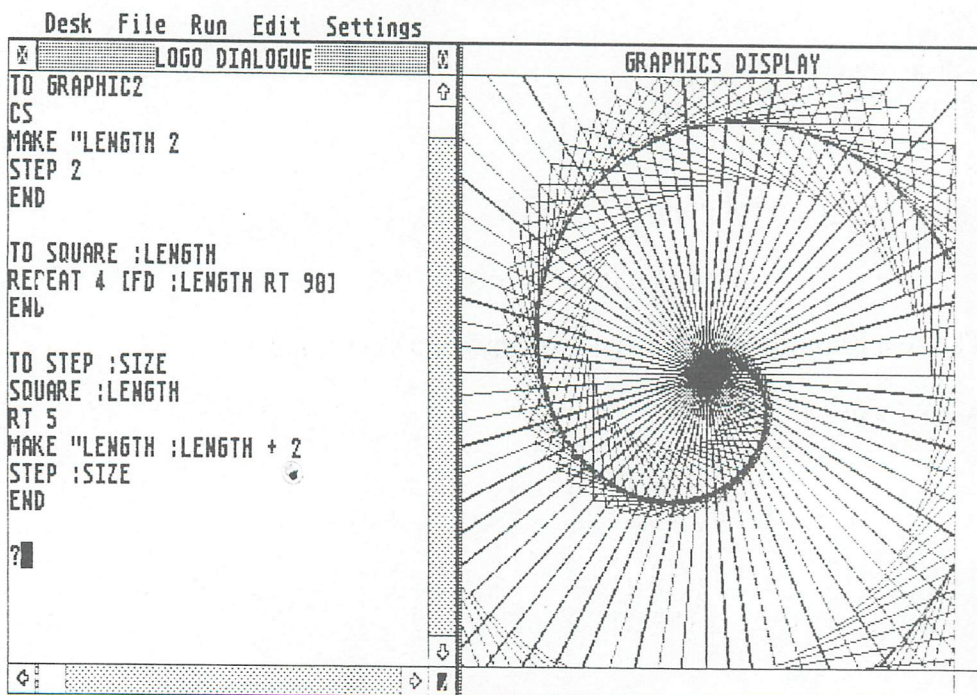


Figure 7: LOGO program GRAPHIC2

```
TO SQUARE
REPEAT 4 [FD :LENGTH RT 90]
END
```

```
TO START
PU HOME
BK :DISTANCE * 2 LT :ANGLE - :INCANGLE
FD :DISTANCE * 2 RT :ANGLE + :INCANGLE
MAKE "ANGLE :ANGLE - :INCANGLE
PD
SQUARE
MAKE "DISTANCE :DISTANCE - 2
MAKE "LENGTH :LENGTH + :STEP
START
END
```

```
TO GRAPHIC3 :STEP :DISTANCE :INCANGLE
CS FS
MAKE "ANGLE 90
MAKE "LENGTH 0
START
END
```

```
MAKE "ANGLE -85
MAKE "LENGTH 170
```

Try entering the following command:

```
GRAPHIC3 5 -20 240
```

The resulting display is...the top view of a skyscraper!

Also try this:

```
GRAPHIC3 20 -20 120
GRAPHIC3 -20 20 240
GRAPHIC3 2 2 2
GRAPHIC3 1 50 240
GRAPHIC3 -2 0 10
```

To produce a number of different designs similar to the previous one, the routine has to be even more universal. Besides entering variables for the size of the basic shape, enlarging factor, and rotation angle, we must input the number of corners the polygon is to have.

LOGO makes this task easy. Think of a circle as being made up out of 360 circle segments, each one degree. Then you can specify the angle between two sides of the polygon through 360, divided by the number of corners (sides).

```
TO TRIANGLE
REPEAT 3 (FD 100 RT 360/3)
END
```

This procedure causes the turtle to draw an equilateral triangle with a side length of 100. The procedure POLYGON in the program GRAPHIC4 allows you to produce various polygons. Note that the angle between two sides is always the same in a regular polygon. This angle is a constant, and has to be calculated only once. This speeds up the program greatly. It's definitely a trick worth remembering!

```
TO GRAPHIC4 :STEP :SIZE :FACTOR :TURN
CS HT
START :SIZE :FACTOR :TURN
END
```

```
TO START :SIZE :FACTOR :TURN
POLYGON :STEP :SIZE
FD 2
RT :TURN
MAKE "SIZE :SIZE + :FACTOR
START :SIZE :FACTOR :TURN
END
```

```
TO POLYGON :STEP :SIDE
LOCAL "TURN
MAKE "TURN (360 / :STEP)
REPEAT :STEP [FD :SIDE RT :TURN]
END
```


The following two figures are the executions of this program with these parameters:

for figure 8	8	60	0	124
for figure 9	100	5	0	10

Again, feel free to experiment with different values.

[Figure page 48]

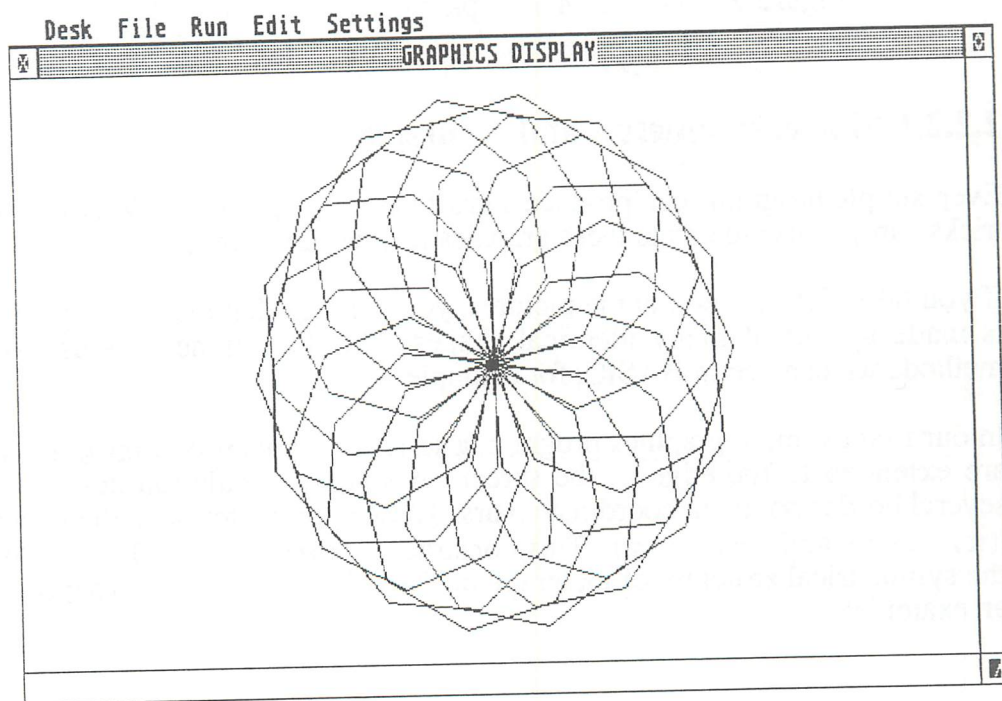


Figure 8: GRAPHIC4 with parameters 8 60 0 100

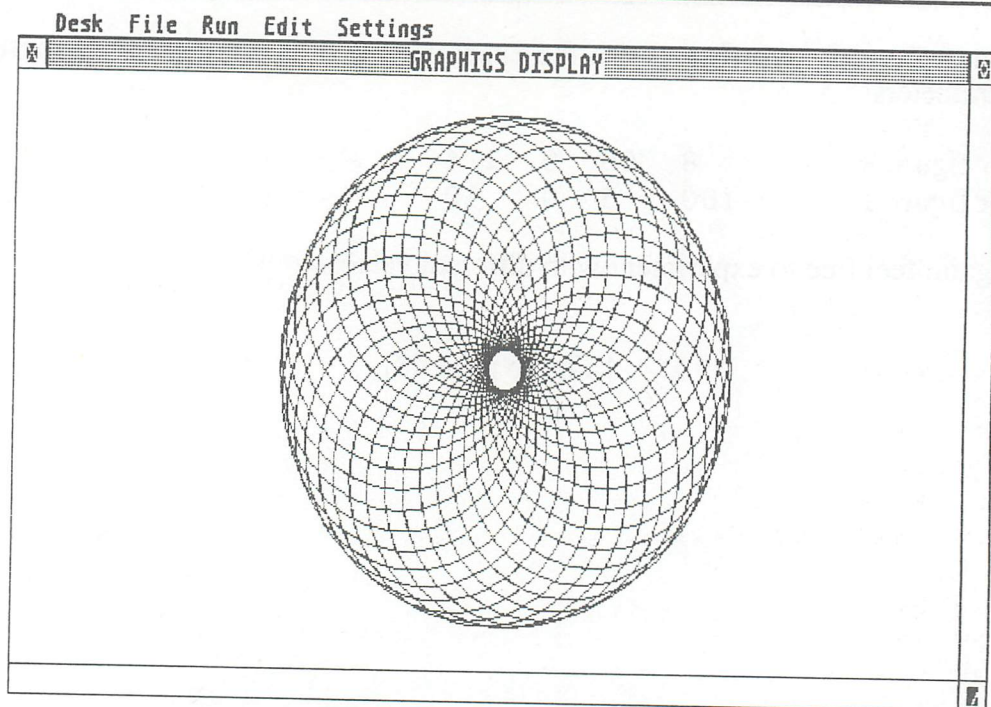


Figure 9: GRAPHIC4 with parameters 100 5 0 10

2.2.2.1 Moire, symmetry, and recursion

Even simple programs can produce a great variety of graphics. What kind of tricks can you use to create these graphics in other languages?

If you take a closer look at the pictures, you'll notice that the whole picture is made up out of many identical shapes. This fact alone tells us what methods we can use in BASIC, for example.

In our first example graphics program, a series of points is chosen and lines are extended to the edge of the screen. A simple calculation designates several border points in coordinate pairs. These coordinates have the format $(x, y-m)$ and $(x, y+m)$. This constructs a *mirror axis*, which forms the symmetrical structure of the graphic. The following BASIC program is an example:

```
10 rem graphic1.bas
20 rem -----
30 fullw 2: clearw 2
35 s=.5 's=1 for hires s=.5 for medres
40 gosub module1:gosub pause:clearw 2:gosub
  module2:gosub pause
50 clearw 2: gosub module3: gosub pause
60 clearw 2:gosub module1:gosub module2:gosub
  pause:end
70 rem-----
80 '
90 module1:
100 for x= 120 to 520 step 4
110 linef x,200*s,640,(x-120)*s
120 linef 640-x,200*s,0,(x-120)*s
130 next
140 return
150 rem-----
160 '
170 module2:
180 for x=120 to 520 step 4
190 linef x,200*s,640,(520-x)*s
200 linef 640-x,200*s,0,(520-x)*s
210 next
220 return
230 rem-----
240 '
250 module3:
260 for x=120 to 520 step 4
270 linef x,200*s,0,(520-x)*s
280 linef 640-x,200*s,0,(520-x)*s
290 next
300 return
310 rem-----
320 '
330 pause:
340 for i= 1 to 8000:next i:return
```


Often several mirror axes are used to create an effect called *moire* (pronounced "mwar-ay"). Moire got its name from a fabric noted for its silky, wavy nature.

You've probably seen this effect on your color TV. It occurs when thin bands of color travel across a finely textured image.

This is usually the fault of the TV picture tube, since it isn't capable of displaying very fine details. The same is true of a color monitor connected to the ST, since the phosphorous pixels in the monitor's tube limit its resolution.

```
10 rem Moire (Color monitor)
20 rem-----
30 clearw 2: fullw 2
40 for x=1 to 640 step 2
50 linef x,1,x,200
60 next
```

The physical construction of the picture tube can cause various color effects, and new colors can even be created when the electron beam hits a pixel covered by the screen mask. This effect varies from monitor to monitor, depending on its resolution. It really can't be calculated. The lower the resolution of a monitor, the greater the moire effects. A high resolution monitor's pixels are closer together and can be lit more precisely.

The following program will give you an idea of how good your color monitor is (if you have one). Its effect is similar to the test picture broadcast by some TV stations.

```

10 rem vertline.bas
20 rem -----
30 clearw 2: fullw 2
40 y1=40: y2=140
50 gotoxy 1,16
60 print"
                                     10      20      4      3      2      1"
                                     5
70 linef 80,y1,550,y1
80 linef 550,y1,550,y2
90 linef 550,y2,80,y2
100 linef 80,y2,80,y1
110 for x=80 to 290 step 21
120 gosub draw
130 next x
140 for x= 290 to 400 step 11
150 gosub draw
160 next x
170 for x=400 to 450 step 5
180 gosub draw
190 next x
200 for x=450 to 490 step 4
210 gosub draw
220 next x
230 for x=490 to 520 step 3
240 gosub draw
250 next x
260 for x=520 to 540 step 2
270 gosub draw
280 next x
290 for x=540 to 550
300 gosub draw
310 next x
320 input dummy$
330 end
340 draw:
350 linef x,y1,x,y2:return

```

The previously mentioned material also applies to the SM124 monochrome monitor as well. Sometimes the computer draws "patterned" lines when you specified nothing but straight lines. This is because a straight line is drawn like a set of stairs if it isn't perfectly vertical or horizontal, as is shown in Figure 10:

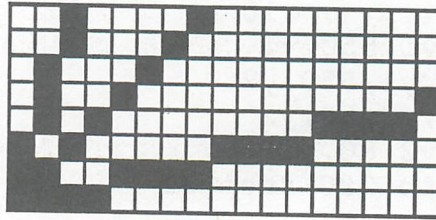


Figure 10: Straight lines—actually like steps

Because of this, a bundle of rays doesn't look like what you'd expect. Instead you get these results:

```

10  rem moire1
20  defint a-z
30  fullw 2 : clearw 2
40  for x=0 to 640 step 2
50  linef 0,0,x,200
60  linef 200,x,0
70  next

```

```

10  rem moire2
20  defint a-z
30  fullw 2 : clearw 2
40  for x=0 to 640 step 2
60  linef 640-x,200,x,0
70  next
80  for y=0 to 200 step 2
90  linef 640,200-y,0,y
100 next

```


These graphics are quite portable. All you need to know to produce graphics like these on another computer system is the coordinate system used by the ST and the commands needed. Armed with this knowledge, you can create all sorts of graphics on a wide range of different computers by making small changes in your program.

If you are using a color monitor, look at the programs above in middle and low resolutions. You'll be surprised how different the same picture can look at different resolutions.

Small changes in the program can produce strikingly different end results. Try changing the values after each step.

For example, the lines don't have to diverge from points on the border as above. The lines can start at any point on the screen. There also may be several vertices. The rays may cross each other, or the step value can be changed from line to line. You can use different colors within a picture as well:

```
10 rem graphic2.bas
20 rem random moire
30 rem -----
40 defint a-z
50 fullw 2
60 clearw 2
80 x = rnd(1)*640 : y=rnd(1)*200
90 s = rnd(1) * 10 + 2
100 for x1=1 to 640 step s
110 linef x,y,x1,200
120 linef x,y,x1,1
130 next x1
140 for y1=200 to 1 step - (s/2)
150 linef x,y,640,y1
160 linef x,y,1,y1
170 next y1
190 for i=1 to 10000:next
200 goto 60
```

If you wish to have several vertices, you must modify the program accordingly.

Try adding another loop to the program. You can designate the number of vertices with the loop end value in line 70. It might be a good idea to raise the minimum step width, so that your graphic won't print out black on black:

```
70  for point=1 to 2
180 next point
200 clearw 2:goto 70
```

Symmetry is not the only tool we can use to create computer art. You can repeat one object several times within one picture, as we did in the first sample LOGO programs. As you may remember, these pictures consisted of a square repeated many times in different positions and sizes. You'll see some good examples of this when we discuss displaying math functions.

The program is simplified by the object's repetition, since only one object that has to be drawn repeatedly. Even if the object is simple, like a square, the end result can often be quite impressive.

We'll present two prime examples of recursion in LOGO. The most simple application is the formation of a circle:

```
TO CIRCLE
RT 1
FD 1
CIRCLE
```

When this procedure has been repeated 360 times, a complete circle is constructed. At this point you can stop the program. Actually, this program doesn't use its full recursive potential. Every time the subroutine is called, the computer uses the variables found in the procedure so that values assigned to the same variables in the second execution do not replace the old data. The values remain and are available to the computer once the second execution is complete.

A typical application of recursive program structures is called HILBERT. It demonstrates a specific function vital to any Pascal user—the Hilbert curve. Our example Hilbert curve was written in LOGO, so you'll have to be a little patient during its execution.

```
TO A :VALUE
IF (:VALUE > 1)
  [D (:VALUE - 1) SETH 270 FD :H
   A (:VALUE - 1) SETH 180 FD :H
   A (:VALUE - 1) SETH 90 FD :H
   B (:VALUE - 1)]
END
```

```
TO B :VALUE
IF (:VALUE > 1)
  [C (:VALUE - 1) SETH 0 FD :H
   B (:VALUE - 1) SETH 90 FD :H
   B (:VALUE - 1) SETH 180 FD :H
   A (:VALUE - 1)]
END
```

```
TO C :VALUE
IF (:VALUE > 1)
  [B (:VALUE - 1) SETH 90 FD :H
   C (:VALUE - 1) SETH 0 FD :H
   C (:VALUE - 1) SETH 270 FD :H
   D (:VALUE - 1)]
END
```

```
TO D :VALUE
IF (:VALUE > 1)
  [A (:VALUE - 1) SETH 180 FD :H
   D (:VALUE - 1) SETH 270 FD :H
   D (:VALUE - 1) SETH 0 FD :H
   C (:VALUE - 1)]
END
```

```
TO START
NAME (:ORD + 1) "ORD
NAME (QUOTIENT :H 2) "H
NAME (:XX + (QUOTIENT :H 2)) "XX
NAME (:YY + (QUOTIENT :H 2)) "YY
PENUP
SETX :XX
SETY :YY
PENDOWN
A :ORD
END
```


TO HILBERT

CS

HT

CLEARTEXT

PR []

PR []

PR [Hilbert Curve Demonstration]

PR []

PR [Algorithm and data structure]

PR [by Niklaus Wirth]

PR []

PR []

MAKE "H 256

MAKE "ORD 1

MAKE "XX 0

MAKE "YY 0

REPEAT 6 [START]

END

MAKE "ORD 6

MAKE "H 8

MAKE "YCOR 82

MAKE "XCOR 170

MAKE "YY 124

MAKE "XX 124

2.2.2.2 LOGO graphic demos

You should now be familiar with the basics of computer graphics and video art. The following pages present several LOGO programs. Most of them are recursive structures, and you've probably seen a couple of them before. We hope that these short programs will encourage you to experiment with your own programs. Some of the functions mentioned earlier in this book are used in these programs, such as the function for drawing a circle. None of the programs should be difficult to understand.

```
TO MOIRE
  (LOCAL "X "Y "R "DW "W "Z)
  HT CS FS
  SETPAN [150 100]
  SETZOOM 1.2
  SETLINE [1 1 1]
  MAKE "R 200
  MAKE "DW 0.01
  MAKE "W 0
  MAKE "Z 180 / PI
  MOIRELINE
  SETZOOM 1
  SETPAN [0 0]
  STOP
END
```

```
TO MOIRELINE
  IF NOT (:W < (2 * PI)) [STOP]
  MAKE "X (:R * COS (:W * :Z))
  MAKE "Y (:R * SIN (:W * :Z))
  MAKE "W :W + :DW
  MAKE "R :R * 0.997
  MAKE "DW :DW / 0.997
  LINEX 100 70 (100 + :X) (70 + :Y)
  MOIRELINE
END
```

```
TO LINEX :X1 :Y1 :X2 :Y2
  PU
  SETPOS LIST :X1 :Y1
  PX
  SETPOS LIST :X2 :Y2
END
```

```
TO THROWNET
(LOCAL "X "Y "XA "YA "VX "VY "V "W "I)
HT CS FS
SETPAN [150 -90]
SETZOOM 1.9
SETLINE [1 1 1]
MAKE "I 0
MAKE "V 17
THROWLINE1
SETZOOM 1
SETPAN [0 0]
STOP
END
```

```
TO THROWLINE1
IF :I > 30 [STOP]
MAKE "W :I * 3
MAKE "VX :V * COS (:W)
MAKE "VY :V * SIN (:W)
MAKE "X 0
MAKE "Y 0
MAKE "XA 0
MAKE "YA 0
MAKE "VY :VY - 0.5
THROWLINE2
MAKE "I :I + 1
THROWLINE1
END
```

```
TO THROWLINE2
IF :Y <= -30 [STOP]
POLY (LIST :XA -(170 - :YA) :X -(170 - :Y))
MAKE "XA :X
MAKE "YA :Y
MAKE "X :X + :VX
MAKE "Y :Y + :VY
MAKE "VY :VY - 1
THROWLINE2
END
```

```
MAKE "XA 100
```

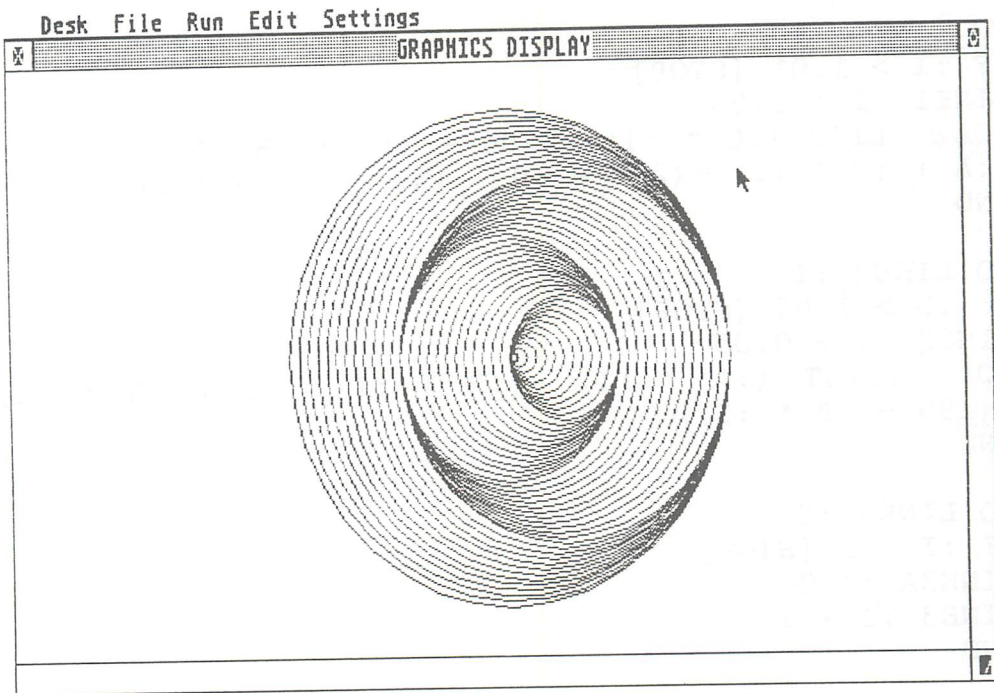



Figure 11: A hardcopy of CONCENTRIC

```
TO PARABOLOID
(LOCAL "A "B "C "D)
HT CS
SETZOOM 1.5
SETPAN [150 -100]
SETLINE [1 1 1]
MAKE "A 250
MAKE "B 150
MAKE "C 70
MAKE "D 50
LINE1 0
LINE2 0
BOX [0 -150 250 150]
BOX [70 -200 250 150]
LINE3 0
SETZOOM 1
SETPAN [0 0]
STOP
END
```

```
TO LINE1 :I
IF :I > 1.01 [STOP]
LINE1 :I + 0.05
POLY (LIST (:C * :I) -(200 - (:D + :B) * (1 - :I))
(:A + :C * :I) -(200 - :D - (:B - :D) * :I))
END
```

```
TO LINE2 :I
IF :I > 1.01 [STOP]
LINE2 :I + 0.05
POLY (LIST (:A * :I) -(:B * :I) (:C + :A * :I)
-(199 - :B * :I))
END
```

```
TO LINE3 :I
IF :I > 1 [STOP]
LINE3A :I 0
LINE3 :I + 1
END
```

```
TO LINE3A :I :J
IF :J > 1 [STOP]
POLY (LIST (:I * :A) -(:J * :B) (:I * :A + :C) -(:J
* :B + :D))
LINE3A :I :J + 1
END
```

```
TO HEX
IF :W1 > (2 * PI) [STOP]
MAKE "W2 (4 * :W1)
MAKE "XA (1.8 * ((:R1 * SIN (:W1 * 180 / PI))
MAKE "XB (:R2 * SIN (:W2 * 180 / PI))
MAKE "X1 :XA + :XB
MAKE "X2 :XA - :XB
MAKE "YA (:R1 * COS (:W1 * 180 / PI))
MAKE "YB (:R2 * COS (:W2 * 180 / PI))
MAKE "Y1 (:YA + :YB) * 2
MAKE "Y2 (:YA - :YB) * 2
POLY (LIST :X1 :Y1 :X2 :Y2)
MAKE "W1 :W1 + 0.05
HEX
END
```

```
TO HEXAVISION
(LOCAL "X1 "X2 "XA "XB "Y1 "Y2 "YA "YB "R1 "R2 "W1
"W2)
HT FS CS
SETLINE [1 1 1]
MAKE "R1 70
MAKE "R2 30
MAKE "W1 0
HEX
STOP
END
```

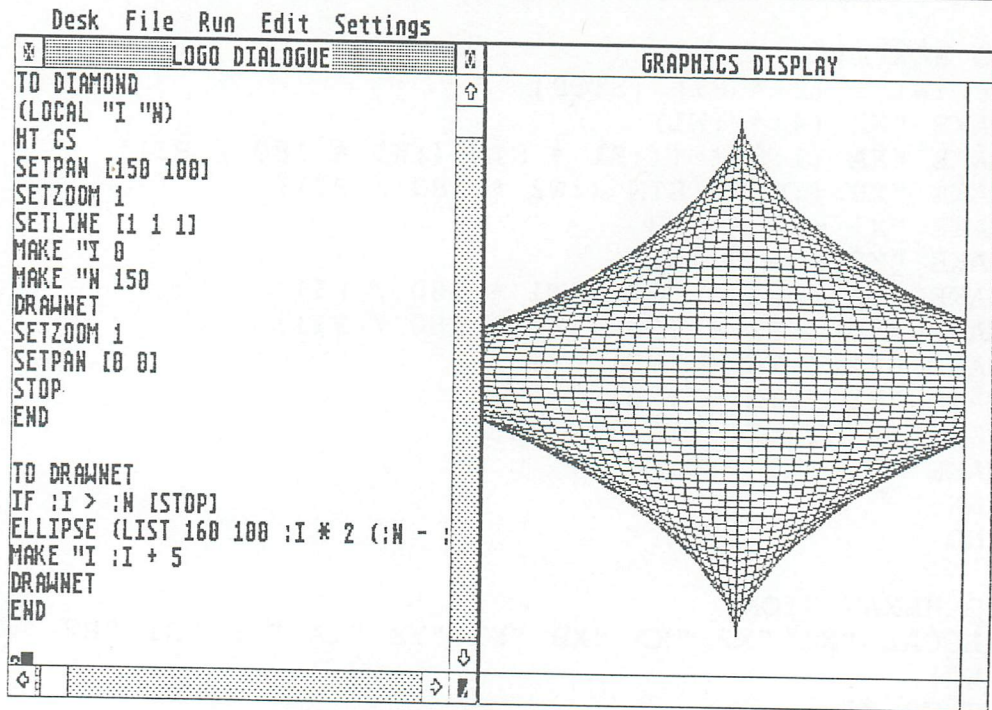



Figure 12: The program NET

```

TO SPIRALSTART
  CS
  SETLINE [1 1 1]
  SETCURSOR -160 -95
  SETHEADING 96
  SPIRALDRAW 330 122 3
  WAIT 100
  CS
  SETCURSOR -160 -75
  SETHEADING 99
  SPIRALDRAW 320 123 3
  WAIT 100
  CS
  SETCURSOR -160 -75
  SETHEADING 99
  SPIRALDRAW 320 123 2
  WAIT 100
  STOP
END

```

```

TO SPIRALDRAW :LEN :DEG :DEC
IF :LEN < 0 [STOP]
FD :LEN LT :DEG
SPIRALDRAW (:LEN - :DEC) :DEG :DEC
END

```

```

TO SETCURSOR :X :Y
PU HOME
SETPOS LIST :X :Y
PD
STOP
END

```

```

TO WAIT :DELAY
IF :DELAY < 1 [STOP]
WAIT :DELAY - 1
END

```

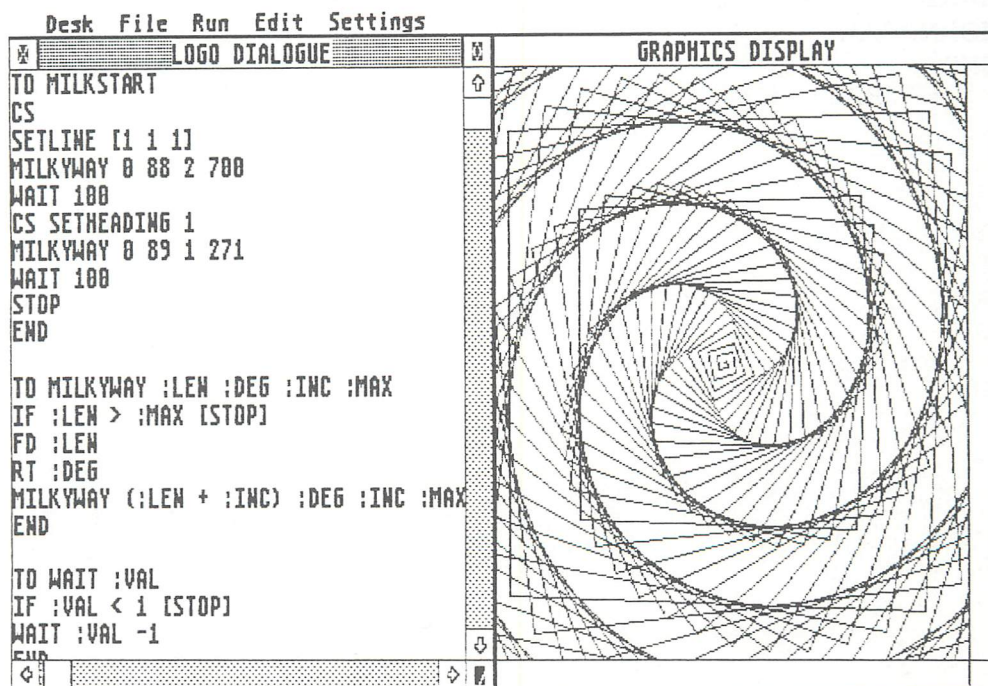


Figure 13: The program MILKYWAY

```
TO HORN
  (LOCAL "X "Y "R "W)
  HT CS FS
  SETPAN [150 100]
  SETZOOM 1
  SETLINE [1 1 1]
  MAKE "R 80
  MAKE "W 0
  HORNDRAW
  SETZOOM 1
  SETPAN [0 0]
  STOP
END
```

```
TO HORNDRAW
  IF :R =< 0 [STOP]
  MAKE "X 160 + 110 * COS (:W * 180 / PI)
  MAKE "Y 110 - 110 * SIN (:W * 180 / PI)
  CIRCLE (LIST :X :Y :R)
  MAKE "W :W + 0.1
  MAKE "R :R - 1
  HORNDRAW
END
```



```
TO INFINITY
(LOCAL "A "B "X "Y "D "XO "YA "XA "W)
HT CS FS
SETPAN [150 100]
SETZOOM 1.5
SETLINE [1 1 1]
MAKE "A 100
MAKE "B 20
MAKE "X 25
MAKE "D 0.05
MAKE "XO 20
MAKE "YA 100
MAKE "XA :XO
MAKE "W 0
INFIDRAW
SETZOOM 1
SETPAN [0 0]
STOP
END
```

```
TO INFIDRAW
IF :XO >= 310 [STOP]
MAKE "XO :XO + :D
MAKE "X :XO + :B * SIN (2 * :W * 180 / PI)
MAKE "Y 100 + :A * SIN (:W * 180 / PI)
POLY (LIST :XA :YA :X :Y)
MAKE "A :A * 0.9998
MAKE "B :B * 0.9998
MAKE "XA :X
MAKE "YA :Y
MAKE "W :W + 0.05
INFIDRAW
END
```

```
TO SNOWSTART
CS
SETCURSOR -30 150
SETTEXT 29
TT [SNOWSTAR]
SETTEXT 0
SETLINE [1 1 1]
SETCURSOR -70 -130
(LOCAL "SIZE "LEVEL)
MAKE "SIZE 250 MAKE "LEVEL 4
SNOW :SIZE :LEVEL
WAIT 100
STOP
END
```

```
TO SNOW :SIZE :LEVEL
IF :LEVEL < 1 [STOP]
S :SIZE :LEVEL
SNOW :SIZE :LEVEL - 1
END
```

```
TO SIDE :SIZE :LEVEL
IF :LEVEL = 0 [FORWARD :SIZE STOP]
SIDE (:SIZE / 3) (:LEVEL - 1)
LEFT 60
SIDE (:SIZE / 3) (:LEVEL - 1)
RIGHT 120
SIDE (:SIZE / 3) (:LEVEL - 1)
LEFT 60
SIDE (:SIZE / 3) (:LEVEL - 1)
END
```

```
TO S :SIZE :LEVEL
REPEAT 3 [SIDE :SIZE :LEVEL RIGHT 120]
STOP
END
```

```
TO SETCURSOR :X :Y
  PU HOME
  SETPOS LIST :X :Y
  PD
  STOP
END
```

```
TO WAIT :TIME
  IF :TIME < 1 [STOP]
  WAIT :TIME - 1
END
```

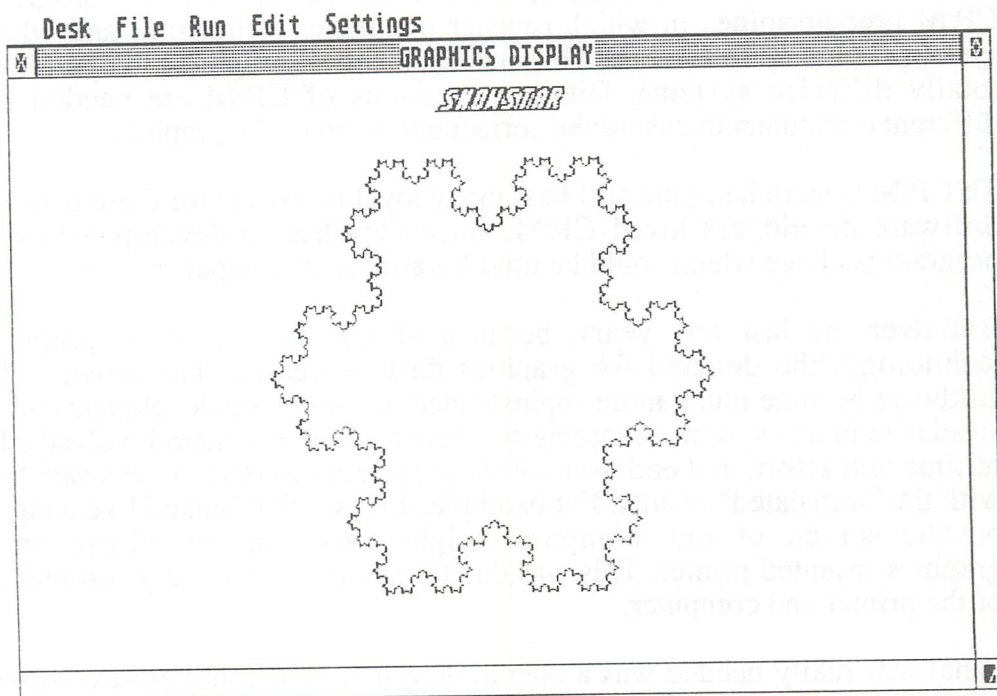


Figure 14: Hardcopy of SNOWSTAR

2.3 Graphics under GEM

If you've worked with the LOGO programs a little, you might have noticed a few special graphic commands that set the ST apart from other computers. These are not really LOGO routines, but are subprograms of the TOS operating system, which consists of GEMDOS and GEM.

The Virtual Device Interface, or VDI, is specifically responsible for these graphic function operations. The VDI is a collection of programs which controls and manages all data sent to the screen.

The idea behind GEM, as well as the VDI, is similar to that of the CP/M operating system. CP/M is a portable operating system. This means that the same version of CP/M will run on many different computers, and is not dependent on the hardware. For instance, when CP/M displays a character on the screen, the computer does not send the code of this letter to some memory location. Instead, the code is written to a specific register of the processor, which in turn calls up the corresponding CP/M subprogram. CP/M predetermines in which register parameter to transmit and which system command to activate. This enables identical programs to operate on totally different systems. Different versions of CP/M are needed for different computers to ensure the correct operation of the graphics.

Th CP/M system had (and still has) many loyal followers for these reasons. Software developers loved CP/M, since they had to develop only one program package which could be used for almost all computers.

But over the last few years, because of rapidly improving graphics technology, the demand for graphics data processing has grown. The hardware became much more sophisticated, but software developers had to specialize in one system. Adapting software to another system involved a lot of time and effort, and end users of these systems weren't always satisfied with the "translated" results. For example, a figure that looked like a circle on the screen of one computer might look like an ellipse on a graphics-oriented printer. This was due to the different x and y resolutions of the printer and computer.

What was really needed was a operating system that could ensure correct proportion and scaling between different peripheral devices. Ideally, this operating system would also add graphic commands (such as set point, draw line) that would be controlled in a way similar to the CP/M system.

GEM is a graphics operating system that handles of input and output in an extremely user-friendly manner. GEM supports many commands for drawing lines, circles, and polygons, for the output of text, and also routines to control a wide range of peripherals.

Programmers use GEM subprograms as standard functions, instead of constantly rewriting routines and inventing the wheel over and over again.

2.3.1 GEM

Right now we'll take a look at the operation of GEM, and especially the operation of the VDI. We'll see how to integrate GEM into our programs, and some typical applications for GEM functions.

First we need to supply the VDI with answers to a few questions. Will the output be sent to a monitor in raster coordinates, or will it be sent as vectors to operate a plotter? How many points can the peripheral display horizontally and vertically in a distance of one centimeter? How many pens (colors) can the peripheral use? This first step is initializing the VDI for the specific peripheral used.

All information about the peripherals is contained in separate peripheral controllers. These GEM components are dependent upon the peripheral hardware and are loaded when needed. The ST's controller for the monitor was integrated into TOS. As a result, the monitor can be used during all GEM operations. To use a printer in the same way, a peripheral controller for a specific printer must first be loaded with the necessary information.

To ensure that the original proportions of a picture are retained by a peripheral device, GEM specifies output in two different coordinate systems:

- NDC Normalized Device Coordinates
- RC Raster Coordinates

Raster Coordinates refer to the existing pixels on the ST monitor screen—320 x 200 and 640 x 400. Raster coordinates are measured in x and y steps for plotters.

In contrast, Normalized Device Coordinates lie on an idealized plane. The NDC orientation is like the Cartesian coordinate system most of us are familiar with. This system places the coordinate pair 0,0 in the bottom lefthand corner, and the highest value for x and y is the top right corner.

Thus the NDC offers a range of 0,0 to 32767,32767, producing a geometrically true plane with very high resolution.

When programming in GEM, you can choose between these two coordinate systems. If you chose NDC, the values you enter are changed by the VDI to the actual coordinates used by the computer. If you chose RC, the VDI will not make any changes to the coordinates. You (or the program) will have to constantly monitor scale changes.

The NDC system offers the greatest advantages where graphics are exchanged between different peripherals.

The RC system has problems associated with transmitting information to the printer, especially in regard to vertical-horizontal proportions. With its coordinate system of 640 x 200 points, the proportion between the coordinates is a ratio of roughly 1:1.8. As a result, squares look like rectangles when displayed in RC.

If the VDI is addressed, the peripheral controller can perform the necessary calculations to convert RC. Then, the graphics transferred between peripherals retain their original proportions. The disadvantage is that this method takes more time, since the points have to be converted to the true coordinate system.

However, NDC is reserved for the use in higher languages like Pascal and Modula, since BASIC and LOGO do not allow you to chose the coordinate system. For this reason it is usually better to use the RC system, unless your program has to be portable.

All of the programs in this book use the raster coordinates.

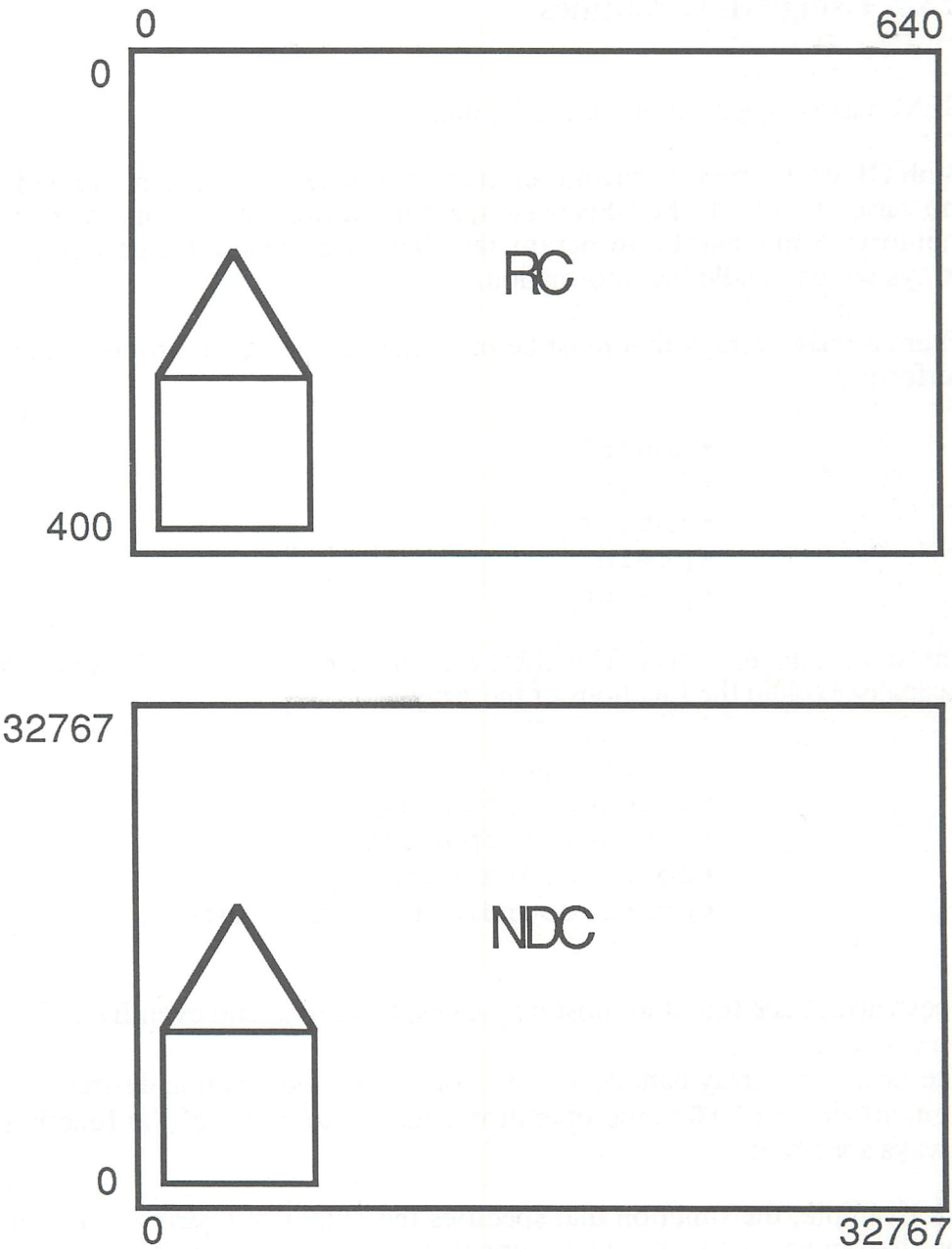


Figure 15: Coordinates in RC and NDC systems

2.3.2 Using GEM routines

GEM makes programming graphics much easier.

With GEM's immense capabilities, many parameters are required to perform the varied functions. For this reason, a specific area of memory is reserved to manage these parameters. The parameters are transferred in arrays which handle the information.

There are five arrays that must be initialized before a VDI routine may be performed:

- `ctrl`
- `intin`
- `intout`
- `ptsin`
- `ptsout`

These are integer arrays. The abbreviations, which you should remember, basically explain the functions of the arrays:

- control array
- integer input array
- integer output array
- point coordinate input array
- point coordinate output array

These arrays are found in most languages: C, Pascal, and even BASIC.

The control array handles all data concerning the function desired. In the element `ctrl(0)`, the operation code (or *opcode*) of the function is always specified.

For example, the function that specifies the output of a string in a special font has an opcode of 8. The routine that specifies a chain of polygons has the opcode 6, etc. The other elements of the control array contain information about the data in other arrays, as well as directions to the peripheral being addressed. For instance, GEM transmits a number for each active window. These numbers are called *handles*. Handles are needed to activate a function at the correct peripheral.

The input array transmits the data that is to be used in the function. Referring to the first example above, the output of a string in a special font, the input array would transmit the string of characters to be printed.

Once the function has been called, `intout` contains the function values. If the function `open_workstation` (sets the VDI to a certain peripheral) is called, you receive 45 different items of information about the peripheral in elements `intout(0)` to `intout(44)`.

`Ptsin` and `ptsout` are reserved for coordinate specifications. These might be a chain of x, y coordinates, such as a string of polygons, or a single coordinate pair to specify the position of text on the screen.

Next we'll show you how the GEM library operates with examples of an `open_workstation` call and `v_gtext`.

2.3.2.1 Example 1: Initializing `Open_workstation`

`Open_workstation` (opcode 1) is the GEM call needed to access all source codes for compiler languages. In the *Atari ST GEM Programmer's Reference* book from Abacus, this GEM call is defined as follows:

<code>contrl(0)</code>	Opcode (1)
<code>contrl(1)</code>	Number of ptsin points (0)
<code>contrl(3)</code>	Length of intin array (11)
<code>intin(0)</code>	Device number, driver
<code>intin(1)</code>	Line type
<code>intin(2)</code>	Color of lines
<code>intin(3)</code>	Marker type
<code>intin(4)</code>	Marker color
<code>intin(5)</code>	Font
<code>intin(6)</code>	Text color
<code>intin(7)</code>	Fill type
<code>intin(8)</code>	Fill pattern index
<code>intin(9)</code>	Fill color
<code>intin(10)</code>	Coordinate flag

The array values are set according to your preferences. A value of 1 in `int_in(1)` sets the lines as solid lines. A value of 0 in `int_in(10)` selects NDC, while a value of 2 selects RC (1 is reserved for future use).

These arrays must always be set. Fortunately, the designers of GEM specified the parameters so that a value of 1 is always the default value, corresponding to the normal setting. This spares you from having to set each parameter every time you use a function.

As a result, the initialization can be done in a single loop. The following example listings show what such a loop looks like in C or Modula-2:

C listing:

```
open_vwork()
{
    int i;
    for (i = 1; <10; i++)
        int_in[i] = 1;
    int_in[10] = 2;
    v_opnwk(int_in, &handle, int_out);
}
```

Modula-2 listing:

```
FROM VDIControls IMPORT OpenVirtualWorkstation,
VAR In:VDIWorkInType;
    Out:VDIWorkOutType;

PROCEDURE open_vwork;
VAR i:INTEGER;
BEGIN
    FOR i:=0 TO 9 DO In[i]:=1 END;
    In[10]:=2;
    OpenVirtualWorkstation(In,handle,Out);
END;
```


2.3.2.2 Example 2: Graphics/text output

The second example is designed to show you how you can print text at a graphics position, and not at the cursor. For example, function 8—`v_gtext (handle, x, y, string)`—instructs GEM to label the axes of function-plotting programs.

In the C listing above, *handle* stands for the output device, *string* is the text to be printed, and *x, y* specify the starting position of the text.

Since these coordinates are pixel coordinates, they must be multiplied by the height and width of the characters, so that you can work with rows and columns. The actual height of the active character set is specified in pixels as a second parameter with `graf_handle`.

Example:

```
handle=graf_handle(&dummy,&height,&dummy,&dummy);
v_gtext(handle,0,height*5,"This is great!!");
```

The font may be selected with `vst_effects (handle, effect)`. The value of *effect* is registered at `intin(0)` and thus determines the font for the next output. The selection of fonts is controlled through bits 0 to 5:

<u>BIT</u>	<u>controls</u>	<u>ACTIVE (=1)</u>	<u>ERASED (=0)</u>
0	bold print	bold	normal
1	light type	light	normal
2	italics	<i>italics</i>	normal
3	underline	<u>underlined</u>	normal
4	outline	outlined	normal
5	shadow	shadowed	normal

The number formed by the combination of the corresponding bits lets the effects be mixed. Some examples:

1	bold print	decimal 1=	000001
2	light print	decimal 2=	000010
3	bold & light print		000011
4	italic		000100
5	italic & bold print		000101
6	italic & light print		000110
7	italic, light print, & bold print		000111
8	underlined & bold print		001001
9	underlined & low intensity		001010

The following LOGO program demonstrates the various character styles available with GEM on the ST. (Settype is the same as vst_effects() in LOGO.)

```

TO TEXTDEMO
  (LOCAL "HI "WI "XC "Z)
  CS HT FS
  SETTEXT 17
  SETCURSOR -100 140
  TT [TEXTSTYLES UNDER LOGO]
  MAKE "Z 0
  MAKE "HI 0
  TEXTPLACE 0
  WAIT 500
  SETTEXT 0
  END

```

```

TO TEXTPLACE :LINE
  IF :LINE > 3 [STOP]
  MAKE "XC (-285 + :LINE * 150)
  SETCURSOR :XC "74
  MAKE "HI :HI + 1
  MAKE "WI :HI
  TEXTOUT :WI * 16
  TEXTPLACE :LINE + 1
  END

```

```
TO TEXTOUT :WI
IF (:Z > (:WI - 1)) [STOP]
SETTEXT :Z
TT (LIST "TextStyle :Z)
RETURN
MAKE "Z :Z + 1
TEXTOUT :WI
END
```

```
TO SETCURSOR :X :Y
PU HOME
SETPOS LIST :X :Y
PD STOP
END
```

```
TO RETURN
SETCURSOR XCOR YCOR - 16
STOP
END
```

```
TO WAIT :DELAY
IF :DELAY < 1 [STOP]
WAIT :DELAY - 1
END
```

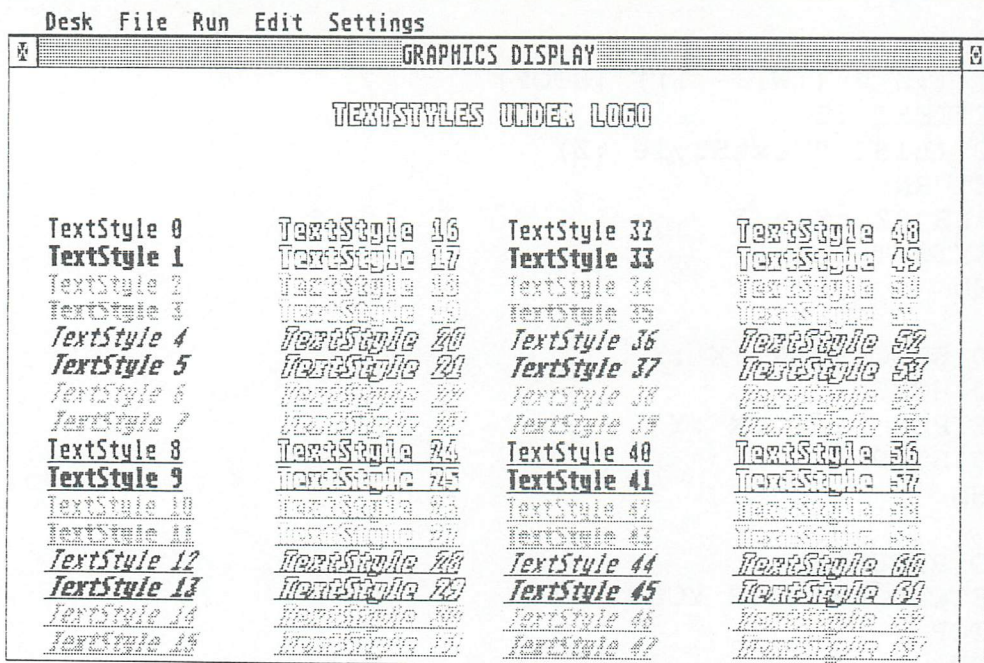



Figure 16: These character styles are available to GEM

2.3.2.3 GEM calls in BASIC

You'll want to know how to use these GEM calls in BASIC. Actually, you probably already know enough to accomplish this, since you only have to initialize the arrays correctly and then execute the VDI call. Machine language subroutines are always invoked by specifying the subroutine's address. But what address does a GEM call have?

Fortunately, this information is not necessary in the case of the ST, since ST BASIC recognizes the arrays. ST BASIC also recognizes the `vdisys` call, which transfers program control to the VDI routine specified in `contrl(0)`. Consequently, the call parameters for `vst_effects` are:

```

contrl(0)  opcode (106)
contrl(1)  number of values in ptsin (0)
contrl(3)  number of values in intin (1)
contrl(6)  device identifier

intin(0)   font style number

```

All you have to do is POKE these values in their corresponding memory locations.

They have to be POKEd because the arrays are not recognized as arrays by BASIC, but as consecutive memory locations located with `control`, etc.

A GEM call in BASIC always looks something like this:

```
10 rem Gem call function 106: Text Special Effects
20 '
100 buildfont:
110 poke control,106 : rem function number
120 poke control+2,0 : rem number of values in ptsin
130 poke control+4,1 : rem number of values in intin
140 poke control+6,1 : rem handle-number: BASIC = 1
150 poke intin, 3 : rem Font: 3 = half intensity
160 vdisys
170 print "Testing Font"
```

Once you know this, it's easy to write a BASIC program similar to the previous LOGO program:

```
10 rem textdemo.bas
20 rem Demonstration of all text styles
30 rem -----
40 clearw 2: fullw 2
50 for font=0 to 31
60 gosub characterstyle
70 print "Character Style ";font;" "
80 next font
90 font = 0: gosub characterstyle:end
100 characterstyle:
110 poke control,106 : rem Function number
120 poke control+2,0 : rem number of values in ptsin
130 poke control+4,1 : rem number of values in intin
140 poke control+6,1 : rem handle-number: BASIC = 1
150 poke intin, font
160 vdisys
170 return
```


Chapter 3

Two-dimensional Graphics

Two-dimensional Graphics

3.1 Points, lines and circles

As you have discovered, there aren't any commands in BASIC to set a single point. That's because GEM doesn't contain such a routine. But all languages have the capability to display polygons since the VDI can connect a series of dots in the pxyarray with the corresponding command:

```
draw_lines()  
{  
    pxyarray[0] = 100  
    pxyarray[1] = 100  
    pxyarray[2] = 100  
    pxyarray[3] = 300  
    pxyarray[4] = 500  
    pxyarray[5] = 300  
}
```

3.1.1 Plot point

BASIC can plot a line that has the same starting and ending points:

```
linef 1,10,1,10
```

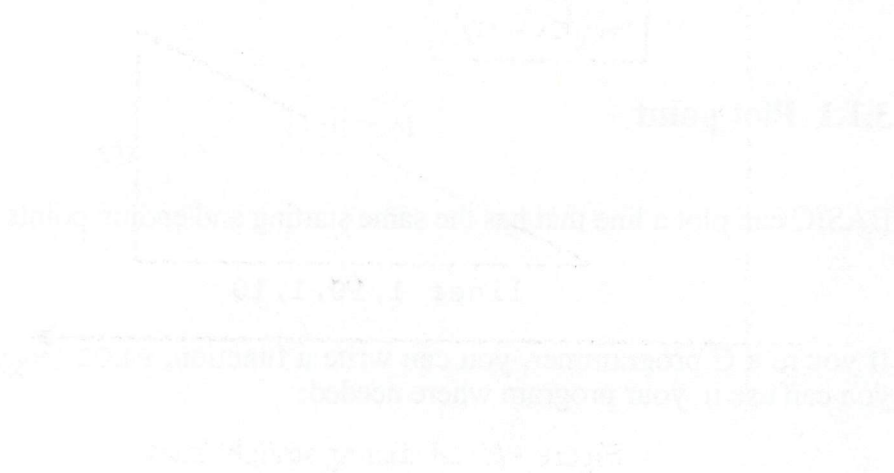
If you're a C programmer, you can write a function, `PLOT(x, y)`, which you can use in your program where needed:


```
plot(x,y)
int x,y;
{
    pxyarray[0] = x;
    pxyarray[1] =y;
    v_pmarker(handle,1,pxyarray);
}
```

3.1.2 Lines

GEM allows you to draw lines as solid, dashed, or dotted. If you aren't too concerned about speed, you can also develop an algorithm that's able to draw dashed or dotted lines on the screen without going through GEM.

Take a look at the following figure:



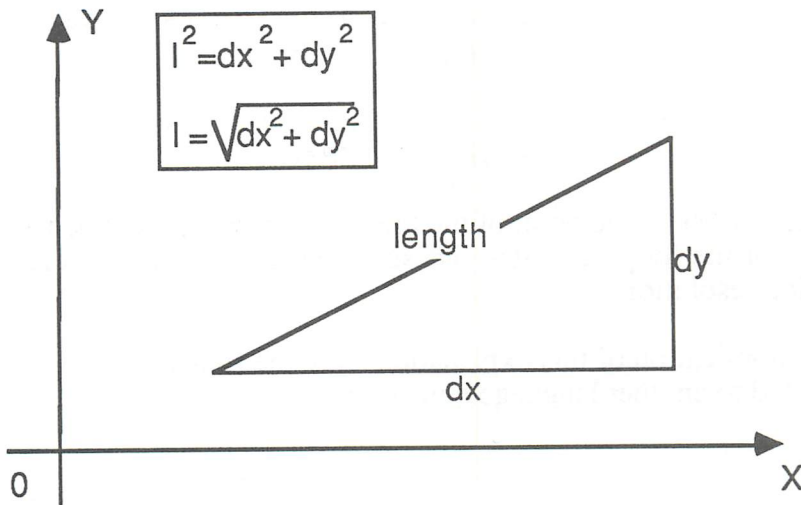
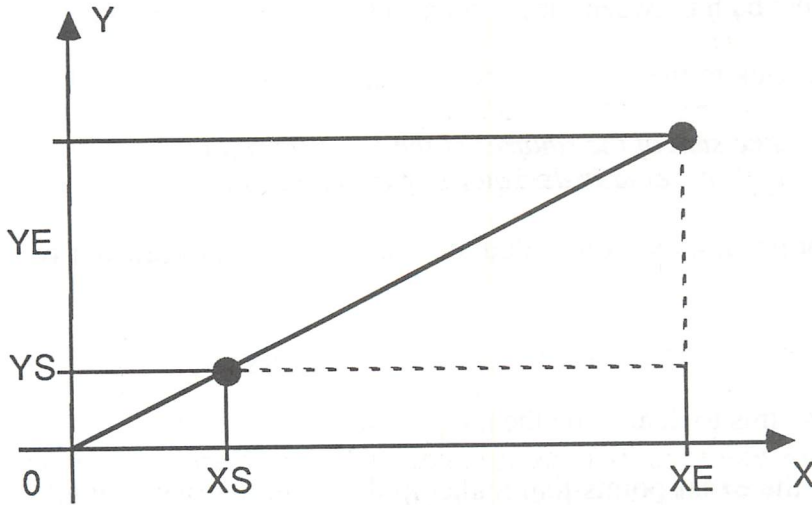


Figure 17: Calculating straight lines.

Somewhere in the picture are two points that we would like to connect with a line. The only information we have is the x and y coordinates of those points. For now, the distance between them can be specified in terms of the x and y axes:

$$dx = xe - xs$$

$$dy = ye - ys$$

The shortest path between these points, a straight line, is not given.

Let's refer back to the Greek mathematician Pythagoras and his theorem:

*The sum of the squares of the legs of a right triangle
equals the square of its hypotenuse.*

This theorem allows us to calculate the distance between the two given points:

$$\text{distance} = \text{SQR}(\text{dx}*\text{dx}+\text{dy}*\text{dy})$$

We can use this to determine the number of dots in the line.

To locate the exact points that make up this line, the line's length must be divided by the distance obtained in the first step. This result is the increased values of the coordinates from the original point.

$$\begin{aligned}\text{increase } x &= \text{dx} / \text{distance} \\ \text{increase } y &= \text{dy} / \text{distance}\end{aligned}$$

$$\begin{aligned}\longrightarrow \quad x &= x_s + \text{STEP}*\text{dx} \\ y &= y_b + \text{STEP}*\text{dy}\end{aligned}$$

In this case, STEP is the point of the line to be drawn. It is dependent on the step value of the loop variables. These variables in turn depends upon the desired line resolution.

To see an application of this explanation, type the following program. It can be converted to another language quite easily:

```
10    rem drawline.bas
20    rem -----
30    defint a-z
40    clearw 2: fullw 2
50    input "from which x";xs
60    input "and which y";ys
70    input "to which x";xe
80    input "and which y";ye
90    input "distance between points";sw
100   dx=xe-xs:dy=ye-ys
110   length = sqr(dx*dx+dy*dy)
120   for distance =0 to length step sw
130   x=x+ distance *dx/length
140   y=ys+distance *dy/length
150   linef x,y,x,y
160   next point
```

3.1.3 Circles

ST BASIC has a `CIRCLE` command, but the result of this command is not exactly what you might expect. Instead of drawing a circle, the ST draws a polygon. In most cases, this extremely fast method produces acceptable results. But this method is not precise enough when we need mathematical exactness. Instead, we can apply the mathematical definition of a circle.

A circle is the set of points in a plane that are the same distance (the radius) from one point in that plane (the center).

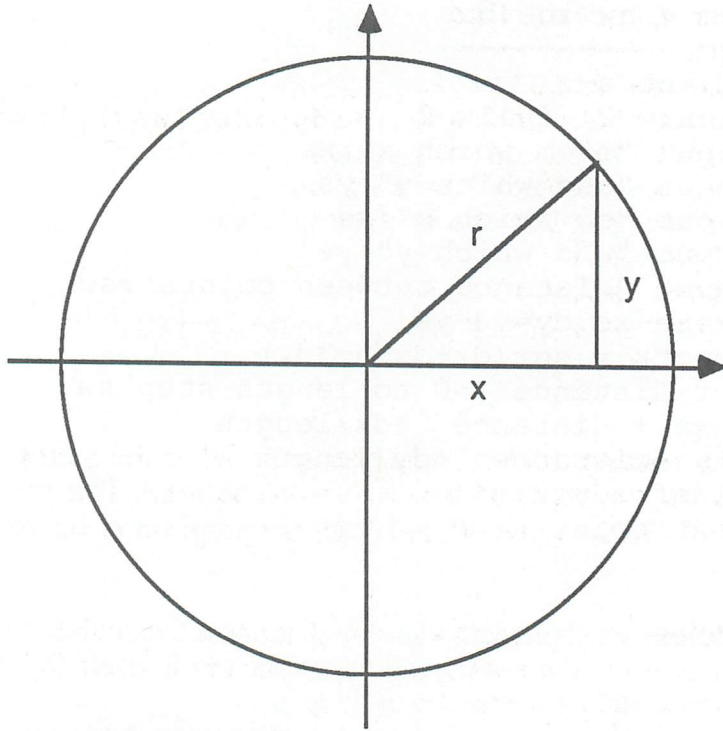


Figure 18: Defining radius

The Pythagorean theorem allows us to calculate the radius in the following manner:

$$\begin{aligned}r * r &= x * x + y * y \\r &= \text{SQR}(x * x + y * y)\end{aligned}$$

This can be used when the coordinates of the center are (0,0)—in other words, the intersection of the x and y axes. This equation is solved for y:

$$\begin{aligned}r * r &= x * x + y * y \\y &= \text{SQR}(r * r - x * x)\end{aligned}$$

To draw a circle, the x values ranging from r to -r must be used. This is most easily accomplished using a loop.

```
10    rem circle1.bas
20    rem -----
30    clearw 2:fullw 2
40    input "around what midpoint (x/y) ";xm,ym
50    input "radius ";radius
60    for x=-radius to radius
70    y=sqr(radius*radius-x*x)+ym
80    x1=x+xm
90    linef x1,y,x1,y
100   linef x1,ym-(y-ym),x1,ym-(y-ym)
110   next x
```

Actually this routine still has some problems. When the radii get larger, the step values are too large and the circle won't be solid. This problem can be solved by STEP .1 in line 40, although the program is slower because of the added work.

It's better to use another method—one that doesn't calculate the individual points in relation to the x-axis, but draws the circle itself (by degrees) and calculates the x and y coordinates directly.

To accomplish this, we need to incorporate the *polar coordinate system*. In this system a point is specified by its distance from the origin (the center) and the angle between this line (origin to point) and the positive x-axis.

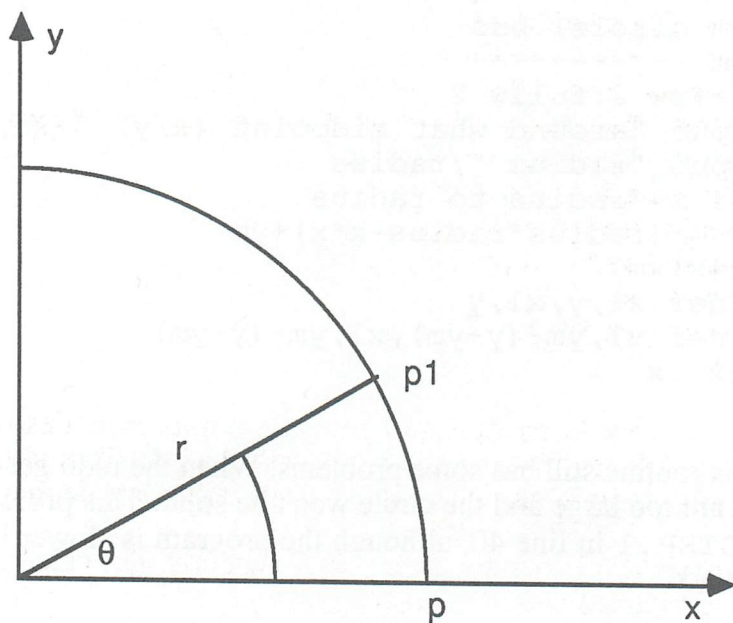


Figure 19: Polar coordinates

This method is ideal for our application. Let's say the coordinates for point p are 0 degrees and 50 units of length. To calculate the remaining points in the circle, the degree value has to be incremented by one until the total 360 degrees has been reached. Since we need x and y coordinates, these conversion equations are used:

$$x = r * \cos(\theta)$$

$$y = r * \sin(\theta)$$

Any circle is easily drawn with this method.

```
10    rem circle2.bas
20    rem -----
30    clearw 2:fullw 2
40    input "around what midpoint (x/y) ";xm,ym
50    input "radius ";radius
60    for grad=1 to 360
70    x=radius*cos(grad)+xm
80    y=radius*sin(grad)+ym
90    linef x,y,x,y
100   next grad
```

Only the relatively large distance between points is undesirable. This mistake is not the fault of the algorithm, but BASIC. This is because all angle functions use radians, whereas our method requires degrees.

Unfortunately BASIC does not have a DEG function. The calculation has to be performed manually.

The circumference of a 1 unit circle (a circle with a radius of 1) is exactly 2π (since circumference = $2\pi r$). Thus an angle of 360 degrees equals $2*\pi$, which is why one degree can be expressed as $2*\pi/360$.

Because of this, these two equations allow us to easily convert degrees to radians and vice versa:

$$\begin{aligned}\text{RAD} &= \text{DEG} * \pi / 180 \\ \text{DEG} &= \text{RAD} * 180 / \pi\end{aligned}$$

A constant factor c , representing π , is used for the multiplication. It's defined before the actual drawing routine (line 40):


```
10    rem circle3.bas
20    rem -----
30    clearw 2:fullw 2
40    c=3.141592/180
50    input "around what midpoint (x/y) ";xm,ym
60    input "radius ";radius
70    for grad=1 to 360
80    x=radius*cos(grad)+xm
90    y=radius*sin(grad)+ym
100   linef x,y,x,y
110   next grad
```

3.1.4 Ellipses

Since ellipses are closely related to circles, we can use the routines above to create ellipses, ovals and similar shapes.

To stretch our circles into ellipses, another factor must be integrated into our conversion equations:

```
10    rem ellipse.bas
20    rem -----
30    clearw 2:fullw 2
40    xm=320:ym=200:c=3.141592/180 rem color ym=100
50    radius =40
60    for grad=1 to 360
70    x=radius*cos(grad*c)*3+xm
80    y=radius*sin(grad*c)+ym
90    linef x,y,x,y
100   next grad
```

VX and VY are responsible for the "distorting" the circle into an ellipse.

Don't use excessively large values in your ellipse programs. Numbers between 0 and 10 should be sufficient.

```
5   rem 3-1-4b
10  rem ellipse demo
20  rem -----
30  clearw 2:fullw 2
40  xm=320:ym=200:c=3.141592/180
50  input "radius";radius
60  input "distortion in x or y direction
    ";direction$
70  if direction$="x" then vx=-1
80  if direction$="y" then vy=-1
90  for grad=1 to 360
100 for factor=.5 to 3.5 step .5
110 x=radius*cos(grad*c)
120 if vx then x=x*factor
130 x=x+xm
140 y=radius*sin(grad*c)
150 if vy then y=y*factor
160 y=y+ym
170 linef x,y,x,y
180 next factor
190 next grad
200 for i=1 to 10000:next
210 end
```

3.2 Applications

The following pages deal with applications of two-dimensional graphics. Software used to graphically display business figures, spreadsheets and the like are probably the most widely-known graphics use. We'll introduce a complete package for creating pie charts, line graphs and bar graphs later in this section.

3.2.1 Pie charts

Lists of numbers are often difficult to work with—pictures are much easier to understand and comprehend at a glance. For these reasons, many businessmen use graphic charts to aid them in their work. Depending on the data and circumstances, specific graphs are chosen to best convey the idea behind the information, and not require a lot of mental calculation.

Line graphs are used to calculate projected figures and values. For instance, the sales department of a company might use the sales of each month over a longer period of time to construct a sales curve. Extending this line graph can help us see past sales trends, and project future sales:

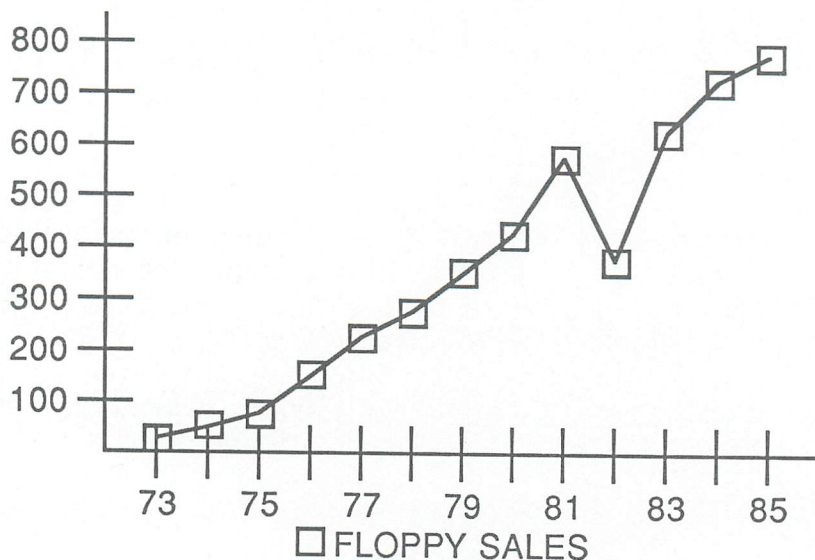


Figure 20: Line graph

Bar graphs and *pie charts* aren't usually used for such interpolation and extrapolation. They're usually used to simplify and clarify complex data. Unfortunately, the amount of data that can be displayed with bar graphs and pie charts is limited. A thirty-foot wall graphed with five hundred bars would look impressive, but would be more confusing than helpful. Because of this, such graphs often display figures that represent an average over a larger amount of time. Here too, the variable to be displayed (sales) depends upon another factor (time span).

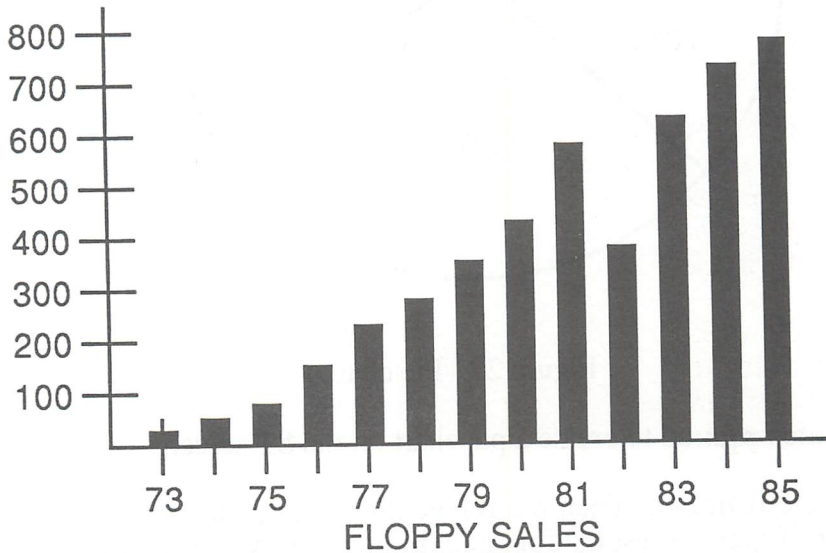


Figure 21: Bar graph

Pie charts are completely different from the others in that they do use numbers and statistics to display values. Instead, only the area of a circle can be used for illustration, so that all figures represent percentages of a whole.

Thus, pie charts are ideal for illustrating independent values. This chart can be used to represent the market shares among competing companies:

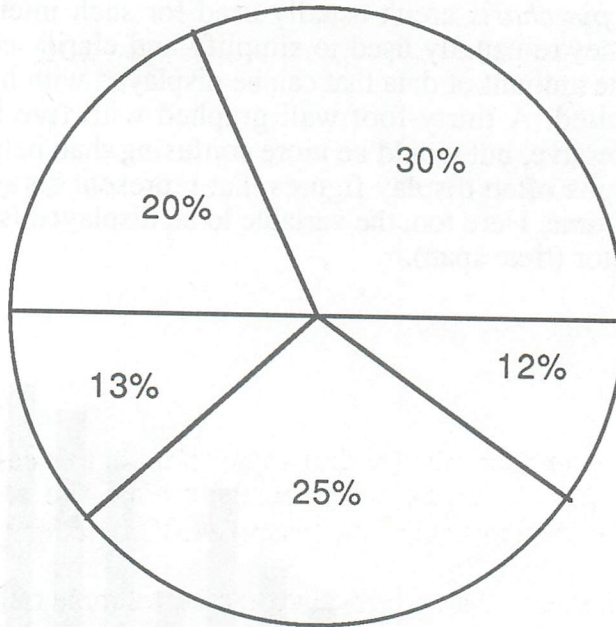


Figure 22: Pie chart

What steps are required to create a pie chart on the ST screen?

First the circle is drawn. Then the relevant data must be transferred to the program for proper representation, and divided into various sections of the drawn circle. Lastly, the sections of the circle are labeled to clarify their meaning of the chart.

So the general data flow of the program looks like this:

- Initialization
- Data entry
- Plotting
- Labeling

Since the initialization reserves memory for the data to be used, and since data arranged in a repeating pattern is most easily entered with a loop, the program would look like the following listing:

```

5      rem piechart.bas
50     xm=320:ym=200 rem for color ym=100
60     c=3.141592/180
1000   input:
1010   clearw 2:print "Enter the data"
1020   print: input"how many data items";items
1030   print:sum=0
1040   for i=1 to items
1050   print i;;input "value";value(i)
1060   sum =sum + value (i)
1070   print
1080   next i

```

We've already found a method for drawing circles. In this case, we want to draw segments starting at the center to the circle. The second plotting method should be used instead of the faster BASIC function `pcircle`.

To determine the size of the individual segments, a simple ratio can be used. The 360 degrees of a circle represent 100 percent—the sum of all individual data.

Accordingly, a single data item then uses a proportion of the total 360 as its percentage of the total 100:

$$\frac{\text{element value}}{\text{total sum}} = \frac{\text{element degrees}}{360 \text{ degrees}}$$

The number of degrees for an individual section is equal to (element value +360/total sum).

```

4000   circle diagram:
4010   clearw 2
4020   for i=1 to items
4030   grad(i)=int (value (i) *360/sum)
4040   next i

```

The field `value (i)` now contains the size of the chart's individual elements in degrees.

Line 4080 checks to see if the end of a sector has been reached.

If so, then a line is drawn to the center of the circle. This values (in degrees) is then added to the end value of the next element, to ensure that the size remains correct.

```

4050  r=120:i=1
4060  for grad=1 to 360
4070  x=r*cos(grad*c)+xm:y=r*sin(grad*c)+ym:
      linef x,y,x,y
4080  if grad(i)=grad then linefx,y,xm,ym:
      grad(i+1)=grad(i+1)+grad(i):i=i+1
4090  next grad
4100  end

```

The labeling could be performed each time an element of the chart is drawn (line 4080), although the format becomes rather sloppy this way. It doesn't look very good when a label covers the chart one time, and then is positioned off in some corner at another.

Ideally, a label should be positioned at about half the height of its section. As a result, the section's degree value is divided by two and used to position the label:

```

4030  grad(i)=int(value(i)*360/sum):
      textpos(i)=int(grad(i)/2)
4050  r=120: i=1: textalt=0
4080  if textalt + textpos(i)=grad then gosub
      labeling

```

The subroutine beginning at line 5000 is called when needed. It first calculates the length of the label. Then it checks to see if the label should be positioned on the left side of the circle (x,320) or on the right side.

This operation produces an "imaginary" graphics cursor that can be addressed from BASIC. Although it's possible to calculate the individual row and column positions, this subroutine is intended to be more universal. This means that different starting coordinates can be used, and the routine is applicable to later graphics programs. As you may have guessed, a VDI function is used.

The GEM-VDI function `v_gtext` (opcode 8) lets you to write a character string to the screen at any position. It is not restricted to the OUTPUT window. It requires a set of x and y coordinates to determine the starting point. The coordinates are passed in `ptsin(0)` and `ptsin(1)`.

The string itself is passed character by character in the `intin` array. The array `ctrl(3)` contains the length of the string passed in `intin`. The string must end with an ASCII zero. For this reason, a 1 is added to the length of the string specified in `ctrl(3)`.

```

5      rem 3-2-1b  placetxt.bas
10     rem calling from Gem function 8:v_gtext
20     rem -----
30     text$="test"
100    poke ctrl,8 :rem function number
110    poke ctrl+2,1
120    poke ctrl+6,len(text$)+1:
      rem length text output
130    poke ptsin,320:poke ptsin+2,200:
      rem coordinates
140    '
150    rem pokes text in array
160    for i=1 to len(text$)
170    poke intin+(i-1)*2,asc(mid$(text$,i,1))
180    next i
190    poke intin+(i-1)*2,0: rem terminating 0
200    vdisys

```

In the next example, the text is positioned in a circle. You must pay attention to the label's position in relation to the circle's center. For instance, if the text is positioned to the left of the circle, the cursor has to be moved left by the length of the string, plus one space. The following subroutine does this:

```

5000   labeling:
5010   x1=(len(title$(i)))
5020   if x>320 then x=x+32
5030   if x<320 then x=x-x1*8-40
5040   if y<200 then y=y+32
5050   if y>200 then y=y+32
5060   text$=title$(i)
5070   gosub v_gtext

```

Before the labels can be printed, they have to be entered into the program. The subroutine `v.gtext` is also added.

The complete program looks like this:


```
10    rem circdiag.bas
20    rem -----
30    clearw 2:fullw 2
40    dim value(20),grad(20)
50    xm=320:ym=200
60    c=3.141592/180
999    '
1000   entering:
1010   clearw 2:print "enter the data"
1020   print:input "how many data items";{{anz
1030   print:sum=0
1040   for i=1 to items
1050   print i;;input"value ";value(i)
1060   input "title";title$(i)
1070   sum=sum+value(i)
1080   print
1090   next i
3999   '
4000   circle_diagram:
4010   clearw 2
4020   for i=1 to items
4030   grad(i)=int(value(i)*360/sum):
      textpos(i)=int(grad(i)/2)
4040   next i
4050   r=120: i=1: textalt=0
4060   for grad=1 to 360
4070   x=r*cos(grad*c)+xm:y=r*sin(grad*c)+ym:
      linef x,y,x,y
4080   if textalt + textpos(i)=grad then gosub
      labeling
4090   if grad(i)=grad then textalt=grad
4100   if grad(i)=grad then linef
      x,y,xm,ym:grad(i+1)=grad(i+1)+grad(i):
      i=i+1
4110   next grad
4120   end
4999   '
5000   labeling:
5010   x1=(len(title$(i)))
5020   if x>320 then x=x+32
5030   if x<320 then x=x-x1*8-40
5040   if y<200 then y=y+32
```

```
5050  if y>200 then y=y+32
5060  text$=title$(i)
5070  gosub v.gtext
5080  return
5999  '
6000  v.gtext:
6010  for digit=1 to len(text$)
6020  poke
        intin+(digit-1)*2,asc(mid$(text$,digit,1))
6030  next digit
6040  poke intin+(digit-1)*2,0
6050  poke contrl,8
6060  poke contrl+2,1
6070  poke contrl+6,len(text$)+1
6080  poke ptsin,x
6090  poke ptsin+2,y
6100  vdisys
6110  return
```

3.2.2 Bar graphs

Rectangles are used to display bar graphs. Many computers have a BOX command in their BASIC command set. With this command, you specify the upper left and lower right corners of the desired rectangle.

Unfortunately, ST BASIC does not include such a command. But it's really no problem. Drawing four perpendicular lines isn't very difficult—at most, four coordinate pairs are specified. But why should you have to bother with all those coordinates if there's another, easier method?

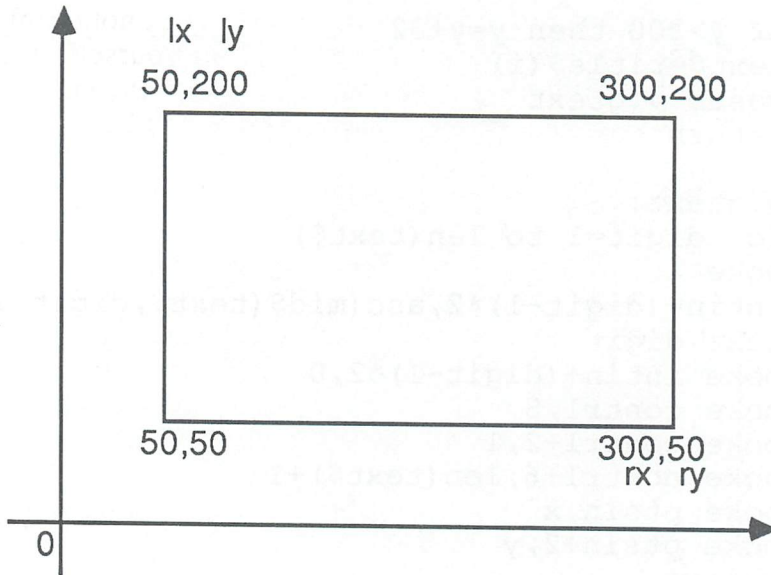


Figure 23: Two points define a rectangle

As you can see in Figure 23, the two missing points' coordinates can be replaced by (lx,ry) and (rx,ly), so that the following subroutine can be written:

```

5      rem drawbox.bas
10     clearw 2:fullw 2
20     input "upper left (x,y) ";lx,ly
30     input "lower right (x,y) ";rx,ry
40     linef lx,ly,rx,ly
50     linef rx,ly,rx,ry
60     linef rx,ry,lx,ry
70     linef lx,ry,lx,ly
80     goto 20

```

Using this routine instead of a built-in function doesn't noticeably reduce the execution time. As you may want to find out for yourself, the order in which you enter the two points makes no difference in the end result. Just make sure that you're using two diagonally opposite points.

A filled BOX can also be drawn quite easily:

```
5      rem fillbox1.bas
10     clearw 2:fullw 2
20     input "corner point 1 (x,y) ";lx,ly
30     input "corner point 2 (x,y) ";rx,ry
40     linef lx,ly,rx,ly
50     linef rx,ly,rx,ry
60     linef rx,ry,lx,ry
70     linef lx,ry,lx,ly
80     for f=ly to ry step 5
90     linef lx,f,rx,f
100    next f
110    goto 20
```

By changing the step value in line 130, you can control the density of the filled area. Unfortunately, this method does have its disadvantages. Because of the loop's structure, the vertices are entered from left to right.

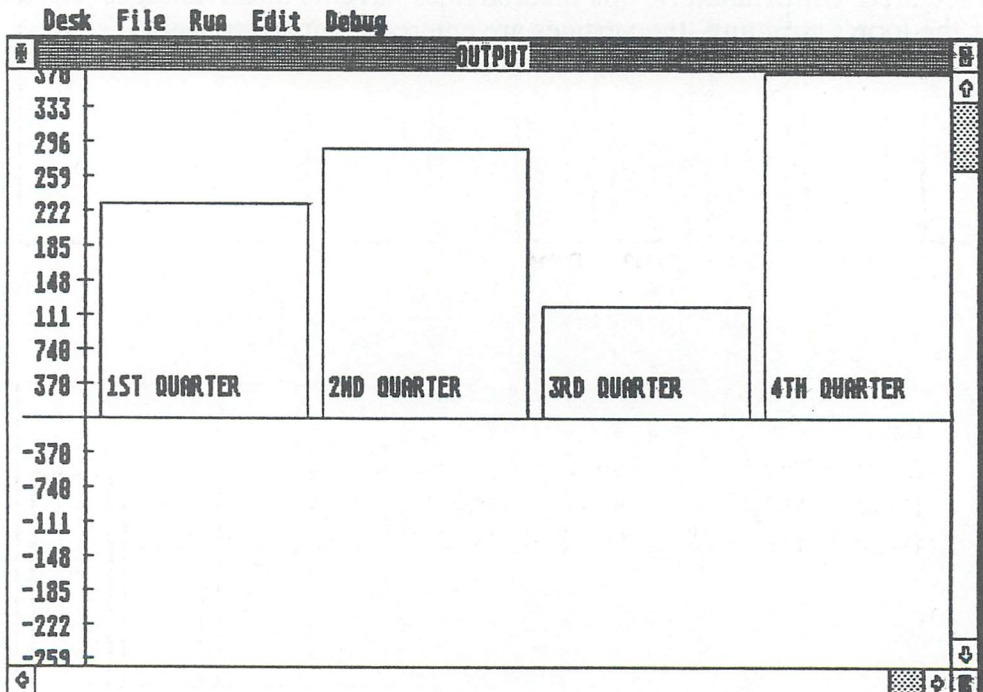
Luckily ST BASIC has the `FILL` command, which enables you to fill areas with various patterns. Only a single point within the area to be filled is specified as a parameter. This is quite easy to accomplish using the vertices:

```

5      rem fillbox2.bas
10     clearw 2:fullw 2
20     input "corner point 1 (x,y) ";lx,ly
30     input "corner point 2 (x,y) ";rx,ry
40     linef lx,ly,rx,ly
50     linef rx,ly,rx,ry
60     linef rx,ry,lx,ry
70     linef lx,ry,lx,ly
80     mx=(lx+rx)/2
90     my=(ly+ry)/2
95     color 1,1,1,1,1
100    fill mx,my
110    goto 20

```

Now you can extend the pie chart program to include the bar chart function. The format we want to use looks like this:



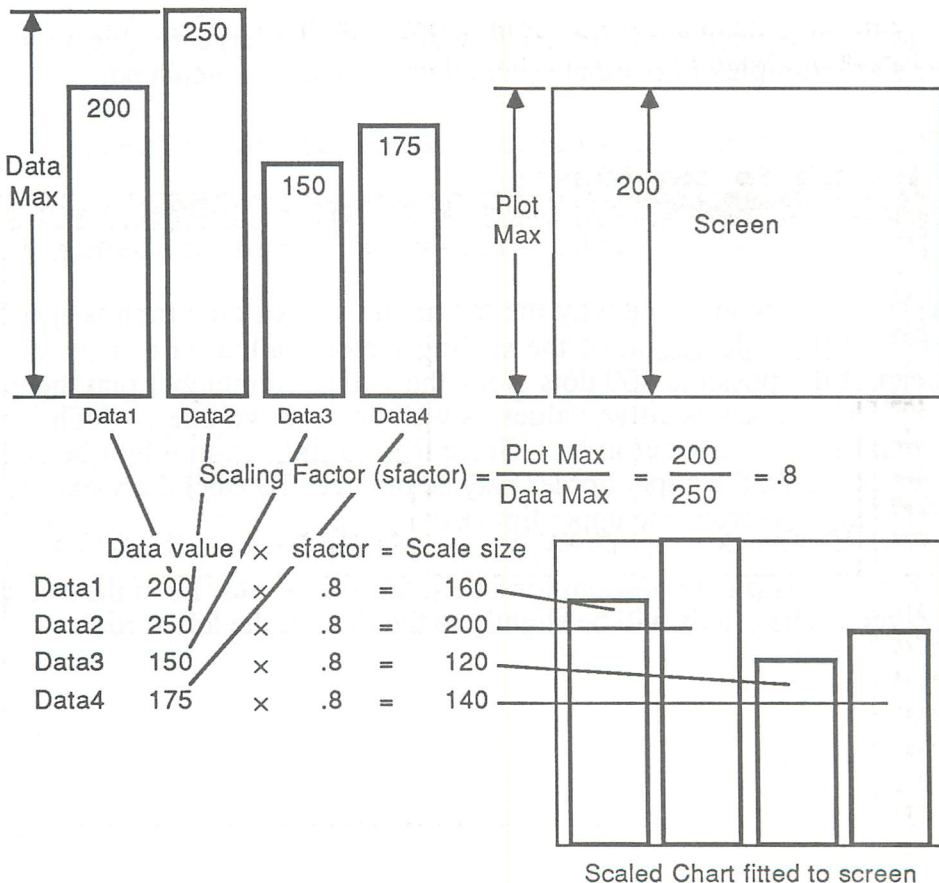
At this point we have the same problem as before: the individual sections have to be proportioned correctly.

Scaling is one of the most important processes in any sort of computer graphics. For this reason, the process is explained in more detail.

What actually takes place is a comparison between two sizes. Thus the chart is assigned an upper limit, such as for the pie chart (360 degrees). All the other values will now be compared to this limit.

A scaling factor is determined by dividing the available screen size by the maximum data size, to make the graph as large as possible:

$$SK = \frac{PLOTMAX}{DATAMAX}$$



To make the graph as large as possible, multiply each value of the graph by this scaling factor (SC). The product of scaling factor * graph value (from the zero-line of the graph) gives you the distance of the value to be displayed.

Before the the main program is executed, the coordinates of this zero-line are determined. The problem is simple since only the distance from this line and only two points have to be known. Here's an example of how the previous program may be altered:

```
5      rem 3-2-1f
40     dim value(20),grad(2),bvalue(20)
50     ym=200
1000   input:
1030   print:sum=0:wmax=0
```

These lines reserve the required memory.

An addition is needed in line 1070, which sets the largest value as vmax. Here's an example of how that scaling factor could be calculated:

```
1070   sum=sum+value(i):if value(i)>value(i-1)
      then vmax=value(i)

2010   clearw 2:mfactor=199/vmax:rx=60:distance=10
```

Maybe you're wondering why the maximum size of the graph is specified as 199 in the calculation of the scaling factor (called sfactor above) instead of the possible 400 dots along the y-axis. Remember that the graph should encompass positive values as well as negative ones, which is why the maximum size is cut in half. Since the multiplication might be slightly larger than 200, the upper border may be omitted. To label the y-axis, it is a good idea to decrease the upper limit even more.

So that there is plenty of room for labels, the x-axis must lie in the middle of the screen. The y-axis will be slightly to the right of the left border:


```
2020  linef 10,200,640,200:rem x axis:
      rem 10,200,400,200 for color monitor
2030  linef 50,0,50,400:rem y axis
```

Depending on how many bars are to be graphed, a loop must be used to calculate their length, and their width must be adjusted before the individual bars are drawn.

```
2040  for i=1 to items
2050  bvalue(i)=value(i)*mfactor
2060  next i
2070  ceilingwidth=580/num-distance
```

The bars are then drawn with the routine developed earlier.

```
2080  for i=1 to items
2090  ly=200:lx=rx+ceilingwidth:ry=200-bvalue(i)
2130  linef lx,ly,rx,ly:linef rx,ry,lx,ry
2140  linef lx,ry,lx,ry:linef lx,ry,lx,ly
2150  rx=rx+distance+ceilingwidth:rem begin the
      next ceiling
2160  next i
```

Now the axes have to be labeled. This is relatively simple for the x-axis, but the y-axis requires more effort.

The scaling factor has to be used if you don't want to use evenly spaced markers, but want to arrange them in accordance with the bars.

The value of the largest bar is equal to v_{max} , which is why one marker is as large as v_{max} , divided by the desired number of markers (10):

```
2170  ' labeling y axis
2180  x = 10 : for y1 = 1 to 10
2190  linef 55,200 + y1 * 20,45,200 + y1 * 20
2200  y = 260 + y1 * 20 - 15 : y2 = vmax/10 * y1*-1
2210  text$ = left$(str$(y2),4)
2220  gosub v.gtext
2230  next y1
```


The negative part of the graph can be supplied with a scale the same way:

```
2240 for y1 = 1 to 10
2250 linef 55,y1 * 20,45,y1 * 20
2260 y = 260 - y1 * 20 -15 : y2 = vmax / 10 * y1
2270 text$ = left$(str$(y2),4)
2280 gosub v.gtext
2290 next y1
```

The x-axis can be labeled like this:

```
2100 x = rx + 5 : y = 245 : if ry > 200
      then y = y + 20
2110 if ry < 200 then y = y - 20
2120 text$ = title$(i) : gosub v.gtext
```

Once you've entered these lines, you have a program that displays a series of bar graphs. All scaling functions and size adjustments are automatic.

Remember that even the ST and the high resolution monochrome monitor have their physical limits. Don't try to create a graph containing too much data.

```
10      rem bar chart
20      rem -----
30      clearw 2 :fullw 2
40      dim value(20),bvalue(20)
50      ym = 200
999      '
1000     input:
1010     clearw 2:print "input the data"
1020     print : input"how many data items ";items
1030     print : sum = 0 : vmax = 0
1040     for i=1 to items
1050     print i; : input"value ";value(i)
1060     input "title ";title$(i)
1070     sum = sum + value(i) : if value(i)
        > valuei-1) then vmax = value(i)
1080     print
1090     next i
2000     barchart:
2010     clearw 2 : mfactor = 199 / vmax :
        rx = 60 : distance= 10
2020     linef 10,200,640,200 : rem x-axis:
        rem 10,200,400,200 for color monitor
2030     linef 50,0,50,400 : rem y-axis
2040     for i=1 to items
2050     bvalue(i) = value(i) * mfactor
2060     next i
2070     barwidth = 580 / items - distance
2080     for i=1 to items
2090     ly=200 : lx = rx + barwidth :
        ry = 200 - bvalue(i)
2100     x = rx + 5 : y = 245 : if ry > 200
        then y = y + 20
2110     if ry < 200 then y = y - 20
2120     text$ = title$(i) : gosub v.gtext
2130     linef lx,ly,rx,ly : linef rx,ly,rx,ry
2140     linef rx,ry,lx,ry : linef lx,ry,lx,ly
2150     rx = rx + distance+ barwidth
2160     next i
2170     ' labelling y axis
2180     x = 10 : for y1 = 1 to 10
2190     linef 55,200 + y1 * 20,45,200 + y1 * 20
2200     y = 260 + y1 * 20 - 15 : y2 = vmax/10 * y1*-1
2210     text$ = left$(str$(y2),4)
```

```
2220 gosub v.gtext
2230 next y1
2240 for y1 = 1 to 10
2250 linef 55,y1 * 20,45,y1 * 20
2260 y = 260 - y1 * 20 -15 : y2 = vmax / 10 * y1
2270 text$ = left$(str$(y2),4)
2280 gosub v.gtext
2290 next y1
2300 e=inp(2)
2310 end
4999 '
5999 '
6000 v.gtext:
6010 for digit=1 to len(text$)
6020 poke intin + (digit - 1)
      * 2,asc(mid$(text$,digit,1))
6030 next digit
6040 poke intin + (digit -1 ) * 2,0
6050 poke contrl,8
6060 poke contrl+2,1
6070 poke contrl+6,len(text$)+1
6080 poke ptsin,x
6090 poke ptsin+2,y
6100 vdisys
6110 return
```

3.3 Manipulation of 2-D graphics

Up to this point only ST BASIC commands have been used. As the previous program demonstrated, good results can be obtained with these functions. The ability to set point enables you to do quite a bit with the graphics of the ST, although creating a whole picture would take too long and be too painstaking. Programs to create intricate graphics would be rather impossible to write in BASIC.

Mathematics can help you create these graphics. Mathematical guidelines and sample programs are presented in the remainder of Chapter 3. Step by step, we'll get closer to the ultimate graphics goal—three-dimensional pictures.

3.3.1 Shapes

Shapes can be found in many home computers as correspondingly-named commands. Unfortunately, the ST doesn't have many of these commands.

However, these functions can be emulated using software and the ST's ample memory. The 68000's high speed allows us to use software for these functions. In the following section you'll see some sample shapes created by the available plotting commands.

Basically, the description of a shape consists of its vertices (or corners) which are then connected with lines. The outline of a house could be detailed with these coordinates:

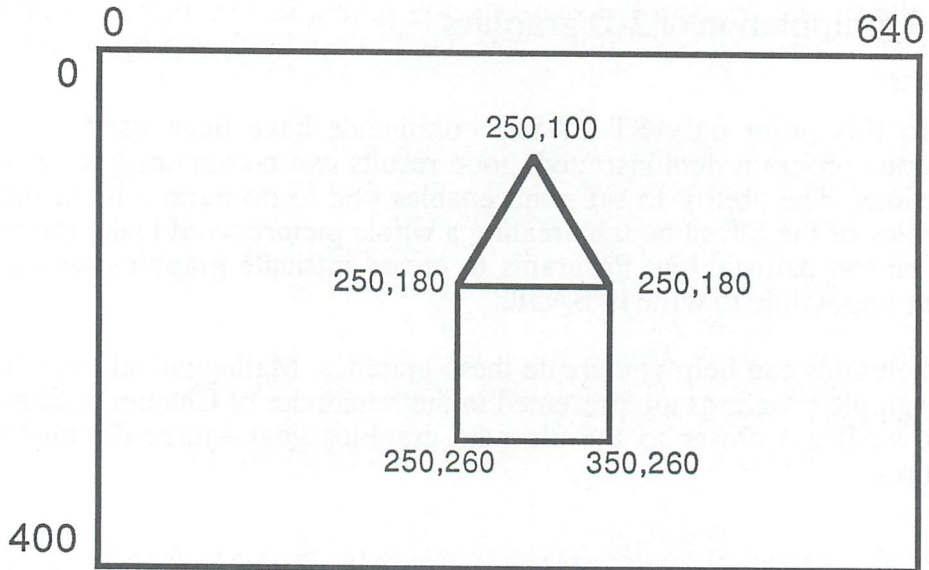


Figure 25: Definitions of a house in RC

To draw this house on the screen, the following program could be used:

```
10      fullw 2:clearw 2 rem house
15      r=2 :rem resolution  1=hires 2=medres
20      linef 250,260/r,350,260/r rem BOTTOM
30      linef 350,260/r,350,180/r rem RIGHT
40      linef 350,180/r,250,180/r rem TOP
50      linef 250,180/r,250,260/r rem LEFT
60      linef 350,180/r,300,100/r rem DIAG RT
70      linef 300,100/r,250,180/r rem DIAG LF
```

An alternative method is to specify the points in one block instead of separate lines. The individual lines would then be drawn by a separate routine:

```
10    fullw 2:clearw 2 rem house2
20    read xv,yv REM starting point
30    read xn,yn REM end point
40    linef xv,yv,xn,yn REM draw line
50    goto 20      REM read next line
1000  REM line data for house
1020  data 250,260,350,
      260,350,260,350,180,350,180,250,180
1030  data 250,180,250,
      260,250,180,300,100,300,100,350,180
```

Since the number of vertices is not known to the program, the program ends with an error message. When dealing with pre-defined shapes, this problem is easy to solve:

```
0      fullw 2:clearw 2 rem house3
20     read num  REM number of lines
30     for lines =1 to num
40     read xv,yv REM starting point
50     read xn,yn REM ending point
60     linef xv,yv,xn,yn REM draw line
70     next lines REM get next line
80     end
1000   REM data for house
1010   data 6
1020   data 250,260,350,260,
      350,260,350,180,350,180,250,180
1030   data 250,180,250,260,
      250,180,300,100,300,100,350,180
```

Since nothing but the shape data is contained in lines 1010 to 1030, the first data statement specifies the number of vertices that the object will have.

For drawing a single picture, this method might seem a little difficult, but it makes the exchange of pictures very easy:

```
10    fullw 2:clearw 2 rem house3a
20    read num  REM number of lines
25    if num<0 then end
30    for lines =1 to num
40    read xv,yv REM starting point
50    read xn,yn REM ending point
60    linef xv,yv,xn,yn REM draw line
70    next lines REM get next line
80    goto 20
999    '
1000  REM data for house
1010  data 7
1020  data 300,350,450,350,450,350,400,300
1030  data 400,300,400,200,400,200,375,150
1040  data 375,150,350,200,350,200,350,300
1050  data 350,300,300,350
1060  data -1
```

A marker to signify the program end is all that needs to be added. This program lets you create any desired number of shapes, and thus complete pictures.

A negative number can be used with ST BASIC, where the origin is always defined as (0,0). Choose a value outside of the usable area if you are using a coordinate system that accepts these values (e.g. LOGO).

```
5      rem house 4
10     fullw 2:clearw 2
20     read num    REM number of lines
25     if num<0 then end
30     for lines =1 to num
40     read xv,yv REM beginning point
50     read xn,yn REM ending point
60     linef xv,yv,xn,yn REM drawing lines
70     next lines REM get next line
80     goto 20
1000   REM the following is the picture data:
1010   data 6
1020   data 250,260,350,260,
      350,260,350,180,350,180,250,180
1030   data 250,180,250,260,
      250,180,300,100,300,100,350,180
1040   data 3
1050   data 280,260,280,220,
      280,220,320,220,320,220,320,260
1060   data 4
1070   data 260,210,290,210,290,210,290,190,
      290,190,260,190,260,190,260,210
1080   data 4
1090   data 310,210,340,210,340,210,340,
      190,340,190,310,190,310,190,310,210
1100   data -1
```


3.3.2 Coordinate conversion

The preceding program may be changed like this:

```
5      rem trans1.bas
6      r = 1 rem r = 2 for color
10     fullw 2:clearw 2
15     for factor=0.1 to 1.9 step 0.2
16     restore
20     read num
25     if num<0 then 80
30     for lines =1 to num
40     read xv,yv
50     read xn,yn
60     linef xv*factor,yv*factor/r,
        xn*factor,yn*factor/r
70     next lines
80     next factor
90     end
1000   REM the following is the picture data:
1010   data 17
1020   data 250,260,350,260,350,260,
        350,180,350,180,250,180
1030   data 250,180,250,260,250,180,
        300,100,300,100,350,180
1050   data 280,260,280,220,280,220,
        320,220,320,220,320,260
1070   data 260,210,290,210,290,210,290,190
        290,190,260,190,260,190,260,210
1090   data 310,210,340,210,340,210,340,190,
        340,190,310,190,310,190,310,210
1100   data -1
```

As you can see, this program produces a whole row of houses in different sizes.

To accomplish the scaling (increase/decrease in size) simply multiply by a constant factor. This is a coordinate conversion achieved through the use of matrices.

The following sections present more possibilities available to you.

3.3.2.1 The coordinate system

With simple calculations in the normal x-y coordinates, shifting shapes, increasing or decreasing their size and also mirror images can be done. We'll have to use more than just the first quadrant of the coordinate plane to observe these functions. For this reason, the origin of the following programs is positioned in the middle of the screen, and to simplify things, the axes are drawn as well.

Since there is no such command in BASIC, the next main problem is how to move the origin to the middle of the screen, Remember how the sizes of the values in the business program were adjusted? Well, here we can use a very similar procedure to change the coordinate system's range to -320 through 320 and -200 through 200. This way the origin will be located in the center of the screen. Just as we used the relationship between PLOTMAX and DATAMAX in bar graphs, we'll relate the length of the standard x and y values to the ones of the new coordinate system (XMIN, XMAX, YMIN, YMAX).

$$\text{For } x: \frac{(X - XMIN)}{(XMAX - XMIN)}$$

$$\text{For } y: \frac{(YMAX - Y)}{(YMAX - YMIN)}$$

To bring these relative coordinates to full scale, the x and y values still have to be multiplied by the maximum resolution factor (640 by 400).

Also, the desired parameters must be specified for the values of XMIN, XMAX, YMIN, and YMAX in the formulas above. To obtain the coordinate system desired in this example, these values are:

```
20 xmin = -320 : xmax = 320
30 ymin = -200 : ymax = 200
```

The conversion formulas would look like this:

```
X = 640 * (X-XMIN) / (XMAX-XMIN)
Y = 400 * (YMAX-Y) / (YMAX-YMIN)
```

To accomplish these tasks, two short subroutines can be written, one of which converts the position of a single point, the other converts the position of two points (which constitute a line). They can be activated with `gosub transpoint` and `gosub transline`.

```
10000 transpoint:
10010 x = 640 * (x - xmin) / (xmax - xmin)
10020 y = 400 * (ymax - y) / (ymax - ymin)
10030 return

10050 transline:
10060 xv = 640 * (xv - xmin) / (xmax - xmin)
10070 yv = 400 * (ymax - yv) / (ymax - ymin)
10080 xn = 640 * (xn - xmin) / (xmax - xmin)
10090 yn = 400 * (ymax - yn) / (ymax - ymin)
10100 return
```

Now it's quite simple to display the new coordinate system on the monitor.

```
5      rem quadrant
10     clearw 2:fullw 2
20     xmin = -320 : xmax = 320
30     ymin = -200 : ymax = 200
40     r = 1 rem r=2 for color
110    xv = -300 : yv = 0:xn = 300 : yn = 0 :
        gosub transline:
120    linef xv,yv/r,xn,yn/r
130    xv = 0 : yv = -200 : xn = 0 : yn = 200 :
        gosub transline:
140    linef xv,yv/r,xn,yn/r
150    end
10000  transpoint:
10010  x = 640 * (x - xmin) / (xmax - xmin)
10020  y = 400 * (ymax - y) / (ymax - ymin)
10030  return
10050  transline:
10060  xv = 640 * (xv - xmin) / (xmax - xmin)
10070  yv = 400 * (ymax - yv) / (ymax - ymin)
10080  xn = 640 * (xn - xmin) / (xmax - xmin)
10090  yn = 400 * (ymax - yn) / (ymax - ymin)
10100  return
```

The axes are calculated in lines 110 and 130.

3.3.2.2 Parallel shifting

Parallel shifting is the simplest kind of conversion of the standard coordinate system. It's accomplished by shifting axes without changing their directions:

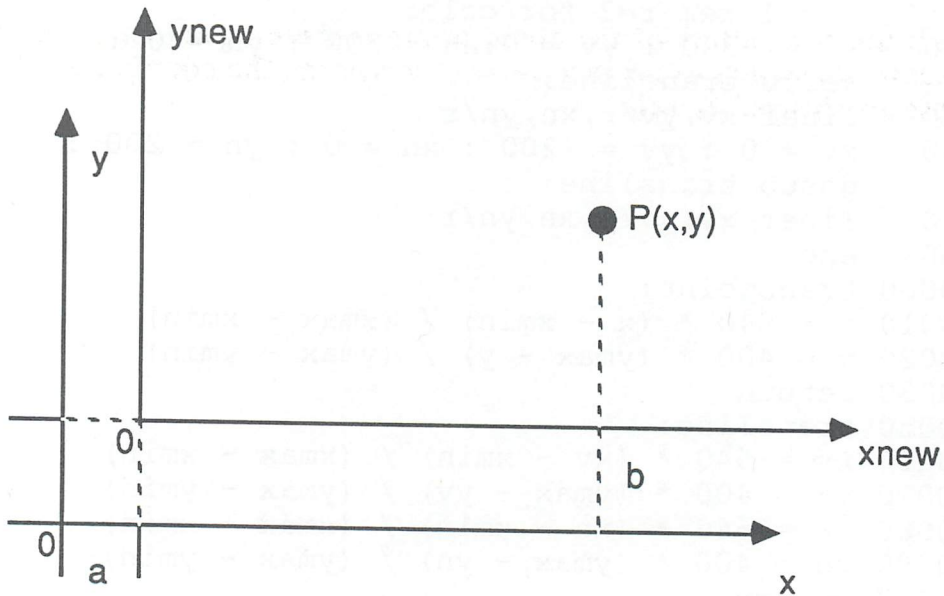


Figure 26: Parallel shifting

The figure above shows that the extent of the movement depends on the values of a and b .

The relationship between the original coordinate system and the new one can be expressed like this:

$$x_{\text{new}} = x - a$$

$$y_{\text{new}} = y - b$$

Since it is not really the axes which we want to shift, but the point(s) on the coordinate plane, the correct formulas are:

$$x_{\text{new}} = x + a$$

$$y_{\text{new}} = y + b$$

Using these simple conversion equations, you can shift any point in the plane and if you wish to work with a third dimension, the following formula would apply for the third axis:

$$z_{\text{new}} = z + c \quad (z_{\text{new}} = z - c)$$

We'll use the outline of the house to prove these equations. Note that because we switch to the new coordinate system, the coordinates of the house are changed as well:

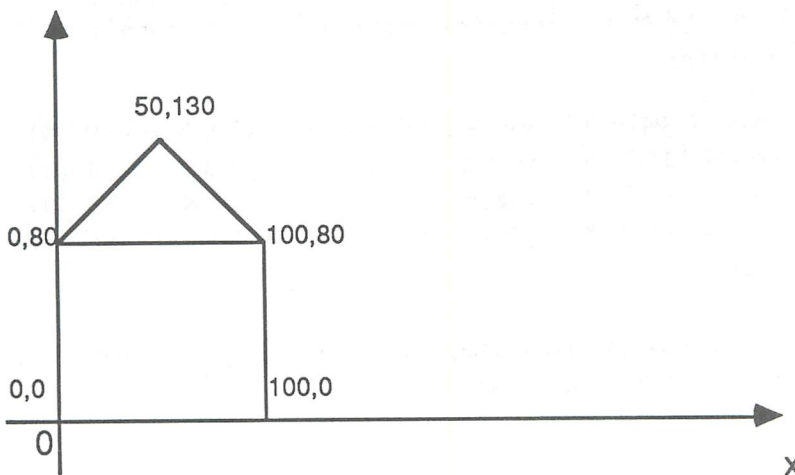


Figure 27: Parallel shifting

```
5      rem pashift
10     clearw 2:fullw 2
20     xmin = -320 : xmax = 320
30     ymin = -200 : ymax = 200
40     r = 1 rem r = 2 for color
110    xv = -300 : yv = 0:xn = 300 : yn = 0 :
        gosub transline:
120    linef xv,yv/r,xn,yn/r
130    xv = 0 : yv = -200 : xn = 0 : yn = 200 :
        gosub transline:
140    linef xv,yv/r,xn,yn/r
150    rem
160    read num REM number of lines
170    for lines =1 to num
180    read xv,yv REM starting point
190    read xn,yn REM ending point
195    gosub transline
200    linef xv,yv/r,xn,yn/r REM draw line
210    next lines REM get next line
220    end
230    REM data for house
240    data 6
250    data 0,0,0,80,0,80,50,130,50,130,
        100,80,100,80,100,0
260    data 100,0,0,0,100,80,0,80
10000  transpoint:
10010  x = 640 * (x - xmin) / (xmax - xmin)
10020  y = 400 * (ymax - y) / (ymax - ymin)
10030  return
10050  transline:
10060  xv = 640 * (xv - xmin) / (xmax - xmin)
10070  yv = 400 * (ymax - yv) / (ymax - ymin)
10080  xn = 640 * (xn - xmin) / (xmax - xmin)
10090  yn = 400 * (ymax - yn) / (ymax - ymin)
10100  return
```

3.3.2.3 Rotation about the origin

This type of rotation is almost as simple to accomplish as parallel shifting an object in degrees. It involves the simultaneous rotation of both coordinate axes around the origin (0,0) on a single plane. The angle θ (theta) determines the amount of rotation in degrees.

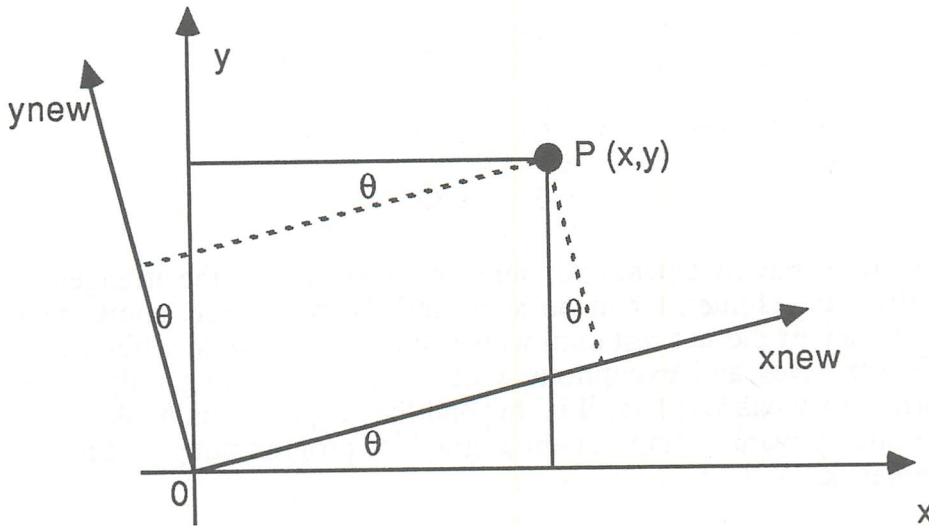


Figure 28: Turning

If you've read about the drawing circles, you'll have no problem understanding the relationship between points x and x_{new} :

$$x_{new} = x * \cos(\theta) + y * \sin(\theta)$$

$$y_{new} = -x * \sin(\theta) + y * \cos(\theta)$$

3.3.3 Matrices

Both shifting and rotation are linear conversions. These are easily displayed in matrices. Here the numbers are arranged in rows and columns. Each element of the matrix can be identified by its position in the pattern. This is identical to the notation in a BASIC array, except that the commas are omitted.

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

The matrix is merely a pattern of numbers in which only the arrangement of its individual elements can be specified. Nevertheless, some inspired mathematicians did not rest until whole shelves of university libraries were filled with rules and exceptions about matrices. Listing only the most important of these would still be beyond the scope of this book. Anyway, you probably want to find out about graphics programming, not advanced mathematics.

For our special applications, the addition and multiplication of square matrices will be most important. Both operations are demonstrated in the following examples incorporating the matrices M1 and M2:

$$M1 = \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix} \qquad M2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

When adding (subtracting) matrices, the numbers in the respective row and column are added (subtracted). Thus the sum of two matrices is:

$$\begin{bmatrix} -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 \end{bmatrix}$$

Multiplication of matrices might seem quite awkward at first, but isn't very difficult either. The elements in the row of the first matrix are multiplied by the elements in the column of the second matrix. The results obtained are then added:

$$\begin{bmatrix} -1 & * & 0 \end{bmatrix} + \begin{bmatrix} 0 & * & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & * & 1 \end{bmatrix} + \begin{bmatrix} 0 & * & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & * & 1 \end{bmatrix} + \begin{bmatrix} 0 & * & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & * & 1 \end{bmatrix} + \begin{bmatrix} 0 & * & 1 \end{bmatrix}$$

$$\begin{matrix} 0 & -1 \end{matrix}$$

$$\begin{matrix} -1 & 1 \end{matrix}$$

Expressed as a formula, this operation would look like this:

$$\begin{array}{ccccc} \begin{bmatrix} A & B \end{bmatrix} & \begin{bmatrix} a & b \end{bmatrix} & A*a+B*c & A*b+B*d \\ * & = & & \\ \begin{bmatrix} C & D \end{bmatrix} & \begin{bmatrix} c & d \end{bmatrix} & C*a+D*c & C*b+D*d \end{array}$$

A conversion such as the previous parallel shifting can be easily expressed as the product of two such matrices. In this case though, a 1 x 2 matrix is multiplied by a 2 x 2 matrix:

$$\begin{matrix} x & y & * & \begin{bmatrix} a & b \\ c & d \end{bmatrix} & = & a*x+c*y & b*x+d*y \end{matrix}$$

X and y are the coordinates of the original point, the sums on the right side represent xnew and ynew.

The nature of the conversion is controlled by the second matrix, as is demonstrated in the following program:

```
5      rem matrix demo
6      rem -----
10     clearw 2:fullw 2
18     ' new screen coordinate system
19     ' with four quadrants
20     xmin = -320 : xmax = 320
30     ymin = -200 : ymax = 200
50     start:
60     clearw 2
70     gosub axisintercept
80     gosub picture.read
90     gosub printing
99     '
100    menus:
110    gotoxy 0,0:print "MATRIX DEMO"
120    input "a ";a : input "b ";b
130    input "c ";c : input "d ";d
140    clearw 2
150    gotoxy 0,0:print "Matrix: "
160    print a,b : print c,d
170    gosub transform
180    input "Original picture ";e$:
    if left$(e$,1)="y" then goto start
190    gotoxy 0,0:for z=1 to 4
        :print string$(20," "):next:goto menus
200    picture.read:
210    restore : read num
220    for point=1 to num
230    read x,y : x(point) = x : y(point) = y
240    next point
250    return
299    '
300    axisintercept:
310    xv = -300 : yv = 0:xn = 300 : yn = 0 :
        gosub transline:
320    linef xv,yv,xn,yn : rem x-Axis
330    xv = 0 : yv = -200 : xn = 0 : yn = 200 :
        gosub transline:
340    linef xv,yv,xn,yn : rem y-Axis
350    return
```

```

399  '
600  printing:
610  for point=2 to num
620  xn = x(point):yn=y(point):
    xv= x(point-1):yv=y(point-1)
630  gosub transline: linef xv,yv,xn,yn
640  next point
650  return
699  '
700  transform:
710  for point=1 to num
720  x(point) = a * x(point) + c * y(point)
730  y(point) = b * x(point) + d * y(point)
740  next point
750  gosub axisintercept
760  gosub printing
770  return
9999  '
10000 transpoint:
10010 x = 640 * (x - xmin) / (xmax - xmin)
10020 y = 400 * (ymax - y) / (ymax - ymin)
10030 return
10050 transline:
10060 xv = 640 * (xv - xmin) / (xmax - xmin)
10070 yv = 400 * (ymax - yv) / (ymax - ymin)
10080 xn = 640 * (xn - xmin) / (xmax - xmin)
10090 yn = 400 * (ymax - yn) / (ymax - ymin)
10100 return
20000 data 8,0,0,0,80,50,130,100,
    80,100,0,0,0,0,80,100,80

```

When you run the `Matrixdemo` program, the axes are drawn first. Then the data defining the house's outline is read to the subscripted variables `x` and `y` (lines 200 through 230) and the original picture is drawn.

In the routine menu, the cursor is positioned at 0,0 and you are asked to input `a`, `b`, `c`, and `d`. Then the resulting matrix is displayed before the new picture is drawn.

Enter different values in the matrix, and try to discover any patterns. It's best to limit the input values between -2 and 2. Don't just enter random numbers—take a systematic approach.

3.3.3.1 Changing size

As you have discovered previously, the multiplication of all x values by a constant factor caused the picture to increase in size along the x-axis (the same goes for the y-axis). To decrease the size of a picture, simply use a negative factor. If you take a look at the matrix in this respect, you might discover that these factors are identical to a and d. (Enter zero for b and c):

$$\begin{matrix} & & [a & 0] \\ x & y & * \\ & & [0 & d] \end{matrix} = a*x+0*y, 0*x+d*y = a*x, b*y$$

Thus, a and d determine the size of the new picture. The variable a is responsible for the change in size along the x-axis, whereas d is responsible for the y-axis. Values between 0 and 1 result in a smaller picture, numbers larger than one will increase the picture's size.

The following figure shows a house with double of its normal width and one-third of its height.

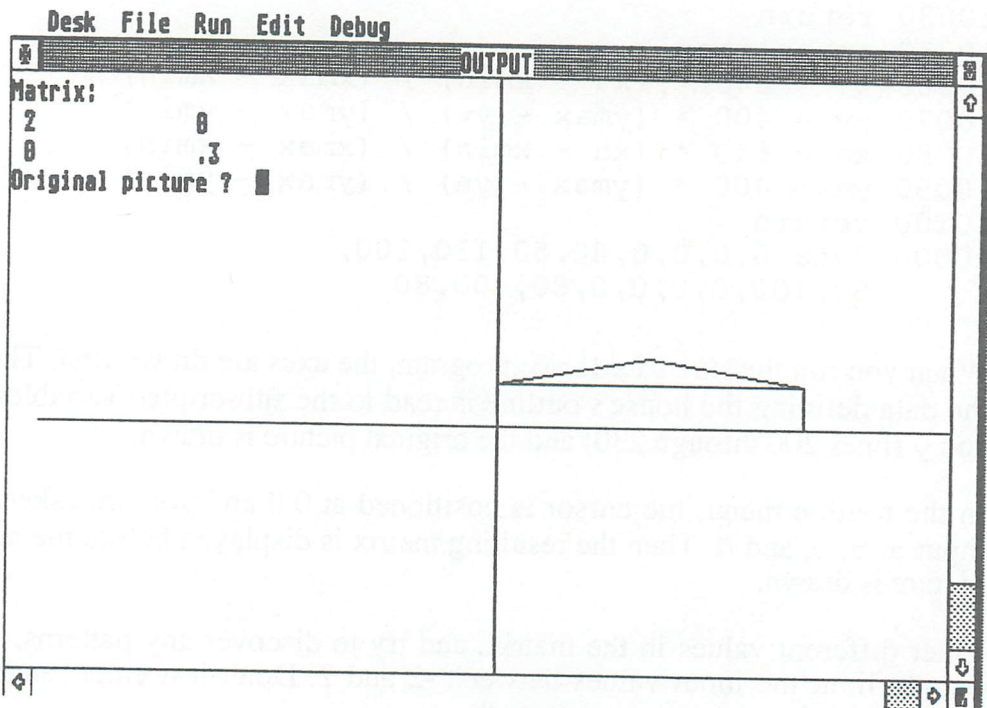


Figure 29: Conversion where $M = \begin{pmatrix} 2 & 0 \\ 0 & 0.3 \end{pmatrix}$

3.3.3.2 Reflections

In matrix operations, there is also an identity matrix. When both factors are one, the new picture is identical to the original.

Depending on the sign of the matrix, the result is a reflection along either the x-axis, the y-axis, or through the origin (compare with Figures 28 and 29).

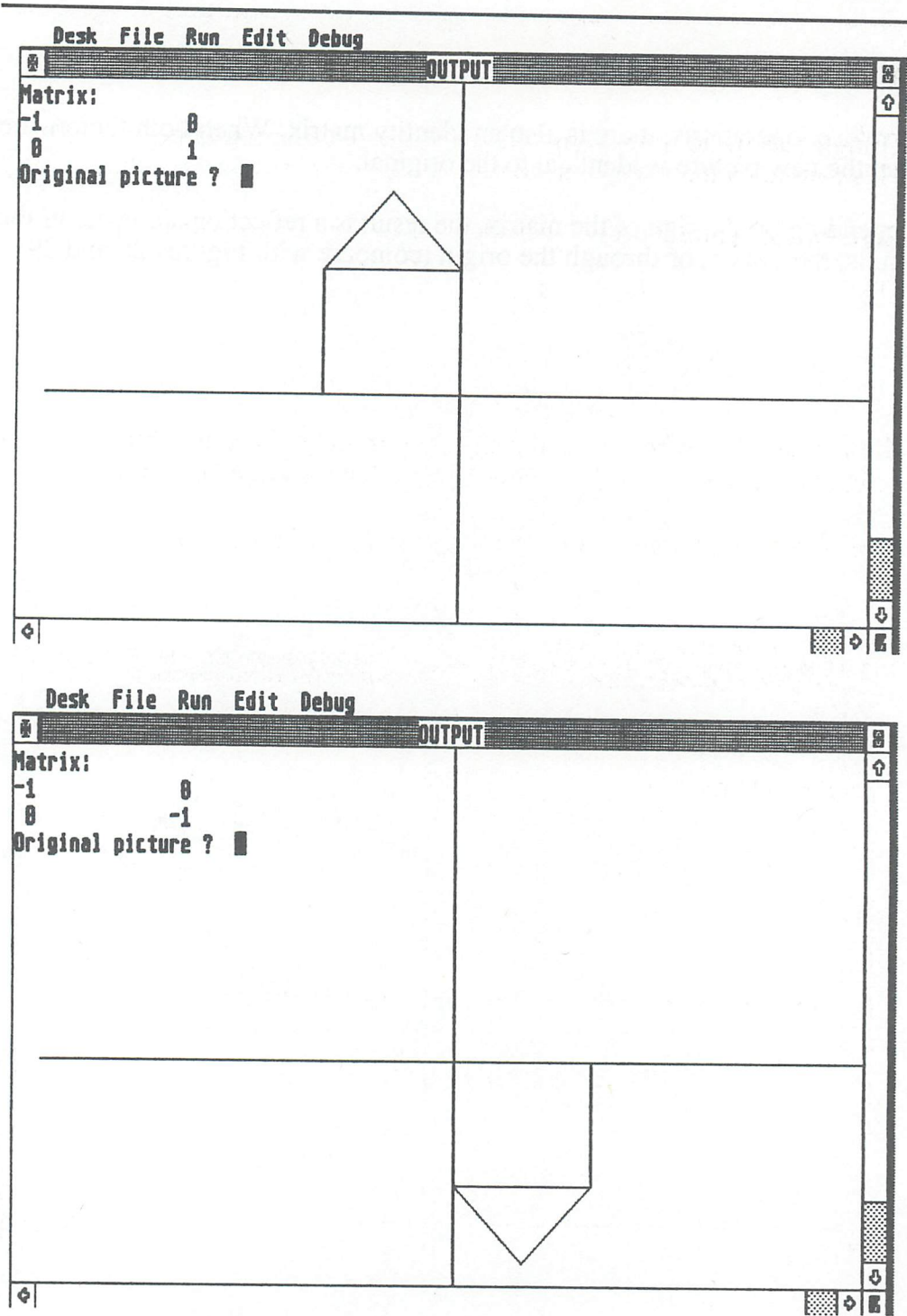


Figure 30: Results of $-1 \ 0 \ 0 \ 1$ and $-1 \ 0 \ 0 \ -1$

3.3.3.3 Oblique graphics

Things gets quite interesting when you enter different values for b and c . This causes pictures to "lean" to one side of the screen or the other.

Here too, the pattern used to calculate the individual coordinates is specified:

$$\begin{matrix} x & y & * \\ & \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \end{matrix} = \begin{matrix} x, & b*x+y \end{matrix}$$

Since a and b are both one, the scale will remain the same, but because of b , the larger the y value, the more distorted the x value becomes.

The same thing happens with the y coordinates when c is not zero.

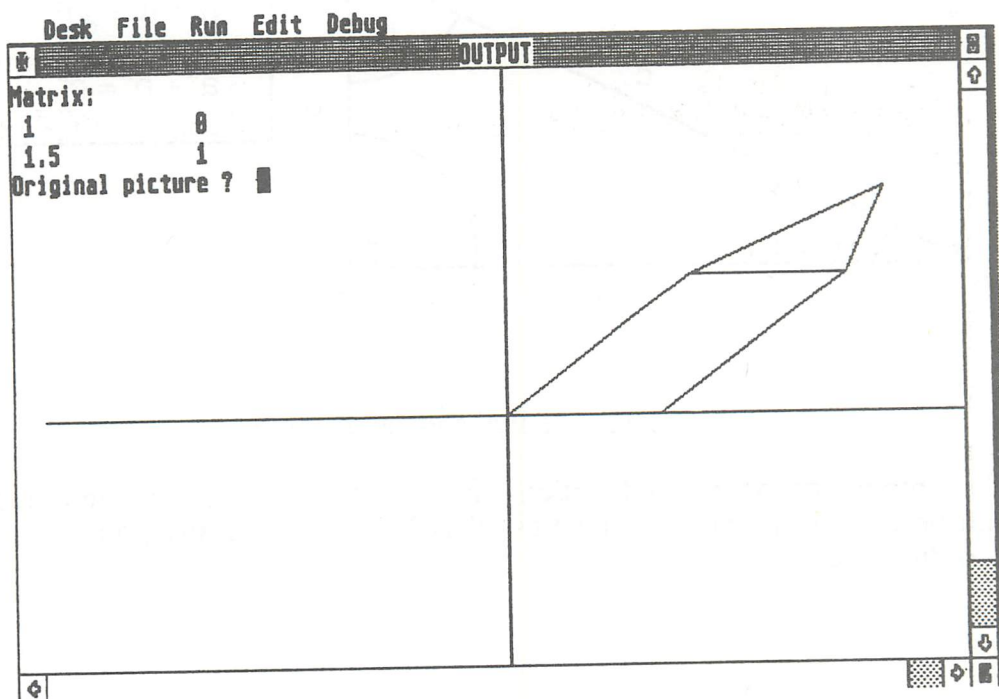


Figure 31: Results of matrix 1 0 1.5 1

3.3.3.4 Matrix Rotation

One basic technique used to manipulate illustrations we haven't talked about yet is the matrix rotation of the object about an axis (often the origin). Actually you've already made such a conversion—the method used to create a circle simply rotates a single point around the origin (centerpoint) at a certain distance (radius).

The matrix for this rotation function is derived from trigonometry.

This refers to the measuring and calculation of triangles, which proves to be especially simple when you're dealing with right triangles. For one, the sum of the two acute angles must always be 90 degrees, and there is also the Pythagorean theorem.

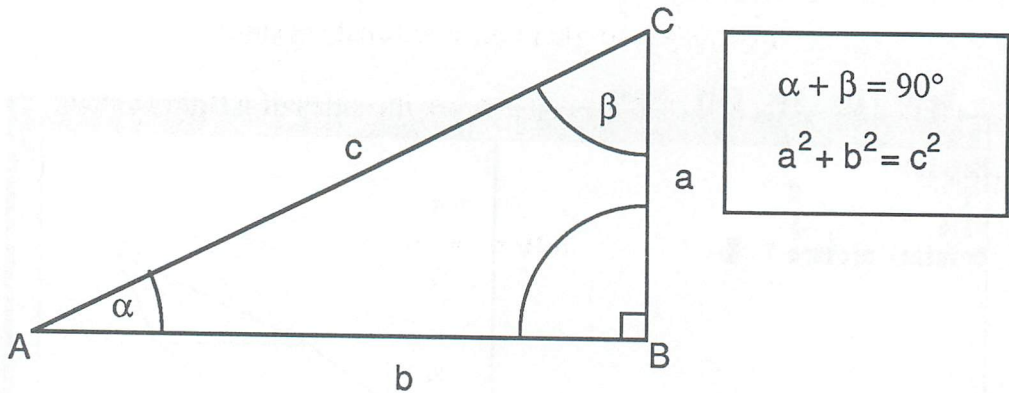


Figure 32: The Pythagorean theorem

To apply the trigonometry functions, any figure consisting of straight lines can be broken down into triangles. This holds true for any point on the coordinate plane.

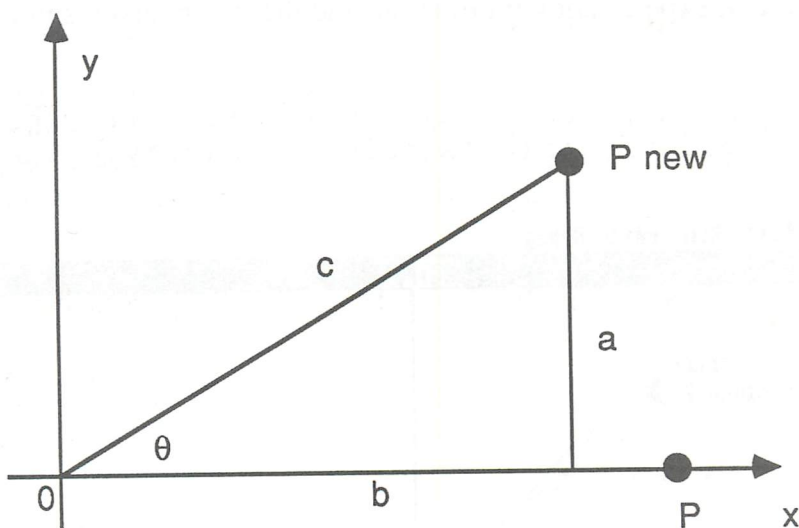


Figure 33: Angle in the coordinate system

The functions SIN and COS allow us to use the sides of a right triangle.

Thus:

$$\sin\theta = \frac{a}{c}$$

$$\cos\theta = \frac{b}{c}$$

θ (theta) is the desired angle of rotation, a represents a part of the y -axis, and b is part of the x -axis.

The matrix looks like this:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

If you'd like to experiment with rotation, add these conversions in lines 720 and 730:

```
720 xnew(point)=cos(a)*x(point)+(-sin(c))*y(point)
730 ynew(point)=sin(b)*x(point)+cos(d)*y(point)
```

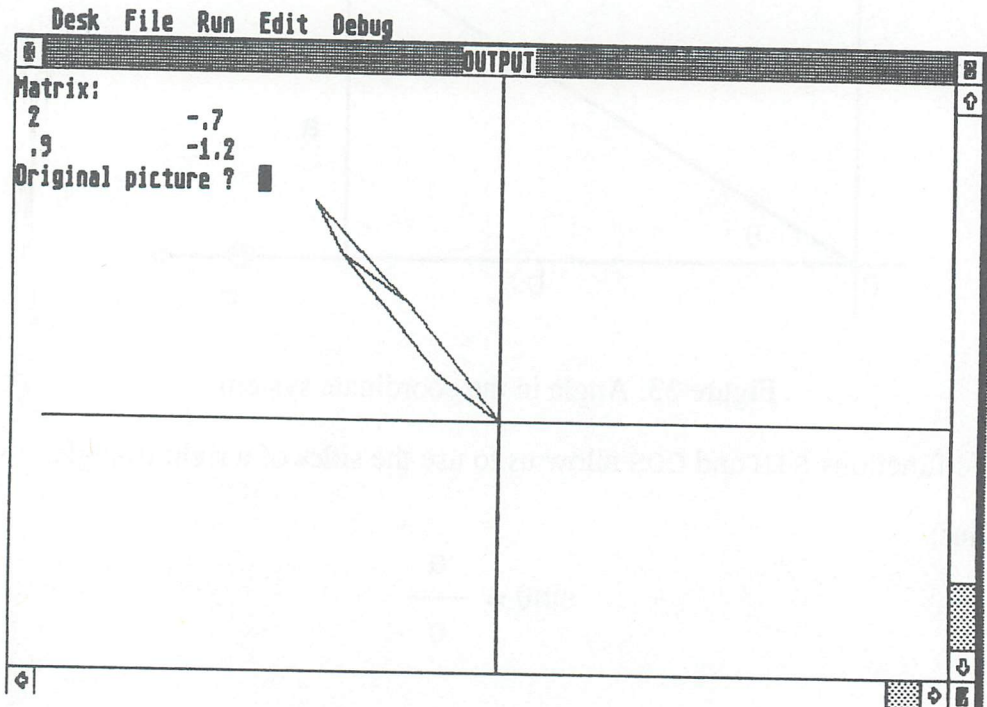


Figure 34: Hardcopy of Matrixdemo

3.3.3.5 Applications

The shapes, matrices, and conversion equations you've just learned are the heart of many impressive graphics programs.

These graphics are usually created by rotating a simple object, such as a triangle, square, or polygon, around one point:


```

10    rem computergraphic 3 ]
20    rem square; rotate & enlarge
30    rem -----
40    clearw 2:fullw 2:res=1 rem res=2 for color
50    x1 = 100 : x2 = 400 : x3 = 400 : x4 = 100
60    y1 = 300 : y2 = 300 : y3 = 100 : y4 = 100
70    sw = 5
80    for value=1 to 100
90      x = x + sw : y = y + sw
100     x1 = x1 - sw : x2 = x2 - sw :
        x3 = x3 + sw : x4 = x4 + sw
110     y1 = y1 + sw : y2 = y2 - sw :
        y3 = y3 - sw : y4 = y4 + sw
120     linef x1,y1/res,x2,y2/res
130     linef x2,y2/res,x3,y3/res
140     linef x3,y3/res,x4,y4/res
150     linef x4,y4/res,x1,y1/res
160     next value

```

The determining factors for this graphic program are the original points (lines 50 and 60), the coordinate origin, the number of repetitions, and the step size. Changing only one value results in a totally different picture!

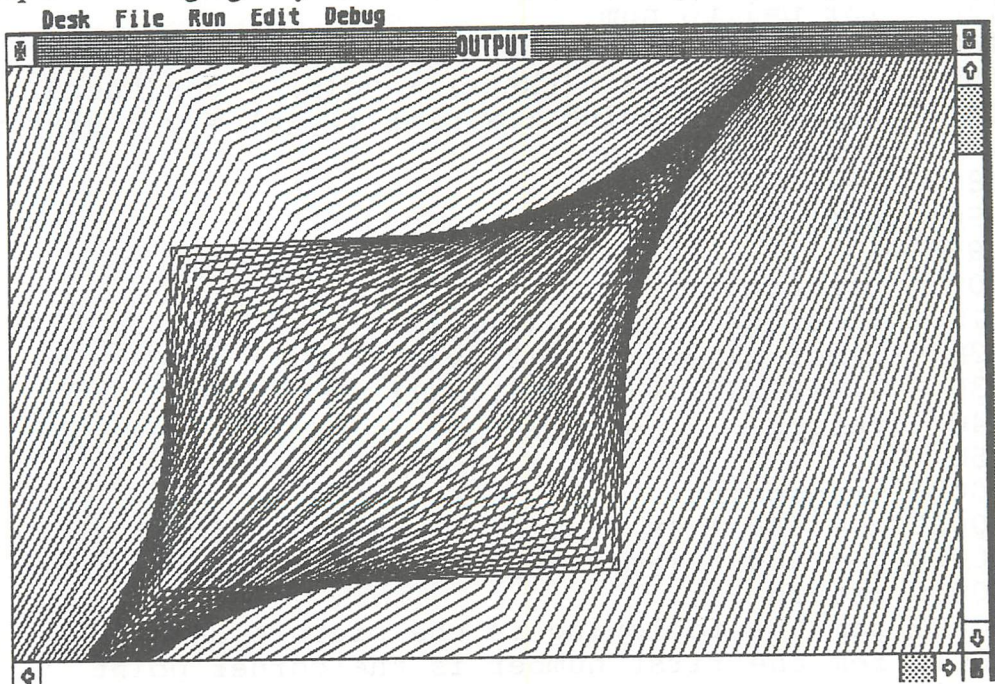


Figure 35: Hardcopy of GRAPHIC3.BAS

This program is rather inflexible and difficult to experiment with. Because the data isn't contained in an array, and the conversion formulas are more difficult to identify. This program is quite different in those respects:

```
10    rem computer graphic 4
20    rem many cornered; rotated and exaggerated
30    rem -----
35    rem coordinate increments in the center of
    the screen
40    xmin=-320:xmax=320:ymin=-200:ymax=200
45    '
49    ' reading the data
50    clearw 2 : fullw 2 : read num
60    for p=1 to num
70    read xo(p),yo(p)
80    next p
89    '
90    rem coordinates transformation
100   rem rotate hom many and how often
110   for theta=1 to 180 step 3
119   '
120   rem calculating all corner points
130   for p=1 to num
140   x = xo(p) * cos(theta) - yo(p) * sin(theta)
150   y = xo(p) * sin(theta) + yo(p) * cos(theta)
160   gosub transpoint : x(p)=x : y(p) = y : next p
170   gosub printing
180   next theta
190   end
199   '
200   printing:
210   linef x(1),y(1),x(2),y(2)
220   linef x(2),y(2),x(3),y(3)
230   linef x(3),y(3),x(1),y(1)
240   return
250   '
10000 transpoint:
10010 x = 640 * (x - xmin) / (xmax - xmin)
10020 y = 400 * (ymax - y) / (ymax - ymin)
10030 return
20000 rem the following is the picture data
20010 rem the first number is the corner point
20020 data 4,-100,-100,100,-100,100,100,-100,100
```

The data containing the shapes which are to be rotated and scaled, is at the end of the program. So that the values are read correctly, the number of vertices is specified first. The coordinates are read and assigned to the variables `x0` and `y0` in the loop of lines 60 through 80. The angle of rotation is determined in line 110, and the lines 140 to 150 contain the familiar conversion equations.

As you might notice, a rotation is performed in this program which in itself results in some interesting graphics. Should you get tired of these pictures, you can enlarge or shrink the shape step by step.

Another change can be achieved with the `transpoint` routine, since the origin doesn't have to be in the center of the screen.

Don't forget that you can also change the original shape. Try this:

```
20020 data 4,-180,100,180,100,180,100,400,180
```

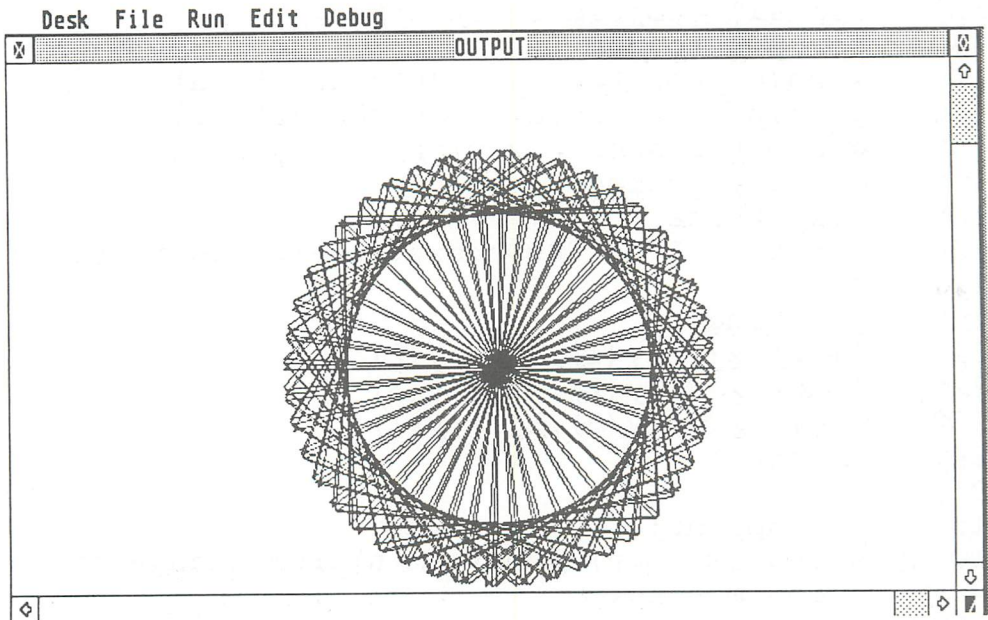


Figure 36: Sample of `GRAPHIC4.BAS`
The varied graphic possibilities of shape rotation is demonstrated by the

The varied graphic possibilities of shape rotation is demonstrated by the following program:

```
10    rem spiral graphic
20    rem rotated around the middle coordinates
30    rem -----
35    rem place coordinates in the middle of the
    screen
40    xmin=-320:xmax=320:ymin=-200:ymax=200:fullw 2
50    dim xo(15),yo(15),x(15),y(15)
59    ' how many corner points
60    start:
70    clearw 2:num=rnd(1)*15
79    ' generate the end of the screen
80    for p=1 to num: xo(p)=rnd(1)*320-160:
    yo(p)=rnd(1)*200-100:next p
90    rem coordinates transformation
99    rem rotate how many and how often
100   thetamax=rnd(1)*320:sw=rnd(1)*10
110   for theta=1 to thetamax step sw
119   '
120   rem all endpoints converted
130   for p=1 to num
140   x=xo(p)*cos(theta)-yo(p)*sin(theta)
150   y=xo(p)*sin(theta)+yo(p)*cos(theta)
160   gosub transpoint:x(p)=x:y(p)=y:next p
170   gosub printing
180   next theta
190   for time=1 to 8000:next:clearw 2:goto start
199   '
200   printing:
210   linef x(1),y(1),x(2),y(2)
220   linef x(2),y(2),x(3),y(3)
230   linef x(3),y(3),x(1),y(1)
240   return
250   '
10000 transpoint:
10010 x=640*(x-xmin)/(xmax-xmin):rem change for
    color monitor
10020 y=200*(ymax-y)/(ymax-ymin):rem change for
    color monitor
10030 return
```


3.4 Functions

Probably the most common use of 2-D graphics is to display functions. Here, the relationship between two numbers, which can usually be expressed as a mathematical equation, is graphically represented.

A function was defined at the beginning of the 18th century as "a variable which depends on another variable." However, this definition was not exact enough for later mathematics. Important for a function aren't just the variables, but rather the arrangement of these variables.

Single numbers aren't the only thing which can be integrated in a function, but whole series of elements, often called sets (such as the set of integers, etc.), can be used. In other words:

A function pairs each element in one set with an element in another set.

For example, one set could consist of a number of cars, another set could contain a number of drivers of whom each could be paired with a car. Unfortunately, this does not really represent a function, since each person is not assigned a specific car. This problem is solved by pairing each car with the currently registered owner.

Let's call the cars x , and the drivers y . This allows us to come up with the following definition of a function:

A function is the representation of two sets, or a set of ordered pairs (x,y) , in which there is exactly one y for each x .

In mathematical notation it looks like this:

$$y = f(x)$$

x are the elements of the known set, and y are the elements of the set to be calculated.

3.4.1 Plotting 2-D functions

Most students learn at least three ways to graphically depict functions.

The *function graph* is often quickly forgotten after grade school. Function graphs represent the sets with circles or ovals, and the pairs between sets are signified with arrows or lines. A function would have only one arrow going from each of its elements, even though more than one arrow could be pointing to elements in the solution set.

Once this graph is drawn, it's quite simple to create a table of values. This table is constructed by listing the elements of one set in a vertical column, and then placing the pairs of those elements from the solution set next to them. In most cases, each of these ordered pairs can be found on the coordinate plane as point p . If all of the ordered pairs of a function are plotted on the plane, then you will see a picture of the function develop.

Usually the standard coordinate system is used for this, where the x -values are found on the horizontal axis, and the y -values are on the vertical. Depending on the domain and range of the function, a series of dots, a line, or a curve is created by the function.

3.4.2 A function plotter

We can utilize the above-mentioned technique in an ST program, while keeping in mind the ST's limitations.

The first step is to create a table of values:

```
10  FULW 2 : CLEARW 2
200 FOR x=1 TO 10
210 y = x * x
220 PRINT x;y
230 NEXT x
```

In line 200 the given set's values (domain) are specified—in this case, 1 through 10. Line 210 contains the function itself, and $f(x)$ is calculated here. Line 220 prints the table.

A simple sketch of the graph is already possible by changing line 220:

```
220 PRINT TAB(y) "*"
```

You'll see that this results in a parabola.

You'd use a similar method to display the function on paper. You chose a perpendicular coordinate system, where the x-values increase from left to right, and the y-values increase from the bottom to the top. You draw the axes first, and then plot the points of the function. Then you connect the points with a smooth curve.

As you know, the coordinate system of the ST is a little different from this type. Consequently, you first use the old conversion to create four quadrants on the screen. Since you've already drawn the axes before, you can simply add that routine and change line 220 so that it sets one single point:

```
20      rem fplot1
30      rem -----
40      clearw 2:fullw 2
50      xmin = -320 : xmax = 320
60      ymin = -200 : ymax = 200
70      gosub axisintercept
200     for x1=-10 to 10
210     x = x1 : y = x1 * x1
220     gosub transpoint
230     linef x,y,x,y
240     next x1
290     end
300     axisintercept:
310     xv = -300 : yv = 0:xn = 300 : yn = 0 :
        gosub transline:
320     linef xv,yv,xn,yn : rem x-Axis
330     xv = 0 : yv = -200 : xn = 0 : yn = 200 :
        gosub transline:
340     linef xv,yv,xn,yn : rem y-Axis
350     return
9999    '
10000   transpoint:
10010   x = 640 * (x - xmin) / (xmax - xmin)
10020   y = 400 * (ymax - y) / (ymax - ymin)
10030   return
```

```

10050 transline:
10060 xv = 640 * (xv - xmin) / (xmax - xmin)
10070 yv = 400/res * (ymax - yv) / (ymax - ymin)
10080 xn = 640 * (xn - xmin) / (xmax - xmin)
10090 yn = 400/res * (ymax - yn) / (ymax - ymin)
10100 return

```

When the program is executed, a parabola appears on the screen. It might be smaller than you expected, but you've already discovered how to make scale changes by multiplying by a constant factor.

Let's create the same parabola on a larger scale. Make the following change:

```

80  mfx=20 : mfy = 5
210 x = x1 * mfx : y = x1 * x1 * mfy

```

The resulting picture is larger, but the dots are so far apart, that the shape of the curve is quite difficult to recognize. We need more dots.

Simply change the step value of the loop in line 200 to STEP .1. The graph is now much more complete.

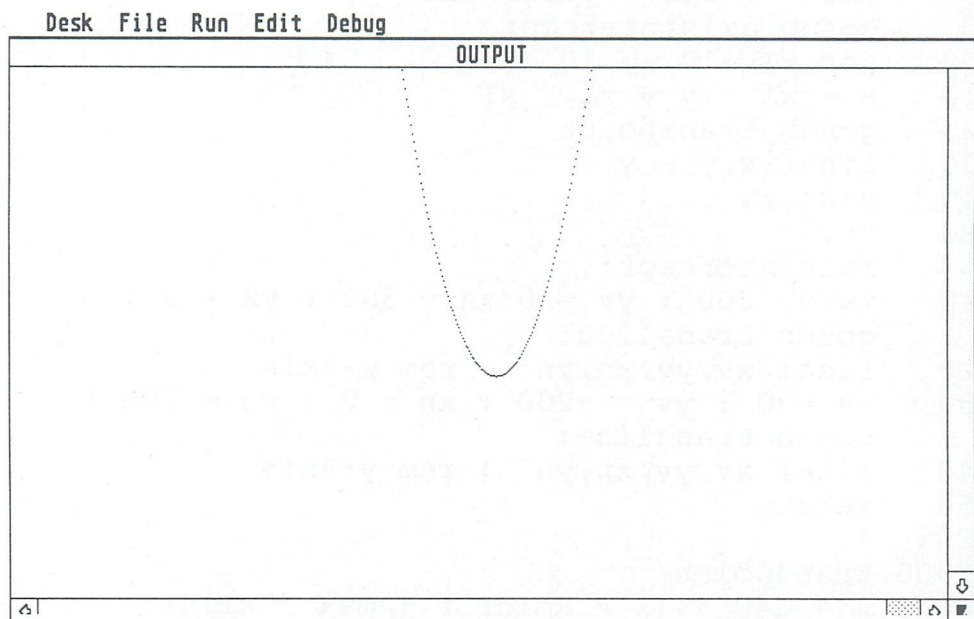


Figure 37: Standard parabola—plotted the fast way

There are two ways to create a solid, smooth curve in this graph. One is to let the computer calculate the remaining points on the curve between the known points. The other method is to connect the known points with lines.

The second method is very fast, but unfortunately very imprecise—the "curve" consists of a linked series of straight lines.

The first method is very precise, but takes a long time to process. That's because the computer has to calculate every point individually. You might want to verify this by changing lines 200 and 210:

```
200 FOR x=10 TO 10 STEP 0.01
210 y=SIN(x)
```

How about a compromise between the two methods? We can design the program so that you can determine the function curve's degree of accuracy, depending to your needs for precision.

Enter the following program and experiment with it. Try changing the function in line 10000 to something other than a standard parabola:

```
10    rem fplot2.bas
20    rem with auto scaling
30    rem -----
40    clearw 2
50    fullw 2
55    res = 1 rem res = 2 for color
60    '
70    ' input plot data
80    input "range from x= ";x: xmin = x:
    gosub function: ymin = x
90    input " up to x= ";x: xmax = x:
    gosub function: ymax = x
100   input "accuracy (0.1 - 0.01) ";accuracy
110   input "enlargement factor (1) ";mf
120   clearw 2:gosub cross
130   x1=xmin:gosub function:y=y*mf:x=x1*mf:
    gosub transpoint:xalt=x:yalt=y
140   for x1=xmin to xmax step accuracy
150   gosub function
160   y = y * mf : x = x1 * mf
170   gosub transpoint
```



```
180   linef xalt,yalt/res,x,y/res:xalt = x:yalt = y
190   next x1 : goto scaling
200   '
210   cross:
220   xv = -300 : yv = 0:xn = 300 : yn = 0 :
      gosub transline:
230   linef xv,yv/res,xn,yn/res : rem x-axis
240   xv = 0 : yv = -200 : xn = 0 : yn = 200 :
      gosub transline:
250   linef xv,yv/res,xn,yn/res :
      rem y-axis
260   return
270   '
280   scaling:
290   xmin = -320 : xmax = 320 :
      ymin = -200 : ymax = 200
300   for xa=-320 to 320 step 64
310   xv = xa : yv = 5 : xn = xa : yn = -5
320   gosub transline
330   linef xv,yv/res,xn,yn/res
340   next xa: end
350   '
10000 transpoint:
10010 x = 640 * (x - xmin) / (xmax - xmin)
10020 y = 400/res * (ymax - y) / (ymax - ymin)
10030 return
10040 transline:
10050 xv = 640 * (xv - xmin) / (xmax - xmin)
10060 yv = 400/res * (ymax - yv) / (ymax - ymin)
10070 xn = 640 * (xn - xmin) / (xmax - xmin)
10080 yn = 400/res * (ymax - yn) / (ymax - ymin)
10090 return
20000 function:
20010 y = sin(x1)
20020 return
```

The one thing still missing from this program is axes labeling, to make the graph clearer. There are several techniques you can use to accomplish this labeling—we'll explain one of them here.

First, you determine how many markers you want on each axis. In this case, we'll use ten in each direction. This means that a mark is required every 64 pixels along the x-axis. The distance between marks on the y-axis is 40 units.

Once you know this, you can construct a loop to draw these markers. The numerical value of each marker equals the number of graphic units divided by the scaling factor. Once this value is calculated, the labeling is accomplished through the graphics cursor.

Also, a different *character mode* must be activated (this is discussed in more detail in the next chapter). This mode involved a logical OR operation managing existing and newly-drawn graphics, to avoid erasing any of the function graph.

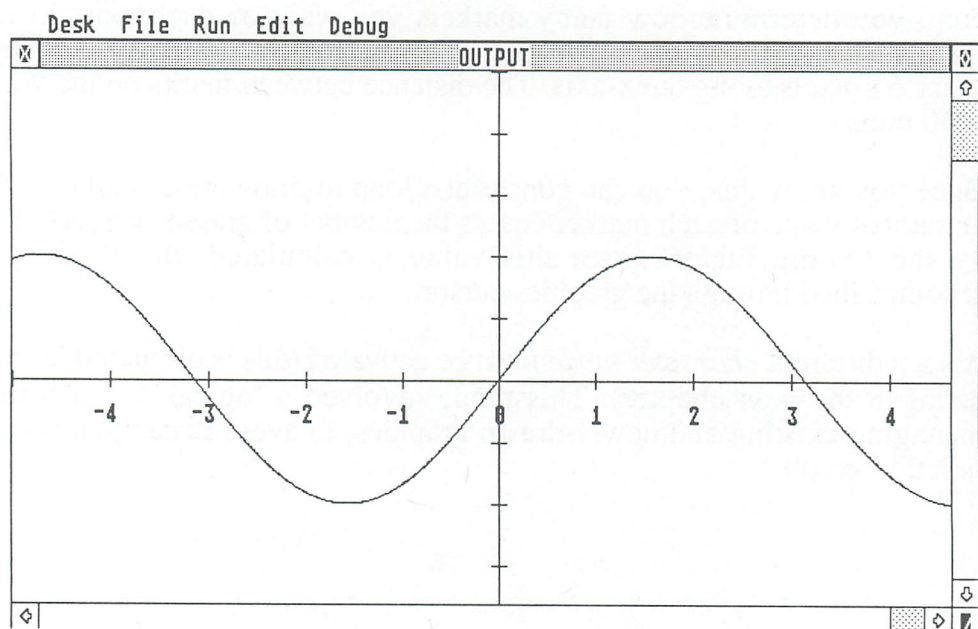


Figure 38: Sample hardcopy of plotter output

One last problem when calculating function values is the division by zero. To solve this problem, you could add a minute factor to each element of the domain (e.g. 0.00001). Alternatively, you can instruct the program to proceed with the next value when such an error occurs.

```
10    rem plotter.bas
20    rem with auto scaling
30    rem -----
40    clearw 2
50    fullw 2
55    r = 1    rem r = 2 for color
60    on error goto 10100 : rem in case of error
70    ' input plot data
80    input"range from x= ";x: xmin = x:
    gosub function : ymin = x
90    input" to x= ";x: xmax = x:
    gosub function : ymax = x
100   input"accuracy (0.1 - 0.01) ";accuracy
110   input"enlargement factor y ";mfy
120   clearw 2:gosub axiscross
130   x1=xmin:gosub function:y=y*mfy:x=x1:
    gosub transpoint:xalt=x:yalt=y
140   for x1=xmin to xmax step accuracy
150   gosub function
160   y = y * mfy : x = x1
170   gosub transpoint
180   linef xalt,yalt/r,x,y/r : xalt = x : yalt = y
190   next x1 : goto scaling
200   '
210   axiscross:
220   xv = -300 : yv = 0:xn = 300 : yn = 0 :
    gosub transline:
230   linef xv,yv/r,xn,yn/r : rem x-axis
240   xv = 0 : yv = -200 : xn = 0 : yn = 200 :
    gosub transline:
250   linef xv,yv/r,xn,yn/r : rem y-axis
260   return
290   '
300   scaling:
310   gosub transparent.modus
320   if xmin<0 then xmin = -xmin:
    diff = xmin + xmax
330   mark=0 : for xa=-xmin to xmax step diff/10
340   mark=mark+1
350   label$(mark)=str$(xa)
360   next xa
370   label$(1)="" : label$(11)="" :
    rem no writing on border
```



```
380   xmin = -320 : xmax = 320 :
      ymin = -200 : ymax = 200
390   mark = 0 : for xa=xmin to xmax step 64 :
      rem 64 = 10 partial marks
400   mark = mark +1
410   xv = xa : yv = 5 : xn = xa : yn = -5
420   gosub transline
430   linef xv,yv/r,xn,yn/r
440   text$=left$(label$(mark),6)
450   y = yn + 60 : x = xn - 10
460   gosub v.gtext
470   next xa
480   for ya = 40 to 200 step 40
490   yv= ya : yn =yv: xv = -5 : xn = 5
500   gosub transline
510   linef xv,yv,xn,yn : linef xv,400-yv,xn,400-yn
520   next ya
530   gosub replace.modus
540   end
5990   '
6000   v.gtext:
6010   for digit=1 to len(text$)
6020   poke intin + (digit - 1) * 2,
      asc(mid$(text$,digit,1))
6030   next digit
6040   poke intin + (digit - 1) * 2,0
6050   poke contrl,8
6060   poke contrl+2,1
6070   poke contrl+6,len(text$)+1
6080   poke ptsin,x
6090   poke ptsin+2,y
6100   vdisys
6110   return
6980   ' set write mode
6990   '
7000   transparent.modus:
7010   poke intin,2
7020   poke contrl,32
7030   poke contrl+2,0
7040   poke contrl+6,1
7050   vdisys 32
7060   return
7070   '
```

```
7080  replace.modus:
7090  poke intin,1
7100  poke contrl,32
7110  poke contrl+2,0
7120  poke contrl+6,1
7130  vdisys 32
7140  return
9990  '
10000 transpoint:
10010  x = 640 * (x - xmin) / (xmax - xmin)
10020  y = 400 * (ymax - y) / (ymax - ymin)
10030  return
10040 transline:
10050  xv = 640 * (xv - xmin) / (xmax - xmin)
10060  yv = 400 * (ymax - yv) / (ymax - ymin)
10070  xn = 640 * (xn - xmin) / (xmax - xmin)
10080  yn = 400 * (ymax - yn) / (ymax - ymin)
10090  return
10100 resume next          : rem just go on
19990 ' f(x1) set inline 20010
20000 function:
20010  y = sin(x1)
20020  return
```


Chapter 4

3-D Graphics

Three-dimensional Graphics

4.1 The third dimension

Over the years, many procedures for representing 3-D objects in a computer have been developed. Usually it is done by creating only an illusion of three dimensions—that's how we make two-dimensional characters appear to have depth.

The following pages discuss the various techniques used to represent and manipulate 3-D objects. First we'll work on the mathematical foundations and use that to build a 3-D function plotter—in essence, we'll develop a small CAD program.

We'll take a look at two methods of representing stereo-vision displays. However, the second method is limited to use on color monitors. Also, you must still use red-blue "3-D" glasses.

4.2 Perspective Drawings

The perspective drawing type 3-D graphic, a kind we see daily on TV and in photos, is the easiest to create on our ST. It doesn't require us to know a lot of technical information, only a little applicable mathematics. To create a perspective drawing, we take a 3-D object and project it onto a 2-D surface:

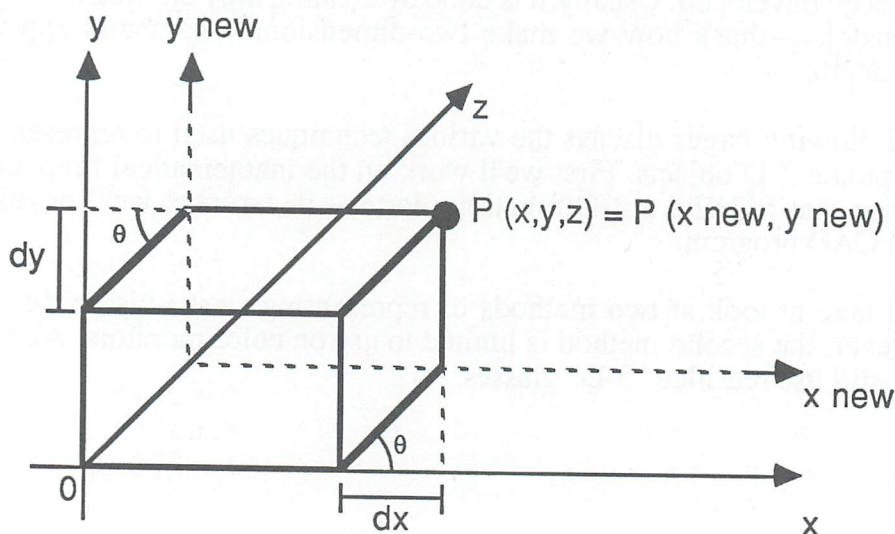


Figure 39: A 3-D object projection

We represent points on the third-axis, the z axis, by building them along the x and y axes. The picture makes it clear that representing the points involves a coordinate transformation. The point $P(x, y, z)$ in a three coordinate system becomes the point $P(x_{new}, y_{new})$ in the two-coordinate system.

Now we have to figure out how to actually implement this using previous knowledge. We learned a formula in the previous chapter that's useful here.

In this way we can keep the position of the z axis constant relative to the x axis and still represent points on both axes. Since we know the length of the line segment to x , we can transform the location of point $P(x, y, z)$ to point $P(x, y)$ using these formulas:

$$\begin{aligned}x &= X2 + z * \text{COS}(\text{THETA}) \\ y &= Y2 + z * \text{SIN}(\text{THETA})\end{aligned}$$

4.3 A 3-D function plotter

Using the previous formulas we can quickly write a function plotter. In two-dimensions, one variable is dependent on another. Now we'll calculate variable z which is dependent on both x and y .

We'll have the first program loop calculate different y values. Eventually we see that each of the three coordinates has its own weighting factor, so the functions display can be enlarged or shrunk:

```
10      'plot3d1.bas
20      '-----
30      clearw 2 : fullw 2
60      mfx =30 : mfy = 30 : mfz = 5 :
        cosine = cos(45) : sine = sin(45)
160     for y=-6 to 6 step .5
170     for x = -6 to 6 step 0.05
180     gosub 430
210     sx = int(x * mfx + y * mfy * cosine)
220     sy = int(z * mfz + y * mfy * sine)
230     linef sx + 320,200-sy,sx + 320, 200-sy
370     next x
380     next y
390     end
430     z = x*x + y * y
440     return
```

You'll recognize the transformation formulas on lines 210 and 220, where the screen coordinates are calculated. Line 430 has the function for a 3-D parabola. If you run the program you'll see a perspective of this parabola, as in Figure 40:

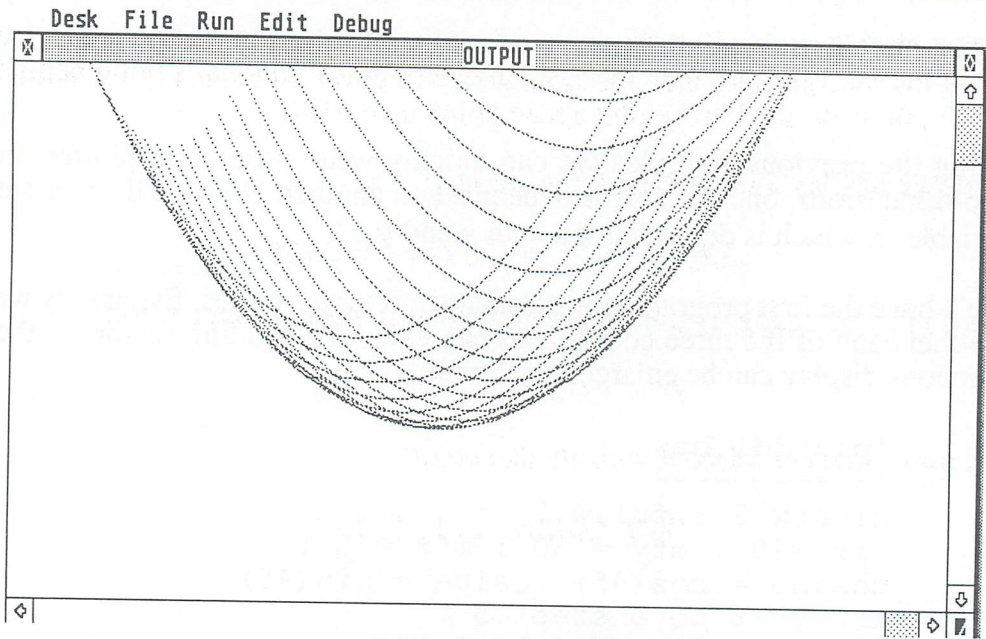


Figure 40: The normal 3-D parabola

The step size in line 160 and 170 determines how close the lines will be to each other. You can change the size of the display by changing the values on line 60. By changing parameters it is possible to make every function look crisp and clean on the screen. It can be annoying when the function curves are all mixed together.

4.3.1 Eliminating hidden lines

Let's rewrite the program again and pay attention to exactly how the individual points are plotted.

Note that it takes a little time before the points are displayed on the screen. You'll also notice that the picture is displayed point by point in the order left to right, and front to back.

Here a simple trick will help us out. If we first display the picture areas that are in the background, then the front area will cover all details lying behind it. All points on the line behind a new point are erased.

First, we display a line from plot position 1:

```
320 LINEF SX,SY-1,SX,400
```

We do this by changing the background color to the character color:

```
290 COLOR 1,1,0,1,1
```

and immediately replace it with another color:

```
340 COLOR 1,1,1,1,1
```

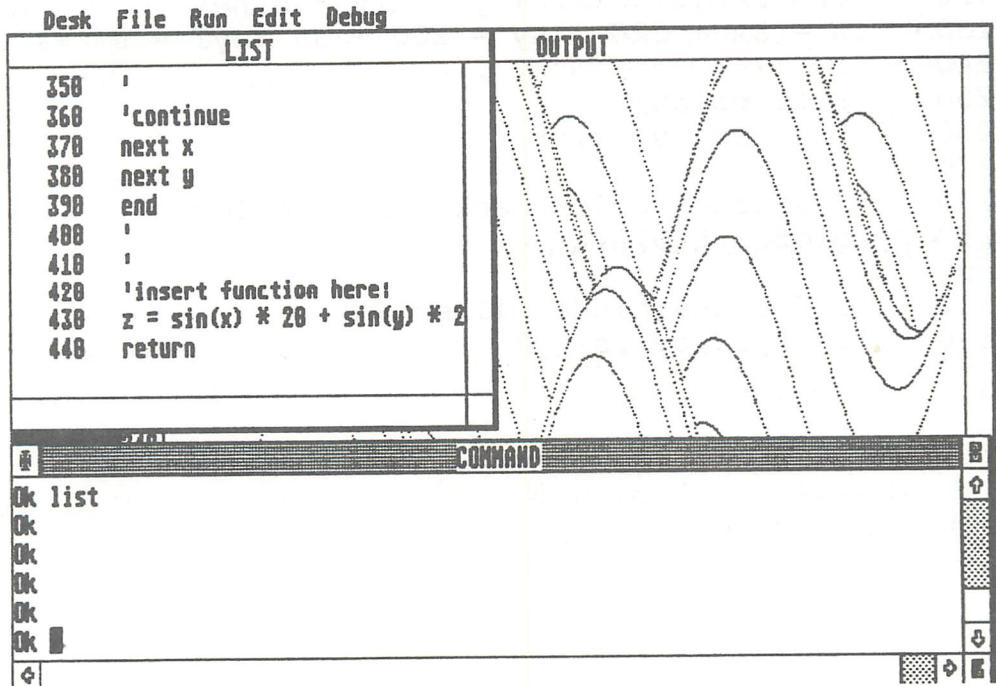


Figure 41: Hardcopy of 3-D function plotter

```
10      ' plot3d with hidden lines
20      ' -----
30      clearw 2 : fullw 2 : color 1,1,1,1,1
40      '
50      'Size factors
60      mfx =30 : mfy = 30 : mfz = 5
70      '
80      'Angle of sight
90      cosine = cos(45) : sine = sin(45)
100     y0 = 400 : 'lowest border of picture
110     '
120     'Help line for step size
130     'linef 1,200,640,200
140     '
150     'Calculation
160     for y=6 to -6 step -.5
170     for x = -6 to 6 step 0.03
180     gosub 430 : ' Find out function value
190     '
200     'Calculate the values of seven coordinates
210     sx = int(x * mfx + y * mfy * cosine)
220     sy = int(z * mfz + y * mfy * sine)
230     sx = sx + 320 : sy = 200 - sy :syl = sy +1
240     '
250     'Plot point
260     linef sx,sy,sx,sy
270     '
280     'Character color=back ground color
290     color 1,1,0,1,1
300     '
310     'and erase hidden lines
320     linef sx,syl,sx,y0
```

```
330   'From now on character color activities
340   color 1,1,1,1,1
350   '
360   'and continue
370   next x
380   next y
390   end
400   '
410   '
420   'Insert function here:
430   z = sin(x) * 20 + sin(y) * 20
440   return
```


4.4 3-D representation of real objects

The next step is to represent real objects on the screen in this manner. We can also do this with our formulas. The object to be represented must first be translated into a numerical model that can be manipulated.

To begin, we use one corner of the object as the zero point. All other points are calculated in relation to this point. Each point is entered as a set of coordinates relative to point $(0, 0, 0)$.

In this manner we can easily describe an object, say a house, as follows:

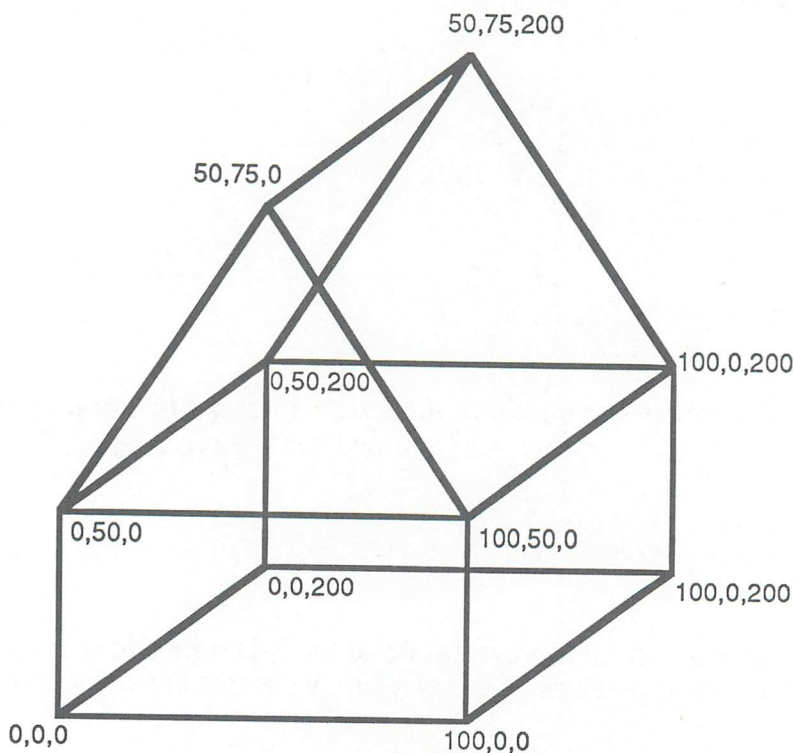


Figure 42: 3-D coordinates of demo house

These points must be given to the computer in such a way that a good representation of the object can be made on the screen. The points must be connected by lines, since the points by themselves don't make a picture. The lines actually compose the figure.

For this reason, we enter the starting and ending coordinates of each line. This not only lets us represent many objects, but also allows us to edit the picture easily. Each line can be lengthened, shortened, moved, or erased.

We can represent the house in seventeen lines using our numerical model:

```
810 data 0,0,0,10,0,0
820 data 10,0,0,10,5,0
830 data 10,5,0,0,5,0
840 data 0,5,0,0,0,0
850 data 10,0,20,10,5,20
860 data 10,5,20,0,5,20
870 data 0,5,20,0,0,20
880 data 0,0,20,10,0,20
890 data 10,0,0,10,0,20
900 data 0,0,0,0,0,20
910 data 0,5,0,0,5,20
920 data 10,5,0,10,5,20
930 data 10,5,0,5,7,0
940 data 0,5,0,5,7,0
950 data 5,7,0,5,7,20
960 data 5,7,20,0,5,20
970 data 5,7,20,10,5,20
```

If we then enter the line numbers, the input can be made into a variable table of the form `xfrom`, `yfrom`, `zfrom` and `xto`, `yto`, `zto`.

```
200 for l=1 to 17
210 read xv(l),yv(l),zv(l),xn(l),yn(l),zn(l)
220 next l
```

Now we have a natural representation of the 3-D object in memory. Before it can be drawn, the values of our screen x,y coordinates must be calculated.

That can be done with the transformation formula:

```
250 for l=1 to 17
260 sxv(l) = xv(l)*zv(l)*cos(45) :sxn(l) = xn(l)
    +zn(l)*cos(45)
270 syv(l) = yv(l) + zv(l) * sin(45): syn(l) =
    yn(l)+zn(l)*sin(45)
280 next l
```

Now we can draw the picture line by line:

```
320 for l=1 to 17
330 xv=sxv(l):yv=syv(l):xn=sxn(l):yn=syn(l)
340 gosub transline
350 linef xv,yv,xn,yn
360 next l
```

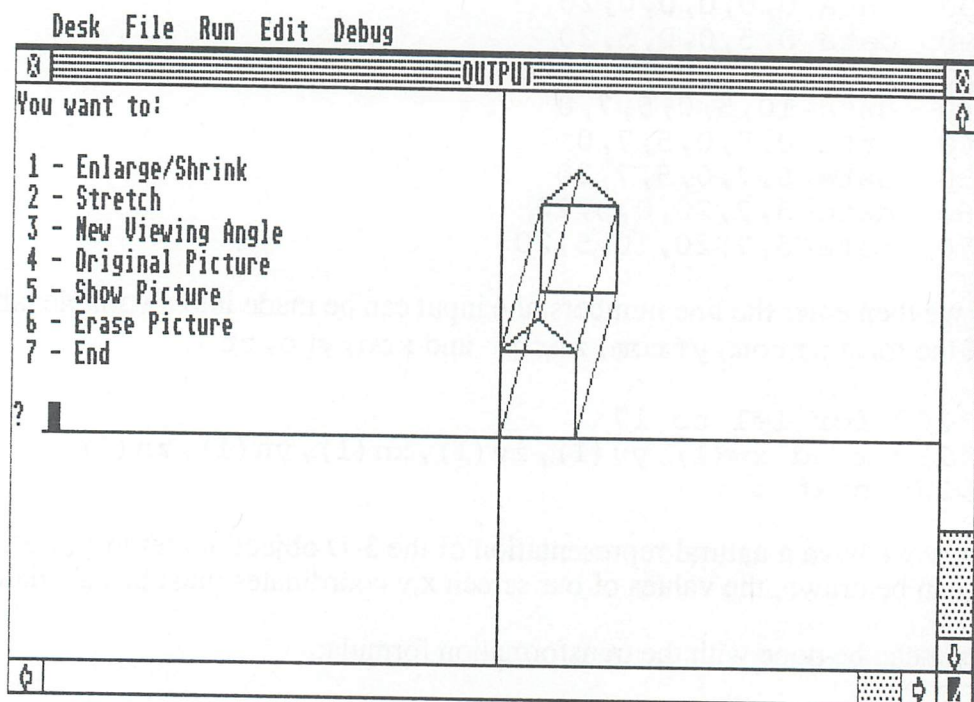


Figure 43: The 3-D demo

```
10 ' 3ddemo.bas
20 ' Manipulation of 3d-Objects
30 ' -----
40 fullw 2:clearw 2:res = 2: '2=med res 1= hi res
50 '
60 'New coordinate system
70 xmin = -320 : xmax = 320
80 ymin = -200 / res : ymax = 200 / res :
   ydif = abs(ymin) + abs(ymax)
90 '
100 'Initialization
110 read num
120 dim xv(num),yv(num),
   zv(num),xn(num),yn(num),zn(num)
130 dim sxv(num),syv(num),sxn(num),syn(num)
140 '
150 start:
160 mfx=5:mfy=5:mfz=5/res:w=45
170 '
180 'Read in picture data
190 restore : read num
200 for l=1 to 17
210 read xv(l),yv(l),zv(l),xn(l),yn(l),zn(l)
220 next l
230 '
240 transform:
250 clearw 2:for l=1 to 17
260 sxv(l)=xv(l)*mfx+zv(l)*cos(w)*mfz:
   sxn(l)=xn(l)*mfx+zn(l)*cos(w)*mfz
270 syv(l)=yv(l)*mfy+zv(l)*sin(w)*mfz:
   syn(l)=yn(l)*mfy+zn(l)*sin(w)*mfz
280 next l
290 '
300 draw:
310 gosub axisintercept
320 for l=1 to 17
330 xv=sxv(l) :yv=syv(l) :xn=sxn(l):yn=syn(l)
340 gosub transline
350 linef xv,yv,xn,yn
360 next l
370 '
380 menu:
390 gotoxy 0,0:print "you want to:":print
```



```
400 print " 1 - Enlarge/Shrink"
410 print " 2 - Stretch"
420 print " 3 - New viewing angle"
430 print " 4 - Original picture"
440 print " 5 - Show picture"
450 print " 6 - Erase picture"
460 print " 7 - END"
470 print:input e
480 if (e<0 or e>7) goto menu
490 on e goto scale,stretch,angle,start,draw,
    cls,exit
500 goto menu
510 '
520 scale:
530 gotoxy 0,10:input"Bigger (x > 1) or
    smaller (x < 0)";mf
540 mfy=mfy+mf:mfz=mfz+mf:mfx=mfx+mf
550 goto transform:
560 '
570 stretch:
580 gotoxy 0,10:input "In which direction:
    x=1, y=2, z=3 ";r
590 input" Stretch (x > 1) or shrink
    (x > 0 and x < 1)",mf
600 if (r<0 or r>3) then goto stretch
610 on r goto 620,630,640
620 for l=1 to num:xv(l)=xv(l)*mf:
    xn(l)=xn(l)*mf:next l:goto transform
630 for l=1 to num:yv(l)=yv(l)*mf:
    yn(l)=yn(l)*mf:next l:goto transform
640 for l=1 to num:zv(l)=zv(l)*mf:
    zv(l)=zv(l)*mf:next l:goto transform
650 '
660 angle:
670 input"How many degrees between the
    x-and z-axis ";w:w=w*3.145926/180
680 goto transform
690 '
700 cls:
710 clearw 2:goto menu
720 '
730 exit:
740 clearw 2:closew 2:end
```

```
750 '
760 ' here is the screen data
770 ' scheme: line coordinates
780 ' line from (xv,yv,zv) - line to (xn,yn,zn)
790 '
800 data 17
810 data 0,0,0,10,0,0
820 data 10,0,0,10,5,0
830 data 10,5,0,0,5,0
840 data 0,5,0,0,0,0
850 data 10,0,20,10,5,20
860 data 10,5,20,0,5,20
870 data 0,5,20,0,0,20
880 data 0,0,20,10,0,20
890 data 10,0,0,10,0,20
900 data 0,0,0,0,0,20
910 data 0,5,0,0,5,20
920 data 10,5,0,10,5,20
930 data 10,5,0,5,7,0
940 data 0,5,0,5,7,0
950 data 5,7,0,5,7,20
960 data 5,7,20,0,5,20
970 data 5,7,20,10,5,20
980 '
990 axisintercept:
1000 xv = -300 : yv = 0 : xn = 300 : yn = 0 :
    gosub transline:
1010 line xv,yv,xn,yn : rem x-axis
1020 xv = 0 : yv = ymin : xn = 0 : yn = ymax :
    gosub transline:
1030 linef xv,yv,xn,yn : rem y-axis
1040 return
1050 '
1060 transline:
1070 xv = 640 * (xv - xmin) / (xmax - xmin)
1080 yv = ydif * (ymax - yv) / (ymax - ymin)
1090 xn = 640 * (xn - xmin) / (xmax - xmin)
1100 yn = ydif * (ymax - yn) / (ymax - ymin)
1110 return
```

4.4.1 CAD Application

As we have seen, a point can be easily manipulated once it is in the computer. In principle, the house demo is just a shape that can be as easily manipulated. It's represented in a different way than it was in Chapter 3. The one difference is that in the examples in Chapter 3, every point except the first was the end point of a line. In this chapter, we specify the beginning point first.

We can enlarge, shrink, and distort our house. What we can't do is change the viewing position—that is, we can't look at the house from a different angle, say, from the side or top.

It's probably already occurred to you that there's a way to do this, too.

Enter the following listing and experiment with different input values:

```
10      'caddemo.bas
20      'perspective drawing
30      'choice of side to view
40      'and viewing distance
50      '-----
60      clearw 2 : fullw 2 : r = 1 :rem r=2 for color
70      maxl = 20 : 'set maximum line number to 20
80      dim xv(maxl),yv(maxl),zv(maxl),
          xn(maxl),yn(maxl),zn(maxl)
90      dim xvon(maxl),yvon(maxl),zvon(maxl),
          xnach(maxl),ynach(maxl),znach(maxl)
100     '
110     ' bwx, bwy, bwz = viewers angle relative to
        axis
120     bwx = 0 : bwy = 0 : bwz = 0
130     '
140     ' bsx, bsy, bsz = coordinates of viewer
150     bsx = 125 : bsy = 100: bsz = 75
160     '
170     new.start:
180     gosub picture.read
190     gosub move
200     gosub matrix
210     gosub twist
220     gosub view
```



```
230 gosub draw
240 '
250 menu:
260 gotoxy 0,0:print "1-viewer's position point "
270 print " 2 - slope/angle of viewer"
280 print:input"what do you want to change ";e
290 if (e<0 or e>2) then goto menu
300 on e goto n.s, n.w
310 '
320 n.s:
330 print "viewers current position is:
    ";bsx;bsy;bsz
340 input "enter new viewer position: ";
    bsx,bsy,bsz
350 goto new.start
360 '
370 n.w:
380 print "current angle is: ";bwx;bwy;bwz
390 input "enter new angle : ";bwx,bwy,bwz
400 goto new.start
410 '
420 'calculate transformation matrix
430 matrix:
440 sx=sin(bwx) : cx=cos(bwx)
450 sy=sin(bwy) : cy=cos(bwy)
460 sz=sin(bwz) : cz=cos(bwz)
470 mat(0,0) = sx * sz * sy + cz * cy
480 mat(0,1) = sx * cz * sy - sz * cy
490 mat(0,2) = sy * cx
500 mat(1,0) = sz * cx
510 mat(1,1) = cz * cx
520 mat(1,2) = -sx
530 mat(2,0) = sx * sz * cy - sy * cz
540 mat(2,1) = sx * cz * cy + sz * sy
550 mat(2,2) = cx * cy
560 return
570 '
580 move:
590 for p=1 to num
600 xv(p) = xv(p) - bsx :
    yv(p) = yv(p) - bsy : zv(p) = zv(p) - bsz
610 xn(p) = xn(p) - bsx :
    yn(p) = yn(p) - bsy : zn(p) = zn(p) - bsz
```



```
620 next p
630 return
640 '
650 twist:
660 for p=1 to num
670 xvon(p) = xv(p) * mat(0,0) + yv(p) * mat(1,0)
+ zv(p) * mat(2,0)
680 yvon(p) = xv(p) * mat(0,1) + yv(p) * mat(1,1)
+ zv(p) * mat(2,1)
690 zvon(p) = xv(p) * mat(0,2) + yv(p) * mat(1,2)
+ zv(p) * mat(2,2)
700 xnach(p) = xn(p) * mat(0,0) + yn(p)
* mat(1,0) + zn(p) * mat(2,0)
710 ynach(p) = xn(p) * mat(0,1) + yn(p)
* mat(1,1) + zn(p) * mat(2,1)
720 znach(p) = xn(p) * mat(0,2) + yn(p)
* mat(1,2) + zn(p) * mat(2,2)
730 next p
740 return
750 '
760 view:
770 for p=1 to num
780 xvon(p) = xvon(p) * (100/zvon(p)) : yvon(p)
= yvon(p) * (100/zvon(p))
790 xnach(p) = xnach(p) * (100/znach(p)) :
ynach(p) = ynach(p) * (100/znach(p))
800 next p
810 return
820 '
830 draw:
840 clearw 2
850 for p=1 to num
860 linef xvon(p)+140,yvon(p)+96/r,
xnach(p)+140,ynach(p)+96/r
870 next p
880 return
890 '
900 'read in object coordinates
910 picture.read:
920 restore
930 read num
940 for p=1 to num
950 read xv(p),yv(p),zv(p),xn(p),yn(p),zn(p)
```

```
960  next
970  return
980  '
990  'object coordinates go here
1000 data 17
1010 data 0,0,0,10,0,0
1020 data 10,0,0,10,5,0
1030 data 10,5,0,0,5,0
1040 data 0,5,0,0,0,0
1050 data 10,0,20,10,5,20
1060 data 10,5,20,0,5,20
1070 data 0,5,20,0,0,20
1080 data 0,0,20,10,0,20
1090 data 10,0,0,10,0,20
1100 data 0,0,0,0,0,20
1110 data 0,5,0,0,5,20
1120 data 10,5,0,10,5,20
1130 data 10,5,0,5,7,0
1140 data 0,5,0,5,7,0
1150 data 5,7,0,5,7,20
1160 data 5,7,20,0,5,20
1170 data 5,7,20,10,5,20
```

4.4.2 Using the mouse as an input device

Professional CAD systems, as well as many new computers on the market today, include a special input device known as a *mouse*. A mouse simplifies the process of data input and manipulation, and replaces more orthodox keyboard equivalents such as the function keys.

In the following pages we'll discover how we can integrate the use of the mouse into these programs.

The GEM VDI handles the mouse operations by looking up its position and reporting them in the `ptsout` array. We'll write a small subprogram that stores information from the operation of the left mouse button in the variables `xm` and `ym`:

```
2010 mouse.input
2020 poke contrl,124
2030 poke contrl+2,0
2040 poke contrl+6,0
2050 vdisys
2060 xm=peek(ptsout)
2070 ym=peek(ptsout+2)
2080 click=peek(intout)
2090 return
```

All that remains, is to develop a suitable way to incorporate the subprogram so that we can use it in our 3-D demonstration program.

After the program starts, we must calibrate the mouse for digital representation of the information.

To do this you give the mouse a *pointer*—a small piece of wire or a needle taped to the lower middle of the screen. The pointer should point to the extreme lower left corner. As soon as you click the left mouse button, this point is assigned the coordinate (0,0).

Move the mouse pointer (slowly, since you're working in BASIC) to the first point to be entered. Click the left mouse button to store this coordinate. Move the mouse pointer to the next position and click the left mouse button again. Since you're using 3-D coordinates you have to enter `z v` and `z n` by hand after guessing values. Then you'll have entered one line.

```
10      ' mus3demo.bas
20      ' Inputting Coordinates with a Mouse
30      ' -----
40      fullw 2:clearw 2:res = 2 :
        ' 1=hires    2=medium resolution
50      '
60      'New Coordinate System
70      xmin = -320 : xmax = 320
80      ymin = -200/res : ymax = 200/res
90      '
100     'Initialization
110     num = 100 : ' Room for 100 lines
120     dim xv(num),yv(num),zv(num),
        xn(num),yn(num),zn(num)
130     dim sxv(num),syv(num),sxn(num),syn(num)
140     '
150     start:
160     mfx=5:mfy=5:mfz=5:w=3.14159/4:l=1
170     gosub calibrate
180     gotoxy0,0:print"Line:
        ";l,xv(1);yv(1);xn(1);yn(1)
190     readin:
200     gosub mouse.input:xv(1)=xm:yv(1)=ym:
        input "Zv ";zv(1)
210     gosub mouse.input:xn(1)=xm:
        yn(1)=ym:input "Zn ";zn(1)
215     gotoxy 0,0:print"Line:
        ";l,xv(1);yv(1);xn(1);yn(1)
220     if zv(1)<>-99 then l=l+1 : goto readin
230     num = l
240     transform:
250     for l=1 to num
260     sxv(l)=xv(l)*mfx+zv(l)*cos(w)*mfz:
        sxn(l)=xn(l)*mfx+zn(l)*cos(w)*mfz
270     syv(l)=yv(l)*mfy+zv(l)*sin(w)*mfz:
        syn(l)=yn(l)*mfy+zn(l)*sin(w)*mfz
280     next l
290     '
300     draw:
310     clearw 2:gosub axisintercept
320     for l=1 to num
330     xv=sxv(l) :yv=syv(l) :xn=sxn(l):yn=syn(l)
340     gosub transline
```



```
350   linef xv,yv,xn,yn
360   next l
370   '
380   menu:
390   gotoxy 0,0:print "You want to:":print
400   print " 1 - Enlarge/Shrink"
410   print " 2 - Stretch"
420   print " 3 - New Viewing Angle"
430   print " 4 - New Sample"
440   print " 5 - Show Picture"
450   print " 6 - Erase Picture"
460   print " 7 - End"
470   print:input e
480   if (e<0 or e>7) goto menu
490   on e goto scale,stretch,angle,
      start,draw,cls,exit
500   goto menu
510   '
520   scale:
530   gotoxy 0,10:input"Enlarge (Number > 1)
      or shrink (Number < 0)";mf
540   mfy=mfy+mf:mfz=mfz+mf:mfz=mfz+mf
550   goto transform:
560   '
570   stretch:
580   gotoxy 0,10:input "In which Direction:
      x=1, y=2, z=3 ";r
590   input" Stretch (Number > 1) or Shrink
      (Number 0...1)",mf
600   if (r<0 or r>3) then goto stretch
610   on r goto 620,630,640
620   for l=1 to num:xv(l)=xv(l)*mf:
      xn(l)=xn(l)*mf:next l:goto transform
630   for l=1 to num:yv(l)=yv(l)*mf:
      yn(l)=yn(l)*mf:next l:goto transform
640   for l=1 to num:zv(l)=zv(l)*mf:
      zn(l)=zn(l)*mf:next l:goto transform
650   '
660   angle:
670   input"How many degrees between the x
      and the z-Axis ";w:w=w*3.14159/180
680   goto transform
690   '

```

```
700  cls:
710  clearw 2:goto menu
720  '
730  exit:
740  clearw 2:closew 2:end
750  '
760  ' Here is the screen data
770  ' scheme: line  coordinates
780  '
790  axisintercept:
1000  xv = -300 : yv = 0:xn = 300 :
      yn = 0 : gosub transline:
1010  linef xv,yv,xn,yn : rem x-Axis
1020  xv = 0 : yv = -200/res :
      xn = 0 : yn = 200/res : gosub transline:
1030  linef xv,yv,xn,yn : rem y-Axis
1040  return
1050  '
1060  transline:
1070  xv = 640 * (xv - xmin) / (xmax - xmin)
1080  yv = (400/res) * (ymax - yv) / (ymax - ymin)
1090  xn = 640 * (xn - xmin) / (xmax - xmin)
1100  yn = (400/res) * (ymax - yn) / (ymax - ymin)
1110  return
1999  '
2000  mouse.input:
2010  poke contrl,124
2020  poke contrl+2,0
2030  poke contrl+6,0
2040  vdisys
2050  xm=int((peek(ptsout)+xdiff)*vfm)
2060  ym=int((ydiff-peek(ptsout+2))*vfm)
2070  gotoxy 1,1:print "Mouse pos. x/y";xm;ym
2080  click=peek(intout)
2090  if click <> 1 then goto mouse.input
2100  print chr$(7):return
2990  '
3000  calibrate:
3010  clearw 2: vfm = 1:xdiff=0 : ydiff = 0
3020  print "Screen Sample with the Mouse:":print
3030  print "Move the mouse until your pointer is"
```

```
3040  print "in the lower left corner of the
      screen." :print
3050  print "Check the numeric coordinates to make"
3060  print "sure the mouse pointer is where you
      want it."
3070  print
3080  print "Then press the left mouse button."
3090  gosub mouse.input
3100  xdiff=xm : ydiff=-ym
3110  clearw 2:print "Great! -
      Now the Picture can be tried."
3120  print:print"Now enter an
      enlargement/reduction factor"
3130  input"(No change = 1); Enter factor :";vfm
3140  print:print"To end the input
      you must enter -99 for ZV!"
3150  print:input "press enter to continue ";e$
3160  clearw 2 : return
```

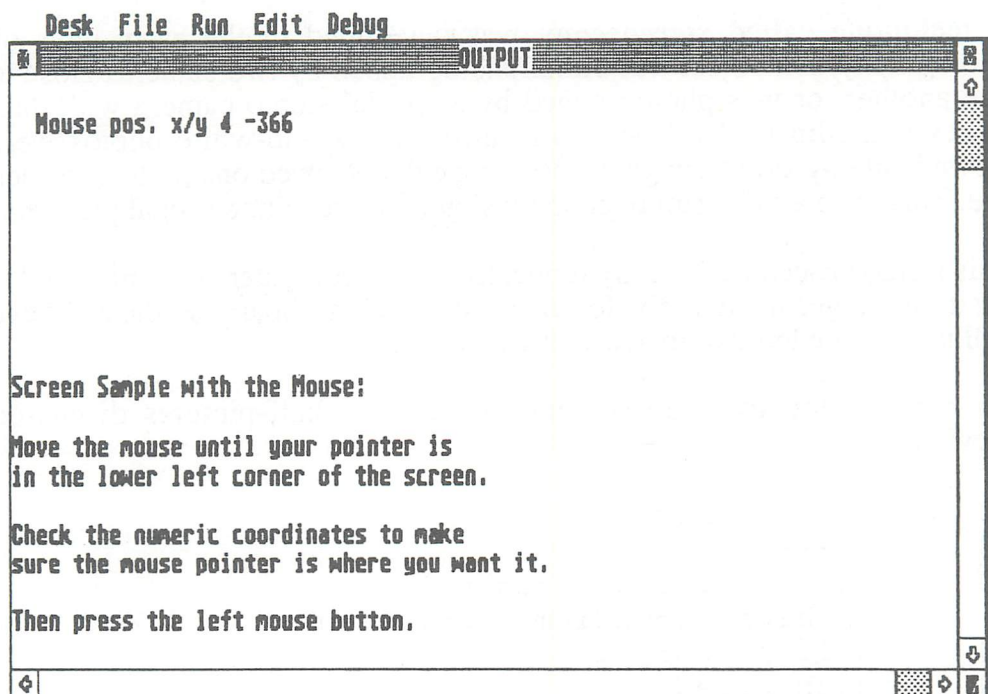


Figure 44: Inputting coordinates with the mouse

4.4.3 Real stereo graphics

Perspective drawings like we've just done work fine, since we don't always need to make something look like a "real" object to get our point across.

You probably know how our 3-D vision works. Your two eyes each register half the picture, since your eyes are separated by a couple of inches and see things from slightly different perspectives. The sight center of the brain puts the two half-pictures together to form a three-dimensional impression.

That's our problem for 3-D representation. We need to show each eye something a little different. We need to divide the picture so that each eye will see half the picture on the screen. We'll try to make it more like standing in front of a real object.

A technique called *stereoscopy* was developed in the early days of photography. An object was either photographed by two cameras close to one another, or was photographed by a special stereo camera with dual lenses and film backs. The two resulting black-and-white photos were viewed side by side through a stereoscope that showed one picture to each eye. This gave a surprisingly good illusion of a three-dimensional picture.

This stereo procedure is easily replicated with a computer. We only need to position images using a divider, like a sheet of cardboard, so the right eye will not see the left-eye image, and vice versa.

To produce the two half-screens, we use the half-pictures discussed previously:

```
10      'stereo1.bas
20      '-----
30      clearw 2:fullw 2:res = 1:
        ' 1=high  2=medium resolution
40      'half picture 1
50      gosub broad
60      read num
70      for l=1 to num
80      read xv,yv,xn,yn:yv=yv/res:yn=yn/res
90      xv=xv * .9 :xn=xn*.9:yv=yv*.9:yn=yn*.9
100     linef xv,yv,xn,yn
110     next
120     'half picture 2
130     read num
140     for l=1 to num
150     read xv,yv,xn,yn:yv=yv/res:yn=yn/res
160     xv=xv * .9 :xn=xn*.9:yv=yv*.9:yn=yn*.9
170     linef xv+320,yv,xn+320,yn
180     next
190     end
200     broad:
210     poke ptsin,6
220     poke ptsin+2,0
230     poke contrl,16
240     poke contrl+2,1
250     poke contrl+6,0
260     vdisys 16
270     return
```

```
280    'frontside
290    data 10
300    data 40,300,225,320
310    data 225,320,225,210
320    data 225,210,40,190
330    data 40,190,40,300
340    'right side
350    data 225,320,300,250
360    data 300,250,300,100
370    data 300,100,225,210
380    data 225,210,225,320
390    'left side
400    data 40,190,135,50
410    data 135,50,300,100
420    'frontside
430    data 10
440    data 60,280,225,320
450    data 225,320,225,210
460    data 225,210,60,170
470    data 60,170,60,280
480    'rightside
490    data 225,320,320,255
500    data 320,255,320,105
510    data 320,105,225,210
520    data 225,210,225,320
530    'leftside
540    data 60,170,150,40
550    'above
560    data 150,40,320,105
```

If you squint, then the two half-pictures on the screen should come together enough so that you can see a cube that looks three-dimensional.

Problems with this method arise because our mind is set up in such away that our two eyes always focus on a single point. As a result, looking at two pictures next to each other for 3-D effect is unnatural and difficult. The deluxe models of the old-style stereoscope had two lenses to make it easier to put the two pictures together in your head.

The effect is even better when the two half-pictures are quite close to each other in the picture:

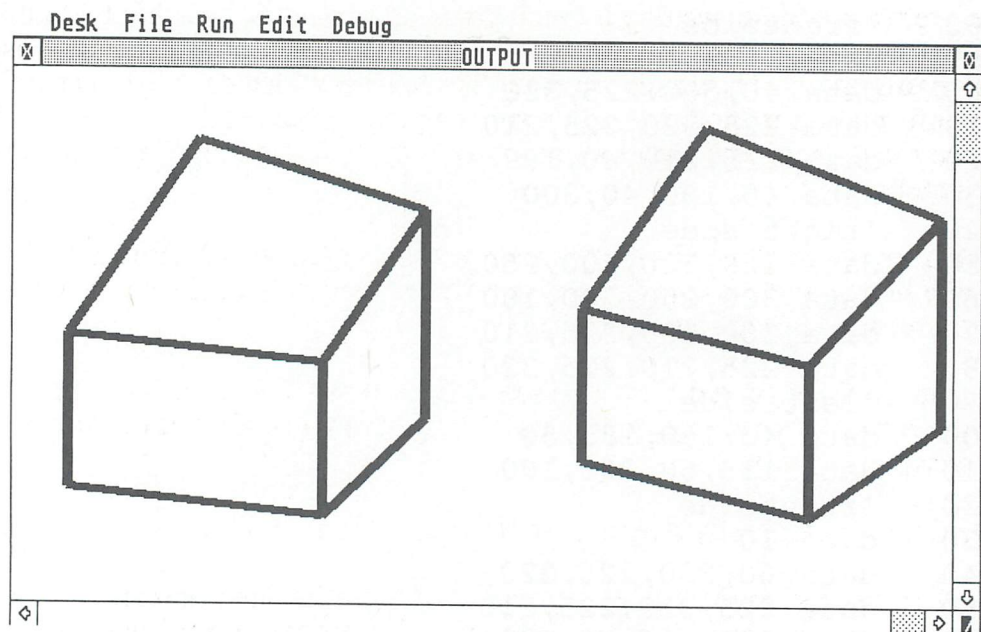


Figure 45: The two halves of a stereo picture

To do this we need a different method of dividing the pictures. In the 1950's the first 3-D films came out. These movies were filmed just like regular films, but by a pair of normal cameras side by side. The major difference was in the method by which they were projected. The two films were synchronized and then projected—one through a red filter, and the other through a green filter (two complementary colors). A viewer's two eyes each saw a slightly different picture.

If you didn't have the 3-D glasses on, it looked like a regular movie with all the curves doubled. However, when you wore the 3-D glasses, each eye got only the information it was supposed to get. The color filter in front of each eye erased one picture, and saw only the image it was supposed to see.

This procedure that can be used on any computer with color graphics capabilities.

Before we continue, we ought to tell you that it could be possible to do this with a monochrome screen. To get 3-D pictures in mono you need to use polarizing filters. Their axes are set up in such a way that each eye sees only the information it should see. Unfortunately, you also need polarized light. Since the monochrome monitor doesn't give off polarized light, we can't use this method.

Actually you don't need a color monitor for a stereo-vision representation. If you don't have a color monitor, you can on a color TV, since it gives out red, green, and blue signals.

If you don't have red/blue glasses handy, that's OK too. You can rig something up with colored plastic—and even use different colors if necessary.

We'll start by correctly aligning the ST and its monitor.

Type in the following two line program and run it. Then open the control panel under the Desk option on the menu bar (if the control panel is not there then you need to reboot with a disk that has that accessory on it, possibly the Language disk that comes with your system). Now put on your 3-D glasses and change colors 2 and 3 until you can see the screen in only one color if you close one eye (i.e. you can only see the left line with the left eye and the right line with the right eye.)

```
10 clearw 2:fullw 2:color 1,1,2,1,1:linef 10,10,10,300
20 color 1,1,3,1,1:linef 20,10,20,300
```

Save this combination. It's what makes your glasses/viewing device work. Better still, save it to your system disk with Save Desktop under the options menu on the Desktop.

Now we're all set to work with real 3-D graphics. Enter the following listing and run it to see the cube as a real-looking 3-D object. (Don't forget to put on fashionable glasses).


```
10      'stereo2.bas
20      '-----
30      clearw 2:fullw 2:res = 1: '1=high
        2=medium resolution
40      'Half picture 1
50      gosub broad
60      read num : color 1,1,2,1,1
70      for l=1 to num
80      read xv,yv,xn,yn:yv=yv/res:yn=yn/res
90      xv=xv * .9 + 180:xn=xn*.
        9 + 180:yv=yv*.9:yn=yn*.9
100     linef xv,yv,xn,yn
110     next
120     'Half picture 2
130     read num : color 1,1,3,1,1
140     for l=1 to num
150     read xv,yv,xn,yn:yv=yv/res:yn=yn/res
160     xv=xv * .9 + 180:xn=xn*.
        9 + 180:yv=yv*.9:yn=yn*.9
170     linef xv,yv,xn,yn
180     next
190     end
200     broad:
210     poke ptsin,3
220     poke ptsin+2,0
230     poke contrl,16
240     poke contrl+2,1
250     poke contrl+6,0
260     vdisys 16
270     return
280     'frontside
290     data 10
300     data 40,300,225,320
310     data 225,320,225,210
320     data 225,210,40,190
330     data 40,190,40,300
340     'rightside
350     data 225,320,300,250
360     data 300,250,300,100
370     data 300,100,225,210
380     data 225,210,225,320
390     'leftside
400     data 40,190,135,50
```

```
410 data 135,50,300,100
420 'frontside
430 data 10
440 data 60,280,225,320
450 data 225,320,225,210
460 data 225,210,60,170
470 data 60,170,60,280
480 'rightside
490 data 225,320,320,255
500 data 320,255,320,105
510 data 320,105,225,210
520 data 225,210,225,320
530 'leftside
540 data 60,170,150,40
550 'above
560 data 150,40,320,105
```

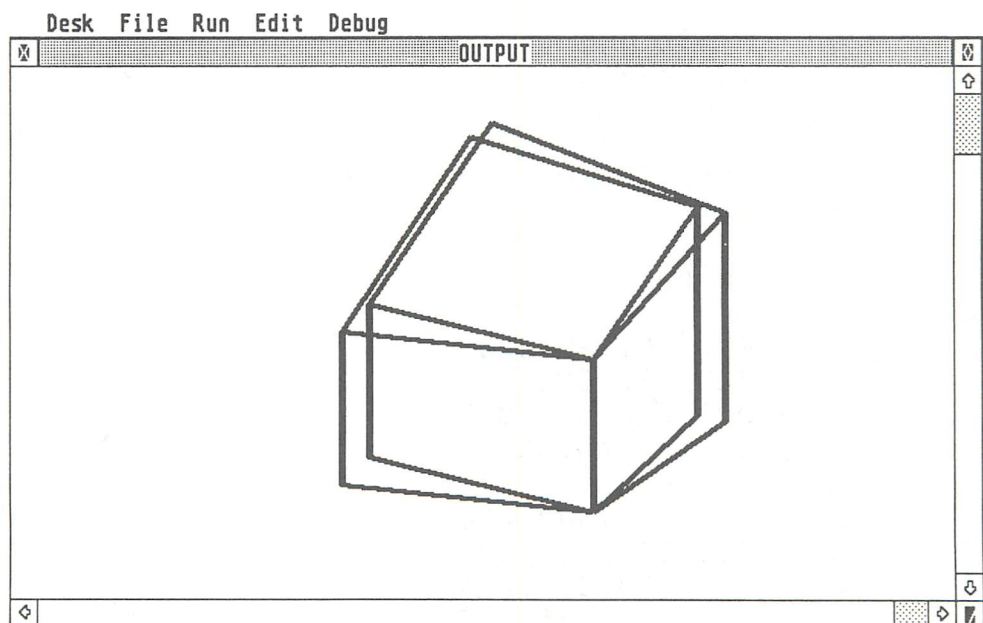


Figure 46: Hardcopy of ST stereo picture

You might ask "How do I go about getting a stereo graphic of a real object?" The easiest, surest way would be to take two photographs of the object you want to show onscreen with the camera in two slightly different positions. Start out by taking the picture pair about 2 inches from each other, and then gradually increase the distance between the cameras.

After you've discovered the best angle to do this, you need only enter the appropriate lines into the ST coordinate system. One way is to draw a grid across the photographs and then translate the most notable lines to the ST. The input routines we've just developed should be a big help there.



Chapter 5

Tricks, Tips... and still more Graphics

5.1 Tips & Tricks

In this chapter, we'd like to answer several questions that you might have about working with the ST's graphics:

1. How does an application differentiate between high or low resolution monitors connected to the ST?
2. How can I write a program in C, Modula-2 or Pascal that will use all the ST's graphics capabilities?
3. How many colored pens are available and how can I use these different colors?
4. How can I simulate different colors on a black-and-white monitor—and how do I work with the different shades of gray?
5. How can I save and load a complete screen?
6. What are the different write modes, and how do I access them?

5.1.1 Determining the resolution

As mentioned in Chapter 2, the monochrome monitor sends a message to the ST to state that it can use high resolution mode only. Originally intended as protection for the standard monitor, this feature can be quite annoying to many of us. Also, we've already mentioned how the software gets the data for the monitor.

Some programs might start with an warning message stating "This application supports a color monitor only."

We know why the program can't run—because the screen coordinates use NDC. A program finds the maximum values for *x* and *y* in the VDI's 45 `intout` files after the call to `open_workstation`. The program also finds a statement about the number of colors supported by the output device

(in `intout(13)`). If the ST is connected to a monochrome monitor, only the background and foreground colors (0 and 1) are available—the return value is 2.

As soon as `intout(13)` is greater than two, a color monitor is required. We can display a message saying that the SM124 monitor is needed. This is illustrated by the upcoming program listings. It was written especially for a coordinate system with 640 X 400 points.

5.1.2 Low, Medium and High Resolution

A program can be designed to work with all three resolution modes—working with normalized device coordinates (32 x 32). This is especially useful when execution speed is important.

We can use the `TRANSPPOINT` and `TRANSLINE` routines, which translate point coordinates from one system to another. When using the ST there is an easier transformation: Mid-Res is related to Hi-Res by a factor of two. That means that the same conversions can be made by multiplying or dividing by two, depending on the sort of operating system the program was developed for.

We must know the `xmax` and `ymax` of the actual workstation. They are stored in `intout(0)` and `intout(1)`.

5.1.3 Inputting colors

You can change the palette colors with the desk top control panel. This lets you define up to sixteen colors by mixing red, green, and blue. The mixing scale has a range of zero to seven.

If you're using BASIC, you can choose these colors by using the command `color t, f, l, pc, p`, where `t` stands for text color, `f` for fill color, `l` for line color, `pc` for pattern color, and `p` for pattern.

LOGO programmers can choose an actual color from the possible spectrum of the 512 hues. Here's the instruction to the control field shifter:

SETPAL pen (color list)

The color list is set by the R, G, and B (red, green, and blue) parts of the color of the pen number. You can enter numbers from 0 to 1000. SETPAL 1 (1000 0 0) will set pen 1 to deep red.

The method is similar under GEM. You must set the three colors of each pen in the array RGB.IN(3):

```
int rgb_in(3);  
  
rgb_in(0)=1000;  
rgb_in(1)=0;  
rgb_in(2)=0;  
vs_color(handle,pen,rgb_in);
```

This C routine also defines the color deep red and assigns it to the desired pen. If the pen number is 0, the background color will be deep red.

If you're working with a monochrome monitor, your choices are limited. Using monochrome, only the colors 0, 0, 0 and 1000, 1000, 1000 are available. You can do little but switch to reverse video, i.e. reverse the screen's background to black and its text to white.

5.1.4 Gray levels on the monochrome monitor

Most of the time you'll want to work with the maximum resolution available. In any case, you'll almost certainly want more than one color or shade on the screen. Even users of the SM124 monochrome monitor don't have to settle for just one color. They can work with different gray levels.

Coordinates in the range from 0, 0 to 320, 200 all have four pixels per point on the hi-res monitor. By using these groupings we can get five shade levels:


```
white:  - -  
        - -
```

```
gray 1:  # -  
         - -
```

```
gray 2:  # -  
         - #
```

```
gray 3:  # #  
         # -
```

```
black:   # #  
         # #
```

5.1.5 Storing screens

You may want to save a screen to disk, especially if it took a long time to produce it. But detailed, intricate screens, such as produced by our 3-D function plotter, may take a long time to load and store.

For this reason you may want to store all of the screen contents in the screen memory area. As we learned in Chapter 1, the ST produces graphics with bits in a specific block of RAM. We can save the whole block to disk with the command `BSAVE` (and later reload it with `BLOAD`).

```
bsave "screen.bin",&H78020,32000
```

```
bload "screen.bin"
```

This command saves an ST screen under the name `screen.bin`. The parameters following the name are its starting address and the length of the stored block.

If we later `bload` without specifying an address, the stored image is reloaded at its original address during loading.

5.1.6 The write-mode

The ST can write new points to the screen, as if you were using the Boolean operators AND, OR, NOT, and XOR. The color of the new point is determined in part by the color of the point that is already there.

Without going into the theory of Boolean algebra, we'll say that four different write modes can be used:

1. The replace mode is the normal mode of operation. Each new point is written and overwrites any previous point.
2. If the transparent mode is activated, old and new points are written over old points and their colors are mixed.
3. If the output function is set to the XOR mode, the two colors are mixed (if they don't cancel each other out). Otherwise the points are erased and the background color is displayed.
4. In the reverse transparent mode, all points previously erased and reversed are shown.

The VDI has the `set_writing_mode` function to set the desired write mode. The call in C looks like this:

```
vswr_mode(handle, mode)
```

An example of setting the various writing modes can be found in the listing of an animated graphics program (Section 5.2.1) in both BASIC and Modula-2.

5.1.7 Filling the screen with patterns

We've already seen how to fill the screen with figures, although so far we can do it on only one screen in one color. GEM has a number of patterns that can be displayed with a short LOGO program:

```
TO FILLSHOW
HT CS
SETLINE [1 1 1]
(LOCAL "XB "YB "W "Z)
MAKE "GFill "TRUE
MAKE "W PI / 6
MAKE "Z (180 / PI)
FILLSET 2 1 0
ATARI
FILLSET 2 13 0
DIAMOND
FILLSET 3 1 0
AB
MAKE "GFill "FALSE
STOP
END

TO FILLSET :SELECT :BEGIN :LRUN
IF :LRUN > 11 [STOP]
SETFILL (LIST :SELECT (:BEGIN + :LRUN) 1)
MAKE "XB (5 * SIN (:W * :Z))
MAKE "YB (5 * COS (:W * :Z))
IF :LRUN < 6 [FILLDRAW :LRUN (-1)]
IF :LRUN > 5 [FILLDRAW :LRUN 1]
MAKE "W (:W + (PI / 3))
FILLSET :SELECT :BEGIN (:LRUN + 1)
END

TO FILLDRAW :DEG :PLACE
ARC (LIST (:PLACE * 153 + :XB) (0 + :YB) 145 (:DEG
* 60) (:DEG * 60 + 60))
STOP
END

TO ATARI
SETFILL [4 1 1]
FILL
```

```
WAIT 50
CS
END
```

```
TO DIAMOND
PPROP "GRAPHICS ".FPT [0 0 128 448 992 2032 4088
8188 16382 8188 4088 2032 992 448 128 0]
SETFILL [4 1 1]
FILL
WAIT 50
CS
END
```

```
TO AB
PPROP "GRAPHICS ".FPT [0 0 0 0 3704 8060 6924 6924
13176 13176 13068 25356 25468 25464 0 0]
SETFILL [4 1 1]
FILL
WAIT 50
CS
END
```

```
TO WAIT :TIME
IF :TIME < 1 [STOP]
WAIT :TIME - 1
END
```

```
MAKE "GFILL "FALSE
```

These patterns can also be accessed from C or BASIC. Using the BASIC color command, the fourth parameter specifies the pattern's color and the fifth parameter specifies the actual pattern.

5.2 Still more Graphics

We'd like to present a couple of very interesting graphic programs—without going into theory and long explanations.

5.2.1 Animated graphics

Animated graphics were the dominion of specialized mainframe computers for a long time, because they require fast calculation of coordinates. Each point must be quickly displayed and erased as needed.

But the ST's 8-Mhz, MC 68000 CPU is well-equipped for the task. We can use the XOR write mode to erase details and get quite respectable results, as the following Modula listing of *Kaleidoscope* will prove. BASIC programmers will find a BASIC version. The BASIC version is not quite as fast, and its effects are as not as impressive.

By the way, if you press the right mouse button as it is running, you'll get a new pattern. Clicking the left mouse button exits the program and returns you back to the desktop.

```
10    defint d,x,y
20    fullw 2
30    New.Pattern:
40    gosub Replace.Mode
50    clearw 2
60    gosub XOR.Mode
70    gosub Linewidth
80    x=0:y=0:dx=1:dy=1
90    xm=310+int(rnd*305):ym=174+int(rnd*170)
100   repeat:
110   if dx<0 then xa=x else xa=xm-x
120   if dy<0 then ya=y else ya=ym-y
130   if xa<ya then x0=x+dx*xa:y0=y+dy*xa:dx=-dx
140   if xa>ya then x0=x+dx*ya:y0=y+dy*ya:dy=-dy
150   if xa=ya then New.Pattern
160   linef x,y,x0,y0
170   linef 615-x,y,615-x0,y0
180   linef x,344-y,x0,344-y0
```

```
190   linef 615-x,344-y,615-x0,344-y0
200   x=x0:y=y0
210   if x<>0 or y<>0 then repeat
220   goto repeat
230   rem -----
240   Replace.Mode:
250   poke intin,1
260   poke contrl,32
270   poke contrl+2,0
280   poke contrl+6,1
290   vdisys
300   return
310   rem -----
320   XOR.Mode:
330   poke intin,3
340   poke contrl,32
350   poke contrl+2,0
360   poke contrl+6,1
370   vdisys
380   return
390   rem -----
400   Linewidth:
410   poke ptsin,1+2*int(rnd*5)
420   poke ptsin+2,0
430   poke contrl,16
440   poke contrl+2,1
450   poke contrl+6,0
460   vdisys
470   return
```

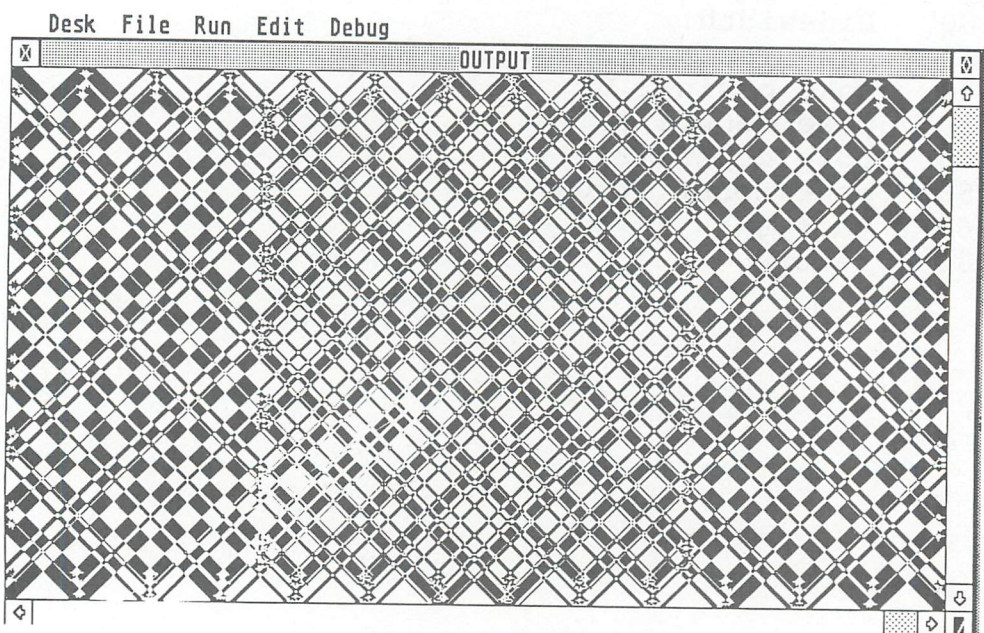
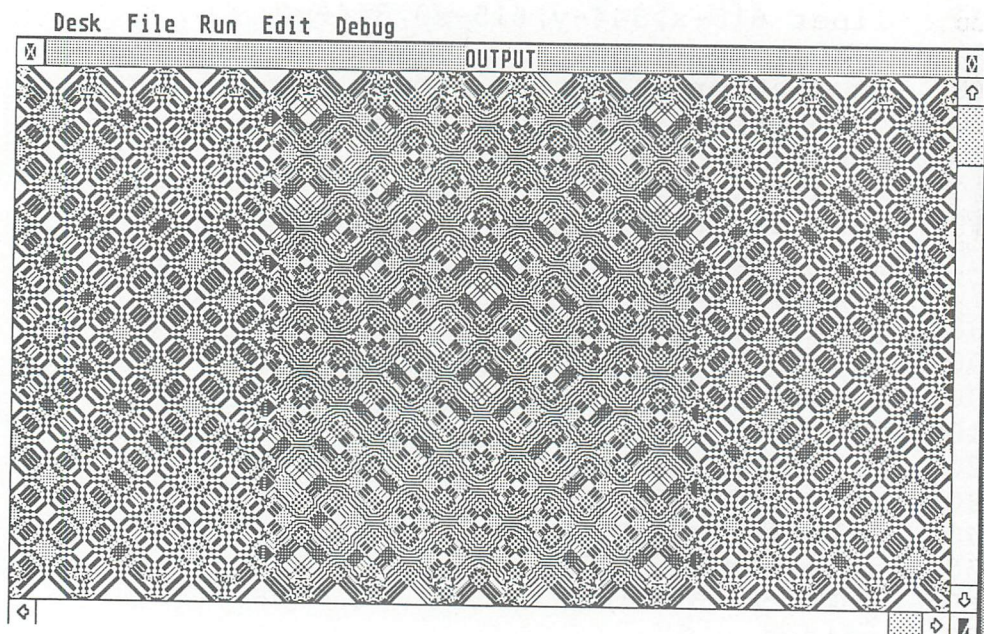



Figure 47: Hardcopy of PATTERN.BAS

```

MODULE Pattern;

FROM VDIControls IMPORT OpenVirtualWorkstation,
                        CloseVirtualWorkstation,
                        ClearWorkstation;

FROM VDIOutputs  IMPORT PolyLine;
FROM VDIInputs   IMPORT HideCursor;
FROM VDIAttribs  IMPORT SetWritingMode;
FROM GEMVDIbase  IMPORT VDIWorkInType,VDIWorkOutType;
FROM AESGraphics IMPORT GrafHandle,GrafMouseKeyboardState;

VAR handle:INTEGER;
    In:VDIWorkInType;
    Out:VDIWorkOutType;

(*=====*)

MODULE RandomNumbers;
EXPORT Random;
CONST M = 1000000000;
      m1 = 10000;
      b = 31415821;
VAR seed : LONGCARD;

PROCEDURE Random ( maxvalue : LONGCARD ) : CARDINAL;

  PROCEDURE Multiply ( p, q : LONGCARD ) : LONGCARD;
  VAR p0, p1, q0, q1 : LONGCARD;
  BEGIN
    p1 := p DIV m1;
    p0 := p MOD m1;
    q1 := q DIV m1;
    q0 := q MOD m1;
    RETURN ((p0*q1+p1*q0) MOD m1) * m1 + p0*q0) MOD M;
  END Multiply;

  BEGIN
    seed := (Multiply (seed, b) + 1) MOD M;
    RETURN CARDINAL((((seed DIV m1) * maxvalue) DIV m1) MOD 65536);
  END Random;

  BEGIN (* MODULE *)
    seed := 349867;
  END RandomNumbers;

```



```
(*=====*)  
  
PROCEDURE Line(x1,y1,x2,y2:INTEGER);  
VAR xyArray:ARRAY [0..3] OF INTEGER;  
BEGIN  
    xyArray[0]:=x1;  
    xyArray[1]:=y1;  
    xyArray[2]:=x2;  
    xyArray[3]:=y2;  
    PolyLine(handle,2,xyArray);  
END Line;  
  
PROCEDURE Init;  
VAR i,dummy:INTEGER;  
BEGIN  
    handle:=GrafHandle(dummy,dummy,dummy,dummy);  
    FOR i:=0 TO 9 DO In[i]:=1 END;  
    In[10]:=2;  
    OpenVirtualWorkstation(In,handle,Out);  
    HideCursor(handle);  
    ClearWorkstation(handle);  
    dummy:=SetWritingMode(handle,3);  
END Init;  
  
PROCEDURE Endit;  
BEGIN  
    CloseVirtualWorkstation(handle);  
END Endit;  
  
PROCEDURE Show;  
VAR x,y,x0,y0,  
    dx,dy,xabs,yabs,  
    xmax,ymax,xrand,yrand,  
    dummy,click: INTEGER;  
BEGIN  
    xrand:=Out[0];  
    yrand:=Out[1];  
    REPEAT  
        ClearWorkstation(handle);  
        x:=0;  
        y:=0;  
        dx:=1;  
        dy:=1;  
        xmax:=(xrand DIV 2)+INTEGER(Random(LONGCARD(xrand DIV 2)));  
        ymax:=(yrand DIV 2)+INTEGER(Random(LONGCARD(yrand DIV 2)));
```

```
REPEAT
  IF dx<0 THEN xabs:=x ELSE xabs:=xmax-x END;
  IF dy<0 THEN yabs:=y ELSE yabs:=ymax-y END;
  IF xabs<yabs THEN
    x0:=x+dx*xabs;
    y0:=y+dy*xabs;
    dx:=-dx;
  ELSIF xabs>yabs THEN
    x0:=x+dx*yabs;
    y0:=y+dy*yabs;
    dy:=-dy;
  ELSE
    x0:=x+dx*xabs;
    y0:=y+dy*yabs;
    dx:=-dx;
    dy:=-dy;
  END;
  Line(x,y,x0,y0);
  Line(xrand-x,y,xrand-x0,y0);
  Line(x,yrand-y,x0,yrand-y0);
  Line(xrand-x,yrand-y,xrand-x0,yrand-y0);
  x:=x0;
  y:=y0;
  GrafMouseKeyboardState(dummy,dummy,click,dummy);
  UNTIL (click<>0) OR (x=0) AND (y=0);
UNTIL click=2;
END Show;

BEGIN
  Init;
  Show;
  Endit;
END Pattern.
```

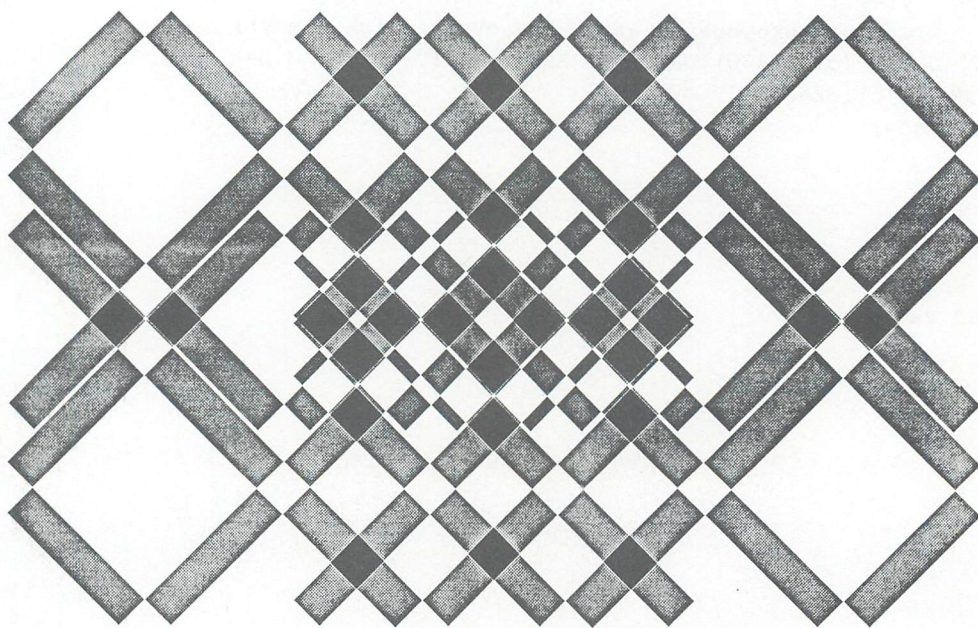
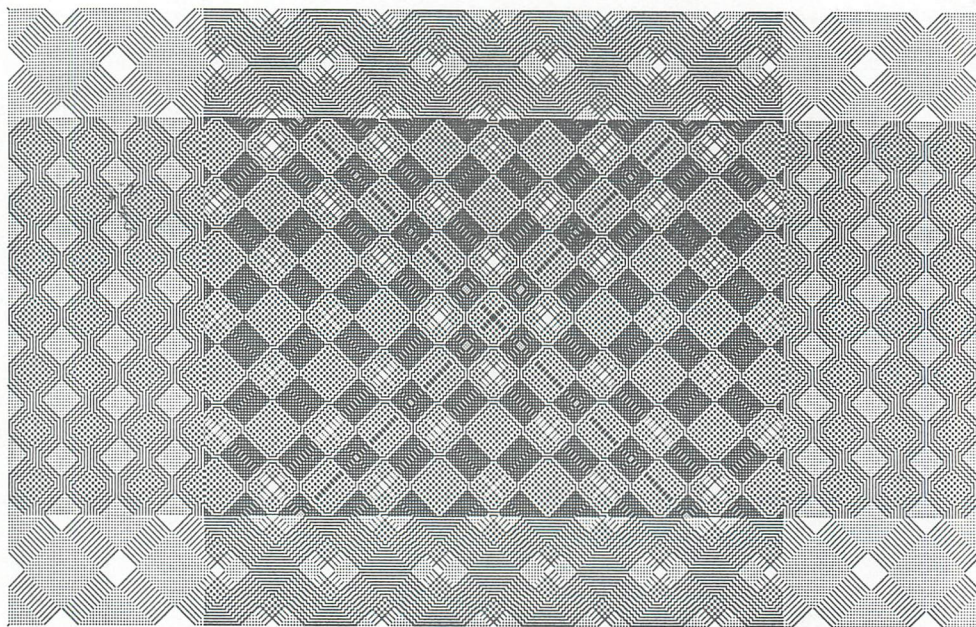



Figure 48: Hardcopy of PATTERN.MOD

5.2.2 Complex Mathematics and Fractals

A fractal is a combination of art and mathematics. Fractals can be produced on the computer by a specialized function plotting program.

Normally we enter a number of values used by a function. In this case, the function values are used again by the same function. Such a relationship between input and program can have the following results:

Either:

- the sequence of function values will converge to a single point
- the values will alternate between several points, or
- the values will fail to converge

The French mathematician Gaston Julia worked with these types of numerical relationships during WWI, and published his findings in the paper *"Squaring and Adding a Number."*

Benoit Mandelbrot looked both at sequences similar to and different from Julia's in his work, *"The Fractal Geometry of Nature."* He explored mathematical relationships in the range of the complex number planes, and published the results in the paper mentioned above. We now have the ability to display the calculations graphically.

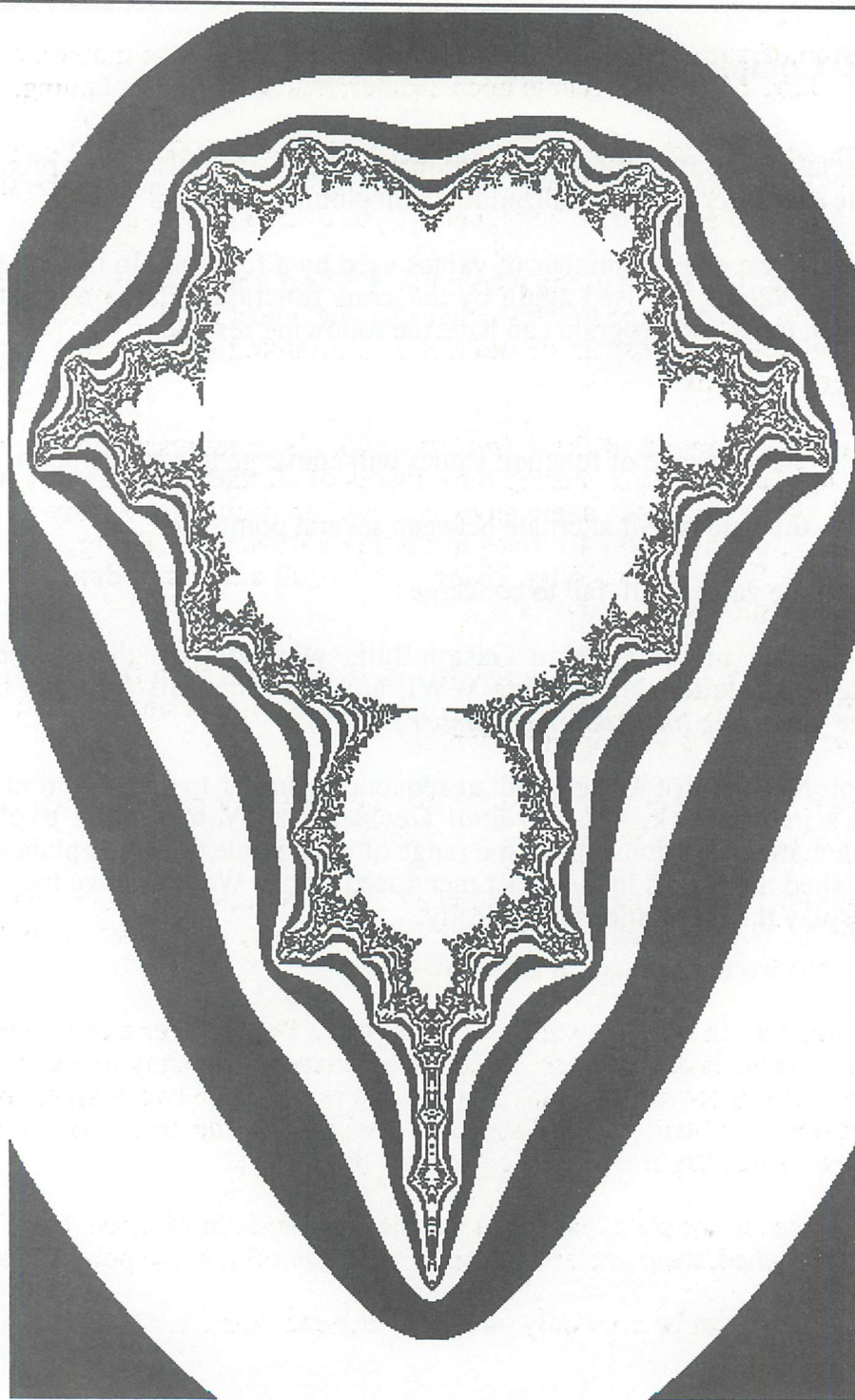


Figure 49: The primitive fractal

The fractal figure comes from the graphic patterns of these mathematical relationships. Mandelbrot came upon a pattern that was self-containing.

Imagine seeing a coastline from an airplane window at 6000 feet. You'll see a particular stretch of the beach. From a satellite you'd see almost the same thing, although since you'd be higher up, you'd see a larger stretch.

This is what the simulations technique is based upon. A large object is represented by an image much smaller than the object itself. For instance, a flight simulator represents mountains as a grouping of lines that can be enlarged or shrunk.

As a simple example of these fractals, think about representing the New York skyline. Instead of storing the outlines of all those skyscrapers (they all look pretty much the same anyway), you can digitize one story of one building. You then loop 40 times to display a 40-story building, and 100 times for a 100-story building. Using just a small amount of data you can build a realistic-looking city.

Back to the fractal. As mentioned, Mandelbrot found such fractals within mathematics. He could use them to represent complex numbers that are made up of a real and an imaginary part.

The complex number is represented by an ordered x,y pair in which every point stands for a particular complex number.

A beginning value, the complex number C , is subtracted from a given set of functions. The difference, also a complex number, is squared. C is then subtracted from the square, and so on.

There are certain function values that converge. That is, after a certain point the same value is produced from then on no matter how many times more it carries out the operation. Until this point is reached the two mathematical operations— subtraction and squaring—will cause the result to alternate between values larger and smaller than the final value.

For this reason we set a maximum calculations depth in advance. Once this point is reached, the calculation should be broken off and the point plotted.

This program can be used only on a monochrome monitor.

If you're using a color monitor you can produce fantastic color graphics. We can calculate the color using the calculation depth. The point's color can be set using `DEPTH MODULO NUMBER OF COLORS`.

How do you get to the different fractals?

Print a hardcopy of a primitive fractal. To do this, input the first value from the outputted example values.

That means you set the length of the x axis, the "real" part of the number. For the imaginary part, you decide the value of the y axis.

The different fractal are enlargements of a section of this output. All you have to do to enlarge a section is change the values of the x and y coordinates that form the boundary of the picture.

Now experience for yourself the working of these fractal relationships, since running it gives you a new picture ... and contains the next picture ...you choose a piece, and ...

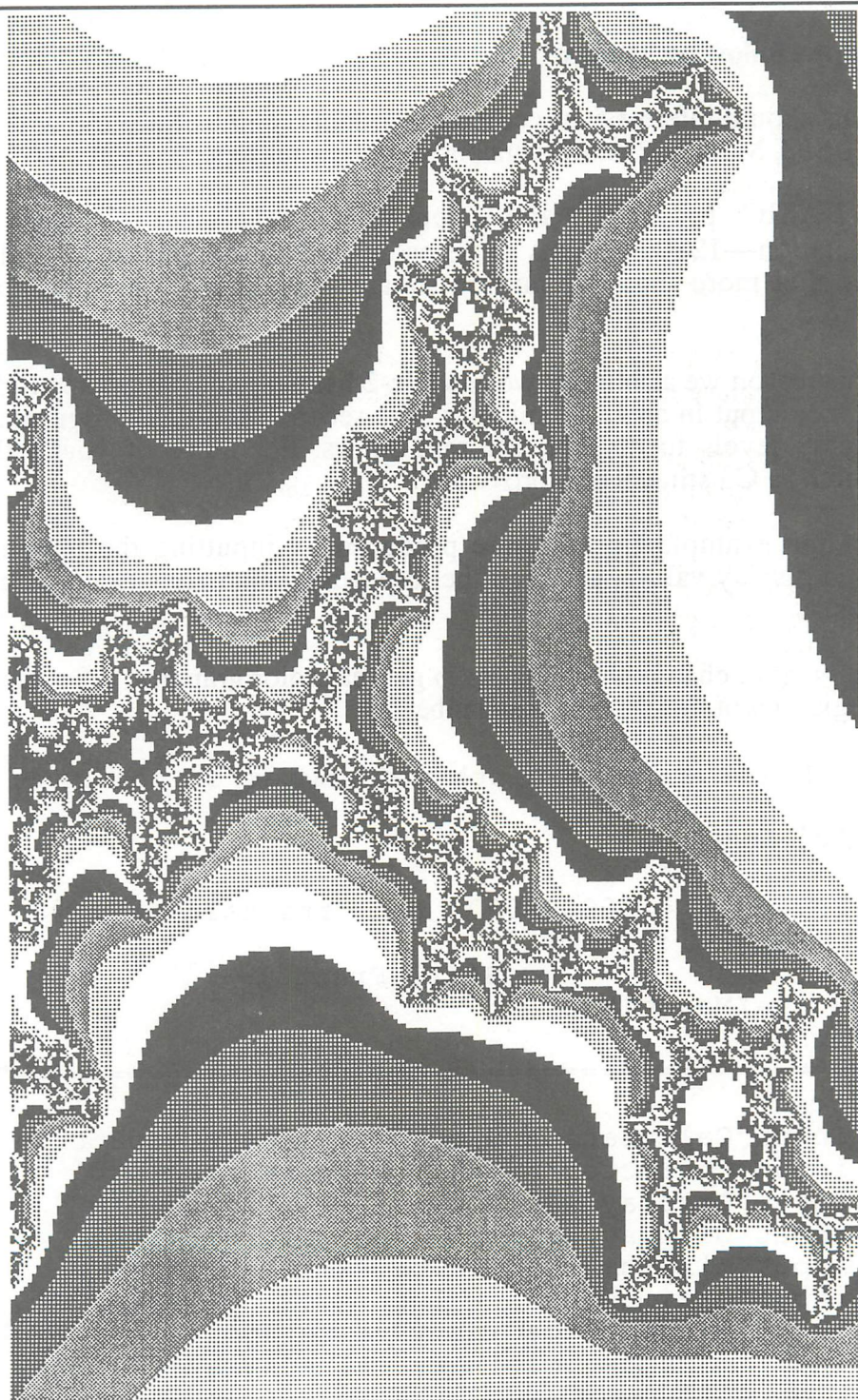


Figure 50: Fractal using parameters 0.0, 0.2, -1.06, -0.9

About the following listings:

Fractals should interest every computer graphics enthusiast. Following are the BASIC, Modula-2 and C versions of the fractal program.

The BASIC program calculations may take several hours of computation—128000 points to be calculated to a depth of 10, which means that more than 1.2 million squares and differences have to be calculated.

For this reason we also have two Modula-2 listings, since they can produce the same output in about 30 minutes. The second Modula-2 program uses four gray levels to display different colors. For those of you with C compilers, a C listing is also provided.

All of the example figures were produced by inputting the maximum allowed display values, and then the respective parameters listed with the figures.

If you want to change the program to get real color graphics, refer back to the beginning of this section. Remember:

$$\text{Point_color} = \text{depth mod number_of_colors}$$

Now the listings:

```
10  infinity = 1000000000  rem  fractal.bas
20  clearw 2 : fullw 2
30  print "          BASIC Fractal--
    Julia/Mandelbrot "
40  print
    "=====
50  print
60  input "Calculation depth "; calculationdepth
70  input "Character depth   "; characterdepth
80  input "Display width     "; xwidth
90  input "Display height    "; height
100 input "minimum real value (xmin) "; remin
110 input "maximum real value (xmax) "; remax
120 input "minimum imaginary value (ymin) "; immin
130 input "maximum imaginary value (ymax) "; immax
140 clearw 2
```

```
150 dreal = (remax - remin) / (xwidth - 1)
160 dimag = (imax - immin) / (height - 1)
170 for iimag=0 to height
180 for ireal=0 to xwidth
190 lreal = remin + ireal * dreal
200 limag = immin + iimag * dimag
210 re = lreal
220 im = limag
230 depth = 0
240 xx = re * re : yy = im * im
250 im = 2 * re * im - limag
260 re = xx - yy - lreal
270 depth = depth + 1
280 if((depth=calculationdepth) or
      ((xx + yy) > infinity)) then 290 else 240
290 if(depth<calculationdepth)then gosub 330
300 next ireal
310 next iimag
315 end
320 'plotting
330 if (depth>characterdepth) then linef
      ireal,iimag,ireal,iimag else gosub 350
340 return
350 if (depth mod 2) then linef
      ireal,iimag,ireal,iimag
360 return
```

Modula-2 Fractal program (black and white)

```
MODULE Apple;

FROM VDIControls IMPORT OpenVirtualWorkstation,
                        CloseVirtualWorkstation,
                        ClearWorkstation;

FROM VDIOutputs  IMPORT PolyMarker;
FROM VDIInputs   IMPORT ShowCursor,HideCursor,SampleMouseButton;
FROM GEMVDIbase  IMPORT VDIWorkInType,VDIWorkOutType;
FROM AESForms    IMPORT FormAlert;
FROM AESGraphics IMPORT GrafHandle;
FROM TextIO      IMPORT WriteString,WriteLn,ReadInt,ReadReal;

VAR handle:INTEGER;
    In:VDIWorkInType;
    Out:VDIWorkOutType;

PROCEDURE Point(x,y:INTEGER);
VAR xyArray:ARRAY [0..1] OF INTEGER;
BEGIN
    xyArray[0]:=x;
    xyArray[1]:=y;
    PolyMarker(handle,1,xyArray);
END Point;

PROCEDURE Init;
VAR i,dummy:INTEGER;
BEGIN
    handle:=GrafHandle(dummy,dummy,dummy,dummy);
    FOR i:=0 TO 9 DO In[i]:=1 END;
    In[10]:=2;
    OpenVirtualWorkstation(In,handle,Out);
    IF Out[13]>2 THEN
        dummy:=FormAlert(1,['[3][Only in Black and White!][Endit]');
        HALT;
    END;
    HideCursor(handle);
END Init;

PROCEDURE Endit;
BEGIN
    CloseVirtualWorkstation(handle);
END Endit;
```

```

PROCEDURE Draw;
CONST infinity=1.0E08;
VAR  breadth,height,ireal,iimag,
      dummy,click,choice,depth,
      calculationdepth,characterdepth:INTEGER;
      remin,remax,immin,imax,
      dreal,dimag,lreal,limag,
      xx,yy,re,im:REAL;

PROCEDURE Input;
BEGIN
  WriteString('Apple Men      -  Julia Men  -  Mandelbrot Men ');
WriteLn;
  WriteString('=====');
WriteLn;
  WriteString('Calculation Depth.....[ex. 10..20] ? ');
  ReadInt(calculationdepth); WriteLn;
  WriteString('Character Depth.....[ex. 5..15] ? ');
  ReadInt(characterdepth); WriteLn;
  WriteString('Picture Breadth.....[max. 640] ? ');
  ReadInt(breadth); WriteLn;
  WriteString('Picture Length.....[max. 400] ? ');
  ReadInt(height); WriteLn;
  WriteString('Z-Re-Min [zB: -0.75  0.0  1.1  1.6 ] ? ');
  ReadReal(remin); WriteLn;
  WriteString('Z-Re-Max [zB:  2.25  0.2  1.27  1.8 ] ? ');
  ReadReal(remax); WriteLn;
  WriteString('Z-Im-Min [zB: -1.2  -1.06 -0.35 -0.05] ? ');
  ReadReal(immin); WriteLn;
  WriteString('Z-Im-Max [zB:  1.2  -0.9  -0.25  0.05] ? ');
  ReadReal(immax); WriteLn;
  dreal:=(remax-remin)/FLOAT(CARDINAL(breadth-1));
  dimag:=(imax-immin)/FLOAT(CARDINAL(height-1));
END Input;

BEGIN
  REPEAT
    ClearWorkstation(handle);
    Input;
    ClearWorkstation(handle);
    iimag:=0;
    REPEAT
      ireal:=0;

```



```

REPEAT
    lreal:=remin+FLOAT(CARDINAL(ireal))*dreal;
    limag:=immin+FLOAT(CARDINAL(iimag))*dimag;
    re:=lreal;
    im:=limag;
    depth:=0;
    REPEAT
        xx:=re*re;
        yy:=im*im;
        im:=2.0*re*im-limag;
        re:=xx-yy-lreal;
        depth:=depth+1;
    UNTIL (depth>=calculationdepth) OR ((xx+yy)>infinity);
    IF depth<calculationdepth THEN
        IF (depth>characterdepth)
            THEN Point(ireal,iimag)
            ELSE IF ODD(depth)
                THEN Point(ireal,iimag) END;
        END;
    END;
    ireal:=ireal+1;
    SampleMouseButton(handle,click,dummy,dummy);
    UNTIL (click=1) OR (ireal>=breadth);
    iimag:=iimag+1;
    UNTIL (click=1) OR (iimag>=height);
    WHILE click<>1 DO SampleMouseButton(handle,click,dummy,dummy); END;
    ShowCursor(handle,0);
    choice:=FormAlert(1,
        '[1][>>  APPLE  MEN  <<|from ABACUS SOFTWARE|Sound &
Graphics][Continue|End]');
    HideCursor(handle);
    UNTIL choice=2;
END Draw;

BEGIN
    Init;
    Draw;
    Endit;
END Apple.

```

Modula-2 Program (four gray levels)

```

MODULE Apple2;
(* Version 2 with Grey Levels from 0=White to 4=Black *)

FROM VDIControls IMPORT OpenVirtualWorkstation,
                        CloseVirtualWorkstation,
                        ClearWorkstation;

FROM VDIOutputs  IMPORT PolyMarker;
FROM VDIInputs   IMPORT ShowCursor,HideCursor,SampleMouseButton;
FROM GEMVDIbase  IMPORT VDIWorkInType,VDIWorkOutType;
FROM AESForms    IMPORT FormAlert;
FROM AESGraphics IMPORT GrafHandle;
FROM TextIO      IMPORT WriteString,WriteLn,ReadInt,ReadReal;

VAR handle:INTEGER;
    In:VDIWorkInType;
    Out:VDIWorkOutType;

PROCEDURE Point(x,y,Greylevel:INTEGER);
VAR xyArray:ARRAY [0..3] OF INTEGER;
BEGIN
    x:=x+x; y:=y+y; (* 640 mal 400 *)
    IF Greylevel=0  (* 0=White 4=Black *)
    THEN RETURN;
    ELSIF Greylevel<>2
    THEN xyArray[0]:=x; xyArray[1]:=y;
        PolyMarker(handle,1,xyArray); END;
    IF Greylevel>1
    THEN xyArray[0]:=x+1; xyArray[1]:=y;
        xyArray[2]:=x;   xyArray[3]:=y+1;
        PolyMarker(handle,2,xyArray); END;
    IF Greylevel>3
    THEN xyArray[0]:=x+1; xyArray[1]:=y+1;
        PolyMarker(handle,1,xyArray); END;
END Point;

PROCEDURE Init;
VAR i,dummy:INTEGER;
BEGIN
    handle:=GrafHandle(dummy,dummy,dummy,dummy);
    FOR i:=0 TO 9 DO In[i]:=1 END;
    In[10]:=2;
    OpenVirtualWorkstation(In,handle,Out);
    IF Out[13]>2 THEN
        dummy:=FormAlert(1,['3][Only in Black & White!][Endit]');
    HALT;

```

```

END;
  HideCursor(handle);
END Init;

PROCEDURE Endit;
BEGIN
  CloseVirtualWorkstation(handle);
END Endit;

PROCEDURE Draw;
CONST infinity=1.0E06;
VAR  breadth,height,ireal,imag,
      dummy,click,choice,depth,
      CalculationDepth,CharacterDepth:INTEGER;
      remin,remax,immin,imax,
      dreal,dimag,lreal,limag,
      xx,yy,re,im:REAL;

PROCEDURE Input;
BEGIN
  WriteString(' Apple  men  - Julia  men  - Mandelbrot  men ');
  WriteLn;
  WriteString('=====');
  WriteLn;
  WriteString('Calculation Depth.....[ca. 10..99] ? ');
  ReadInt(CalculationDepth); WriteLn;
  WriteString('Character Depth.....[ca. 1..33] ? ');
  ReadInt(CharacterDepth); WriteLn;
  WriteString('Picture Width.....[max. 320] ? ');
  ReadInt(breadth); WriteLn;
  WriteString('Picture Height.....[max. 200] ? ');
  ReadInt(height); WriteLn;
  WriteString('Z-Re-Min [zB:  -0.75  0.0   1.1   0.14] ? ');
  ReadReal(remin); WriteLn;
  WriteString('Z-Re-Max [zB:   2.25  0.2   1.27  0.18] ? ');
  ReadReal(remax); WriteLn;
  WriteString('Z-Im-Min [zB:  -1.2  -1.06 -0.35 -1.05] ? ');
  ReadReal(immin); WriteLn;
  WriteString('Z-Im-Max [zB:   1.2  -0.9  -0.25 -1.02] ? ');
  ReadReal(immax); WriteLn;
  CharacterDepth:=(CharacterDepth DIV 5)*5+3;
                      (*Grey 3,then Black*)
  dreal:=(remax-remin)/FLOAT(CARDINAL(breadth-1));
  dimag:=(immax-immin)/FLOAT(CARDINAL(height-1));
END Input;

```

```

BEGIN

```

```

REPEAT
  ClearWorkstation(handle);
  Input;
  ClearWorkstation(handle);
  iimag:=0;
  REPEAT
    ireal:=0;
    REPEAT
      lreal:=remin+FLOAT(CARDINAL(ireal))*dreal;
      limag:=immin+FLOAT(CARDINAL(iimag))*dimag;
      re:=lreal;
      im:=limag;
      depth:=0;
      REPEAT
        xx:=re*re;
        yy:=im*im;
        im:=2.0*re*im-limag;
        re:=xx-yy-lreal;
        depth:=depth+1;
      UNTIL (depth>=CalculationDepth) OR ((xx+yy)>infinity);
      IF depth<CalculationDepth THEN
        IF (depth>CharacterDepth)
          THEN Point(ireal,iimag,4)
          ELSE Point(ireal,iimag,depth MOD 5) END;
      END;
      ireal:=ireal+1;
      SampleMouseButton(handle,click,dummy,dummy);
      UNTIL (click=1) OR (ireal>=breadth);
      iimag:=iimag+1;
      UNTIL (click=1) OR (iimag>=height);
      WHILE click<>1 DO SampleMouseButton(handle,click,
                                          dummy,dummy); END;

      ShowCursor(handle,0);
      choice:=FormAlert(1,
        '[1][>> APPLE MEN <<|from ABACUS SOFTWARE|Sound &
          Graphics][Continue|End]');
      HideCursor(handle);
      UNTIL choice=2;
    END Draw;

BEGIN
  Init;
  Draw;
  Endit;
END Apple2.

```

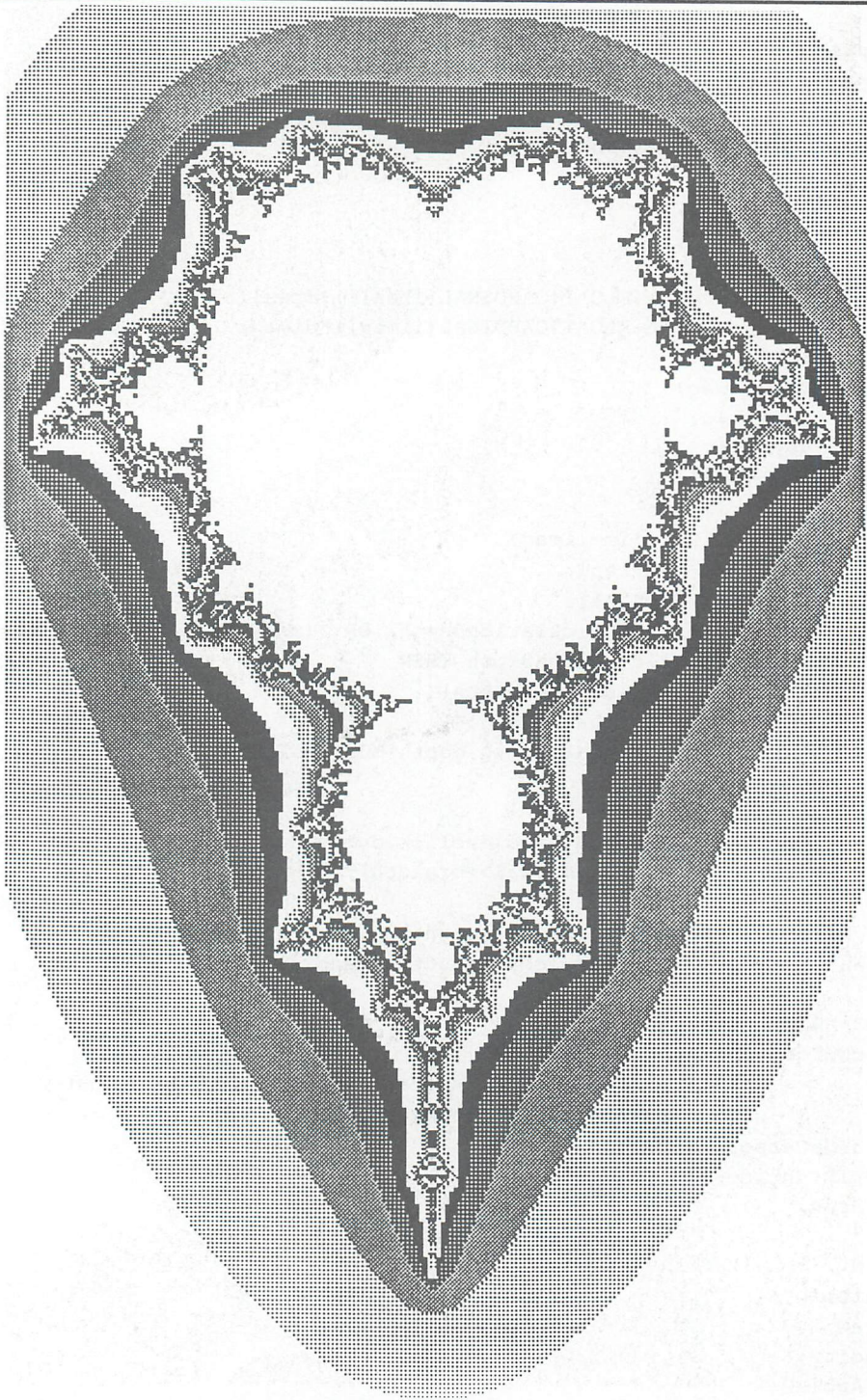



Figure 51: Sample fractal printout in monochrome (four gray levels)

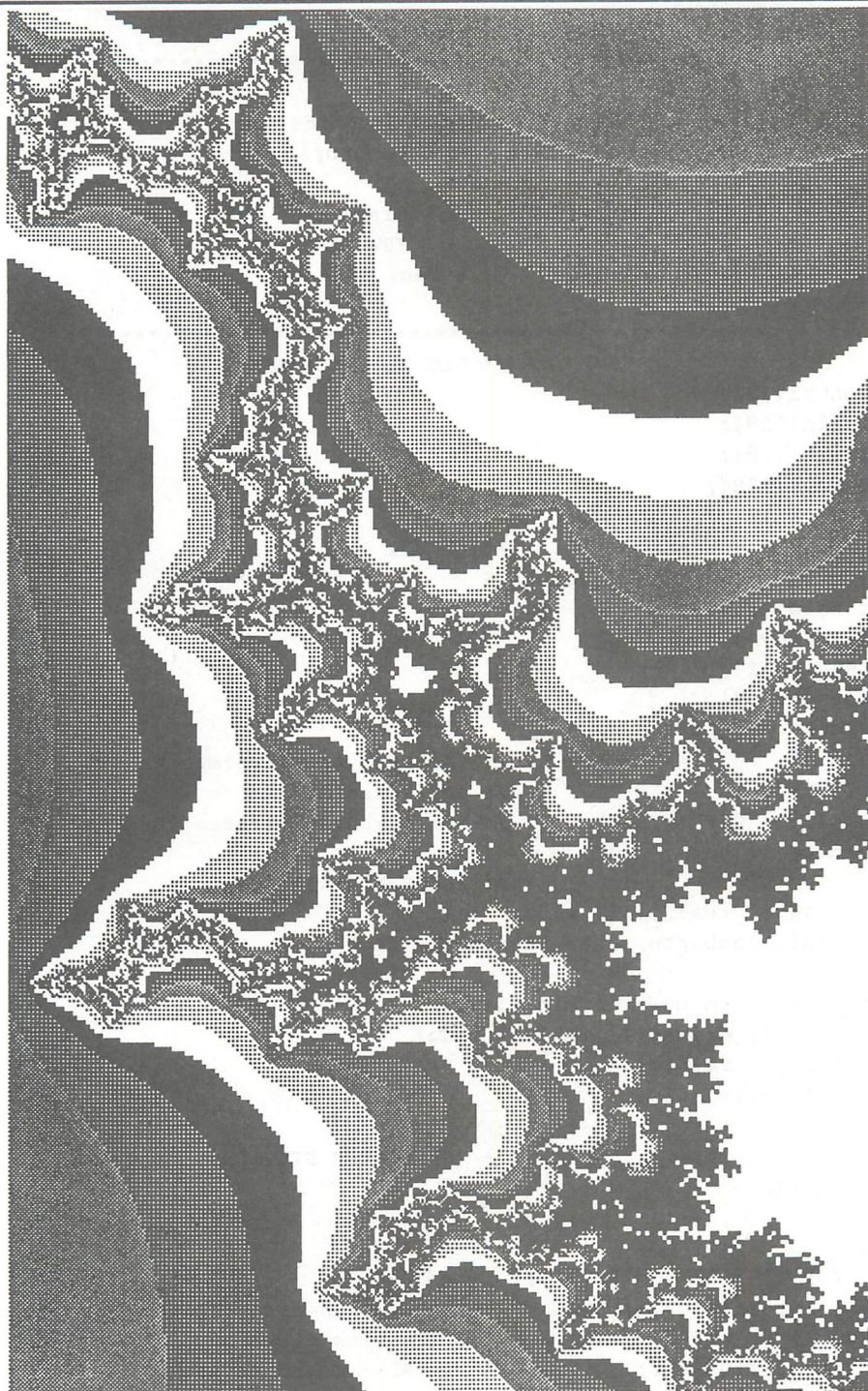


Figure 52: Fractal using parameters 1.1, 1.27, -0.35, -0.25

```

/*****
High Resolution Graphics for
Monochrome Monitors

from: ST Sound & Graphics
ABACUS SOFTWARE
V1 - Dec. 84

*****/

/* GLOBAL VARIABLES

int contrl[12];
int intin[128];
int ptsin[128];
int intout[128];
int ptsout[128];

/* GEMVDI & AES
GLOBALS */

int handle,i;
int pxyarray[1];
int int_in[11];
int int_out[57];
int width, height;
int dummy; /* for everything unimportant */
int click; /* for Mouse Click */
int choice = 0; /* for Number of Chosen Button */
int endit;

/* PROGRAM VARIABLES */
int i_imag, i_real,tall, wide;
int CalculationDepth, CharacterDepth, depth;

float re_min, re_max, im_min, im_max;
float d_real, d_imag, l_real, l_imag;
float xx, yy, re, im;
float infinity=1.0E08;

/* OPEN WORK STATION */
open_vwork()
{
int i;
for (i = 1; i <10; i++){
int_in[i] = 1; /* init int_in array: linetype, color, */
} /* fillstyles etc. */
int_in[10] = 2; /* use RC - Coordinates */
v_opnvwk(int_in, &handle, int_out); /* Now we can go... */
}

```



```

/* MAIN PROGRAM */
main()
{
    appl_init();
    handle=graf_handle(&width,&height,&dummy,&dummy);
    open_vwork();
    v_clrwk(handle);
    input();
    v_hide_c(handle);
    v_clrwk(handle);
    d_real=(re_max - re_min) / (wide - 1.0);
    d_imag=(im_max - im_min) / (tall - 1.0);
    i_imag = 0;
    do{
        i_real = 0;
        do{
            l_real = re_min + i_real * d_real;
            l_imag = im_min + i_imag * d_imag;
            re = l_real;
            im = l_imag;
            depth = 0;
            do{
                xx = re * re;
                yy = im * im;
                im = 2.0 * re * im - l_imag;
                re = xx - yy - l_real;
                depth++;
            }while ((depth < CalculationDepth) && ((xx + yy) <
                infinity));
            if (depth < CalculationDepth)
                if (depth > CharacterDepth)
                    plot(i_real,i_imag);
                else
                    if ((depth %2 ) != 0) /* % = modulo - Operator */
                        plot(i_real,i_imag);
                    i_real++;
                    vq_mouse(handle,&click,&dummy,&dummy);
            }while ((click == 0) && (i_real < wide));
            i_imag++;
        }while ((click == 0) && (i_imag < tall));
        while (click != 1){
            vq_mouse(handle,&click,&dummy,&dummy);
        }
    }
    desktop();
}

```



```

/* END AND BACK TO THE DESKTOP */
desktop()
{
    v_clsvwk(handle);
    appl_exit();
}

/* PLOT POINT */
plot(x,y)
int x,y;
{
    pxyarray[0] = x;
    pxyarray[1] = y;
    v_pmarker(handle,1,pxyarray);
}

/* INPUT THE USER DATA */
input()
{
    text(1,1," APPLE MEN - JULIA MEN - MANDELBROT MEN ");
    text(2,1,"=====");
    text(4,1,"CalculationDepth ..... [ca. 10..20] ?");
    CalculationDepth = mouse_input(0,100,4,43);
    text(6,1,"CharacterDepth ..... [ca. 5..15] ?");
    CharacterDepth = mouse_input(0,100,6,43);
    text(8,1,"PictureWidth..... [ max. 640] ?");
    wide = mouse_input(0,640,8,43);
    text(10,1,"PictureHeight..... [ max. 400] ?");
    tall = mouse_input(0,400,10,43);
    text(12,1,"Z-Re-Min [z. B. -0.75 0.0 1.1 1.6] ?");
    re_min = mouse_input(-3,3,12,43);
    text(14,1,"Z-Re-Max [z. B. 2.25 0.2 1.27 1.8] ?");
    re_max = mouse_input(-3,3,14,43);
    text(16,1,"Z-Im-Min [z. B. -1.2 -1.06 -0.35 -0.05?]");
    im_min = mouse_input(-2,2,16,43);
    text(18,1,"Z-Im-Max [z. B. 1.2 -0.9 -0.25 0.05 ?");
    im_max = mouse_input(-2,2,18,43);
}

mouse_input(w_min,w_max,line,column)
int w_min, w_max; /* Min & Max Values */
int line, column; /* Input at Position */
{
    int status; /* Lower Key = 1 else 0 */
    int mx,scale;
    double s_f,number;

```

```

if (w_min < 0){
    w_min = w_min * -1;
    w_max = w_max + w_min;
}
s_f = w_max / 639.0; /* Scaling */
do{
    vq_mouse(handle,&status,&mx,&dummy); /* sample mouse */
    number = mx * s_f - w_min;
    print(number,line,column);
}while (status == 0);
evnt_button(1,1,1,&dummy,&dummy,&dummy,&dummy);
evnt_button(1,1,0,&dummy,&dummy,&dummy,&dummy);
return(number);
}

/* Output of a Variables Contents */
/* Passed Value MUST BE of type DOUBLE */

print(var,line,column)
int line,column;
double var;
{
    char decimal[80];

    ftoa(var,decimal,6); /* 6 = Descendant Position */
    decimal[79]=0; /* initialized as...char [80] */
    v_gtext(handle,width * column,height * line,decimal);
}

/* addressed TEXT OUTPUT */
/*****/

text(line,column,string)
int line,column;
char string[80];
{
    v_gtext(handle,width * column,height * line,string);
}

```

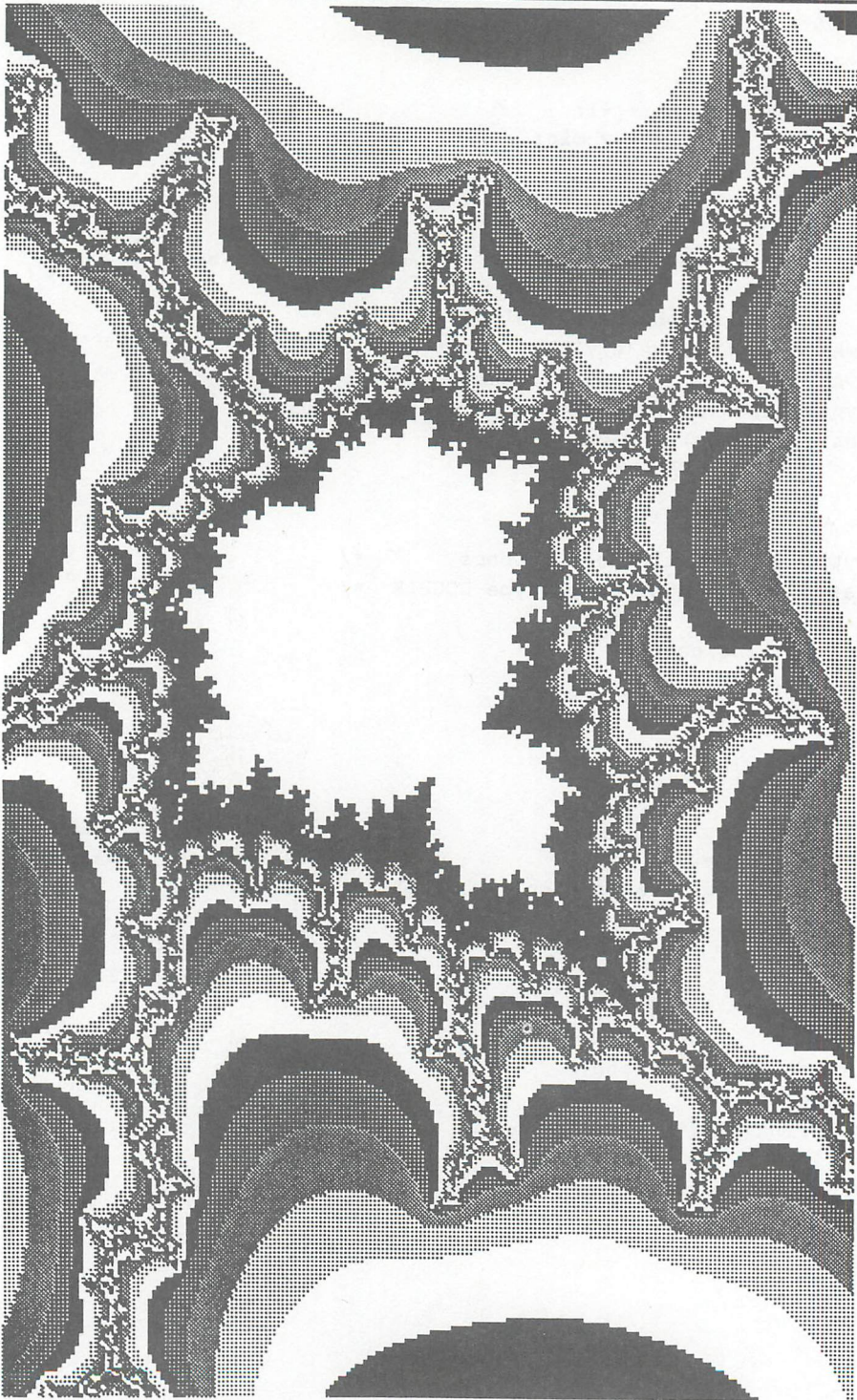



Figure 53: Fractal using parameters 0.14, 0.18, -1.05, -1.02

Chapter 6

Sound

6.1 Sound

Computer games without sound? Unthinkable—especially if we're talking about high-quality arcade-type games.

The distinctive "ping" and "pong" tones were the first commercial use of computer sound in the popular PONG arcade game of the mid-Seventies. Later, home computers like the Apple II let you synthesize these tone effects and hook up a loudspeaker to couple these "flip-flop" tones in the computer to create sound. Other computers like the TRS-80 sent several bytes in a timed sequence through a port containing an amplifier. These methods were pretty primitive by today's standards—they could only produce one voice, and the tonal quality left much to be desired.

But interest in computer sound was piqued. Hobbyists wanted better equipment for sound synthesis, and professionals worked to develop a miniature synthesizer that could emulate pipe organs and a whole range of classical instruments.

Then a breakthrough sound chip from General Instrument appeared: the AY-3-8910. It appeared shortly after the development of the Apple II and the TRS-80, and consequently was used in the home computers of the next generation (Colour Genie, Schneider CPC and MSX computer), where its performance won many ardent supporters.

Today's computers have multi-function sound chips built right into them. For instance, the Yamaha YM-2149 chip is used in the ST. It produces a tone, sound, an envelope generator, and a mixer—giving us all the tools we need to create a wide spectrum of sounds and music.

But how do we use this chip to produce various tones and sounds? How do we simulate instruments so they sound real? How can we play a whole melody, or a piece with several voices?

And what has Atari done to simplify the use of the YM-2149? How do we access it from BASIC and LOGO?

All these questions are answered in the following pages, which give important information about using the ST's sound capabilities.

6.2 The tone makes the music

The *tone* produced by the computer is the most important aspect of sound production using an electronic medium.

As you might know, a tone is produced by *oscillations*. These oscillations enter our ears after travelling through the air. The movement within the eardrum caused by the oscillations forms an electric pulse that is sent to the brain and produces the impression of a sound. The higher the tone, the faster the ear drum vibrates, and hence the higher the brain perceives the tone to be.

Why then does a string in a piano, which vibrates 440 times per second sound different from a string of the same length and same vibration speed on a banjo? Simple, someone may say, the string in an instrument has a soft "touch" and the other has a hard "touch".

How do these different touches produce different tone? To answer this question we must look into the characteristics of a tone.

We need to differentiate between periodic sound and aperiodic noise.

The difference lies in the instrument that produces the sound. For example, the same tone that sounds pleasant coming from a flute may sound harsh coming from a different instrument.

Of course, no instrument produces an absolutely pure tone. Rather they produce a combination of several tones out of basic vibrations and overtones.

This combination of sounds was explained in the early 1800's by French mathematician Joseph de Fourier in his work "*Theorie analytique de la chaleur*". His theory combined mathematics, physics and the current technology. He developed the harmonic analysis of sound by looking at pure sine vibrations and a constant. He theorized that the basic sounds are a combination of harmonics and basic frequencies.

The use of Fourier sequences makes possible the harmonic synthesis in an electronic medium. You can add a single vibration and get a result. In this way we can construct *rectangular* and *sawtooth* vibrations, for example.

6.3 The envelope

It's not only the overtones that are important for the typical sound of a particular instrument. The *envelope* is just as important to an instrument's distinctive sound. Each instrument has a distinctive envelope.

We differentiate between four different phases: *attack*, *decay*, *sustain* and *release*. For this reason, the envelope is often simply called an ADSR.

The different phases can be described as follows:

- **Attack** is the time period in which the tone begins at sound level zero until it reaches the maximum sound strength.
- **Decay** is the time period immediately following attack in which the sound level falls to a constant level.
- **Sustain** is the constant sound level described above, where the tone remains constant for an interval until it begins to fade away.
- **Release** is the time period in which the tone falls from the sustain level to silence.

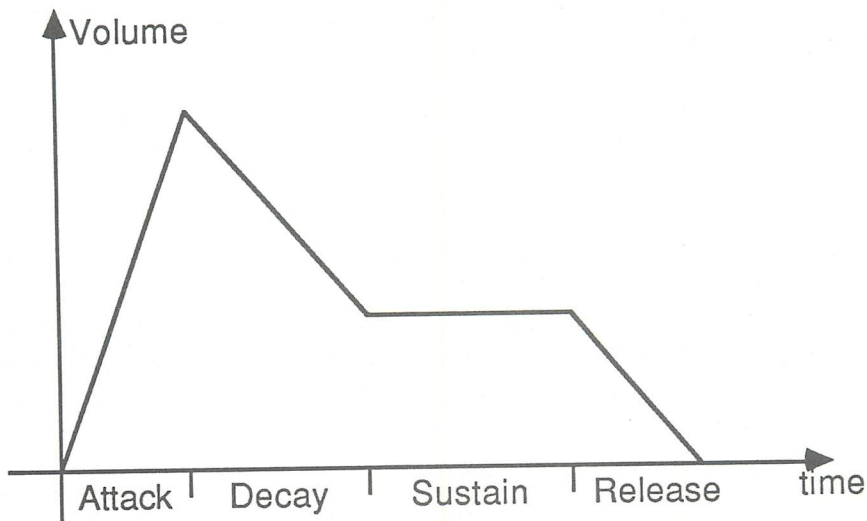


Figure 54: Example of an envelope

Think about blowing a trumpet. If you open the valve and blow, you don't immediately get the full sound strength. The level is more dependent upon how hard you blow the trumpet. If you increase the air pressure, the sound level will begin to rise (Attack), then begin to fall for a short time (Decay). It will temporarily maintain a certain sound level (Sustain), and finally fall off again (Release).

These four elements of an envelope account for the different sound produced using the same frequencies.

The envelope of a drum, for instance, will theoretically appear as a triangle on the envelope's graphic representation. The drumstick will strike the drum head which will start the curve. The drum head inhibits reverberations so there is no sustain phase. The tone will fall as quickly as it rose.

These curves also explain natural noise, of course. For example, a gunshot will very quickly reach its full noise level, then the level will fall and reverberate for a while. Theoretically it can be thought of as a percussion sound, like the drum. Natural noises are mostly completely aperiodic however—for example, the sound of a freight train.



6.4 The Synthesizer

Robert Moog used this theory in 1964 to build his famous Moog Synthesizer. He built a device that allowed him to experiment with tone, frequency, overtones, and sound level, and to change them at will. This Moog device was analog—i.e. the different parameters were attained by using different electric potentials.

Now, about twenty years later, we have a single IC that produces sound superior to the original Moog synthesizer. The ST contains a Programmable Sound Generator (PSG) with three voices. We can only play what a three-fingered piano player could—but we still have a lot of possibilities open to us.

The PSG also has an element that functions as a fourth voice—a *sound generator*—intended for use in the synthesis of aperiodic sounds, and which also acts as a mixer for the tone channel. It uses a *programmable envelope generator* to produce the various effects. The volume of each individual tone channel can be controlled.

The sound generator always produces a rectangular signal. Before the signal is output, a D/A transformer converts this analog signal so that the chips of the AY-3 can serve as a synthesizer.

6.5 The Chip

The ST's Programmable Sound Generator AY-3-8910 from General Instruments (replacement type: Large Scale Integration chip) can be used to produce complex tones and sounds under software control.

The PSG can do various tasks depending on how it is initialized. The AY-3 sound generator works independently of the 68000. All control signals given to the chip are in digital form. This is why an additional analog component is not needed, and why the PSG is directly connected to the 68000 microprocessor.

The connection between the microprocessor and the PSG is made using memory locations. The tone generator uses a memory-mapped component. Register 16 is used for communication. The CPU can write to and read from this area. Similar functions of the PSG are taken care of using registers \$00 through \$0F, which the PSG controls by itself. As soon as the processor has loaded the register, the PSG begins its work. It can produce a tone for as long as the instruction stays in the register. In the meantime, the 68000 can continue running the program.

Now lets take a look at the components of the sound chip. There are six functional blocks:

1. The Tone Generator

This component produces the desired frequency for the three channels: A, B, and C. It always produces rectangular waves.

2. The Sound Generator

At the output from the sound generator, a rectangular wave is available. It's frequency can be changed.

3. The Mixer

There are three mixers, one for each channel. This functional block mixes the output of the tone generators with the sound frequency.

4. The D/A Transformer

As mentioned before, the AY-3-PSG works digitally, which means the signals are converted into analog form before it goes through the loudspeaker into our ears. The output of the digital/analog transformer is between 0 and 1 volt, depending on the desired sound volume. So that it sounds right to the human ear, the number is not treated as an absolute level but as a logarithm.

Naturally the D/A transformer must know which sound level is desired. This information is given using four bits. The control parameters of the sound level is represented by values 0 through 15.

The frequencies to be transformed come to the D/A transformer from the mixer (tone and/or sound).

5. The Amplitude Control

The amplitude—the level of a vibration—is set by the sound level of the tone and is passed to the D/A transformer from this functional block. There are two distinctly different ways of doing this. To produce a constant sound level (in the range 0 to 15) by loading the appropriate control register, the exit amplitude can be controlled directly by the CPU. To produce a variable sound level, the output amplitude can be controlled by the envelope generator.

6. The Envelope Generator

To allow for amplitude modulation of the mixer's output signal we can use the envelope generator.

These previous descriptions apply to several sound chips: General Instruments AY-3-8910, AY-3-8912, AY-3-8913, and the Yamaha YM-2149. Using chips YM-2149 and AY-3-8910, we discover that two functional blocks that have nothing to do with tone production. There are two 8-bit I/O ports in this area that have important uses in the ST.

The AY-3-8912 differs from the two types mentioned in that it can only work through one port. The AY-3-8913 has no port.

The following is the Block Diagram of the PSG:

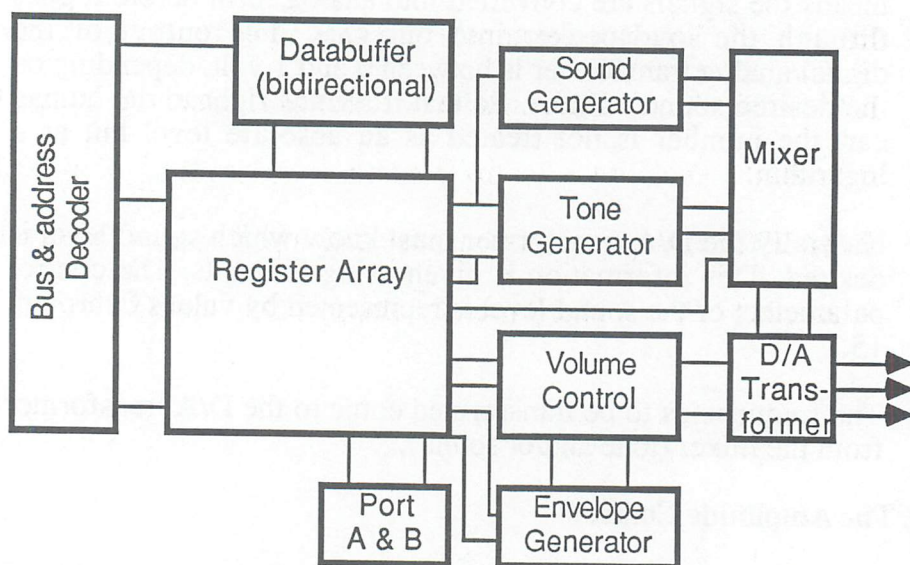


Figure 55: PSG Block Diagram

6.5.1 The Register Array

If you look at the Block Diagram, you will notice that an important function block is omitted. We left out the register array, which is the sound generator's "command post". Here the functions of the chip are activated. We must pay close attention to these registers.

The 68000 processor uses 16 read and write registers as a memory block to accomplish this task. In the following diagram, we refer to these registers as R0 to R15 (hexadecimal \$00 to \$0F). Earlier we saw another data page—pay attention to the fact that that register is represented in octal. That means that register 07 is followed by register 10. The highest register then has the value 17.

The registers of this function block are set up as follows:

- R0—R5 Tone generator sets the frequency
- R6 Sound generator sets the sound frequency
- R7 Mixer turns on tone or sound in channels A, B, and C
- R8—R10 Amplitude control controls sound level for A, B, and C
- R11—R13 Envelope sets frequency and appearance of the envelope
- R14—R15 Port A and B used for floppy I/O in the ST

REG	FUNCTION	VALUE	BIT 7	6	5	4	3	2	1	0
0 1	Frequency Channel A	0 - 4095	8 - bit Fine Adjustment x x x x Coarse							
2 3	Frequency Channel B	0 - 4095	8 - bit Fine Adjustment x x x x Coarse							
4 5	Frequency Channel C	0 - 4095	8 - bit Fine Adjustment x x x x Coarse							
6	Sound Period	0 - 31	x x x 5 - bit Period							
7	Enable		Port Sound Tone B A C B A C B A							
8	Volume A	0 - 15 or H	x	x	x	H	1	1	1	1
9	Volume B	0 - 15 or H	x	x	x	H	1	1	1	1
10	Volume C	0 - 15 or H	x	x	x	H	1	1	1	1
11 12	Envelope Period	16 bits = 0 - 65535	8 - bit Fine Adjustment 8 - bit Coarse Adjustment							
13	Envelope		x	x	x	x	h	h	h	h
14	I/O Port A		8 - bit Parallel Port							
15	I/O Port B		8 - bit Parallel Port							

Figure 56: The Sound Chip Register

6.5.1.1 The register

1. Control of the tone generator

Registers R0 through R5 are used to set the tone frequency for the three channels: A, B, and C. Each channel has two 8-bit registers. Hence, each channel has 16 bits available—although only 12 bits are used. The values may range is from 0 to 4095. The rectangular vibrations are produced in the tone generator as follows:

The clock-signal in the PSG is divided by 16. The result is divided by the 12-bit value from the register. These are divided in the register for coarse adjustment (R1, R3, and R5, also known as Coarse Tune Register, the 4 bits, MSB of the 12-bit word) and the fine adjustment (R0, R2, and R4, Fine Tune, 8 Bit, LSB).

2. Control of the sound generator

Five of the 8 bits in register R6 (0 to 4) are used to control the sound generator. In other words, the values of the sound generator range from 0 to 31. Again, the clock signal is first divided by 16 and then by the value in R6.

3. Control of the mixer

R7 is used as the multi-function register to turn various functional blocks (and the two ports) on and off. This means that this register has a specific bit which determines whether a function is on or off.

The low-order bits (0—2) control tone generators A, B, and C respectively. If the PSG finds only zeros here, the pure square frequency stored in R0 to R5 is output.

The bits 3 to 5 mix the channel sound. If we erase this, the pure square frequency is output.

B6 and B7 are important to the I/O ports (R14 and R15). Bit 7 is especially important in the ST. This bit controls port B, which is responsible for disk access. For this reason, this register should be read before each access to the sound chip, the value stored elsewhere, and then later restored. Now you know why the LED's of the disk

drive light up when you use any tone features using LOGO--an erroneous value was stored in this register.

4. Controlling the volume

R8, R9 and R10 control the output signal of the D/A transformer, and hence the volume of channels A, B, and C. Bits 5 to 7 aren't used. Bit 4 indicates the particular envelope pattern to be used (then B4 must be set), or whether the tone should be played at a constant volume. If B4 = 0, then the value in bits 0 to 3 set the level of the signal, ranging from 0 to 15.

5. Controlling the envelope generator

Three registers—R11, R12, and R13—are used for sound operations. They store the different envelopes. The various envelope options available in the PSG are implemented by two independent methods.

The first method uses a variation of the envelope's frequency in register R13 and R14. The value is attached to a 16-bit word, which allows values from 0 to 65535. The register is again divided in Envelope Coarse Tune (R12) and Envelope Fine Tune (R11). To set the envelope frequency, the PSG first divides clock frequency by 256 and combines the result with the programmed 16-bit value.

The second method is chosen from the different ADSR envelopes (Envelope/Cycle). The four low bits of register R13 serve as variables for hold, alternate, attack, and continue. Bits 4 to 7 aren't used. The envelope frequency be divided by 16. That's so the envelope will be divided into a cycle with 16 individual steps. The envelope control uses a counter to keep track of the steps up or down. The amplitude controller will be given the values of this counter.

There is a repeat command, but there are also envelopes which only have only a single cycle.

The following chart gives you an idea of the envelopes that are available for your use:

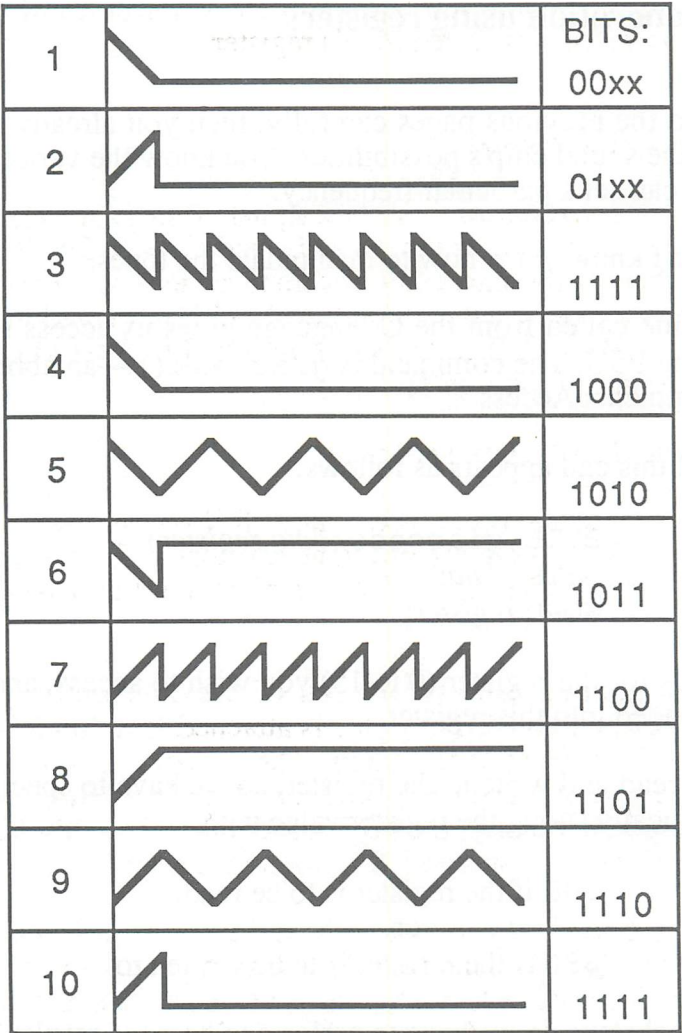


Figure 57: The PSG's envelopes

6.6 Tone generation using registers

If you've read the previous pages carefully, then you already have a good overview of the sound chip's possibilities. You know the various envelopes and how you can set a particular frequency.

What you don't know yet is how to manipulate the tones.

A GEM routine called from the C language gives us access to individual registers of the PSG. The command is `giaccess()`—an abbreviation for General Instruments Access.

The syntax of this call appears as follows:

```
BYTE giaccess(data,register)  
BYTE data;  
WORD register;
```

register stands for the register (0 to 15) you wish to access, and *data* is the value to be loaded into this register.

We can both read and write to the register, so we have to specify this. This is done by logically ORing the register value with:

```
$00 if the register is to be read,  
    or  
$80 if the register is to be written to
```

What additional steps do we have to perform to hear the results ?

There are at least three steps yet:

- Set the tone frequency (R0 to R5)
- Activate the desired channel (R7)
- Set the loudspeaker to a nonzero value (R8 to R10)

Before we decide on an exact value, let's give a command to the PSG :


```

for 1:
giaccess(0,0 + 128);
giaccess(100,1 + 128);

for 2:
giaccess(...,7 + 128);

for 3:
giaccess(15,8 + 128);

```

In this case we add 128 to each register. We used a mask value to read into the register instead of writing to it. Remember, we need to be careful about the status of register 15 (set with the value in R7) so that we can restore it later—we don't want to lose contact with our disk drive. Now we can write the following short C program, which produces a tone on channel A with the maximum sound level:

```

/* include files */
#include "obdefs.h" /* object definitions */
#include "define.h"
#include "gemdefs.h"
#include "gembind.h"
#include "osbind.h"

/*global variables */

int contrl[12];
int intin[128];
int ptsin[128];
int intout[128];
int ptsout[128]; /* enough space for all cases */
int handle,i; /* virtual workstation handle */
int width, height;
int pxyarray[12]; /* Array for x,y coordinates */
int pxy[4];
int int_in[11]; /* Input into GSX Array */
int int_out[57]; /* Output from GSX Array */
int dummy;

int ports; /* Status of PSG Register 14/15 */

main()
{
    appl_init(); /* Initialize Workstation */
    handle=graf_handle(&width,&height,&dummy,&dummy);

```



```

    open_vwork(intin,&handle,intout);
    sound(); /* Run the tone routine */
    click(); /* and wait for a click of the mouse */
    Giaccess(0,8 + 128); /* then: Change volume from A to 0 */
    Giaccess(ports, 7 + 128); /* and: Reset the port condition */
    v_clsvwk(); /* Close the workstation */
    appl_exit(); /* and return to the desktop */
}

open_vwork()
{
    int i;
    for (i = 1; i <10; i++){
        int_in[i] = 1; /* init int_in array: linetype, color, */
    } /* fillstyles etc. */
    int_in[10] = 2; /* utilize RC - Coordinates */
    v_opnvwk(int_in, &handle, int_out); /* Now things get going ... */
}

click() /* Wait for mouse click (left) */
{
    evnt_button(1,1,1,&dummy,&dummy,&dummy,&dummy);
}

sound()
{
    ports = Giaccess(dummy,7); /* Read Portstatus */
    Giaccess(0x8E,0 + 128); /* Frequency = */
    Giaccess(0,1 + 128);
    Giaccess(0x3E,7 + 128); /* Only produce tone
on channel A */
    Giaccess(15,8 + 128); /* Full sound volume */
}

```

Running the compiled program will produce a loud, pure tone from the monitor speaker. Now let's take a look at the tone we've produced.

As mentioned earlier, the sound generator produces tones by frequency divisions that are supplied by the master clock frequency. In the case of the ST, they come from a 2 Mhz clock. Internally this clock is first divided by 16, which produces a frequency of 125,000 Hertz. A further division by the contents of registers R0 and R1 (for channel A) produces the output frequency.

After the assignment:

$$\text{Tone period} = \frac{\text{clock frequency}}{\text{frequency} * 16}$$

Then the period can be calculated by the given clock frequency (here 2,000,000 Hertz).

Now we do another calculation:

$$\text{Frequency} = \frac{\text{clock frequency}}{\text{period} * 16}$$

We loaded R0 with \$1C and R1 with \$01, this hexadecimal number has the value 284. If we set the value in the above assignment, we get a value of about 440 Hz. Voila—we've produced an A note.

We can produce every tone on the scale, and then some. Let's take a closer look. The frequency of our tone generators A, B, and C are set by a 12-bit value so the values range ran 0 to 4095. Our system produces sound between 125000 Hz (period 0) and 30.53 Hz, ranging from ultra-low bass tones to high tones unable to be heard by humans.

Sometime we can't get a frequency exactly right, and must accept a certain amount of variance from what we really want. Since the PSG can accept only integer values, we can't produce all of the tones exactly. (For reference, look at the table in Appendix B).

To help you with the necessary calculations, here is a BASIC program. This program will output the two hex numbers to be passed to the tone generator register to produce a specified frequency.

```
0   ' CHANGE.BAS : Change Dec. # in Memory from
20   '             (0-32767 (16-bit))to hexadecimal
30   '
40   clearw 2 : fullw 2
50   dim digit$(16)
60   data 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
70   for digit=0 to 15 : read digit$(digit) : next
80   input "Decimal Number: ";decimal
90   if decimal < 0 then goto 80
100  if decimal<=255 then hbyte=0 :
      lowbyte=decimal: goto 130
110  lowbyte = decimal mod 256
120  hbyte = (decimal - lowbyte) / 256
130  dec = hbyte : gosub change :
      dec = lowbyte : gosub change
140  print
150  goto 80
160  '
170  rem decimal number between 0 und 255 into hex
180  change:
190  z1=int(dec/16)
200  z2=dec and 15
210  hexnum$=digit$(z1) + digit$(z2)
220  print hexnum$;
230  return
```

To finish up this section let's take another look at the C listing. To analyze each `giaccess` call, you'll either need to memorize all the register numbers or use the information in Figure 56.

In either case we get a program that's very difficult to read. To improve program's readability, the first essential step is to use *symbolic constants* extensively.

That means we set the up constants such as `CHANNEL_A` and `VOLUME_A` that we can use in every sound program. To avoid having to retype it, we store these definitions in the file `SOUND.H`. This file, already known from GEM definition files, uses `include#` to merge the values into our program.

The include file appears as follows:


```
#define CHANNELA_FINE 0
#define CHANNELA_COARSE 1
#define CHANNELB_FINE 2
#define CHANNELB_COARSE 3
#define CHANNELC_FINE 4
#define CHANNELC_COARSE 5
#define SOUNDCHANNEL 6
#define CHANNELNUM 7
#define PORT_STATUS 7
#define VOLUME_A 8
#define VOLUME_B 9
#define VOLUME_C 10
#define ENV_FINE 11
#define ENV_COARSE
#define ENV_SHAPE 13
#define ENV_VOLUME 15
#define ENV_9 9
#define WRITE 128

#define A_ON 0x3E /* Tone production */
#define B_ON
#define C_ON
#define AB_ON
#define AC_ON
#define BC_ON
#define ABC_ON
#define RA_ON /* Sound */
#define RB_ON
#define RC_ON
#define RAB_ON
#define RAC_ON
#define RABC_ON

#define A_OFF 0x00 /* Tone production */
#define B_OFF
#define C_OFF
#define AB_OFF
#define AC_OFF
#define BC_OFF
#define ABC_OFF
```



```
#define RA_OFF /* Sound */
#define RB_OFF
#define RC_OFF
#define RAB_OFF
#define RAC_OFF
#define RBC_OFF
#define RABC_OFF
```

Using these definitions, `sound()` appears as follows:

```
ports=Giaccess(dummy,PORT_STATUS);
Giaccess(0x1c,CHANNEL_FINE + WRITE)
Giaccess(0x01,CHANNEL_COARSE + WRITE)
Giaccess(A_ON,CHANNELNUM + WRITE)
Giaccess(15,VOLUME_A + WRITE)
```

It's more typing, but a lot more readable, isn't it?

6.7 Practical application

Let's say that we want a device that can produce as broad a frequency band as possible. We don't want to type in all the frequencies, because it's too slow and tedious. Instead, we want to use the mouse to enter values. That means we'll display a horizontal scale with a left border of the frequency 30.5 Hz, and a right border for a maximum value. Also, we want our variables displayed on screen as a check to make sure we've entered the correct values.

So we have four tasks:

1. Determine the mouse position
2. Calculate the x-coordinate of the period
3. Load the sound register
4. Display the variable contents on the screen

The first task is quite easy, since the GEM library has a command `evnt_mouse`. It gives the exact x- and y-coordinates of the mouse when it is clicked. With this value and our knowledge of scaling from the earlier graphic section, task two is rather easy as well. Also, the third task shouldn't give us any problem.

The biggest problem, at least for C novices, is displaying the period value on the screen. Digital Research's C compiler—in contrast to some other C packages—doesn't have a `printf()` function. As a result, those of you with the Digital Research compiler must do without `printf()`.

Luckily, we already know an equivalent function from the graphic section: `v_gtext()`. This function outputs a string. All we need to do is change our integer into a string. To do this we use the C function `ftoa(a, b, c)`—floating point to ASCII. Its parameters are:

- a the number to be changed into a string
- b the buffer of type char which will hold the character
- c the number of character positions

By using these functions we can write the function `printvar(x)`. The function does exactly what its name says.

We've also got a few other things to do, like opening the `work_station` and similar functions—we'll do this as we did in the preceding program for use in the following programs.

After compiling the C programs on the following pages, we can make practical use of the ST's sound capabilities. For example, we can use the programs to test a group of cassette recorders for response and sensitivity when recording and playing back the tones we generate with the ST.

6.8 Sound manipulation using the envelope

Enough pure tones for now. We'll move on to another interesting aspect of the ST's synthesizer: manipulating the square waves to mimic real sounds.

If we write a machine gun simulator, then pulling the trigger should make a true firing sound. If we program a melody, it would be nice if our ST could sound like a piano instead of like a computer.

To get closer to this goal we'll experiment with the different envelopes. Take a look at Figure 57, which shows ten different envelopes. The diagrams show how the tone's loudness in the PSG varies from beginning to end. Some of them repeat the envelope (e.g. 3, 5, 7, and 9). They move from an original state to another value state, then return to the original state and repeat the process several times.

Let's take a look at envelope 9.

Initially the PSG has a 0 volume, so we hear nothing. Then the volume goes up to the maximum level of 15—represented by the ascending section of the line. Then the loudness begins to diminish again until you can't hear it any more. The process is then repeated over and over.

This effect can be used to simulate the Doppler effect of a train coming closer to you before beginning to move away again.

We'll test out how the envelope number works in a test program:


```

/*****
/* sub2.c - Programming an Envelope
*****/
sound()
{
    ports = Giaccess(dummy,PORT_STATUS); /* Read Port Status
    Giaccess(0x1c,CHANNEL_FINE + WRITE); /* Frequency = 440 Hz
    Giaccess(0x01,CHANNEL_COARSE + WRITE);
    Giaccess(A_ON,CHANNELNUM + WRITE); /*Tone Production with channel
A*/
    Giaccess(ENV_VOLUME,VOLUME_A + WRITE);/* Use Envelope 9
    Giaccess(ENV_9,ENV_SHAPE + WRITE); /* Active Envelope 9
    click(); /* Wait for click of the mouse*/
    Giaccess(ALL_OFF,CHANNELNUM+WRITE); /* and turn PSG off
    Giaccess(ports,PORT_STATUS + WRITE); /* Reset port condition
}

```

Using the earlier definition file should make further explanation unnecessary.

Our next topic is tone shape.

In the register overview, you discovered something called EP, which we haven't said anything about for a while. EP stands for *Envelope Period*. We can use the EP to give the sound chip information about how long the attack of the sound level will be. In practice, this is usually done with the values in R11 and R12. A 16-bit value can be put in the register, which means the period can range from 1 to 65535. If we convert to Hertz, this gives us frequencies between 125,000 and 1.9 Hz.

Try out the following example, where we exchange 0xFF for 0x00:

```
/* **** */
/* sub3.c - Manipulating the Envelope */
/* **** */
sound()
{
    ports = Giaccess(dummy,PORT_STATUS);/* Read Port status */
    Giaccess(0x1c,CHANNEL_FINE + WRITE); /* Frequency = 440 Hz */
    Giaccess(0x01,CHANNEL_COARSE + WRITE);
    Giaccess(A_ON,CHANNELNUM + WRITE); /* Tone Production with channel
A*/
    Giaccess(ENV_VOLUME,VOLUME_A + WRITE);
    Giaccess(ENV_9,ENV_SHAPE + WRITE); /* Use Envelope 9 */
    Giaccess(0xFF,ENV_FINE + WRITE); /* Active Envelope 9 */
    Giaccess(0xFF,ENV_COARSE + WRITE); /* Period = 65535 */
    click();
    Giaccess(ALL_OFF,CHANNELNUM+WRITE); /* and turn off the PSG */
    Giaccess(ports,PORT_STATUS + WRITE);
}
```

By experimenting with the envelopes and envelope frequencies, we can produce sounds similar to musical instruments and many other sounds. Most natural sounds are aperiodic—which is why we're using the sound generator instead of the tone generator.

```
/******  
/* sub4.c - Sound of the Sea */  
/******  
sound()  
{  
    ports = Giaccess(dummy,PORT_STATUS);/* Read Port  
    Status */  
    Giaccess(0,0 + WRITE);  
    Giaccess(0,1 + WRITE);  
    Giaccess(0,2 + WRITE);  
    Giaccess(0,4 + WRITE);  
    Giaccess(0,5 + WRITE);  
    Giaccess(31,6 + WRITE);  
    Giaccess(199,7 + WRITE);  
    Giaccess(16,8 + WRITE);  
    Giaccess(16,9 + WRITE);  
    Giaccess(16,10 + WRITE);  
    Giaccess(0,11 + WRITE);  
    Giaccess(32,12 + WRITE);  
    Giaccess(14,13 + WRITE);  
    click();  
    Giaccess(A_OFF,CHANNELNUM+WRITE);  
    Giaccess(ports,PORT_STATUS + WRITE);  
}
```

These sea sounds have one problem: the noise always decays in the exact same way. To improve on the program, you should vary the output level, e.g. between 5 and 15.

Another example is synthesizing a gunshot. When the gunpowder explodes, the noise level quickly reaches its maximum level, and then quickly drops to 0. This is perfect for envelope 4:

```
/******  
/* sub5.c - BANG! */  
/******  
sound()  
{  
    ports = Giaccess(dummy,PORT_STATUS);  
    Giaccess(0,0 + WRITE);  
    Giaccess(0,1 + WRITE);  
    Giaccess(0,2 + WRITE);  
    Giaccess(0,4 + WRITE);  
    Giaccess(0,5 + WRITE);  
    Giaccess(15,6 + WRITE);  
    Giaccess(7,7 + WRITE);  
    Giaccess(16,8 + WRITE);  
    Giaccess(16,9 + WRITE);  
    Giaccess(16,10 + WRITE);  
    Giaccess(0,11 + WRITE);  
    Giaccess(16,12 + WRITE);  
    Giaccess(0,13 + WRITE);  
    click();  
    Giaccess(A_OFF,CHANNELNUM+WRITE);  
    Giaccess(ports,PORT_STATUS + WRITE);  
}
```


That should be enough examples. Next we want to experiment with the other envelopes. For this we enter lines from the subprogram TEST6.C, which activates an envelope when the left mouse button is clicked.

```
/******  
/* sub6.c - ENVELOPE DEMO                               */  
/******  
sound()  
{  
int curve;  
ports = Giaccess(dummy,PORT_STATUS);/* READ PORT STATUS */  
curve = 0;  
do{  
    Giaccess(0,0 + WRITE);  
    Giaccess(0,1 + WRITE);  
    Giaccess(0,2 + WRITE);  
    Giaccess(0,4 + WRITE);  
    Giaccess(0,5 + WRITE);  
    Giaccess(31,6 + WRITE);  
    Giaccess(199,7 + WRITE);  
    Giaccess(16,8 + WRITE);  
    Giaccess(16,9 + WRITE);  
    Giaccess(16,10 + WRITE);  
    Giaccess(0,11 + WRITE);  
    Giaccess(32,12 + WRITE);  
    Giaccess(curve,13 + WRITE);  
    click();  
    curve++;  
}while (curve<16);  
click();  
Giaccess(A_OFF,CHANNELNUM+WRITE);  
Giaccess(ports,PORT_STATUS + WRITE);  
}
```

6.9 Programming music from BASIC

You now know the capabilities of the ST's sound chip, and know how to use it as well. If you're writing professional games, you'll probably be working in C.

But if you're programming for your own purposes, you'll want to simplify things by working in an interpreted language. And you'd probably like an explanation and some examples of how to use the sound capabilities in BASIC and LOGO. Atari took your desires into consideration. Access to the sound generators is not reserved for hardware specialists and computer wizards.

The user who wants to use the tone generator in BASIC can avoid calculating the desired frequencies period and working with a high and a low value. He only needs to pay attention to following:

C	C#	D	D#	E	F	F#	G	G#	A	A#	B
1	2	3	4	5	6	7	8	9	10	11	12

The table above contains the values for an octave. The octave itself can be entered as a value between 1 and 8. All the chip then needs to know before it can play the desired tone is the tone channel (1, 2, or 3), and the the volume (0 to 15). The values are passed in this form:

`SOUND Channel, Volume, Note, Octave, Play_length`

To get BASIC to play the note A, we enter:

```
SOUND 1, 15, 10, 4, 50
```

You'll recognize this tone as our old test sound. It will continue to play until you press a key.

The same tone range is given by the following program:

```
10 REM Tonerange
20 data 1,3,5,6,8,10,12,-1
30 for octave = 1 to 8
40 restore
50 read note
60 sound 1,15,note,octave,50
70 if note = 12 then 80 else 50
80 next period
90 end
```

You can also play an entire melody. Enter and run the listing on the following pages:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100

```
10  fullw 2: clearw 2
20  print " *****      YOU CAN WIN IF YOU
    WANT *****"
30  read volume,note,octave,length
40  if volume=-1 then end
50  sound 1,volume+4,note,octave,length/3
60  i=i+1
70  if i<> 101 then 30
80  restore 280
90  read volume,note,octave,length
100 if volume=-1 then end
110 sound 1,volume+4,note,octave,length/3
120 i=i+1
130 if i<> 229 then 90
140 restore 420
150 read volume,note,octave,length
160 if volume=-1 then end
170 sound 1,volume+4,note,octave,length/3
180 i=i+1
190 goto 150
200 data 7,3,4,50, 7,3,5,100, 0,0,0,5, 7,3,5,50,
    7,1,5,50
210 data 7,10,4,50, 7,8,4,25
220 data 7,10,4,25, 7,10,4,100, 7,3,5,50,
    7,1,5,50
230 data 7,10,4,50, 7,8,4,50, 0,0,0,5, 7,8,4,50,
    7,10,4,25
240 data 7,6,4,25, 7,6,4,75, 7,8,4,50, 7,10,4,25
250 data 7,6,4,25, 7,6,4,75, 7,8,4,50, 7,10,4,25
260 data 7,8,4,25, 7,8,4,50, 7,10,4,25, 7,1,5,25
270 data 0,0,0,5, 7,1,5,85
280 data 7,1,5,25, 7,3,5,25, 0,0,0,5, 7,3,5,50,
    0,0,0,5
290 data 7,3,5,25, 0,0,0,5, 7,3,5,50, 0,0,0,5,
    7,3,5,25
300 data 7,1,5,25, 7,5,5,25, 7,3,5,50, 7,1,5,25,
    0,0,0,5
310 data 7,1,5,50, 7,10,4,25, 0,0,0,5, 7,10,4,25,
    7,1,5,25
```



```
320 data 0,0,0,5, 7,1,5,25, 0,0,0,5, 7,1,5,25,  
    0,0,0,5  
330 data 7,1,5,25, 0,0,0,5, 7,1,5,25, 7,10,4,50,  
    0,0,0,5  
340 data 7,10,4,25, 0,0,0,5, 7,10,4,85, 0,0,0,5,  
    7,10,4,25  
350 data 0,0,0,5, 7,10,4,25, 7,3,5,25, 0,0,0,5  
360 data 7,3,5,25, 0,0,0,5, 7,3,5,25, 0,0,0,5,  
    7,3,5,25, 0,0,0,5  
370 data 7,3,5,25, 7,12,4,75, 7,6,5,25, 0,0,0,5,  
    7,6,5,25  
380 data 7,5,5,25, 7,3,5,25, 7,3,5,25, 0,0,0,5,  
    7,3,5,25, 7,1,5,25  
390 data 7,3,5,25, 0,0,0,5, 7,3,5,25, 0,0,0,5,  
    7,3,5,25, 0,0,0,5  
400 data 7,3,5,25, 7,3,5,25, 0,0,0,5, 7,5,5,25,  
    7,6,5,25, 7,3,5,25  
410 data 0,0,0,5, 7,3,5,95,0,0,0,5  
420 data 10,10,4,25, 10,1,5,25, 10,3,5,50,  
    10,3,5,25, 10,1,5,25  
430 data 10,3,5,50, 10,4,5,25, 10,1,5,25,  
    10,3,5,25, 0,0,0,5  
440 data 10,3,5,25, 0,0,0,5, 10,3,5,25,  
    10,1,5,25, 10,3,5,50  
450 data 10,5,5,25, 10,6,5,25, 10,3,5,50,  
    0,0,0,5, 10,3,5,25  
460 data 10,1,5,25, 10,3,5,25, 0,0,0,5,  
    10,3,5,25, 10,5,5,25  
470 data 10,6,5,25, 10,8,5,25, 10,6,5,25, 0,0,0,5  
480 data 10,6,5,25, 10,5,5,25, 10,3,5,50, 0,0,0,5  
490 data 10,3,5,25, 10,1,5,25, 0,0,0,5,  
    10,1,5,50, 0,0,0,5  
500 data 10,1,5,25, 10,12,4,25, 10,1,5,50,  
    10,8,4,25  
510 data 10,12,4,25, 10,1,5,25, 0,0,0,5,  
    10,1,5,25, 0,0,0,5  
520 data 10,1,5,25, 10,12,4,25, 10,1,5,50,  
    10,3,5,25, 10,5,5,25  
530 data 10,3,5,75, 10,1,5,25, 10,10,4,120,  
    0,0,0,100  
540 data 10,10,4,25, 10,1,5,25, 10,3,5,50,  
    0,0,0,5
```

```
550 data 10,3,5,25, 10,1,5,25, 10,3,5,50,  
    10,10,4,25  
560 data 10,1,5,25, 10,3,5,25, 0,0,0,5,  
    10,3,5,25, 0,0,0,5  
570 data 10,3,5,25, 10,1,5,25, 10,3,5,25,  
    10,10,4,50  
580 data 10,11,4,25, 10,8,4,50, 0,0,0,5,  
    10,8,5,25, 10,6,4,25  
590 data 10,8,4,50, 10,3,4,25, 10,6,4,25,  
    10,8,4,25, 0,0,0,5  
600 data 10,8,4,25, 0,0,0,5, 10,8,4,25,  
    10,6,4,25, 10,8,4,50  
610 data 10,3,5,25, 0,0,0,5, 10,3,5,25,  
    10,1,5,50, 0,0,0,5  
620 data 10,1,5,25, 10,12,4,25, 10,1,5,50,  
    10,8,4,25  
630 data 10,12,4,25, 10,1,5,25, 0,0,0,5,  
    10,1,5,25, 0,0,0,5  
640 data 10,1,5,25, 10,12,4,25, 10,1,5,50,  
    10,3,5,25  
650 data 10,5,5,25, 10,3,5,50, 10,1,5,50,  
    10,10,4,50  
660 data 10,8,4,50, 10,10,4,25, 10,8,4,25,  
    10,6,4,50  
670 data 0,0,0,50, 10,3,5,50  
680 data 0,0,0,0, -1,-1,-1,-1
```

Now that you know what the chip can do, you won't want to work with just pure tones. To help you out ST BASIC has the command WAVE. The syntax for WAVE is:

WAVE Enable, Sound_Level_regulation,Envelope,Env.period,Length

Only the parameters *Sound_Level_regulation* and *Env.period* need remark. The second value asks if the PSG should control the volume for a particular channel through an envelope, or if the SOUND command should use an input value.

Notice the difference: here a single data value sets the values in register R8, R9, and R10. That means that some coding is needed here.

The three low-order bits are important. Bit 1 stands for Channel 1 (A), bit 2 for channel 2 (B), and bit 3 for channel 3 (C). The envelope generator will be activated if the corresponding bit is set (= 1). If all the bits are erased (hence a 0 is passed), all three voices will be controlled by a SOUND command by means of a WAVE command.

The procedure for setting the envelope hasn't changed—it's only been made easier. Now the 16 bit value can be passed as a number in the range between 0 and 65535. Distribution in LSB and MSB isn't important any longer.

As the examples on the following page show, we can get the desired effects quickly and without a lot of calculations by using just these two commands.

```
10  clearw 2: fullw 2
20  print "MINI ORGAN"
30  print:print "We will play with
    s,d,f,g,h,j,k."
40  print:print "The tone level will be varied
    with + and -."
50  print: print "END = CTRL-Z."
60  octave=4
70  x=inp(2)
75  if x = 26 then end
80  if x = 115 then n=1 : goto 170
90  if x = 100 then n=3 : goto 170
100 if x = 102 then n=5 : goto 170
110 if x = 103 then n=6 : goto 170
120 if x = 104 then n=8 : goto 170
130 if x = 106 then n=10: goto 170
140 if x = 107 then n=12: goto 170
150 if x = 43 then octave = octave +1:
    if octave >=8 then octave =8
160 if x = 45 then octave = octave -1:
    if octave <=1 then octave = 1
170 sound 1,12,n,octave,25
180 goto 70
190 for i=1 to 10:next:return
```



```
10 clearw 2: fullw 2 : z$=string$(60," ")
20 gotoxy 1,1: print " S O U N D E D I T O
   R":print
30 input "Octave          ";octave
40 input "Note            ";note : h=0
50 input "Duration        ";duration
60 input "Envelope      (y/n) ";e$:if e$="y"
   then h=-1
70 if h=0 then enable=0:print z$:print z$:
   print z$:goto 110
80 input "Envelope  (0-15) ";envelope
90 input "Period      ";hp
100 input "Enable  (1-7)   ";enable
110 input "Channel  (T/R)  ";channel
120 wave channel,enable,envelope,hp,duration
130 goto 20
```


6.10 The sound chip under LOGO

Programming the sound chip in LOGO is similar to BASIC. Here too, the high level language LOGO can produce everything from simple tones to complicated musical pieces.

However, working with LOGO isn't quite as easy as in BASIC, since it works with the registers directly. This means you have to set registers R0 to R13 of the PSG.

An example should make it clear how the LOGO syntax and programming works:

```
TO SEA
SOUND [0 0 1 0 2 0 3 0 4 5 6 1500 07 199 8 16 9 16
10 16 11 0 12 20 13 9]
END
```

Everything clear?

We can use 14 registers for sound synthesis and exchange the numbers 0 to 13 in the length of time it takes to produce a sound the way LOGO does it. In between this there is another number and this one will be written in the previously named register.

You can also use sound in LOGO programs by changing the register contents. It's similar to the method used when programming the sound chip in C. This means that sound effects from C listings can easily be transferred to LOGO programs. Turn back a few pages and try it once.

You'll find similar programs on the following page.

TO GUN

SOUND [0 0 1 0 2 0 3 0 4 0 5 0 6 15 7 7 8 16 9 16
10 16 11 0 12 16 13 0]

END

TO EXPLODE

SOUND [0 0 1 0 2 0 3 0 4 0 5 0 6 31 7 199 8 16 9 16
10 16 11 0 12 50 13 9]

END

TO SEA

SOUND [0 0 1 0 2 0 3 0 4 0 5 0 6 15 7 199 8 16 9 16
10 16 11 0 12 32 13 14]

END

TO PIANO

SOUND [0 200 1 0 2 201 3 0 4 100 5 0 6 0 7 248 8 16
9 16 10 0 11 0 12 20 13 9]

END

TO FOOTSTEPS

SOUND [0 0 1 0 2 0 3 0 4 0 5 0 6 31 7 199 8 16 9 16
10 16 11 0 12 6 13 12]

END

6.11 What is MIDI?

So far you've learned how to use the ST's sound chip to synthesize sound and play musical pieces.

But it's also possible to use the ST's built-in tone generator to produce a musical background. The operations can even be carried out independently of the processor, e.g. using envelope 1110 or 1100. Alternatively, the envelope generator can be used (however user-unfriendly it might be). This means more than one musical production can be stored under another production of every single envelope phase. This means a high-quality synthesized tone of nearly every musical instrument can be produced.

The output of the three-voiced sound chip should amuse any (semi-)professional musician. But the PSG would be a little boring for a musician/programmer. However, the ST's built-in MIDI interface would be of great interest to him. The MIDI interface allows you to hook up several instruments to the ST, like a stack of Roland synthesizers, and have them play together. This is only one of many MIDI possibilities.

Technically, the Musical Instrument Digital Interface is no different than a normal serial interface used for data transfer from computer to computer, computer to modem, etc. The only difference is that MIDI's transfer rate is very high.

You've no doubt used the RS-232 interface in the course of your programming. The RS-232 transfer rate is from 300 to 9600 Baud. One Baud means eight bits per second.

Since every character sends a start and stop bit to differentiate between single characters, about 30 to 980 characters per second can be transferred over the RS-232. For real-time applications, that's much too slow. Any instruments using the RS-232 could only produce a few notes a minute.

In contrast, the MIDI Specification 1.0 established in 1982 standardized a transfer rate of 31250 Baud.

Maybe you're asking why you can't use parallel interfaces. Then you can send eight bits to one device and then eight bits to another. That would certainly speed things up.

There are several problems with this solution. The first problem is that the cables used for this can't be much longer than 10 feet. Another problem is that you need a platform on which the instruments can be connected with broad (at least nine-strand), stiff cable. Working with this cable is quite a hassle when compared to working with three-strand wiring.

For these reasons, we use a serial interface—MIDI uses a 5-pin DIN port.

How is the transferred data processed? MIDI 1.0 is a standardization of data transfer. Here is MIDI's technical data:

Transfer rate:	31250 baud
Data bits:	8; in addition, 1 start and 1 stop bit
Transfer method:	asynchronous

Asynchronous means that only data that is actually needed is transferred. There are no zero bytes sent to synchronize the destination device. This means that the transfer parameters are set so that the only data relevant to the device is transferred.

Every controller must be able to participate in the transfer protocol. The master device is responsible for the direction of the transferred signal. The master device runs the whole MIDI network. synthesizers, rhythm devices, etc. are the slaves of the impulse it sends.

Imagine that we have two synthesizers, both connected to the MIDI interface and to the ST. In addition to the ports on both these devices, MIDI IN and MIDI OUT, there is another port labeled MIDI THROUGH.

To connect the two synthesizers, we hook up one end of a cable to one synthesizer in the MIDI OUT port, and then hook up the other end of the cable to the other synthesizer's MIDI IN port. This arrangement designates the first synthesizer as the master device, and the second synthesizer as the slave device.

Normally only three data items are transferred through MIDI:

- Note (or pitch)
- Attack
- Soundwave (or timbre)

These optional parameters can also be sent through MIDI:

- Synchronization
- Reset
- Start/stop signal for percussion, etc.
- Volume
- Manufacturer device data

We now use the ST to determine the device configuration. As soon as we have the IN connection of a synthesizer connected to the OUT connection of the ST, the computer becomes the master device. Then the synthesizer is completely at the command of the ST—assuming we have the proper software.

Before the synthesizer can start to output tones, it waits for three bytes:

- byte as command for note playing
- byte specifying tone (note and octave)
- byte indicating percussion

To turn the table around, the ST (with proper software) can also be used as a tape recorder. If you use it as the slave (via MIDI IN) and attach the other end of the cable to the MIDI OUT connection of a synthesizer, the tones will be faithfully reproduced in the computer in real time.

This opens fantastic opportunities for the creative musician. Not only can routine work be made easier, a melody can also be stored in the computer. All the notes of a musical piece can be stored on diskette and edited at a later time.

Since MIDI can play 16 computer-controlled musical instruments at the same time, the ST could be used to control an entire band. Musicians could enter all the relevant data, and the ST could then be hooked up as the master device. The ST would then play the instruments—perfectly synchronized and exactly as ordered.

Appendix C contains a bibliography of MIDI information resources.

Appendix

Appendix A:

Program: Compute the Periodic Values of the Tone Generators

```
10      'Note, Period & Frequency
20      '-----
30      clearw 2:fullw 2
40      dim scale$(12)
50      for note = 1 to 12:read scale$(note):next
60      for octave = -3 to 4
70          print:print "OCTAVE: ";octave
80          print"Note          Frequency          Period"
90          for note = 1 to 12
100         frequency = 440
            * (2 ^ (octave + ((note-10)/12)))
110         period = int(125000 / frequency)
120         print " ";scale$(note),frequency,period
130     next note
140 next octave
150 data "C","C#","D","D#","E","F",
        "F#","G","G#","A","A#","B"
```


Appendix B: Table of Periodic Values

OCTAVE: -3

NOTE	FREQUENCY	PERIOD
C	32.70321	3822
C#	34.64785	3607
D	36.70812	3405
D#	38.89089	3214
E	41.20347	3033
F	43.65355	2863
F#	46.24932	2702
G	48.99945	2551
G#	51.91311	2407
A	55	2272
A#	58.2705	2145
B	61.73544	2024

OCTAVE: -2

NOTE	FREQUENCY	PERIOD
C	65.40642	1911
C#	69.29567	1803
D	73.41621	1702
D#	77.78177	1607
E	82.40691	1516
F	87.30709	1431
F#	92.49863	1351
G	97.99889	1275
G#	103.8262	1203
A	110	1136
A#	116.541	1072
B	123.4709	1012

OCTAVE: -1

NOTE	FREQUENCY	PERIOD
C	130.8128	955
C#	138.5913	901
D	146.8324	851
D#	155.5635	803
E	164.8138	758
F	174.6141	715
F#	184.9972	675
G	195.9978	637
G#	207.6524	601
A	220	568
A#	233.0819	536
B	246.9417	506

OCTAVE: 0

NOTE	FREQUENCY	PERIOD
C	261.6256	477
C#	277.1827	450
D	293.6648	425
D#	311.127	401
E	329.6276	379
F	349.2282	357
F#	369.9944	337
G	391.9954	318
G#	415.3047	300
A	440	284
A#	466.1638	268
B	493.8833	253

OCTAVE: 1

NOTE	FREQUENCY	PERIOD
C	523.2511	238
C#	554.3653	225
D	587.3295	212
D#	622.254	200
E	659.2551	189
F	698.4565	178
F#	739.9888	168
G	783.9908	159
G#	830.6093	150
A	880	142
A#	932.3273	134
B	987.7664	126

OCTAVE: 2

NOTE	FREQUENCY	PERIOD
C	1046.502	119
C#	1108.73	112
D	1174.659	106
D#	1244.508	100
E	1318.51	94
F	1396.913	89
F#	1479.977	84
G	1567.981	79
G#	1661.218	75
A	1760	71
A#	1864.654	67
B	1975.533	63

OCTAVE: 3

NOTE	FREQUENCY	PERIOD
C	2093.004	59
C#	2217.46	56
D	2349.318	53
D#	2489.015	50
E	2637.02	47
F	2793.825	44
F#	2959.954	42
G	3135.962	39
G#	3322.436	37
A	3520	35
A#	3729.309	33
B	3951.066	31

OCTAVE: 4

NOTE	FREQUENCY	PERIOD
C	4186.008	29
C#	4434.92	28
D	4698.634	26
D#	4978.03	25
E	5274.038	23
F	5587.649	22
F#	5919.909	21
G	6271.924	19
G#	6644.871	18
A	7040	17
A#	7458.616	16
B	7902.127	15

Appendix C: MIDI Resources

The following is a list of organizations with MIDI information and software products:

Dr. T's Music Software 66 Louise Road Chestnut Hill, MA 02167	(617) 244-6954
---	----------------

Hippopotamus Software 985 University Ave., Suite #12 Los Gatos, CA 95030	(408) 395-3190
--	----------------

Hybrid Arts 11920 W. Olympic Blvd. Los Angeles, CA 90064	(213) 826-3777
--	----------------

Micro-W Distributing 1342B Route 23 Butler, NJ 07405	(201) 838-5606
--	----------------

Mimetics Corporation P.O. Box #60238 Sta. A Calabasas, CA 91302	(408) 741-0117
---	----------------

Club MIDI P.O. Box 93895 Hollywood, CA 90093	(213) 876-6725
--	----------------

MusicData 8444 Wilshire Blvd. Beverly Hills, CA 90211	(213) 655-3580
---	----------------

Opcode Systems 707 Urban Lane Palo Alto, CA 94301	(415) 321-8977
---	----------------

Index

ADSR	216
Amplitude Control	219
AND	181
animation	184
Apple II	213
ASCII	233
Attack	215
axis	28
bar graphs	75
BASIC	58,162,178,184,196
Bload	180
Boolean algebra	181
BOX	81
Bsave	180
byte	8
CAD	4,145,158,162
caddemo.bas	158
change.bas	230
character mode	125
circle	67
circle1.bas	69
circle2.bas	71
circle3.bas	72
color	7,12,178,183
complex numbers	191,194
contrl	79
Control	52
Colour Genie computer	213
CP/M	48
D/A transformer	219
Datamax	97
Decay	215
DIN port	251
Doppler effect	235

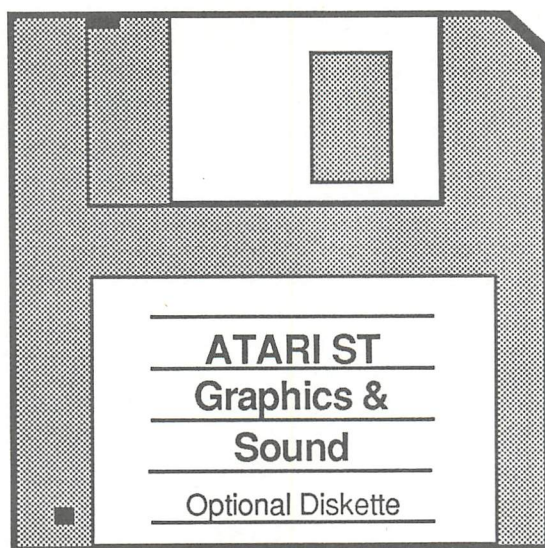
Ellipses	72
envelope	215
Envelope generator	219, 224
evnt_mouse	233
EP	236
FILL	84
fillshow.log	182
films, three-dimensional	170
Flight Simulator	3,193
Fourier, Joseph de	214
font.bas	59
fractal	191-196
frequency	229
function plotter	120, 147
GEM	48,49,161,182, 226
GEMDOS	48
General Instrument	213
giaccess()	226, 230
glasses, three-dimensional viewing	145,170,171
graphic1.bas	29
graphic2.bas	33
gray levels	179
handles	52,55
hidden lines	148
Hilbert curve	34
Hi-res to Mid-res conversion	178
input	53
intin	55,79
intout	53,177
Julia, Gaston	191
Kaleidoscope	184
linef	63
line graphs	74
LOGO	16,35,38,178,181, 248

loop	34
Lucasfilm	6
luminance	8
Mandelbrot, Benoit	191-193
matrices	104
matrixdemo	107
MIDI	250
Mid-res to Hi-res conversion	178
Mixer	218, 223
moire	30
monochrome screen	171, 177, 179, 193
mouse	162
Moog, Robert	217
mouse.input	162
MSX computer	213
music	241-247
NDC	49, 50, 177
NOT	181
object projection, three-dimensional	146
octal	220
octave	241
opcode	52
open-workstation	53, 171
OR	181
oscillations	214
output	78
Pappert, Seymour	20
pattern.bas	184
pattern.mod	187
parabola	128
parabola, three-dimensional	147
parallel shifting	100
perspective drawings	146
pcircle	77
pie charts	75
Pixels	7
plotmax	97
plotter	147
plot3d1.bas	147

plot3d with hidden lines	150
pointer	161
polar coordinates	69-71
Pointe Reyes	6
PONG game	213
Programmable Sound Generator	217,220, 229
printf()	233
printvar (x)	233
PSG	217, 220, 229
ptsin	53
ptsout	161
pxyarray	63
Pythagoras	66
 Ram	 7,9,11
Raster	8
RC	49,50
rectangular vibrations	214
Release	215
 sawtooth vibrations	 214
save_work	171
saving screens	180
Schneider CPC computer	213
setpal	179
set_writing_mode	181
shapes	91
simulations technique	193
sound	241, 246
Sound Chip Register	222
Sound generator	218, 223
step	66
stereo graphics, real objects	174
stereoscopy	167
stereo1.bas	168
stereo2.bas	172
string	55
sub2.c	236
sub3.c	237
sub4.c	238
sub5.c	239
sub6.c	240
Sustain	215

symbolic constants	230
Symmetry	34
3d-demo	155
tomograph	6
tone	214
Tone generator	218, 223, 241
tone period	229
TOS	48
transline	178
transpoint	123, 178
triangle	26
Tron	4
TRS-80	213
turtle	16, 21
v_gtext	78
v_gtext()	233
VDI	48, 49
vdisys	58
Vector	9
vertline.bas	31
vision, three-dimensional	167
vmax	87
volume control	224
wave	245, 246
work_station	234
Works, The	4
xmax	98
xmin	98
xnew	103
XOR	181
Yamaha YM-2149 chip	213
ymax	98
ymin	98
ynew	103
zero point	152

Optional Diskette



For your convenience, the program listings contained in this book are available on an SF354 formatted floppy disk. You should order the diskette if you want to use the programs, but don't want to type them in from the listings in the book.

All programs on the diskette have been fully tested. You can change the programs for your particular needs. The diskette is available for \$14.95 plus \$2.00 (\$5.00 foreign) for postage and handling.

When ordering, please give your name and shipping address. Enclose a check, money order or credit card information. Mail your order to:

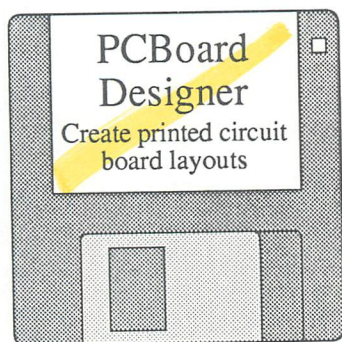
Abacus Software
P.O. Box 7219
Grand Rapids, MI 49510

Or for fast service, call 1- 616 / 241-5510.



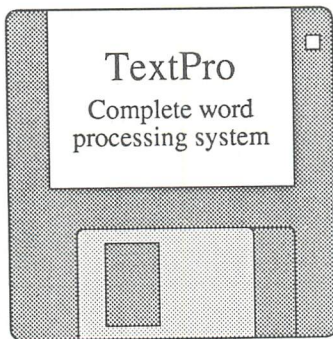
The WAIT is over!

New ST software from a name you can count on...



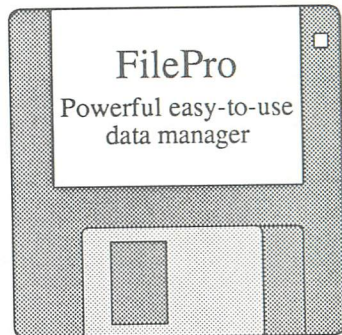
PCBoard Designer

Interactive, computer-aided design package that automates layout of printed circuit boards. **Auto-routing** with 45° or 90° traces; two-sided boards; pin-to-pin, pin-to-BUS or BUS-to-BUS. Rubberbanding of components during placement. Prints board layout, pinout, component list, net list. Output to Epson printer at 2:1. Pays for itself after first designed board. **\$395.00**



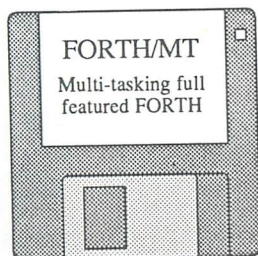
ST TextPro

Wordprocessor with professional features and easy-to-use! Full-screen editing with mouse or keyboard shortcuts. High speed input, scrolling and editing; sideways printing; multi-column output; flexible printer installation; automatic index and table of contents; up to 180 chars/line; 30 definable function keys; metafile output; much more. **\$49.95**



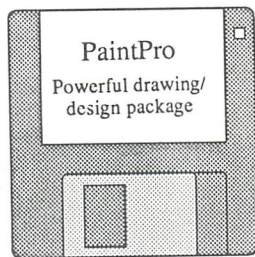
ST FilePro

A simple-to-use and versatile database manager. Features help screens; lightning-fast operation; tailorable display using multiple fonts; user-definable edit masks; capacity up to 64,000 records. Supports multiple files. RAM-disk support for 1040ST. Complete search, sort and file subsetting. Interfaces to TextPro. Easy printer control. **\$49.95**



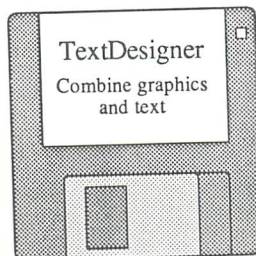
ST Forth/MT

Powerful, multi-tasking Forth for the ST. A complete, 32-bit implementation based on Forth-83 standard. Development aids: full screen editor, monitor, macro assembler. 1500+ word library. TOS/LINEA commands. Floating point and complex arithmetic. Available July '86. **\$49.95**



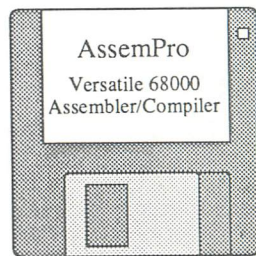
ST PaintPro

A GEM™ among ST drawing programs. Very friendly, but very powerful. A *must* for everyone's artistic or graphics needs. Use up to **three** windows. Free-form sketching; lines, circles, ellipses, boxes, text, fill, copy, move, zoom, spray, paint, erase, undo, help. **\$49.95**



ST Text Designer

An ideal package for page layout on the ST. Accepts prepared text files from TextPro or other ASCII wordprocessors. Performs block operations—copy, move, columns. Merges bit-pattern graphics. Tools to add borders & separator lines, more. Available August '86. **\$49.95**



ST AssemPro

Professional developer's package includes editor, two-pass inter-active assembler with error locator, online help including instruction address mode and GEM parameter information, monitor-debugger, disassembler and 68020 simulator, more. Available July '86. **\$59.95**

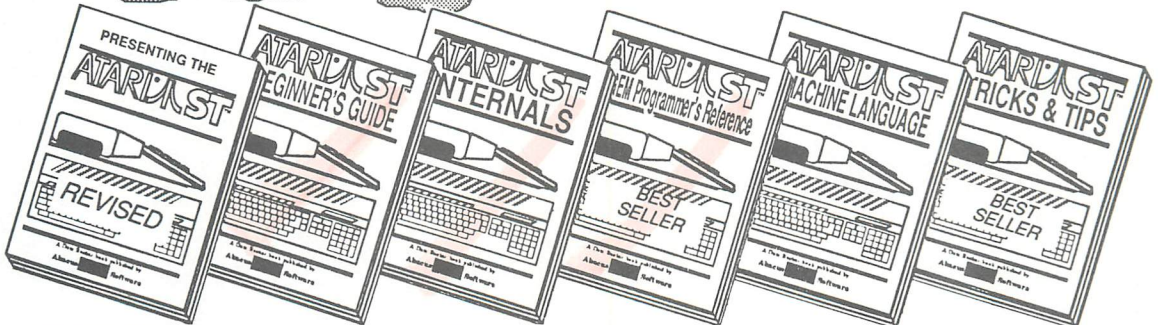
ST and 1040ST are trademarks of Atari Corp.
GEM is a trademark of Digital Research Inc.

Abacus Software

P.O. Box 7219 Dept. N8 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Call **now** for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add \$4.00 per order for postage and handling. Foreign add \$10.00 per item. **Other software and books coming soon.** Call or write for your **free** catalog. Dealer inquiries welcome—over 1400 dealers nationwide.

HOT OFF THE PRESS



PRESENTING THE ST
Gives you an in-depth look at this sensational new computer. Discusses the architecture of the ST, working with GEM, the mouse, operating system, all the various interfaces, the 68000 chip and its instructions, LOGO. 200pp \$16.95

ST Beginner's Guide
Written for the firsthand ST user. Get a basic understanding of your ST. Explore LOGO and BASIC from the ground up. Simple explanations of the hardware and internal workings of the ST. Illustrations, diagrams. Glossary. Index. 200pp \$14.95

ST INTERNALS
Essential guide to the inside information of the ST. Detailed descriptions of sound and graphics chips, internal hardware, I/O ports, using GEM. Commented BIOS listing. An indispensable reference for your ST library. 450pp \$19.95

GEM Programmer's Ref.
For serious programmers needing detailed information on GEM. Presented in an easy-to-understand format. All examples are in C and assembly language. Covers VDI and AES functions. No serious programmer should be without. 410pp \$19.95

MACHINE LANGUAGE
Program in the fastest language for your ATARI ST. Learn 68000 assembly language, its numbering system, use of registers, structure & important details of instruction set, and use of internal system routines. Geared for the ST. 280pp \$19.95

ST TRICKS & TIPS
Fantastic collection of programs and info for the ST. Complete programs include: super-fast RAM disk; time-saving printer spooler; color print hardcopy; color plotter output hardcopy; creating accessories. Money saving tricks and tips. 260pp \$19.95



ST GRAPHICS & SOUND
Detailed guide to graphics and sound on the ST. 2D & 3D function plotters, Moté patterns, graphic memory and various resolutions, fractals, recursion, waveform generation. Examples written in C, LOGO, BASIC and Module2. 250pp \$19.95

ST LOGO GUIDE
Take control of your ST by learning ST LOGO—the easy to use, powerful language. Topics include: file handling, recursion-Hilbert & Sierpinski curves, 2D and 3D function plots, data structure, error handling. Helpful guide for ST LOGO users. \$19.95

ST PEEKS & POKES
Enhance your programs with the examples found within this book. Explores using different languages BASIC, C, LOGO and machine language, using various interfaces, memory usage, reading and saving from and to disk, more. 280pp \$16.95

BASIC Training Guide
Thorough guide for learning ST BASIC programming. Detailed programming fundamentals, commands descriptions, ST graphics & sound, using GEM in BASIC, file management, disk operation. Tutorial problems give hands on experience. 300pp \$16.95

BASIC TO C
Move up from BASIC to C. If you're already a BASIC programmer, you can learn C all that much faster. Parallel examples demonstrate the programming techniques and constructs in both languages. Variables, pointers, arrays, data structure. 250pp \$19.95

3D GRAPHICS
FANTASTIC! Rotate, zoom, and shade 3D objects. All programs written in machine language for high speed. Learn the mathematics behind 3D graphics. Hidden line removal, shading. With 3D pattern maker and animator. \$24.95

The ATARI logo and ATARI ST are trademarks of Atari Corp.

Abacus Software

P.O. Box 7219 Dept. A8 Grand Rapids, MI 49510 - Telex 709-101 - Phone (616) 241-5510

Optional diskettes are available for all book titles at \$14.95

Call now for the name of your nearest dealer. Or order directly from ABACUS with your MasterCard, VISA, or Amex card. Add \$4.00 per order for postage and handling. Foreign add \$10.00 per book. Other software and books coming soon. Call or write for your free catalog. Dealer inquiries welcome—over 1400 dealers nationwide.

ATARI® ST™

GRAPHICS & SOUND

Learn how to create graphics and use the built-in sound facilities of your Atari ST. Examples are written in BASIC, C, LOGO, and Modula 2. Here's just some of the topics you'll learn about:

- Mirror and rotation
- Graphics under GEM
- Coordinate transformations
- Raster and vector graphics
- Principles of music synthesis
- The sound chip
- Plotting 2D & 3D functions
- Moiré patterns
- Bar and pie charts
- Fractals
- Waveform generation
- The ST as a synthesizer

About the author:

Jörg Walkowiak is a member of the experienced Data Becker Product Development team, based in Duesseldorf, W. Germany. He is also a best selling book author in the U. S. and Europe.

ISBN 0-916439-49-6

The ATARI logo and ATARI ST are trademarks of Atari Corp.

A Data Becker book published by

You Can Count On  Software

P.O. Box 7219 Grand Rapids, MI 49510 • Telex 709-101 • Phone 616/241-5510