graphic art

for the BBC computer

turtle graphics and art

boris allan

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12-13 Little Newport Street,
London WC2R 3LD

ISBN 0 946408 08 4

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

DIGITALLY REMASTERED ON ACORN RISC OS COMPUTERS, NOVEMBER 2011

To Ruth, Mark, and Maggie

In the dime stores and bus stations. . .

CONTENTS

Contents in detail

INTRODUCTION

Artist’s Tools

Functions, local and global parameters, procedures with multiple parameters. The resolution of text and characters in varying modes; pixels and graphics resolution; text and graphics windows; the use of ‘split screen’ graphics; the first use of VDU commands; setting the graphics origin.

CHAPTER 1

Turtle Graphics 1

The drawing of a shape with normal BASIC commands, and problems therewith; the idea of an intrinsic geometry, dependent upon the internal structure of a shape; turtle geometry as an intrinsic geometry. The basic turtle commands MOVE and TURN; Version 1.1 of my Turtle Graphics routines.

CHAPTER 2

Turtle Geometry

Angles and ratios of sides are the key intrinsic aspects of geometry; the trigonometrical ratios, sine, cosine and tangent; measuring angles in degrees and radians. How the turtle graphics routines use trigonometrical ratios. Drawing regular polygons; an examination of different plotting styles.

CHAPTER 3

Turtle Graphics II

Modes 1 and 5, the four colour modes; the design of the colours to be used by multi-coloured turtle graphics. Implementing multi-coloured turtles;

example effects, and the game NORT.

CHAPTER 4

Driving Graphics

The VDU command, and what it is; the reasoning behind ASCII codes; control codes on computers. Routines to assist in the design of shapes; saving the character routines as ASCII files; saving and loading user-designed shapes

CHAPTER 5

Drawing Charts and Graphs

The normal distribution, the Central Limit Theorem; a simulation of random sampling; the drawing of histograms and frequency polygons, using graphics windows; the effects of sample size, and number of categories.

CHAPTER 6

Turtle Graphics III

The square as a special rectangle; modifying turtle graphics to allow simple sketching of designs; a rectangle is a stretched square, and an ellipse is a stretched circle.

CHAPTER 7

Generative Graphics

Programming style, creative programming does not follow rules; ideas for extensions to the work in this book; suggested reading; Cities in Flight.

INTRODUCTION
Artist's Tools

> The science of painting begins with the
> point, then comes the fine, the plane
> comes third, and the fourth the body in
> its vesture of planes. This is as far as
> the representation of objects goes.
>
> Leonardo da Vinci, *Trattato delta pittura*

This is a book about the BBC computer, and how to use graphics which fit that machine.

This is not a book about graphics, which then uses the BBC computer merely as an example of how graphics can be implemented. What is it, then, that makes the BBC computer so different when we come to discuss the artistic uses of graphics? Not only is it the way in which its graphics are implemented; but also, more importantly, it is the way in which the programming language (BBC BASIC) differs from all other versions of BASIC on other microcomputers.

The single most important feature of BBC BASIC, and the feature which has to influence any implementation of any graphics system, is the ability to use procedures and functions with multiple parameters. This ability produces a highly effective, yet very compact system of great power and flexibility.

In this chapter, therefore, I will first discuss just why BASIC on the BBC computer is so different, then why and how this affects the way in which we produce a graphics system.

Functions

Suppose we wish to calculate the square root of the number 3, and we wish to store the value of the square root in a variable called SQUAREROOT. We can enter a fine in a program such as

```
1000 SQUAREROOT = SQR(3)
```

and the value of the square root is then stored in the desired variable.

If we wished to find the square roots of all the values from 1 to 10, we would use a loop from 1 to 10 and have a line

```
1000 PRINT SQR(I)
```

where I was the loop counter (ie FOR I = 1 TO 10). At each time the SQR function was encountered, the variable I would be replaced by the value stored there, and the square root calculated. The value stored in I would not be affected. In jargon, I was called by value and not by name. If a variable is called by name then it is possible to alter the value stored at that name: to call by value means that the function only knows about the value stored, and not the name.

The function SQR has only one parameter (ie there can only be one input value), so suppose that we need to define a function which will give the square root of the product of two values. To work out the square root of 2 multiplied by 3 we can simply enter

```
1000 PRINT SQR(2*3)
```

But suppose that we wanted to appear rather more sophisticated (note: I say only appear). We define a function of our own to produce the square root of the product — the function we call FNsqr(X,Y).

Function FNsqr has two parameters (X and Y) but, before we discuss the status of the symbols X and Y, the function definition:

```
1000 DEF FNsqr (X,Y)
1010 X = X*Y
1020 = SQR(X)
1030 REM end of FNsqr
```

in which we see that (1000) the function is defined as having two parameters, X and Y. In fine 1010 the two values are multiplied together, and the result stored in the variable the function calls X (this variable has no relationship to any other X outside the function definition). In fine 1020 there is an assignment statement (ie there is an =) with no variable to the left of the assignment: the system assumes, therefore, that the assignment is to the function.

If the function is then used for the variables I and J by

```
100 PRINT FNsqr(I,J), I, J
```

then (when this line is activated) the value stored at I is substituted into the temporary variable X in the function definition, and the value at J is stored at Y. The variables X and Y take values, and have an existence, which is "local" to that function. The values printed out for I (in particular) and J are not affected by the action of the function (ie I does not become equal to I* J).

If lines in the function definition are altered, eg:

```
1010 Z = X*Y
1020 = SQR(Z)
```

to then have a line

```
100 Z= 0 : PRINT FNsqr(I,J), Z
```

shows that the value Z is modified in the function. The value stored at Z is modified because it is not one of the parameters, and has not been defined as being local to the function: the scope of operation of Z is global, compared with the function. Finally add one line,

```
1005 LOCAL Z
```

and then activate line 100. In this case the global value of Z is unaltered, as the Z in the function definition has been explicitly defined as being local to that function.

Procedures

A key difference between functions and procedures is that a function always produces a value (or, "delivers a result"). In many BASICs another key difference is that functions take parameters.

A few BASICs now allow multiple parameters for functions (and some even allow function definitions to extend over more than one statement) but the use of parameters in routines is rare. On mainframes and minicomputers (eg Hewlett-Packard) there are various very sophisticated BASIC systems, but BBC BASIC stands out from most other microcomputer BASICs in terms of its sophistication.

Nearly all of the comments we would wish to make about parameters for procedures have been already made about functions, but the distinctions between the various types of variable in procedures are far more important than they are for functions. After all, functions do not tend to be used as much as procedures, functions deliver a result, whereas procedures do something.

In BBC BASIC (and from now on this will be shortened to BB) the command to plot at a certain point is

```
PLOT 69, X, Y
```

where the coordinates are X and Y (see page 320 of the User Guide, or alternatively (UG page 386)

```
VDU 25, 69, X; Y;
```

VDU driver commands will be discussed in later chapters. To define a procedure which will plot a diagonal fine from coordinates s,s to f,f without using parameters is obviously possible, but to use parameters makes it so much simpler. Watch:

```
1000 DEF PROCdiag(s,f,inc)
1010 LOCAL i
1020 FOR i= s TO f STEP inc
1030 PLOT 69, i, i
1040 NEXT i
```

```
1050 ENDPROC : REM diag
```

and to draw an instant diagonal line from 0,0 to 1000,1000 we enter (in immediate mode, so no fine numbers)

```
PROCdiag (0, 1000, 10)
```

where the points are plotted in gaps of 10 units.

PROCdiag has three parameters s, f, and inc, the scope of which is merely that of the procedure; there is another variable i (used as the loop counter) whose scope is also defined as being local to the procedure; and there are no global variables. If there is a variable i in the main program (or s, f, or inc) the values of these variables are unaffected — the procedure does not even recognise their existence.

If there were variables s and f in the main program (with the correct interpretation), it might be possible to define a different procedure, which did more than just draw a diagonal line of blobs. It might make the finish of the fine (f) the start of a new line (s):

```
1000 DEF PROCnewdiag(inc)
1010 LOCAL i
1020 FOR i = s TO f
1030 PLOT 69, i, i
1040 NEXT i
1050 s = f
1060 ENDPROC : REM newdiag
```

For this procedure both s and f are global to the procedure: they have to be so, because we need to modify their values. With PROCdiag we are able to use variables with names other than s or f in the main program. As long as the names are in the correct position in the list of parameters, the name does not matter.

Recursion

Sometimes (not very often) when writing a program, or a routine, there may be a case where what you want to do is fairly simple, but to achieve the result by normal methods seems to be overly tedious in terms of the mental commitment necessary to solve the problem (a classic example is the Kasner snowflake, discussed in the next chapter).

It is not that the solution is impossible to find by normal methods, it is just that the solution is difficult to code simply. One way sometimes used to solve such problems is called ‘recursion’; and, though there is a great mystique surrounding the term, it is very simple to use recursion — though remarkably wasteful.

Examine this function and routine:

```
1000 DEF FNwalk = RND(55) - 28
2000 DEF PROCrndwalk(x, y ,z)
```

```
2010 x = x + FNwalk : y = y + FNwalk
2020 PLOT z,x,y : PROCrndwalk(x,y,z)
2030 ENDPROC : REM rndwalk
```

where the function is without parameters because it does not depend upon any input value to produce its result.

The procedure has three parameters: two (x and y) are co-ordinates, and the third sets the style of plotting (U G page 319). The input coordinate values are modified by FNwalk up to a limit of plus or minus 27 units (independently); the new coordinates are then used to plot to new coordinates (with the style of plotting set by z); and then the new values are used as parameters for yet another call to the same procedure (ie PROCrnd walk)

The call to PROCrndwa1k within PROCrndwalk is what is termed a recursive call. If you use PROCrndwaIk in immediate mode, eg

```
MODE 4 : PROCrndwalk (500,500,5)
```

the screen will show randomly drawn lines, usually called a ‘random walk’ (similar to Brownian motion). After a short while the plotting will stop with an error message at line 1000: by repeatedly calling itself (and the BASIC system having to remember where it has been) the program runs out of room.

Repeating the above fine (still in Mode 4) for different values of z (eg 5, 6, 21, 22, 69, 70, 85, or 86) is a very good way of investigating the effects of the flexible plotting command. My favourite is 86, plot triangles in the logical inverse colour. Mode 5 is also worth trying at this stage, possibly using GCOL to change the graphics colour (UG pages 163, 262).

Using Modes 0,1, and 2, produces a surprise: the plotting is over much more quickly than for Modes 4 and 5. A glance at the memory map (UG page 500) shows that, as the graphics memory increases, so the memory available for BB is less. As the memory map shows, the BASIC stack reaches from the graphics memory boundary (ie HIMEM) until it reaches the BB program. When it reaches the program, we find there is no room at fine 1000.

Recursion can be fun, but — unless you are very careful — your program will crash, particularly with a large program in higher resolution modes. However, as I noted above, PROCrndwalk is a useful way of investigating plotting styles. Another useful exercise is to write PROCrndwalk ‘iteratively’ , ie by use of a FOR. . .NEXT loop (or possibly a REPEAT . . . UNTIL loop). Also worth trying is to write the (iterative) PROCdiag in a recursive manner.

It is interesting to note that in the definition of PROC (UG page 329) there is an example of recursion — without any explanation — so the designers of BB must have thought that recursion was important. The definition of recursion is, of course,

RECURSION : See Recursion

Screen resolution

To draw pictures on the screen we need to know something about the screen. Start by trying out this procedure

```
1000 DEF PROC_ORIGIN TO_(X,Y)
1010 MOVE 0,0 : DRAW X, Y
1020 ENDPROC : REM ORIGIN_TO_
```

and then use the procedure by entering (in direct mode, ie without line numbers)

```
PROC_ORIGIN_ TO_(800,800)
```

which will draw a fine from the bottom left corner of the screen to somewhere towards the top right corner. Holding the RETURN key down succeeds in moving the fine up the screen: there is no distinction between the drawing of lines on the screen, and the entering in of the characters. To repeat the call of the procedure is to draw a second fine, parallel to the first.

If nothing has happened, you are probably in mode 3, 6, or 7, none of which allow high resolution graphics: in fact, if in mode 7, the space symbol '_' will appear as a hyphen '-'. At this point it is worth turning to UG page 59, which gives the number of characters per line, and numbers of lines, for the various modes. Keeping note of the numbers there: Figure 0.1 Character Resolution

Figure 0.1 Character Resolution

MODE	CHARACTERS	LINES	TOTAL CHARS
0	0 to 79 (80)	0 to 31 (32)	= 2560
1	0 to 39 (40)	0 to 31 (32)	= 1280
2	0 to 19 (20)	0 to 31 (32)	= 640
3*	0 to 79 (80)	0 to 24 (25)	= 2000
4	0 to 39 (40)	0 to 31 (32)	= 1280
5	0 to 19 (20)	0 to 31 (32)	= 640
6*	0 to 39 (40)	0 to 24 (25)	= 1000
7*	0 to 39 (40)	0 to 24 (25)	= 1000

Note: The * indicates that this is not a graphics mode.

Figure 0.2 Graphics Resolution

MODE	RESOLUTION	COLOURS	
0	640 x 256	2	= 20K
1	320 x 256	4	= 20K
2	160 x 256	16	= 20K
4	320 x 256	2	= 10K
5	160 x 256	4	= 10K

The total characters column shows that in mode 0 there can be 2560 characters on the screen at one time. As each character (U G page 170) is 8 elements wide by 8 elements high, then an 80 by 32 characters mode is the same as 640 by 256 elements.

If you now refer to the top of page 161 in the UG, you find that in the graphics modes the screen is divided up into imaginary rectangles: mode 0, we are told, has 640 x 256 squares. The number of squares correspond to what I termed elements, or what in other places are called 'pixels'. The 'higher' the resolution of the graphics (or the greater the number of characters on the screen) the larger the memory needed. Different modes also have different numbers of colours: the greater the number of colours which can be used on the screen at the same time, the greater the memory.

The memory requirements are those given at the bottom of page 160 of the UG, and it is worth noting how the calculation is made. Each group of 8 pixel/elements occupies one byte (one memory location); each bit within the byte can be set or not (two colours); if another byte is associated with that byte, there are now two bits per pixel (four colours); and to have 16 colours requires 4 bits per pixel. Therefore, to calculate the requirement for mode 2:

(160/8) x 256 x 4 20480 or 20K (= 20480/1024)

which — if not immediately obvious — should be studied carefully.

The theoretical dimensions of the plotting screen are (in the same order as we gave the characters and fines) 0 to 1279 across and 0 to 1023 upwards (or downwards). For each mode, therefore, there is a minimum resolution: for mode 5 (with 160 pixels across) each pixel is 1280/160 units wide, and thus the maximum discrimination in the X direction is 8 units. The maximum discrimination in the Y direction is thus 1024/256 = 4 units: each pixel is of size 8 x 4. There are 32 fines of text, so each fine will be 1024/32 = 32 units wide.

Figure 0.3 Pixel Resolution

MODE	PIXEL SIZE	LINE WIDTH
0	2 × 4	32
1	4 × 4	32
2	8 × 4	32
4	4 × 4	32
5	8 × 4	32

Splitting the screen

As there are two ways in which the screen may be used, there are two pointers to where we are using the screen. There is a text cursor, which points to where the next character is to be placed (usually flashing); and there is a graphics cursor which gives the start of the next graphics plot. Normally the two cursors are distinct, but it is possible to use a command which allows text to be entered at the position of the graphics cursor. The VDU command

```
VDU 5
```

will mean that only one cursor is active (ie the graphics cursor), and text can be entered at any part of the screen — without the screen scrolling up a line when on the bottom line.

As the normal discrimination for text in mode 5 is 1280/20= 64 units wide, characters can only be placed in increments of 64 units, whereas characters can be placed in increments of 8 units (see Figure 0.3) by use of the graphics cursor. On page 173 of the UG, an example is given of a rocket rising more smoothly, due to the increased discrimination obtained by using the graphics cursor. To separate the cursors we use VDU 4.

Though the association of text with the graphics cursor is useful, sometimes it is even more useful to make sure that text and graphics never occupy the same place on the screen. What is frequently needed is a ‘text space’ and a distinct ‘graphics space’. Many computers (eg the Apple II) provide this distinction automatically. We need to use text windows and graphics windows (see pages 56-61, and 385-388 of the UG).

The question is: where should the two spaces be situated? Following the example of the Apple, and other computers, I propose that the best place is with the text at the bottom of the screen (often four fines of text), and graphics to fill the rest of the screen. The reason I like this arrangement is that this allows the user to enter, interactively, drawing commands and it allows the user to study the command (it does not disappear) if something goes wrong — as it often does.

The origin for text is at the top left hand corner and (using mode 4 as an example) extends to 39 across (x axis) and 31 down (y axis). To place a window in the bottom four fines, we need to occupy lines 28, 29, 30, and

31; and to use the full width we need to occupy character positions 0 to 39.

Note that for all modes which use graphics (Figure 0.1) there are 32 fines, so that the only difference for other modes to mode 4 is in the numbers of characters across the screen.

To set the text window we use a VDU command

```
VDU 28, leftchar, bottomline, rightchar, topline
```

where the labels are as they say. To set a text window in the manner we have discussed, we enter

```
VDU 28, 0, 31, 39, 28
```

to give a window which extends from character 0 to 39, and fine 28 to 30. One ofthe following VDU commands is for mode 5, and one is for mode 0: work out which is which -

```
VDU 28, 0, 31, 79, 28
VDU 28, 0, 31, 19, 28
```

No answers supplied.

To set up the graphics window is similar, but different. The command is

```
VDU 24, leftcoord; bottomcoord; rightcoord; topcoord;
```

and, whereas in the text command we differentiated between fines and characters, we only have coordinates in the graphics command.

The most important distinction is that between the use of the comma ',' in the VDU 28 command, and the use of the semi-colon ';' in the VDU 24 command. The dimensions of the text window (in any direction) are never greater than 255 (check Figure 0.1 if you are not sure); whereas the dimensions of the coordinates often extend beyond 255. BB, therefore, makes a distinction between numbers of 255 or less (which can be treated as one byte — 8 bits), and numbers which might be up to 65535 (two bytes of 16 bits). The difference is explained on page 386 of the UG, by reference to the command

```
VDU 24, 150; 300; 100; 700;
```

and then

```
VDU 25, 4, 100; 500;
```

which is actually equivalent to the command PLOT 4, 100, 500. The VDU 25 command given is (the UG claims) the same as

```
VDU 25, 4, 100, 0, 244, 1
```

because $100 = 100 \times 1 + 0 \times 255$ and $500 = 244 \times 1 + 1 \times 255$. The comma sends the preceding number to the system as if it were one byte (and if the number is greater than 255 it sends the value MOD 256). A semi-colon informs the system that the preceding number has to be sent as two bytes

— there is no need for the last comma, but BB always has to have the last semicolon.

The above VDU 25 could be written more explicitly as

```
VDU 25, 4, X MOD 256, X DIV 256, Y MOD 256, Y DIV 256
```

and, later — Chapter 4 — I will analyse the use of VDU commands in detail; for now, however, all we really need is the important difference ’twixt comma and semi-colon.

Each fine of characters corresponds to 32 units in graphics, so the graphics screen (if it is not to overlap with the text screen) will have to start up 4 x 32 units (ie 128) and then extend from side to side and to the top

```
VDU 24, 0; 128; 1279; 1023;
```

will do very nicely. To play with graphics constructively, therefore, we need to start with the two screens and then clear them both:

```
VDU 28, 0, 31, 39, 28 : CLS
VDU 24, 0; 128; 1279; 1023; : CLG
```

We are away, apart from one little extra chore.

If we now use PROC_ORIG IN_TO we would find that part of the line was missing: the origin is outside the graphics window . We change the graphics origin to a new point by use of VDU 29 (U G page 388). To set the origin to 640,566 (ie the middle of the graphics window) we

```
VDU 29, 640; 566;
```

and now we really are away.

CHAPTER 1
Turtle Graphics

There is
one art,
no more,
no less:
to do
all things
with art-
less ness

Piet Hein, *Ars brevis*

In *Mindstorms*, Seymour Papert (1980 : 219) gives a short BASIC program to draw a house. Converted to BB, this little program is

```
1000 MOVE 0,0
1010 DRAW 100,0
1020 DRAW 75,150
1030 DRAW 0,100
1040 DRAW 0,0
1050 END
```

and Papert notes that this is not suitable as a general method for drawing a house, for it also requires quite a good deal of work to prepare. "This demand would be less serious if the program, once written, could become a powerful tool for other projects . . . the BASIC program allows one particular house to be drawn in one position. In order to make a BASIC program that will draw houses in many positions, it is necessary to use algebraic variables. . ."

Papert proves his own point (perhaps deliberately) by giving a routine for drawing a lopsided house — try the program to see.

How is the house constructed? Essentially the house is a square (or rectangle) with a triangle on top: forget about inessential aspects such as doors and windows for the moment. To draw a house, therefore, we draw a square, and place the triangle on top. Start with the square. Let our square be of side SIDE, and let it be drawn from any arbitrary coordinate X,Y: make it into a procedure.

```
1000 DEF PROCL-SQUARE(X,Y, SIDE)
1010 PLOT 0,X,Y : REM MOVE TO X, Y
1020 PLOT 1,SIDE,0 : REM A RELATIVE PLOT
1030 PLOT 1,0,SIDE
1040 PLOT 1,-SIDE,0
1050 PLOT 1,0,-SIDE : REM BACK TO BASE
```

```
1060 ENDPROC : REM SQUARE COORD VERSION 1
```

As you will note, I have used ‘relative’ plots in constructing the routine (UG page 319), partly because it means that we only refer to X and Y at one point. Suppose that the graphics cursor was already at the point X, Y? We can ignore line 1010, and treat X and Y as global to the procedure (really we never have to refer to X or Y):

```
1000 DEF PROC_SQUARE(SIDE)
1010 PLOT l,SIDE,0 : PLOT 1,0,SIDE
1020 PLOT 1,-SIDE,0 : PLOT 1,0,-SIDE
1030 ENDPROC : REM SQUARE COORD VERSION 2
```

lines 1010 then become similar. PROC_SQUARE can be prettified

```
1000 DEF PROC_SQUARE (SIDE)
1010 PROC LSHAPE (SIDE) : PROC_LSHAPE(-SIDE)
1020 ENDPROC : REM SQUARE COORD VERSION 3
1025
1030 DEF PROC_LSHAPE (LIMB)
1040 PLOT 1,LIMB,0 : PLOT 1,0,L1MB
1050 ENDPROC : REM LSHAPE (by Lynne Reid Banks?)
```

and we now have two routines where once there was one. The second routine (ie PROC_LSHAPE) could be used for other shapes, other than the square: PROC_LSHAPE could become part of a library of ‘useful routines’ but really there is little point to the exercise. PROC_LSHAPE is not exactly memorable: can you remember which way it bends, and could you remember it in aeons to come, together with many other little gobbets of program?

How are we to tilt the house? How are we even to tilt the LSHAPE?

Intrinsic geometry

To draw a square we go forward a fixed distance and turn left (or fight) through 90 degrees. This we do four times: why not draw a square in this way? There seems to be an internal logic to a square, which does not depend upon sides being horizontal and vertical.

A geometrical figure, such as a square, has a certain ‘intrinsic’ property, which depends only upon that type of figure: that a square has four equal angles, and four equal sides, is independent of position and orientation. To say that the sides of a square must be parallel to the axes (as in the above routines) is an ‘extrinsic’ property: an external frame of reference is needed to decide which direction is horizontal.

Rather than concentrating on external properties, which is what one has to do if one concentrates on coordinates and transformations of coordinates, the routines herein concentrate upon the intrinsic properties of shapes. As Piet Hein implies, simplicity is the way to true art (‘Ars brevis’ means ‘Art in short’).

A procedure:

```
1000 DEF PROCL-SQUARE (SIDE)
1010 LOCAL I : FOR I=1 TO 4
1020 PROC_MOVE(SIDE,I) : PROC_TURN(90)
1030 NEXT I
1040 ENDPROC : REM SQUARE INTRINSIC VERSION
```

which uses two other procedures PROC_MOVE and PROC_TURN, of which more later. To draw a square of side 100, instantly we have to enter.

```
PROC_SQUARE(100)
```

where the square is drawn at the current cursor position. Highly complex shapes can be produced by use of a very few essential procedures. Design a procedure to draw an equilateral triangle:

```
1100 DEF PROC_TRIANGLE(SIDE)
1110 LOCAL I : FOR I = I TO 3
1120 PROC_MOVE(S1DE,1) : PROC_TURN(l20)
1130 NEXT I
1140 ENDPROC : REM TRIANGLE INTRINSIC VERSION
```

and it is simplicity itself. We can draw a triangle in exactly the same way as we drew the square, entering

```
PROC_TRIANGLE(SIDE)
```

then to tilt the square through 25 degrees counter clockwise

```
PROC_TURN(25) : PROC_SQUARE(l00)
```

in instant mode. To move to coordinates 20,250 (without plotting) and then draw a square turned through 43 degrees counter clockwise

```
PROC_MOVETO(20,250,0): PROC_TURN(43):
PROC_SQUARE(100)
```

- no calculation whatever, purely a concentration on the intrinsic nature of the problem. Note that in PROC_MOVE and PROC_MOVETO the final parameter is 1 for plot; and 0 for do not plot, just move.

One last example:

```
1200 DEF PROC_QUIZ(SIDE)
1210 LOCAL I : FOR I = I TO 6
1220 PROC_MOVE(SIDE) : PROC_TURN(60)
1230 NEXT I
1340 ENDPROC : REM QUIZ - GUESS WHAT
```

to whet your analytical appetite.

Turtle geometry

The above form of analysis, which concentrates on intrinsic properties of geometrical figures, is commonly called Turtle geometry. Papert's book, to which I earlier referred, is the classic text for explaining the reasons behind turtle geometry. The subtitle to Papert's book is 'Children, computers, and powerful ideas' and this has led some to believe that turtle geometry is 'kid's stuff', to be ignored, or left to little children.

This is not only wrong, but very short sighted: Turtle Geometry (Abelson and diSessa, 1980) is far beyond a child's picture book. Like any powerful methodology (or powerful computer language) the scope is vast: from the very simple (but never trivial) triangle to, eg, Lorentz transformations in relativistic mechanics. Turtle geometry is a powerful and accessible means of producing computer graphics which is not only creative but also of great utility.

Though using turtle graphics is a powerful tool, turtle graphics is also part of a philosophy and style of using computers. Two fundamental ideas which underlie much of the work in this book are:

> It is possible to design procedures which make the communication with computers a more natural process than is possible with more traditional methods of programming graphics;
> Learning to use graphics in this manner can assist in the learning of general thinking about program design — by considering the intrinsic nature of the problem and not its merely extrinsic aspects.

Experience is an important element in learning how to perform any task, and the computer can assist tremendously in producing many varied forms of experience. Many of the procedures (eg PROC_INSPIRALI) produce results which surprise myself, so expect to be surprised.

The approach through turtle graphics, and similar styles of application, are not the old 'discovery methods' they used to practise at school. With discovery methods the teacher or author of the book is supposed to know what is to be discovered — often the results of turtle routines are news to me as well.

In turtle graphics the user has control of a little (hypothetical) creature called a turtle, and the turtle fives on the surface of the visual display (TV or VDU). The turtle responds to a few very simple commands MOVE forward, and TURN through and angle (in the LOGO and SMALLTALK languages the commands are the same but differently named, see Papert, 1980, and Smalltalk-80 by Goldberg and Robson, 1983). The intrinsic routines PROC_SQUARE, PROC_TRIANGLE, and PROC_QUIZ could all be followed by a little turtle on the screen. Turtle geometry (eg constructing shapes by use of turtle(s)) is a useful alternative to traditional methods of 'doing graphics' — this will become clearer as we progress.

At more advanced levels, in the geometry of curved surfaces (ie differential geometry), the turtle always faces and turns in the tangent plane of the point on the surface at which the turtle is located. As the geometry of space in Einstein's theory of relativity is a differential geometry, the turtle can explore Einstein's universe — but we will not (for more details see Abelson and diSessa, 1980).

Turtle commands

The basic turtle commands are very few, a move command, and a turn command: theoretically these are all that are needed (a moveto command is not really necessary). The turtle commands I have implemented for **Version 1.1** are — apart from housekeeping commands — the following:

PROC_TURN (A)	Turn through A degrees
PROC_TURNTO(A)	Turn to angle A degrees
PROC_MOVE(D,S)	Move forward D units, S=1 to plot, S=0 to just move.
PROC_MOVETO(X,Y,S)	Move to coords X,Y, S=1 to plot, S=0 to move

plus a command

PROC_LOC	What are the coords, and what is the present angle?

However, to use these commands the screen needs to be organized in such a manner that text and graphics can be separated.

The screen housekeeping commands are:

PROC_START	Clear the screen and set up separate text and graphics screens, centre cursor
PROC_RESTART	Clear only the graphics screen and set the cursor to the centre
PROC_CENTRE	Centre the cursor
PROC_INVERT	Change the drawing colour from black to white or vica versa

But a program is worth a thousand words — so enter in routines Turtle **Graphics Version 1.1**. I will then discuss the routines in the order in which they appear in the listing.

```
1000 REM--------------------------------
1010
1020
1030 REM   G R A P H I C   ART
1040
1050 REM   (c) Boris Allen, 1983
1060
1070
1080 REM--------------------------------
1090
1100 REM   Turtle Graphics : 1.1
```

```
 1110
 1120 REM------------------------------
 1130
 1140 DEF PROC_CLRSCR
 1150 PROC_CLS : PROC_CLG
 1160 ENDPROC : REM CLRSCR
 1170
 1180 DEF PROC_CLG
 1190 GCOL 0,PEN : GCOL 0,129-PEN
 1200 VDU 24,0;128;1279;1023; : CLG
 1210 REM Clears an upper graphics windo
w
 1220 VDU 29,640;566;
 1230 REM Sets the origin to centre of g
raphics window
 1240 ENDPROC : REM CLG
 1250
 1260 DEF PROC_CLS
 1270 COLOUR 1-PEN : COLOUR 128+PEN
 1280 VDU 28,0,31,39,28 : CLS
 1290 REM Clears lower text window
 1300 ENDPROC : REM CLS
 1310
 1320 DEF PROC_COL(PE)
 1330 PEN=PE
 1340 ENDPROC : REM COL
 1350
 1360 DEF PROC_CENTRE
 1370 MOVE 0,0 : ANGLE=0 : X=0 : Y=0
 1380 ENDPROC : REM CENTRE
 1390
 1400 DEF PROC_RESTART
 1410 PROC_CLG : PROC_CENTRE
 1420 ENDPROC : REM RESTART
 1430
 1440 DEF PROC_START
 1450 PROC_COL(0) : PROC_CLRSCR : PROC_C
ENTRE
 1460 ENDPROC : REM START
 1470
 1480 DEF PROC_INVERT
 1490 PEN=1-PEN : GCOL 0,PEN
 1500 ENDPROC : REM INVERT
 1510
 1520 DEF PROC_TURNTO(A)
 1530 ANGLE=FN_ANGLE(A)
 1540 ENDPROC : REM TURNTO
 1550
 1560 DEF PROC_TURN(A)
 1570 ANGLE = FN_ANGLE(ANGLE+A)
 1580 ENDPROC : REM TURN
 1590
 1600 DEF PROC_LOC
 1610 PRINT "COORDINATES ARE ";X,Y'"ANGL
E IS "ANGLE
 1620 ENDPROC : REM LOC
 1630
 1640 DEF PROC_MOVE(DISTANCE,STYLE)
 1650 X=X - DISTANCE*SIN(RAD(ANGLE))
 1660 Y=Y + DISTANCE*COS(RAD(ANGLE))
 1670 IF STYLE=1 THEN DRAW X,Y ELSE MOVE
 X,Y
 1680 ENDPROC : REM MOVE
 1690
 1700 DEF PROC_MOVETO(XN,YN,STYLE)
 1710 LOCAL XDIF,YDIF : XDIF=XN-X : YDIF
=Y-YN
 1720 IF YDIF<>0 THEN PROC_TURNTO(DEG(AT
N(XDIF/YDIF))+180*(YN<Y)) ELSE PROC_TURN
TO(SGN(-XDIF)*90)
 1730 X=XN : Y=YN
 1740 IF STYLE=1 THEN DRAW X,Y ELSE MOVE
 X,Y
 1750 ENDPROC : REM MOVETO
 1760
 1770 DEF FN_ANGLE(A)
 1780 IF A MOD 360 <0 THEN =A MOD 360 +
360 ELSE =A MOD 360
 1790 REM ANGLE
 1800
 1810 DEF PROC_NEW
 1820 VDU 26 : CLS
 1830 ENDPROC : REM NEW
 1840
```

PROC_CLRSCR consists of two routines PROC_CLS and PROC_CLG, defined later.

PROC_CLG sets the graphics foreground and background colours by use of GCOL (UG page 262) and a global variable PEN. Usually the foreground is black and the background white. Line 1210 sets a graphics window (see above) and clears the graphics window; line 1220 sets the origin to the centre of the window.

PROC_CLS sets the text foreground and background colours (UG page 222) to the reverse of those normally set by PROC_CLG. A text window, for the lower four lines, is established and the text window is cleared.

PROC_COL is the first routine with a parameter, and is merely a way of altering the value stored in the global variable PEN. PEN normally has the value 0, but this allows that value to be changed procedurally (another way — less consistent — is PEN = PE).

PROC_CENTRE moves the cursor to the centre, and sets the global variables X,Y, and ANGLE all to zero.

PROC_RESTART clears the graphics window (PROC_CLG) and then centres the cursor (PROC_CENTRE) — the text is unaffected. The global variable PEN is not affected.

PROC_START sets the PEN to zero, clears both screens (PROC_CLRSCR) and centres the cursor (PROC_CENTRE). This routine has to be activated before any of the others, otherwise some of the global variables might be uninitialized.

PROC_INVERT changes the foreground colour in graphics, without altering the background colour (black to white and vice versa). To alter the value from O to 1 (or vice versa) the value is subtracted from I. Another way of changing the value might be PROC_COL(1-PEN). This routine is used for erasing existing lines.

PROC_TURNTO sets the value stored in the global variable ANGLE to the value supplied. To make sure that values do not go outside the range 0 to 359, the value is normalized by FN_ANGLE. Not a true turtle command because it refers to an absolute (rather than relative) angle, the name of this command derives from UCSD Pascal (Bowles, 1977), as do most of the routine names herein.

PROC_TURN takes the value of the parameter, and adds it onto the global variable ANGLE, where counting of angles is in a counterclockwise direction and uses FN_ANGLE.

PROC_LOC is simply an environmental enquiry ‘What is my location, and in which direction am I facing?’ . In my turtle graphics procedures you cannot see the turtle, and so this is to help when lost.

PROC_MOVE has two parameters: the first (DISTANCE) gives the distance to be moved, and the second (STYLE) indicates whether there is to be a plot or a move (many versions of turtle graphics have PENUP and PENDOWN commands). The calculations in fines 1650 and 1660 need not detain us.

PROC_MOVETO is not a true turtle command, because it is absolute not relative taken from UCSD Pascal. The first two parameters are the coordinates, and the final parameter is the plotting style (comparable to PROC_MOVE). Do not bother about the calculations needed, except to notice that the IF in line 1720 is to trap a possible division by zero.

PROC_NEW returns both graphics and text windows to their full screen values, and both cursors are horned (see UG page 387).

FN_ANGLE constrains the angle between 0 and 359, and the conditional (in line 1780) is to account for negative as well as positive angles — all negative angles become positive.

A square dance

As I have already noted, the only really essential commands are PROC_MOVE(DISTANCE,STYLE) and PROC_TURN(A): though the routines I shall discuss in the rest of this chapter make use of many of BB's facilities, they use only the basic two turtle commands. The routines are designed to operate in mode 4, though they will run under mode 0.
The routines are fisted as Turtle Routines 1.1, and the first to be examined will be that to draw a square. To draw a square we first draw the side of a square (ie PROC_SIDESQ), which — as we have seen — is move forward a distance and then turn through 90 degrees. The distance move forward corresponds to the parameter SIDE, and is the length of the side of the square.

```
2000 REM-------------------------------
2010
2020
2030 REM   G R A P H I C   ART
2040
2050 REM   (c) Boris Allen, 1983
2060
2070
2080 REM-------------------------------
2090
2100 REM   Turtle Routines : 1.1
2110
2120 REM-------------------------------
2130
2140 DEF PROC_SIDESQ(SIDE)
2150 PROC_MOVE(SIDE,1) : PROC_TURN(90)
2160 ENDPROC : REM SIDESQ
2170
2180 DEF PROC_SQUARE(SIDE)
2190 LOCAL I : FOR I=1 TO 4
2200 PROC_SIDESQ(SIDE) : NEXT I
2210 ENDPROC : REM SQUARE
2220
2230 DEF PROC_SQTURN
2240 LOCAL I,A$ : FOR I=1 TO 600
2250 PROC_MOVE(I,0) : PROC_SQUARE(I)
```

```
 2260 PROC_TURN(30) :A$=INKEY$(0)
 2270 IF A$="F" THEN ENDPROC ELSE NEXT I
 2280 ENDPROC : REM SQTURN
 2290
 2300 DEF PROC_INSPIRALR(SIDE,ANG,INC)
 2310 PROC_MOVE(SIDE,1) : PROC_TURN(ANG)
 2320 PROC_INSPIRALR(SIDE,ANG+INC,INC)
 2330 ENDPROC : REM INSPIRALR
 2340
 2350 DEF PROC_INSPIRALI(SIDE,ANG,INC)
 2360 REPEAT
 2370 PROC_MOVE(SIDE,1) : PROC_TURN(ANG)
 2380 ANG = ANG + INC : A$ = INKEY$(0)
 2390 UNTIL A$="F"
 2400 ENDPROC : REM INSPIRALI
 2410
 2420 DEF PROC_CIRCLE
 2430 LOCAL I : FOR I = 1 TO 360
 2440 PROC_MOVE(1,1) : PROC_TURN(1)
 2450 ENDPROC : REM CIRCLE
 2460
 2470 DEF PROC_PERIFIXED(PERIM,SIDES)
 2480 LOCAL ANG,I
 2490 ANG = 360/SIDES
 2500 FOR I=1 TO SIDES
 2510 PROC_MOVE(PERIM/SIDES,1) : PROC_TU
RN(ANG)
 2520 NEXT I
 2530 ENDPROC : REM PERIFIXED
 2540
 2550 DEF PROC_POLYGONS
 2560 LOCAL I
 2570 FOR I = 6 TO 36 STEP 6 : PROC_PERI
FIXED(1080,I)
 2580 PROC_CENTRE : PROC_TURN(I*10) : NE
XT I
 2590 ENDPROC : REM POLYGONS
 2600
 2610 DEF PROC_SNOWFLAKE(ODER,INC)
 2620 PROC_DECISION(ODER,60,INC)
 2630 PROC_DECISION(ODER,120,INC)
 2640 PROC_DECISION(ODER,120,INC)
 2650 ENDPROC : REM SNOWFLAKE
 2660
 2670 DEF PROC_DECISION(ODER,ANG,INC)
 2680 PROC_TURN(ANG)
 2690 IF ODER>0 THEN PROC_POINT(ODER-1,I
NC) ELSE PROC_MOVE(INC,1)
 2700 ENDPROC : REM DECISION
 2710
 2720 DEF PROC_POINT(O,I)
 2730 PROC_DECISION(O,0,I)
 2740 PROC_DECISION(O,-60,I)
 2750 PROC_DECISION(O,120,I)
 2760 PROC_DECISION(O,-60,I)
 2770 ENDPROC : REM POINT
 2780
 2790 DEF PROC_OUTSPIRAL(A,INC)
 2800 LOCAL I,A$
 2810 PROC_CENTRE : REPEAT
 2820 PROC_TURN(A) : PROC_MOVE(I,1) : I
= I + INC
 2830 A$=INKEY$(100) : UNTIL A$="F"
 2840 ENDPROC : REM OUTSPIRAL
```

To draw a square we have to draw four sides, which is just what PROC_SQUARE does. The loop counter (I) is declared as being local to the procedure, as it is only operative within that procedure. Another way of writing the procedure would be on the one fine, say. Now to use PROC_SQUARE.

First, we clear the decks for action by setting up the special text and graphics screens, and, second, we draw the square (of side 200):

PROC_START : PROC_SQUARE (200)

We enter this interactively, and see a square (almost) instantly appear. To draw a bigger square, tilted through 50 degrees, keeping the first square on screen:

PROC_TURN(50) : PROC_SQUARE(300)

a tilted square appears. Try

PROC_TURN(-100) : PROC_SQUARE(300)

and a similar larger square will appear — tilted in the opposite direction (50 - 100 = -50). Now clear the graphics, keep the text, and move the square away from the centre:

PROC_RESTART : PROC_MOVE(l00,0)
PROC_SQUARE(200)

and it is worth entering the two lines separately.

The latter two lines will: (a) clear the graphics screen; (b) move 100 units in a upwards direction — without plotting; and (e) draw a square of side 200 units at that point. Note the different effect.

PROC_RESTART : PROC_TURN(160) : PROC_MOVE(200,0)
PROC_SQUARE(200)

produces (a tilted square, bottom leftish). Finally, try

```
PROC_RESTART : PROC_MOVETO(200,-300,0)
PROC_SQUARE(200)
```

to see how it is possible to use non-basic turtle commands. Try to let the squares dance by using PROC_SQTURN.

In this routine I and A$ are local to the procedure: I is used as a loop counter, and A$ is used as a means to produce early termination (the F key is pressed). For up to 600 times the cursor (or turtle) moves forward a distance I (without drawing), draws a square of side I, and then the turtle turns through 30 degrees. The keyboard is checked by INKEY$(0) to see if a key is pressed (saved in A$); if the key was an F, then ENDPROC else the loop counter is incremented.

The routine is activated by

```
PROC_RESTART : PROC_SQTURN
```

and it runs remarkably quickly. If you want to slow it down, put a pause in the INKEY$, eg INKEY$(20), but this does not slow down the drawing of the squares — it just increases the time between squares (see UG page 276, for INKEY$). This is a tediously predictable routine — it is the same every time. The predictability is shown in **Icon 1.1**.

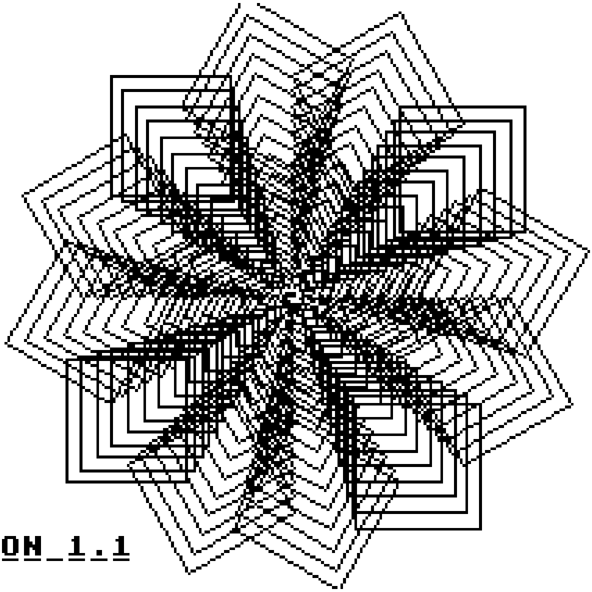
Note that Figures are not computer output, they may be diagrams or tables which are there to assist in the understanding of the text. An Icon — which is ‘an image, picture, or representation’ according to the dictionary — is an exact copy of a display on the screen, it is a screen dump onto a printer.

Need a rest? Take Five.

An unsquare dance

I walk forward a certain, fixed, distance and turn through a certain, changing, angle: what happens?
Solution: see PROC_INSPIRALI.

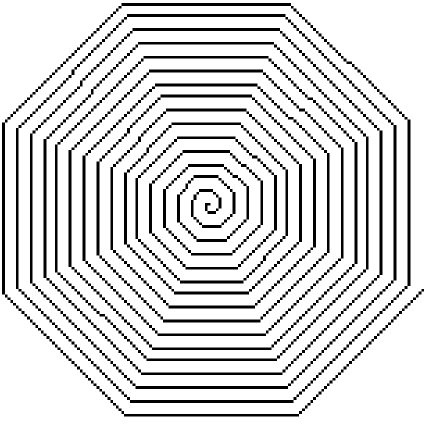
A$ is local again, tedious but safe, and the routine repeats until the F key is pressed. The distance is kept constant (ie SIDE), but the angle (ANG) is incremented (by INC) at each pass through the indefinite loop (ie REPEAT. . .UNTIL. . .). This routine produces a vast array of unpredictable results, which, once known, are completely predictable. It is named PROC_ANSPIRALI because it is an INward SPIRAL, coded Iteratively. Iteration means, as explained in the previous chapter, that the control of the procedure is by a loop mechanism — in this case REPEAT. . .UNTIL.

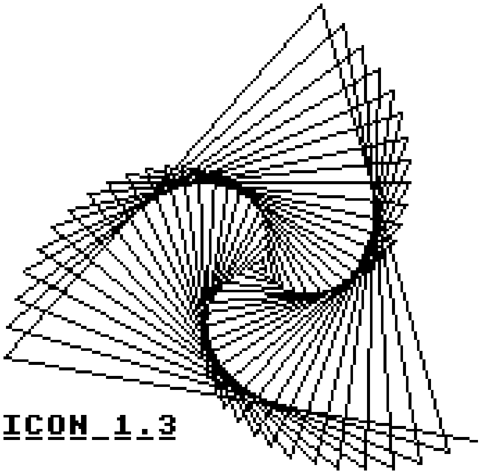


ICON_1.1

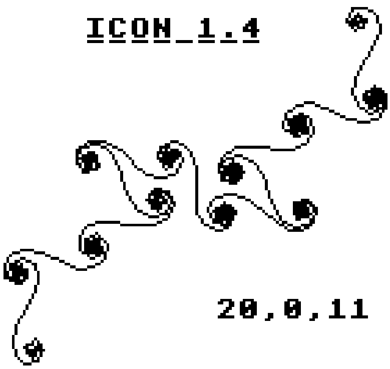
An outward SPIRAL is shown by PROC_OUTSPIRAL, in which the angle remains constant and the distance increases, and is what we normally mean by a spiral. **Icons 1.2** and **1.3** show two examples of outward spirals for varying values of the fixed angle.

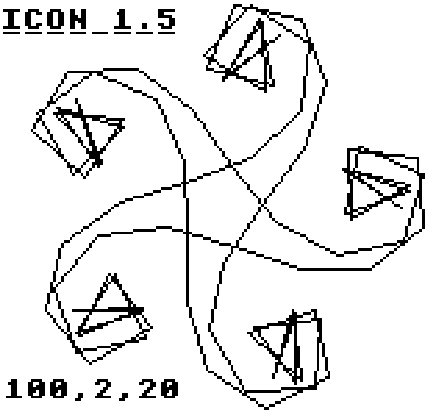
ICON_1.2

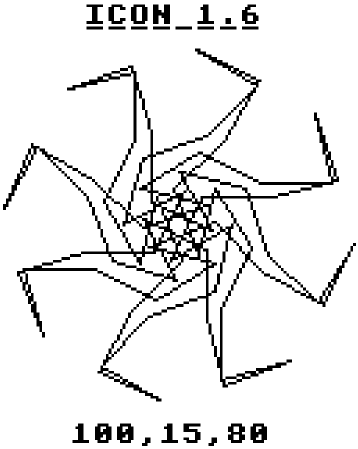


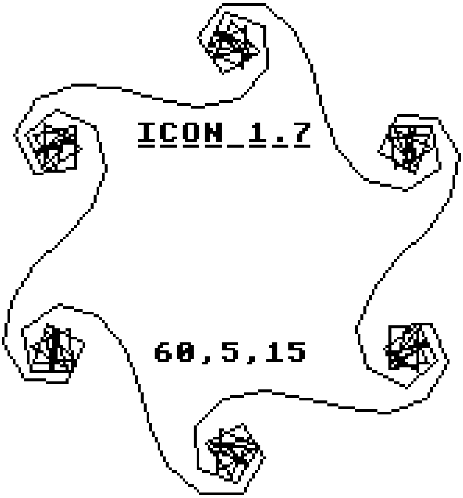


An outward spiral — as the name spiral suggests — keeps on spiralling outwards, but an inward spiral does nothing as common as that. Icons 1.4 to 1.7, are examples of four highly different shapes. Try to watch what happens as the plotting unfolds: if it helps to slow down the process, change the value of the parameter in the INKEY$.




ICON_1.5
100,2,20


ICON_1.6
100,15,80



The path begins by spiralling inwards, that is, turning in on itself. After a varying number of turns (depending upon INC) the angle becomes greater than 180 degrees, and so the path appears now to unwind. Allowing PROC_INSPIRALI to carry on until completion always results in a closed figure, ie the path ends up where it started, and it retraces the original path. Using PROC_INSPIRALI is a beneficial exercise for the imagination.

As a further way of producing the same (well, almost) effect, examine PROC_INSPIRALR. The final R is for Recursive, and this is a less efficient method for producing the same effect as the iterative version. The routine is recursive because in its own body it calls itself (ie PROC_INSPIRALR(SIDE,ANG + INC,INC)). That it is less efficient can be seen when some of the effects are duplicated — we are forever running out of room before it has ended. It's a Raggy Waltz.

Both sides now

Between the inward and the outward spirals there is the limbo spiral (if you are not in or out you must be in limbo). The limbo spiral has a rather more familiar designation as the circle. To draw a circle you can use the formula X^2 + Y^2 = R^2, or use the two equations X = R*COS (ANGLE) and Y = R*COS(ANGLE). The best way to draw a circle is to move forward a fixed distance, turn through a fixed angle, and carry on until you are back where you started. Try the Circle Game, use PROC_CIRCLE.

It is a circle it is true, but remember that the resolution on the screen is not perfect: perhaps it is possible to get away with less work?

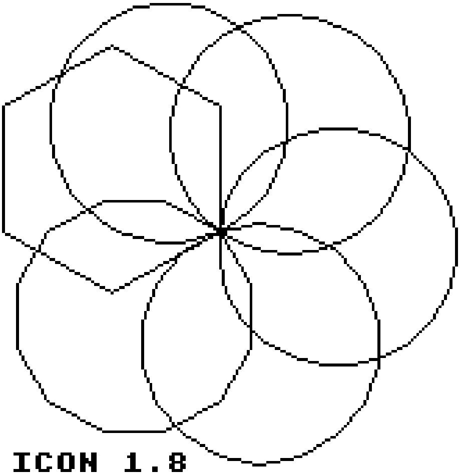
PROC_PERIFIXED takes as a parameter the perimeter of the shape you wish to draw, and as the second parameter the number of sides in the shape.

To draw a regular (ie equilateral) triangle of side 400, therefore, we enter.

```
PROC_RESTART : PROC_PERIFIXED(400,3)
```

and a triangle is drawn. To draw a succession of shapes, all with the same perimeter being the parameter. It can be seen that the shape soon approaches that of a circle, given the imperfections of the screen resolution. A circle is a shape of constant curvature — a limbo spiral.

By examination of the result of PROC_POLYGONS, I think that a polygon of 30 sides is more than adequate (even for large perimeters), and 24 sides is a good approximation (see **Icon 1.8**). We will develop the circle motif further, next chapter: note that to draw a circle of 30 sides with PROC_PERIFIXED (not an optimal method of drawing a circle) takes about two seconds, depending on size.



ICON_1.8

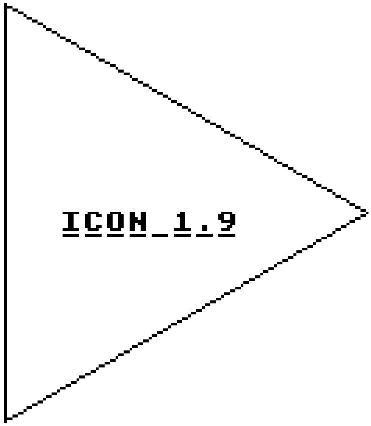
The snowflake curve

In *Mathematics and the Imagination* (Kasner and Newman, 1940), there is a chapter on "Change and changeability", and what we have been discussing is all about change and changeability

There is an appendix to their chapter, on Pathological Curves, ". . .each of whom has an individual history resembling no other", and Kasner and Newman start by discussing a normal and healthy set of curves: the polygons which approach closer and closer to circles, as we found with

PROC_POLYGONS.

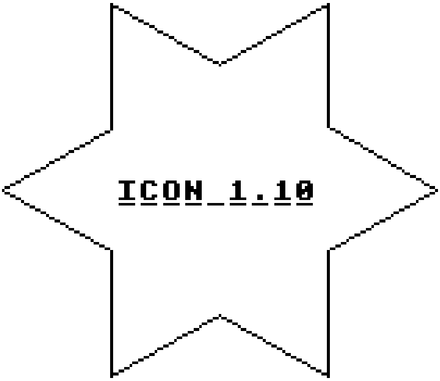
The first pathological curve they discuss is the Snowflake curve, which starts out life as an equilateral triangle (**Icon 1.9**). Each side of the triangle is trisected (cut into three equal parts), and on each middle third an equilateral triangle is drawn (a first order snowflake, **Icon 1.10**). The trisection process is repeated again on each side, to produce a second order snowflake (**Icon 1.11**). The process then continues as long as desired.

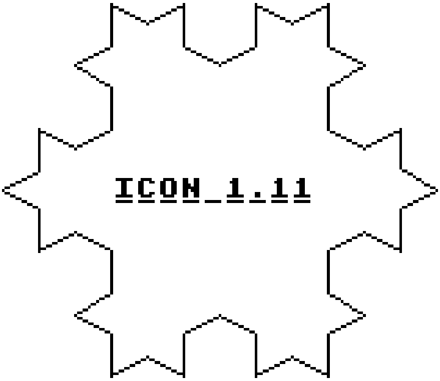


This curve is called 'pathological' because — as you can see — the perimeter of the snowflake increases with each trisection (it is an extra third larger), however, — as is also clear to see — the area enclosed by the snowflake is not infinite, it approaches a limit (as do the polygons). Ultimately, we have a curve of an "infinite" perimeter enclosing a finite area: this is why the snowflake is pathological.

The snowflakes are very pleasing and satisfying to draw: this is why I draw them. As you may be able to appreciate from my description, the snowflaking process is "Take one snowflake, and then modify it". This is an example of recursion: which is another reason why I chose this subject. To clarify what happens I have split the drawing of the snowflakes into three routines.

Just consider what happens along any side. One goes forward a third of the distance, and turns through 60 degrees; one moves forward the same distance, and turns through — 120 degrees; the same distance again, and turn through 60 degrees; and then that distance again. If the final stage of the snowflaking has been reached, a straight line (without bumps) is drawn.


ICON_1.10


ICON_1.11

There are three procedures: (a) to set the basic shape of the equilateral triangle (PROC_SNOWFLAKE); (b) to decide upon whether a straight line or bump is to be drawn (PROC_DECISION); and (c) to draw a bump, with possibly smaller bumps on the bump (PROC_POINT which makes four calls to PROC_DECISION).

These procedures are recursive in a slightly different manner, no procedure refers immediately to itself. What happens is that PROC_DECISION refers to PROC_POINT which refers to PROC_DECISION (and so it continues, until ODER is zero — not ORDER because OR is a BB keyword).

There are now three ways we can draw an equilateral triangle: we can use a purpose built routine, we can use PROC_PERIFIXED with the number of sides being three, or we can use PROC_SNOWFLAKE with the order equal to zero. The latter:

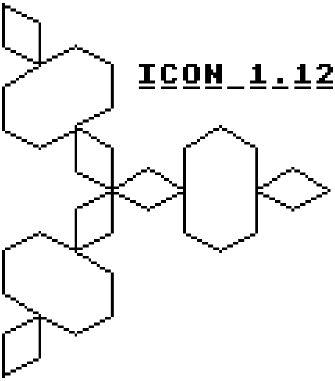
```
PROC_START : PROC_SNOWFLAKE(450,0)
```

and to draw snowflake of order 1 on top of the triangle

```
PROC_SNOWFLAKE(150,1)
```

The rest is up to you. Try to move the snowflake to other positions, other sizes, and with different orientations.

As a last task, design routines to draw the Anti-snowflake Curve (Kasner and Newman, 1940 : page 299, and also my **Icon 1.12**). The triangles are drawn inward, towards the centre, rather than outward. Hint: the only routine which needs altering is PROC_POINT. No answer is supplied.



CHAPTER 2
Turtle Geometry

> Don’t talk to me about mathematics —
> I’ve come to the conclusion that I’ve
> learnt to live without it.
>
> *Prince Philip, The Duke of Edinburgh*
>
> There is no ‘royal road’ to geometry.
>
> *Euclid*

The turtle procedures are simple to use because they accentuate the intrinsic properties of shapes.

When we study the intrinsic properties of shapes, we look at the angles (which never vary, though the size be doubled), and at the ratios of the size of one side compared to another. To talk of angles and ratios of sides, is to talk of trigonometry.

Triangulation

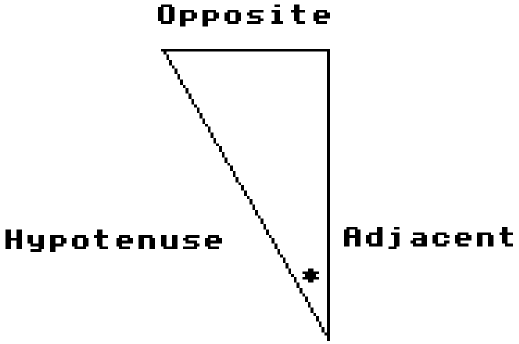
Enter the following procedure, which is not well-written TG because it uses too many moveto and turnto commands:

```
4000 DEF PROC_TRI(SIDE)
4010 PROC_MOVETO(200,-200,0) : PROC_TURNTO(30)
4020 PROC_MOVE(SIDE,1) : PROC_TURN(-120)
4030 PROC_MOVE(SIDE/2,1) : PROC_TURN(-90)
4040 PROC_MOVETO(200,-200,1)
4050 ENDPROC : REM TRI
```

(it is assumed that the basic TG procedures are already there). If the following instant one-line program is entered, eg

```
FOR I = I TO 7 : PROC_TRI(100*I) : NEXT I
```

then there are seven superimposed triangles drawn, each getting bigger enclosing the smaller triangles. The angles in the triangle are (bottom) 30 degrees, (top right) 90 degrees, and (top left) 60 degrees; these angles are the same for each triangle — see **Icon 2.1**, which shows a similar type of triangle.



ICON_2.1

The longest of the lines (ie the first to be drawn) is always twice as long as the shortest line (second to be drawn) — how long (relatively) is the other fine? Hint: use PROC_LOC to work out the coordinates, after the second fine is drawn (ie after fine 4030). What happens if we try to

PRINT SIN(RAD(30)); COS(RAD(60))

should not be very surprising, but has to be explained.

Both SIN(RAD(30)) and COS(RAD(60)) are equal to 0.866025404, and if the value of sine 30 degrees is found from a book of tables, it is 0.8660 (as is the value of cosine 60 degrees). Check that the value of sine 60 degrees (and of cosine 30 degrees) is 0.5. The question now is to relate this to the triangle.

Enter

PROC_RESTART : PROC_TRI(500)

where the coordinates of the lower apex of the triangle are (as one would expect from the content of the routine) 200, — 200 (X axis, Y axis). By dextrous use of PROC_LOC (twice) it can be found that the upper left apex is at -50,233.012702 with the upper right apex at 200,233.012702.

We know from the routine that the long side of the triangle is 500. This is called the hypotenuse of the right angled triangle (the right angle is that of 90 degrees at the top fight apex). The hypotenuse is always opposite the right angle. The vertical side (ie the medium sized side) is 233.012702 — 200 — 433.012702, and the small side is 250 (both from the coordinates, and from the routine, ie 500/2).

If we concentrate on the angle at the lower apex, an angle of 30 degrees, then we can give names to the sides of the right angled triangle. The side opposite the angle (in this case the shortest side) is called the opposite; the side next to the angle, which is not the hypotenuse, is called the adjacent (**Icon 2. 1**). The sine and cosine of an angle are defined by

sine = opposite/hypotenuse
cosine = adjacent/hypotenuse

so that

sine(30) 250/500 = 0.5
cosine(30) = 433.012702 = 0.866025404

which checks. Check that it checks.

Degrees of radians

We have been working in degrees, and this is the means by which most of us find it easiest to conceptualise angles. In mathematics (and computer arithmetic) it is easier to work in terms of radians rather than degrees. For example (and do not lose any sleep over it), if x is given in radians

sine(x) = x - x^3/3! + x^5/5 - x^7/7! + . . .
cosine(x) 1- x^2/2! + x^4/4! - x^6/6! + . . .

which is very nice and simple (x^3 — for example — is x*x*x and 3! is 1* 2*3). So that we can still work in degrees, even though the calculations are simpler for the machine in radians, BB (UG page 331) has a RAD function to change an angle measured in degrees to its equivalent in radians.

To turn completely round is to go through 360 degrees, or to turn through 2*PI radians (PI is a BB constant equal to 3. 14159265, see UG page 318). A radian is also called a circular measure. One radian is the angle at the centre produced by an arc on the circumference of a circle, where the length of the arc is the same as the radius — a form of circular equilateral triangle.

You would expect that, as the circumference is curved, the other two sides would be slightly closer together than for a normal equilateral triangle. They are. The angle is about 57 degrees for the arc, as against 60 degrees for the equilateral triangle (compare UG page 331). If you know that the length of the circumference is 2*PI*RADIUS, you might like to work out why the radian is equal to 180/PI degrees.

Note that in the TO routines we always use degrees, and the conversion into radians is done within the routines. This is how it should be, though radians have a certain mathematical felicity degrees have an overwhelming familiarity.

Further functions

The classic three functions of trigonometry are the sine, cosine, and the tangent, with the definition of the tangent being

tangent = opposite/adjacent

which, for our example triangle, produces

tangent(30) = 250/433.012702
= 0.577350269

in agreement with

```
PRINT TAN(RAD(30))
```

the way we refer to the tangent in BB (UG page 362). We use all three functions in the TO routines, and, interestingly, in the notes of Kasner and Newman's discussion of 'Change and Changeability' (used last chapter, when examining pathological curves) they give a good brief explanation of what they term trigonometrical 'ratios' (ie functions).

Actually we do not use the tangent in the routines, rather we use the 'arctangent'. The arctangent (written as ATN in BB — see UG page 210) gives the angle corresponding to a tangent value. T AN(RAD(30)) is equal to 0.577350269, and so ATN(0.577350269) is 30 degrees, but expressed in radians: DEG(ATN(.577350269)) converts the result so that it is given in degrees — DEC is the reverse of RAD. ACS (arccosine) and ASN (arcsine) are also available in BB (U G pages 201, 209). Another name for the arctangent is the 'inverse' tangent, and similar for the others.

At this point it is worth examining PROC_MOVE and PROC_MOVETO.

Moving commands

If we return to the triangle example, it can be seen that if we start at the lower apex (call it Xl, Yl), then the coordinates of the upper left apex (X2,Y2) are related to the distance between the two apexes (DIST) by

```
X2 = X1 - DIST*SIN(RAD(30))
Y2 = Y1 + DIST*COS(RAD(30))
```

If there is no need to remember where are X1 and Y1 then there is no need to distinguish between, say, Xl and X2 in the equations; and if instead of 30 degrees we insert any angle, then we arrive at

```
X = X - DIST ANCE*SIN(RAD(ANGLE))
Y = Y + DIST ANCE*COS(RAD(ANGLE))
```

(see PROC_MOVE).

Though this is, at least on the face of it, perfectly acceptable, we do not know that it will work for angles outside the range of 0 degrees to 90 degrees. To prove that it works, we need to find the sign of the trigonometrical ratios for various angles: try it for yourself, and prove to yourself that the equations always work. If the equations do not work, let me know.

To move to a certain position at first appears to be a simple exercise: merely a use of the commands DRAW or MOVE from BB. This is true, but we need to know at what angle the turtle is pointing after the move. We need to know the angle — this is the key. We know the coordinates, the distances, we do not know the angle.

If we know the sides of a triangle, and we want to know angles, we use the inverse trigonometrical functions: in this case we know the opposite and the adjacent sides, so we use the arctangent. In PROC_MOVETO the local variables XDIF and YDIF correspond to distances along the side of a right angled triangle, in the X and Y directions respectively. The angle (in radians) corresponding to these distances is the arctangent of (XDIF/YDIF), with certain adjustments.

The first adjustment is to try to account for angles outside the range 0 degrees to 90 degrees. If

```
PRINT DEG(ATN(1)); DEG(ATN(-1))
```

is tried, the results are 45 and -45 (ie degrees). Relating this to the value of the ratio (XDIF /YDIF) indicates that, when the angle is 45 degrees, both XDIF (equivalent to X2 - Xl) and YDIF (ie Yl - Y2) are negative, so that the ratio is positive. This is correct so far. When both XDIF and YDIF are positive (bottom left corner) the angle is 225 degrees.

For -45 degrees, XDIF is positive and YDIF is negative, so the ratio is negative: again correct. When XDIF is negative and YDIF is positive, the ratio is also negative, but the angle corresponds to 135 degrees.

If YDIF is negative, the angle lies between -90 (270) degrees and 90 degrees (ie is upwards) — the range of values given by ATN. When YDIF is positive (ie Y1 is greater than Y2) then 180 degrees has to be added to the angle between -90 degrees and 90 degrees — the adjustment being accomplished by +180*(YN<y) in PROC_MOVE. (On the use of logical comparisons see UG pages 99 to 101, and 369).

The second adjustment is of a different nature: it stops the computer having to divide by zero. In the division XDIF/YDIF, if YDIF is zero (a horizontal fine) then the result is indeterminate, and the computer produces an error. The check in the IF statement is to stop such an error, and after the ELSE the turn is either 90 degrees (if XDIF is negative) or -90 degrees (if XDIF is positive), that is, SGN(-XDIF)*90. FN_ANGLE turns the negative values into the correct positive angle.

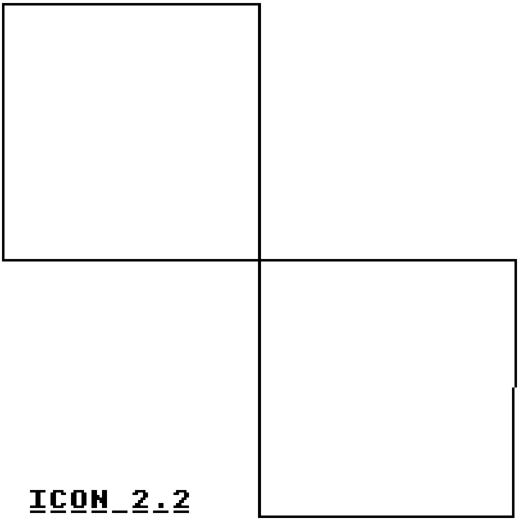
Circular motion

A knowledge of trigonometry is not necessary to use TG commands, but to advance in graphics a knowledge of trigonometry is valuable if not totally necessary . In this section we will investigate the drawing of a circle, with an adjustable centre and variable radius.

Start simple. What is a circle? A circle is no more than a polygon with 30 sides — at least the way we will draw it. 30 sides is too many to easily examine: why not start with a square (a polygon of four sides)? Two calls to PROC_PERIFIXED :

PROC_PERIFIXED(1600,4) : PROC_PERIFIXED(-1600,4)

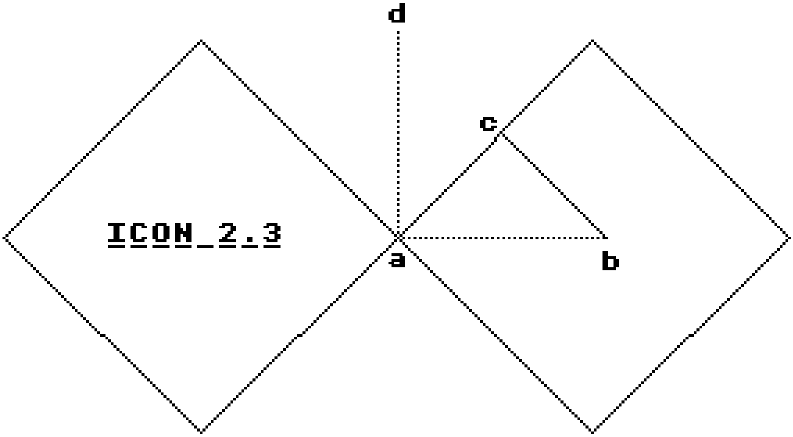
to produce the arrangement of **Icon 2.2**. If instead of four sides we request 30 sides, then we also get two slightly skewed circles (only the skewness of the squares amplified).



To draw circles, we need to 'de-skew' them, only we do not de-skew the 30 sided polygons at first: we start simple, and de-skew the squares. To de-skew the squares all we need to do is tilt the squares through 45 degrees:

PROC_TURN(45) : PROC_PERIFIXED(1600,4) :
PROC_PERIFIXED(-1600,4)

This arrangement is shown in **Icon 2.3**, with the addition of a few extra fines and labels. The dotted line from a to d (ie ad), is at an inclination of 0 degrees; the line ab is at 270 degrees; the fine be is at 45 degrees; and the line an is at 3 1 5 degrees.



These angles are only true for a square, but certain intrinsic properties are true for all polygons:

ac = ab*sine(abc)
angle(cad) = angle(abc)

so that if ab is the radius, then the distance through which the turtle moves at each side is twice the length of ac. The distance is thus 2*ac = 2*radius* sine(abc) — the angle a be for a 30-sided polygon is 360/30*2 = 6 degrees (why?). The distance moved along each side is thus 2*radius*sine(6). We also have to start by turning through six degrees.

It makes sense to work through the preceding argument, if it is not too clear, because this is the justification of PROC_C30 (shown in **Turtle Routines 1.2**). The routine is called C30 as it draws a circle by actually drawing a 30-sided polygon.

The local variable J refers to the size of each side of the polygon, where R is the radius, and the local variable I is a loop counter. The routine assumes that the turtle is at the centre of the circle to be drawn, but that the angle at which the turtle is facing is unknown. The turtle moves forward a distance R without drawing, and is turned through 96 degrees (ie to the left 90 degrees and down six degrees from this).

The circle is drawn in a conventional manner as a 30-sided polygon, where the move is J forward and the turn is through 12 degrees. Finally the turtle turns to point back, returns to the centre of the circle, and is then left pointing in the original directIon.

Changing plotting styles

I suggested that when investigating the random walk that different values of the PLOT parameter be tried, to see their differing effects. The MOVE and MOVETO routines in Version 1.1 only offer two styles — draw a line, or do not draw a fine — so why not expand the scope to use more of the BB plotting variants?

In **Turtle Graphics Version 1.2** most of the routines are the same, the only differences come in the two moving commands. If the UG (pages 3 I 9, 320) is consulted, it can be seen that plotting commands come in groups of eight: within each group of eight we want the sixth command — which always draws a fine absolute in the current graphics foreground colour.

Wanting to keep the commands MOVE and MOVETO compatible with those of Version I. I, I decided that 0 should remain as move, do not draw, and kept I as draw a fine. In terms of the plotting commands, 0 in MOVE corresponds to 0 in PLOT, and I in MOVE corresponds to 5 in PLOT. This explains how the plotting mode is calculated in the new moving commands — the style parameter of the moving commands, and their action, are shown in Figure 2. I.

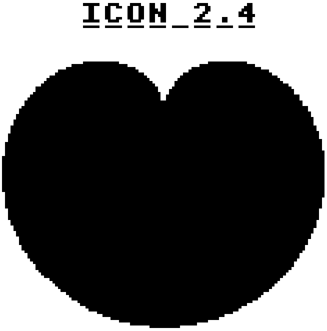
Figure 2.1 Drawing Styles

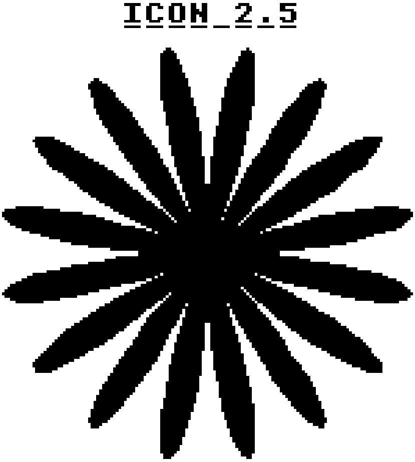
MOVE	PLOT	ACTION
0	0	Just move
1	5	Draw line
2	13	Draw line without last point
3	21	Dotted fine
4	29	Dotted fine without last point
9	69	Plot point
11	85	Fill triangle

It is also possible to use the PLOT parameters directly as a parameter in MOVE and MOVETO, but using my method is fairly simple, and comprehensive: if you want to do more, then do so.

These different styles are illustrated by **Icons 2.4** to **2.8.** All these examples are obtained by use of the same routine PROC_SINFN, with different parameters. PROC_SINFN is an example of the use of TG to provide ‘polar’ plots: a polar coordinate of a point is the distance of the point from the origin, together with the angle at which the point is in relation to the origin (sounds very TG?).

PROC_SINFN relates the angle of inclination to the distance away from the origin by some function of the sine of the angle, with the factor being variable. The first example (**Icon 2.4**) shows a ‘cardoid’, with a factor of 1/2 and a plotting style of 11 (ie PLOT with 85, filling in triangles, effectively filling the shape). **Icon 2.5** also fills triangles, and the factor is 8.

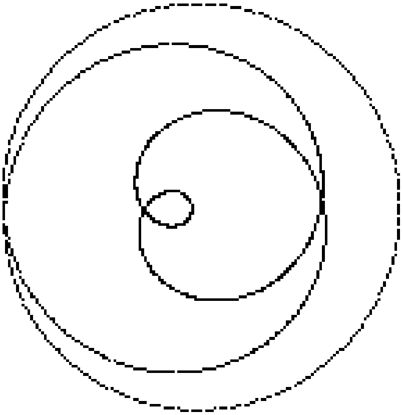




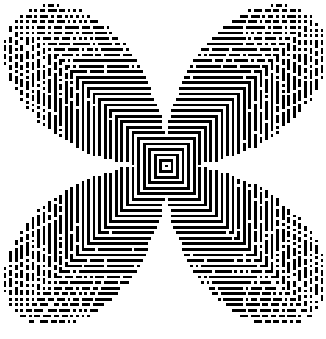
Icon 2.6 uses a style of 9 (ie PLOT with 69), that is, plot a point: the factor is less than 1/2, I will let you discover which value it is. A dotted line (style 3, PLOT with 21) is shown in **Icon 2.7**, the factor is greater than I, but work it out (note the differences between even and odd factors greater than I). The use of the dotted fine produces some interesting interference patterns.

Last of all, we have another circle — **Icon 2.8**. In this case the style is I (PLOT with 5) — draw a fine. It reinforces the relationship between trigonometrical ratios, and shapes in general. The interference patterns are again quite intriguing, and also known as Moire Effects.

ICON_2.6



ICON_2.7



ICON_2.8

The time has now come to move up a gear, to a mode of many colours.

```
 1000REM-----------------------------
 1010
 1020
 1030REM   G R A P H I C   ART
 1040
 1050REM   (c) Boris Allen, 1983
 1060
 1070
 1080REM-----------------------------
 1090
 1100REM   Turtle Graphics : 1.2
 1110
 1120REM-----------------------------
 1130
 1140 DEF PROC_CLRSCR
 1150 PROC_CLS : PROC_CLG
 1160 ENDPROC : REM CLRSCR
 1170
 1180 DEF PROC_CLG
 1190 GCOL 0,PEN : GCOL 0,129-PEN
 1200 VDU 24,0;128;1279;1023; : CLG
 1210 REM Clears an upper graphics windo
w
 1220 VDU 29,640;566;
 1230 REM Sets the origin to centre of g
raphics window
```

```
 1240 ENDPROC : REM CLG
 1250
 1260 DEF PROC_CLS
 1270 COLOUR 1-PEN : COLOUR 128+PEN
 1280 VDU 28,0,31,39,28 : CLS
 1290 REM Clears lower text window
 1300 ENDPROC : REM CLS
 1310
 1320 DEF PROC_COL(PE)
 1330 PEN=PE
 1340 ENDPROC : REM COL
 1350
 1360 DEF PROC_CENTRE
 1370 MOVE 0,0 : ANGLE=0 : X=0 : Y=0
 1380 ENDPROC : REM CENTRE
 1390
 1400 DEF PROC_RESTART
 1410 PROC_CLG : PROC_CENTRE
 1420 ENDPROC : REM RESTART
 1430
 1440 DEF PROC_START
 1450 PROC_COL(0) : PROC_CLRSCR : PROC_C
ENTRE
 1460 ENDPROC : REM START
 1470
 1480 DEF PROC_INVERT
 1490 PEN=1-PEN : GCOL 0,PEN
 1500 ENDPROC : REM INVERT
 1510
 1520 DEF PROC_TURNTO(A)
 1530 ANGLE=FN_ANGLE(A)
 1540 ENDPROC : REM TURNTO
 1550
 1560 DEF PROC_TURN(A)
 1570 ANGLE = FN_ANGLE(ANGLE+A)
 1580 ENDPROC : REM TURN
 1590
 1600 DEF PROC_LOC
 1610 PRINT "COORDINATES ARE ";X,Y'"ANGL
E IS "ANGLE
 1620 ENDPROC : REM LOC
 1630
 1640 DEF PROC_MOVE(DISTANCE,STYLE)
 1650 X=X - DISTANCE*SIN(RAD(ANGLE))
 1660 Y=Y + DISTANCE*COS(RAD(ANGLE))
 1670 IF STYLE<>0 THEN PLOT (STYLE-1)*8+
5,X,Y ELSE MOVE X,Y
 1680 ENDPROC : REM MOVE
 1690
 1700 DEF PROC_MOVETO(XN,YN,STYLE)
 1710 LOCAL XDIF,YDIF : XDIF=XN-X : YDIF
=Y-YN
 1720 IF YDIF<>0 THEN PROC_TURNTO(DEG(AT
N(XDIF/YDIF))+180*(YN<Y)) ELSE PROC_TURN
TO(SGN(-XDIF)*90)
 1730 X=XN : Y=YN
 1740 IF STYLE<>0 THEN PLOT (STYLE-1)*8+
5,X,Y ELSE MOVE X,Y
 1750 ENDPROC : REM MOVETO
 1760
 1770 DEF FN_ANGLE(A)
 1780 IF A MOD 360 <0 THEN =A MOD 360 +
360 ELSE =A MOD 360
 1790 REM ANGLE
 1800
 1810 DEF PROC_NEW
 1820 VDU 26 : CLS
 1830 ENDPROC : REM NEW
 3000REM-------------------------------
 3010
 3020
 3030REM   G R A P H I C   ART
 3040
 3050REM   (c) Boris Allen, 1983
 3060
 3070
 3080REM-------------------------------
 3090
 3100REM   Turtle Routines : 1.2
 3110
 3120REM-------------------------------
 3130
 3140 DEF PROC_C30(R)
 3150 LOCAL I,J
 3160 J = 2*R*SIN(RAD(6)) : PROC_TURNTO(
0) : PROC_MOVE(R,0) : PROC_TURN(90+6)
 3170 FOR I = 1 TO 30 : PROC_MOVE(J,1) :
 PROC_TURN(12) : NEXT I
 3180 PROC_TURN(90-6) : PROC_MOVE(R,0) :
 PROC_TURN(180)
 3190 ENDPROC : REM C30
 3200
 3210 DEF PROC_SINFN(SIZE,FACTOR,STYLE)
 3220 LOCAL I,MAX
 3230 IF FACTOR <= 1 THEN MAX=180/FACTOR
 ELSE MAX=180-(INT(FACTOR/2)*2=FACTOR)*1
80
 3240 FOR I=1 TO MAX : PROC_CENTRE : PRO
```

```
C_TURNTO(I) : PROC_MOVE(SIZE*SIN(RAD(I*F
ACTOR)),STYLE) : NEXT I
 3250 ENDPROC : REM SINFN
```

CHAPTER 3
Turtle Graphics III

> Whatever colors you have in your mind
> I'll show them to you and you'll see them
> shine
>
> Bob Dylan, "Lay, Lady, Lay"

The next obvious extension to TG is to have more than one turtle, with each turtle drawing lines of a specific colour — a colour which might be different from the other turtles. We need a four colour mode: one colour for the background, and three colours for the turtles.

Mode 1 takes up more memory, though with an increase in resolution, so we will normally use mode 5 to allow more space for complex applications. The question is then which four colours? (and then how to control three different turtles at once). The choice of colours, and the three turtles, incorporates the distinction between 'actual' and 'logical' colours.

Choosing colours

There are four colours available in modes 1 or 5 at any one time. These are the four 'logical' colours labelled 0, 1, 2, and 3: normally these colours are black, red, yellow, and white. Black has an 'actual' colour number 0, red is 1, yellow is 3 and white is 7 (see UG pages 162-168, 222-224, and 262).

It is possible, therefore, to change the logical colour 0 from black (actual colour 0) to flashing cyan-red (actual colour 14), by assigning the logical colour to a new actual colour number. The logical number is like the address of a house, which never varies, but whose occupant (the actual colour) may vary, without the house number altering.

The logical numbers 0 to 3 are 'foreground' colours, they define the colours of the text, or fines in graphics: corresponding to logical foreground colour 0 is background colour 128, and for 3 there is 131. If we printed in logical 2 on a logical 130, we would not be able to see what had been printed (text and background would be the same actual colour).

The normal association of logical and actual colours for mode 5 (as well as mode 1) is shown in Figure 3.1.

Figure 3.1 Colours in Modes 1 and 5

LOGICAL NUMBER		ACTUAL NUMBER	COLOUR
0	128	0	Black
1	129	1	Red
2	130	3	Yellow
3	131	7	White

To change the association we use another VDU command (UG pages 224, 382)

```
VDU 19, logicalnumber, actualnumber, 0, 0, 0
```

that is:

```
VDU 19, logicalnumber, actualnumber; 0;
```

where the second sends two bytes at a time by use of ";". To understand, an example might help — Mode 5

```
1000 COLOUR 131 : CLS : REM SET THE BACKGROUND TO
WHITE (LOGICAL 3)
1010 REPEAT COLOUR 0 : PRINT "********************" :
REM PRINT IN FIRST LOGICAL COLOUR
1020 COLOUR I : PRINT "********************" : REM AND
SECOND
1030 COLOUR 2 :PRINT "********************" : REM AND
THIRD
1040 INPUT LGICAL, ACTUAL : REM COLOUR NUMBERS
1050 VDU 19,LGICAL,ACTUAL;0; : REM WATCH THE
OUTPUT CHANGE COLOUR
1060 UNTIL FALSE : REM AND AGAIN
```

This little program helps you to investigate the effects of various actual colours, in differing combinations. An important consideration with colour graphics is how well the different colours can be distinguished on a monochrome television. (My colour TV is used for watching TV, most of the time).

By dint of much playing around with the above program, I came to the conclusion that I would use the following logical and actual associations

```
VDU 19, 0, 7; 0;
VDU 19, 1, 1; 0;
VDU 19, 2, 6; 0;
VDU 19, 3, 3; 0;
```

that is:

Figure 3.2 New colour associations

LOGICAL NUMBERS		ACTUAL NUMBER	COLOUR
0	128	7	White
1	129	1	Red
2	130	6	Cyan
3	131	3	Yellow

I propose to use 131 for the graphics background (ie yellow), and £29 for the text background (ie red); 0, 1, and 3, for graphics foreground (the three turtles), and 0 (ie white) for the text. The means to change these assignments are provided — for graphics — within the program.

The three turtles

Within the programs for the first version of turtle graphics, there were four global variables: X, Y, ANGLE and PEN. Though some of the other routines have to be slightly altered to cope with mode 5 rather than mode 4 (eg PROC_CLS) not much else needs great alteration — as long as it is remembered which turtle is which.

The way I propose to code the routines is to have only one set of routines, without any extra parameters for whichever turtle is to be used. The reason I intend to implement the graphics in this way is universality: all I want to say is PROC_CIRCLE(SIZE) for the routine to work with whichever turtle is being used at that time.

To have routines such as PROC_CIRCLE(SIZE, T_NUMBER) makes the whole programming process far more tedious than needs be. There will be special procedures to define the turtle to be used; and once so defined, the definition will stay until another turtle is defined. We will, however, within the basic TG routines have to keep track of where each turtle is located, and in which direction it faces.

We need at least one extra global variable: TURTLE, which holds the number of the turtle being used at that time (0, I, or 2). Instead of PEN we need three PENs, one for each turtle, just as we need three Xs, three Ys, and three ANGLEs. This all sounds like the place to use arrays, and it is for X, Y , and ANGLE — but not for PEN. The difference comes from the fact that PEN is a small integer less than 255 (it can fit in a byte), whereas X, Y , and ANGLE can be fractional numbers.

We need to have three examples of each of the above, where PEN is a set of three small integers, and the others are sets of three fractional numbers. We set aside space for these sets of three by a BB statement such as

```
1000 DIM PEN 2, X(2), Y(2), ANGLE(2)
```

and we refer to each element in this manner:

Figure 3.3 TURTLE global variables

TURTLE	PEN	X	Y	ANGLE
0	PEN?0	X(0)	Y(0)	ANGLE(0)
1	PEN?1	X(1)	Y(1)	ANGLE(1)
2	PEN?0	X(2)	Y(2)	ANGLE(2)

Note that PEN, which was ‘dimensioned’ differently to the others, and is different to the others in the way in which elements are named. The I’th element of PEN is shown by PEN? I, whereas the I’th element of ANGLE is shown by ANGLE(I). PEN is a ‘byte vector’ (see UG pages 237, 409-413), whereas ANGLE is an ‘array’ (see UG pages 120— 125, 236): PEN is effectively a succession of bytes, and ANGLE is a succession of real numbers.

If we are using a turtle called TURTLE, (where TURTLE is the logical number) then to use the correct actual colour for that wee beastie we say PEN?TURTLE; the turtle is at coordinates X(TURTLE), Y (TURTLE) facing in a direction ANGLECTURTLE). TURTLE always denotes the logical colour number, which is why 3 and 131 refer to the background logical colour (TURTLE lies between 0 and 2).

The new routines

The routines in Version 2.1 are similar but different. For a start, as arrays and vectors have to be dimensioned, we start (using PROC_START) by running the short program part. We could have started Versions 1.1 and 1.2 in this manner, but with arrays and vectors this is the cleanest way. Here goes.

```
 1000REM-------------------------------
 1010
 1020
 1030REM   G R A P H I C   ART
 1040
 1050REM   (c) Boris Allen, 1983
 1060
 1070
 1080REM-------------------------------
 1090
 1100REM   Turtle Graphics : 2.1
 1110
 1120REM-------------------------------
 1130
 1140 REM MAIN PROGRAM
 1150 DIM PEN 2, CLR 2, X(2), Y(2), ANGL
E(2)
 1160 PEN?0 = 7 : PEN?1 = 1 : PEN?2 = 6
: BACK = 3
```

```
 1170 FOR I = 0 TO 2
 1180 VDU 19, I, PEN?I; 0; : CLR?I = PEN
?I
 1190 X(I) = 0 : Y(I) = 0 : ANGLE(I) = 0
 1200 NEXT I
 1210 VDU 19, 3, BACK; 0;
 1220 PROC_TURTLE(0) : PROC_CLRSCR
 1230 END : REM OF MAIN PROGRAM
 1240
 1250 DEF PROC_CLRSCR
 1260 PROC_CLS : PROC_CLG
 1270 ENDPROC : REM CLRSCR
 1280
 1290 DEF PROC_CLG
 1300 GCOL 0, TURTLE : GCOL 0, 128+BACK
 1310 VDU 24, 0; 128; 1279; 1023;
 1320 VDU 29, 640; 566; : CLG
 1330 MOVE 0,0
 1340 ENDPROC : REM PROC_CLG
 1350
 1360 DEF PROC_CLS
 1370 COLOUR 0 : COLOUR 129
 1380 VDU 28, 0, 31, 19, 28 : CLS
 1390 REM It is VDU 28, 0, 31, 39, 28 :
CLS for mode 1
 1400 ENDPROC : REM PROC_CLS
 1410
 1420 DEF PROC_TURTLE(LGCL)
 1430 TURTLE = (LGCL MOD 3 + 3) MOD 3 :
GCOL 0,TURTLE
 1440 MOVE X(TURTLE),Y(TURTLE)
 1450 ENDPROC : REM TURTLE
 1460
 1470 DEF PROC_START
 1480 RUN
 1490 ENDPROC : REM START
 1500
 1510 DEF PROC_COL(ACT)
 1520 VDU 19, TURTLE, ACT; 0;
 1530 ENDPROC : REM COL
 1540
 1550 DEF PROC_CENTRE
 1560 MOVE 0,0 : ANGLE(TURTLE) = 0
 1570 X(TURTLE) = 0 : Y(TURTLE) = 0
 1580 ENDPROC : REM CENTRE
 1590
 1600 DEF PROC_RESTART
 1610 LOCAL I
 1620 FOR I=2 TO 0 STEP -1
 1630 PROC_TURTLE(I) : PROC_CENTRE
 1640 NEXT I : PROC_CLG
 1650 ENDPROC : REM RESTART
 1660
 1670 DEF PROC_INVERT
 1680 IF PEN?TURTLE = CLR?TURTLE THEN PE
N?TURTLE=BACK ELSE PEN?TURTLE=CLR?TURTLE
 1690 PROC_COL(PEN?TURTLE)
 1700 ENDPROC : REM INVERT
 1710
 1720 DEF PROC_TURNTO(A)
 1730 ANGLE(TURTLE) = FN_ANGLE(A)
 1740 ENDPROC : REM TURNTO
 1750
 1760 DEF PROC_LOC
 1770 PRINT "TURTLE "TURTLE
 1780 PRINT "COORDS ";INT(X(TURTLE)+.5);
" ";INT(Y(TURTLE)+.5)
 1790 PRINT "ANGLE "INT(ANGLE(TURTLE)+.5
)
 1800 ENDPROC : REM LOC
 1810
 1820 DEF PROC_TURN(A)
 1830 ANGLE(TURTLE) = FN_ANGLE(ANGLE(TUR
TLE)+A)
 1840 ENDPROC : REM TURN
 1850
 1860 DEF PROC_MOVE(DISTANCE,STYLE)
 1870 X(TURTLE) = X(TURTLE) - DISTANCE*S
IN(RAD(ANGLE(TURTLE)))
 1880 Y(TURTLE) = Y(TURTLE) - DISTANCE*C
OS(RAD(ANGLE(TURTLE)))
 1890 IF STYLE=0 THEN MOVE (X(TURTLE),Y(
TURTLE) ELSE PLOT (STYLE-1)+5, X(TURTLE)
,Y(TURTLE)
 1900 ENDPROC : REM MOVE
 1910
 1920 DEF PROC_MOVETO(XN,YN,STYLE)
 1930 LOCAL XDIF,YDIF : XDIF = XN-X(TURT
LE) : YDIF = Y(TURTLE)-YN
 1940 IF YDIF<0 THEN PROC_TURNTO(DEG(ATN
(XDIF/YDIF))+180*(YN<Y(TURTLE))) ELSE PR
OC_TURNTO(SGN(-XDIF)*90)
 1950 X(TURTLE) = XN : Y(TURTLE) = YN
 1960 IF STYLE=0 THEN MOVE X(TURTLE),Y(T
URTLE) ELSE PLOT (STYLE-1)+5, X(TURTLE),
Y(TURTLE)
 1970 ENDPROC : REM MOVETO
 1980
 1990 DEF FN_ANGLE(A)
 2000 IF A MOD 360 <0 THEN =A MOD 360 +
360 ELSE = A MOD 360 : REM ANGLE
```

PROC_START (ie main program) dimensions two vectors (ie PEN and CLR), and three arrays (ie X, Y, and ANGLE): all have three elements (ie 0, 1, and 2). PEN?() (corresponds to first turtle) is set to actual colour number 7 (white); the second turtle is set to colour I (red), and the third turtle is set to colour 6 (cyan); and the BACKground colour is set to 3 (yellow).

The actual colours are assigned to the logical numbers by use of the VDU 19 command; the original CoLouR is remembered (used in PROC_INVERT); and the coordinates and angle are initialised to zero for all turtles. The background colour is set by the VDU 19, 3, BACK; 0; command; the initial turtle is set (by PROC_TURTLE) to be turtle 0, and the screen is set up by PROC_CLRSCR.

PROC_CLRSCR merely calls the two routines PROC_CLS and PROC_CLG.

PROC_CLG first sets the current logical colour to the operative turtle number, and sets the background to the operative background number (altering the value of BACK can produce some interesting effects). The rest of the routine matches that of the monochrome version.

PROC_CLS sets colours for text and background (white on red) and sets up a text window for mode 5 : if the VDU 28 command is kept the same as that in the first version, it is possible to use mode 1.

Next in the (boring?) list is PROC—TURTLE, which takes as parameter the new LoGiCaL number for the graphics, and makes that the new TURTLE number. The equation with MOD is to account for negative values — FN_ANGLE uses a different method — in case by chance out of bounds numbers are given. This routine is the one used to change from one turtle to another, and so the graphics cursor is moved to the last coordinates for that turtle (ie X(TURTLE), Y(TURTLE).

```
 3000REM----------------------------
 3010
 3020
 3030REM   G R A P H I C   ART
 3040
 3050REM   (c) Boris Allen, 1983
 3060
 3070
 3080REM----------------------------
 3090
 3100REM   Turtle Routines : 2.1
 3110
 3120REM----------------------------
 3130
 3140 DEF PROC_SQTURN(F)
 3150 LOCAL I,A$
 3160 FOR I=0TO2 : PROC_TURTLE(I): PROC_
TURNTO(120*I):NEXT I
 3170 REPEAT : PROC_TURTLE(I MOD 3)
 3180 PROC_TURN(90) : PROC_MOVE(F*I,0)
 3190 PROC_SQUARE(F*I) : I = I+1 : UNTIL
 INKEY(-68)
 3200 *FX15,0
 3210 ENDPROC : REM SQTURN
 3220
 3230 DEF PROC_SQUARE(SIDE)
 3240 LOCAL I : FOR I = 1 TO 4
 3250 PROC_MOVE(SIDE,1) : PROC_TURN(90)
 3260 NEXT I
 3270 ENDPROC : REM SQUARE
 3280
 3290 DEF PROC_MOIRE
 3300 LOCAL I
 3310 FOR I=0 TO 1079 : PROC_TURTLE(RND(
3)-1) : PROC_CENTRE : PROC_TURNTO(I/3)
 3320 PROC_MOVE(400,1) : NEXT I
 3330 ENDPROC : REM MOIRE
 3340
 3350 DEF PROC_INITIALIZE
 3360 REMDIM D(2)
 3370 PROC_TURTLE(0): PROC_MOVETO(-32,-3
68,0) : PROC_TURNTO(0)
 3380 PROC_TURTLE(1): PROC_MOVETO(32,-36
8,0) : PROC_TURNTO(0)
 3390 D(0) = 10 : D(1) = 10 : VDU 7
 3400 ENDPROC : REM INITIALIZE
 3410
 3420 DEF PROC_DEVIATION
 3430 PROC_TURTLE(1): PROC_TURN(90*INKEY
(-89)-(INKEY(-88)))
 3440 PROC_TURTLE(0) : PROC_TURN(90*INKE
Y(-51)-(INKEY(-66)))
 3450 ENDPROC : REM DEVIATION
 3460
 3470 DEF PROC_ACCELN
 3480 D(0) = D(0) - 18*INKEY(-82) : D(1)
 = D(1) - 18*INKEY(-73)
 3490 ENDPROC : REM ACCELN
 3500
 3510 DEF PROC_TRAVEL
 3520 LOCAL I%
 3530 PROC_DEVIATION : PROC_ACCELN
 3540 I% = RND(1) +.5
 3550 PROC_GO(I%) : PROC_GO(1-I%)
 3560 ENDPROC : REM TRAVEL
 3570
 3580 DEF PROC_GO(I)
 3590 LOCAL J : FOR J = 1 TO 8
 3600 PROC_DEVIATION
 3610 PROC_TURTLE(I)
 3620 PROC_MOVE(D(I),1)
 3630 PROC_MOVE(4,0):IF POINT(X(TURTLE),
Y(TURTLE))<>3 THEN PROC_END
 3640 PROC_MOVE(-4,0) : REM Note that it
 is -4 for mode 1, -8 for mode 5
 3650 PROC_TURTLE(1-I)
 3660 PROC_MOVE(D(1-I),1)
 3670 PROC_MOVE(4,0):IF POINT(X(TURTLE),
Y(TURTLE))<>3 THEN PROC_END
 3680 PROC_MOVE(-4,0) : NEXT J : REM See
 remark above
 3690 ENDPROC : REM GO
 3700
 3710 DEF PROC_END
 3720 VDU 7 : VDU 7
 3730 PRINT "TURTLE ";TURTLE;" LOSES ":
VDU 7
 3740 FOR I = 1 TO 500 : NEXT I : *FX15,
0
 3750 END
 3760
 3770 DEF PROC_NORT
 3780 LOCAL A$ : A$ = GET$
 3790 PROC_INITIALIZE
 3800 REPEAT PROC_TRAVEL
 3810 UNTIL FALSE
 3820 ENDPROC : REM NORT
```

Some highly interesting results can be achieved by changing the actual colours of the turtles, and PROC_COL is the means by which the change is made — see Figure 3.4 — on this, more later.

PROC_CENTRE centres the current turtle (it does not affect the other two turtles).

PROC_RESTART centres all turtles and clears graphics by PROC_CLG: a routine without PROC_CLG would centre all turtles without affecting the display.

The operation of PROC_INVERT is rather different from the routine of the same name in the monochrome version. Whereas the monochrome version changes the plotting mode from foreground to background colour (or vice versa), this routine either hides all the plotting of the turtle — thus far — or makes it all reappear again.

If the present actual colour of the PEN is the same as the original colour (CLR?TURTLE) then the new actual colour becomes that of the background, or else the colour of the PEN becomes the same as the original colour. This is a rather useful way of hiding a piece of plotting, for the plotting to suddenly appear, as if by magic.

PROC_TURNTO is the first of the TG routines proper (the rest being housekeeping, more or less). All it does is make the present angle for the present turtle equal to the parameter of the procedure (within 0 to 359). PROC_TURN is as simple in operation.

PROC_LOC has had to be modified to fit on the lesser space of mode 5: the use of INT is also to reduce space. Note the addition of the information which tells which is the operative turtle.

The main difference in PROC_MOVE (apart from distinguishing between turtles), is to do with the STYLE of plotting. If the style is 0 then a move does not plot; if 1 then plot in the normal turtle colour; and if the style is 3 then plot in the background colour (ie erase, or the old version of PROC_INVERT). The different use of STYLE is again the only real change to PROC_MOVETO. FN_ANGLE is unchanged.

Finally here is the list of actual colour numbers, and the colours to go with the number.

Figure 3.4 Actual Colours and Numbers

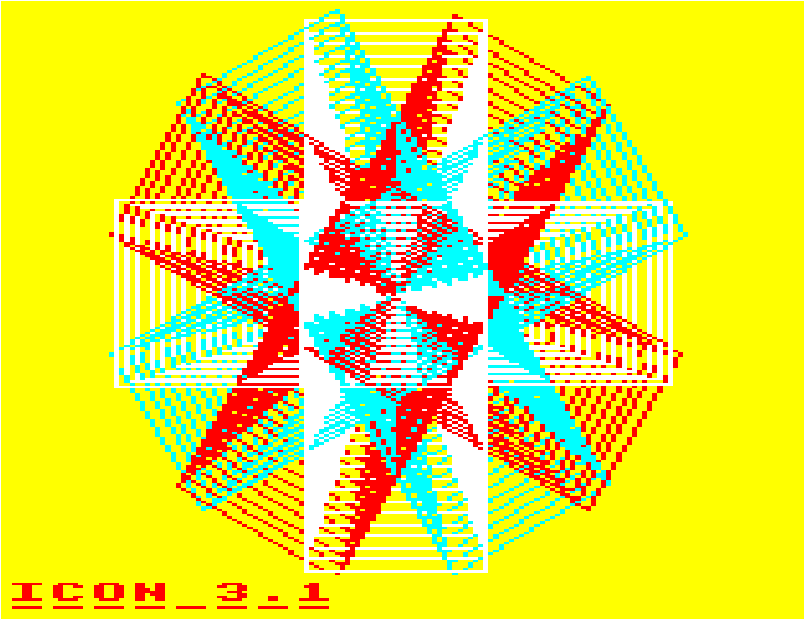
ACTUAL NUMBER	COLOUR
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Flashing BIack/White
9	Flashing Red/Cyan
10	Flashing Green/Magenta
11	Flashing Yellow/B1ue
12	Flashing BIue/Yellow
13	Flashing Magenta/Green
14	Flashing Cyan/Red
15	Flashing White/Black

Example pictures

Before the game is designed, here are two different examples of coloured TG.

Though the name is the same as a Version I routine, PROC_SQTURN is different. For a start PROC_SQTURN in the coloured version has a parameter (F), but first examine the routine. Each turtle is turned to a different angle, turtle0 to 0 degrees, 1 to 120 degrees, and 2 to 240 degrees (the first loop).

For a large number of times (1800) a turtle is chosen (the loop index MOD 3), this turtle is turned through 90 degrees from where it was previously, it is moved through a distance F*I (without plotting), and then a square of side F*I is drawn — by use of PROC_SQUARE. PROC_SQUARE is a simple routine to draw a square. An example is shown in **Icon 3.1**, but it does not do justice to the multi-coloured effects.



This routine produced multi-coloured effects, of a varying nature, depending upon the value of F . Further effects can be investigated by using the combination

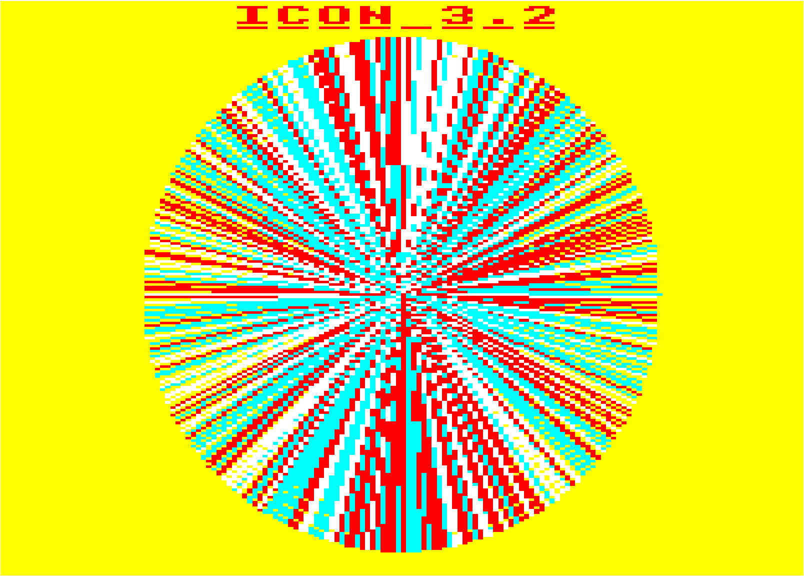
PROC_TURTLE(A_VALUE) :
PROC_COL(ANOTHER_VALUE)

The various values of ANOTHER_VALUE can be chosen with the assistance of **Figure 3.4**.

The other example is a circular Moire demonstration: PROC_MOIRE. In the loop (which is activated 3*360 times), a TURTLE is chosen at random; the turtle is centred; it turns to 1/3 degrees (I is the loop counter); and the turtle moves 400 units forward. A multi-coloured circle is drawn. Modifying the PROC-JFURTLE parameter, eg

PROC_TURTLE((I DIV 3) MOD 3)

(though the MOD 3 is not necessary), produces various intriguing effects. As with the PROC_SQTURN, using PROC_COL can produce startling effects. The effects, as shown in **Icon 3.2,** are far better in colour than in monochrome — my screen dump routine only works in black and white.



Both routines show how easy it is to program with three coloured turtles.

The game

One of the less important reasons for producing multi-coloured graphics is the design of video games — the most important reason is the pure delight of experimenting with artistic effects. So here is a game.

The game is called NORT, and involves only two of the three turtles. It is a turtle race, but remember these turtles are cybernetic beasties and so can move pretty speedily. The trail that each turtle leaves is noxious, and so to travel over your own or your opponent's trail is calamitous. You lose.

The two turtles start out side by side, at the same speed: the left turtle is controlled by the 'A' , 'S' , 'D' ,keys; and the right turtle is controlled by the ';' , ':' , ']' , keys. In each case the left key turns the turtle through 90 degrees to the left, the fight key turns the turtle through 90 degrees to the fight, and the middle key speeds up the turtle. Once a turtle has speeded up then it stays at that speed (it does not slow down), though it is always possible to increase the speed further.

The faster the turtle goes, the easier it is to cut across the other turtle, but the easier it is to get out of control. As you will realise, this game is extremely simple to implement using coloured TG.

I do not claim that this is a definitive game, and it can stand much improvement, but it indicates what can be done. An improved method of reading the keyboard, and controlling the turtle racers would greatly help. To the routines.

The first routine, in order, is PROC_INITIALIZE: the distances the turtles have to move are stored in the array D which has two elements, D(0) and D(1). The first turtle (ie 0) is moved to its starting point at -2, -368, and turned to face upwards. The second turtle (ie 1) is moved to 32, -368, facing upwards, and the initial speeds are set at 12 (ie D(0) and D(1)). There is a starting beep (VDU 7).

PROC_DEVIATION senses the keyboard for both turtles, and turns accordingly (uses PROC_TURN and INKEY()).

Another short routine is PROC_ACCELN, which modifies the two speeds (only upwards) by sensing the keyboard via INKEY().

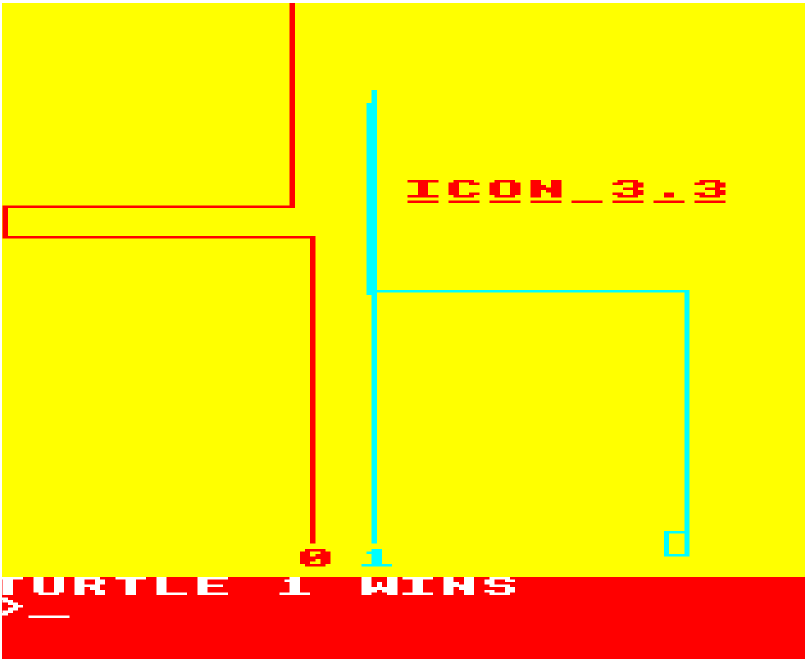
PROC_TRAVEL has a local variable I%, and establishes the deviation and speed before setting I% randomly to 0 or 1. The routine PROC_GO is called with I% as parameter, and this establishes the order in which the two turtles are moved.

The turtles move in eight segments — so sometimes it is possible to 'jump' over a trail. These eightjumps are produced by the loop J = 1 TO 8, and at each jump the keyboard is sensed, and then the turtles jumped in order — given by the parameter to the routine. After each jump, there is a move forward of four units (without plotting), and that location is checked to see if the colour there is not logical 3 (the background). If that square is the wrong colour, then the game ends. If the colour is that of the background, then there is a move (back) of -4 units.

PROC_END double beeps (VDU 7), tells you who has lost, waits and then flushes the buffers (*FX15,0).

To start the game you have to enter RUN and then PROC_NORT: the game commences with the press of a key (GET$), when we PROC_INITIALIZE and then REPEAT PROC_TRAVEL forever — or at least until we hit PROC_END.

The two trails shown in **Icon 3.3** give some idea of the courses of a typical game. The game is more effective in colour than in black and white, though quite possible to follow because of the colours chosen.



```
TURTLE 1 WINS
>_
```

CHAPTER 4
Driving Graphics

Nil posse creari de nilo
[Nothing can be created from nothing]

Lucrecius, *Be rerun Natura*

We have already encountered the control of graphics by use of VDU commands (the VDU ‘drivers’) — it is impossible to ignore them. What is the function of a VDU command? Hold down the CTRL key and press the G key: you beep. Enter

```
PRINT CHR$(7);
```

and you beep, appearing on the next line. When you enter

```
PRINT CHR$(7)
```

(ie without the semicolon) you beep and appear on the next line but one. To enter

```
VDU 7
```

again produces a beep, this time you are on the next fine. Entering

```
VDU 7:VDU 7
```

produces two beeps, as does

```
VDU 7,7
```

This simple example shows that the VDU command is a special kind of PRINT command — it is a PRINT CHR$ command, where the command is terminated by a semicolon.

Asking about ASCII

The inquiring mind might then ask: ‘Why is the PRINT CHR$ combination worth all this trouble?’. As you ask, it's ASCII, the American Standard Code for Information Interchange. Go into any mode other than 7, turn to the UG page 490, and then

```
VDU 66,111,114,105,115
```

to which the response is, how I love it,

```
Boris
```

and to reajly get it correct

```
VDU 66,1 11,1 14,105,105,1 15,32,65,108,108,97,110
```

- try it out to see what is produced, gorgeous really.

The numbers which follow a VDU command are the numbers which correspond to the characters in the ASCII. The command to produce Boris could be written

```
VDU 66 : VDU 111 : VDU 114 : VDU 105 : VDU 115
```

that is, as a series of commands — each with one number following — but it is simpler to run the numbers together if at all possible. If you look at the codes on page 490 of the UG, you will see that from ASCII 0 to ASCII 31 there are no characters. These are the control characters, which include the beep at 7 (the diagram says "beep").

Now turn to page 378 of the UG, where there is a summary of the meaning of the VDU codes from 0 to 31, plus code 127, There is an exact correspondence between the two summaries (ie the VDU on page 378, and the ASCII on page 490).

The original ASCII codes only extended from 0 to 127, as there were only seven holes on paper tapes plus one hole as a 'parity' check (ie count up the holes and punch an extra hole depending on whether the result was odd or even). This is why the meaning of codes 128 to 255 is "undersigned initially" as the UG says. Codes 0 to 31 were given names, and suggestions for standard meanings, but these codes were 'control' codes and partly depended on the system.

If you refer to the VDU code summary on page 378 of the U G, you will see a third column which is labelled CTRL: the keys in this column — when pressed at the same time as the CTRL — produce the ASCII control code given as the ASCII abbreviation (column four). The ASCII code 0 (for example) is abbreviated to NUL, and means 'null' do nothing; according to the last column of the figure on page 490, for the BBC computer it means 'does nothing'.

Looking at the ASCII abbreviations can explain some strange pairings. To disable the VDU is code 21 (CTRL U) which is abbreviated to NAK (Negative Acknowledge); whereas to activate the VDU one uses code 6 (CTRL F), or ACK (Acknowledge). Each computer has its own conventions for some codes but usually code 13 (&0D) is return and code 10 (&0A) is line feed, and so on. Otherwise the computer would have trouble talking to devices in languages they could understand.

Character generation

We have been talking about characters, and the characters from 128 to 255 are undefined in the figure on page 491 of the UG. However, unless a special operating system (OS) command is given — of which more later — only characters with codes 224 and 255 can be redesigned. To store each user-designed shape takes up 8 bytes, and so to store 128 shapes is 8* 128 bytes or 1K — to redesign the 32 shapes from 224 to 255 takes only 256 bytes.

Memory is again a problem.

The first thing we now have to find out is how to define a character, and details are given in the UG on pages 170-171, 384-385, and 389. The VDU command to design a character is 23 (&17), which is CTRL W (ETB — End transmission block). On page 378 of the UG we discover that, once the system has been alerted by VDU 23, it expects 9 bytes to follow: the first byte gives the number of the character to be redesigned; and tht$remaining 8 bytes described what is the shape of the new character.

The work comes in working out the description of the shape, as Lucrecius said ‘Yer can’t get owt from nowt’. The routines I have developed are designed to make the design the important element, with the description of the design being made as simple as possible. These are the **Character Routines 1.**

First: how to design a character. In **Figure 4.1**, I show the ubiquitous man from the UG (page 170)

Figure 4.1 The Little Man

THE SHAPE	THE BINARY	THE DECIMAL
. . . *** . .	00011100	28
. . . *** . .	00011100	28
. . . . * . . .	00001000	8
. *******	01111111	127
. . . . * . . .	00001000	8
. . . * . *. .	00010100	20
. . * . . . *.	00100010	34
. *. . . . . *	01000001	65

The numbers in the decimal column are the equivalents of the corresponding binary numbers, with the binary numbers being exactly related to the filled and empty squares in the character design.

The character number 224 is designed to be a ‘man’ shaped by the VDU command

```
VDU 23,240,28,28,8,127 ,8,20,34,65
```

which can be re-written in a more transparent form as

```
VDU 23 : REM Prepare to send a character definition
VDU 240 : REM and the character number is ASCII 240
VDU 28 : REM This is the top line defined
VDU 28,8,127 ,8,20,34 : REM the middle six
VDU 65 : REM and the last line
```

If you think back to the discussion of screen resolution, the 8*8 character frame is no newcomer. *Note*: as long as the VDU numbers are in the correct order, then the bytes can be split over more than one VDU statement. The most tedious part (and least artistic) is the conversion from binary numbers to decimal equivalents.

Character designer

The first of the Character Routines is PROC.—INB YTES, and this routine allows the user speedy and easy input of binary numbers to be converted into decimal equivalents. These converted numbers are then made available for other routines, to be assigned to character numbers, and to be manipulated.

The PROC_INBYTES routine uses two highly useful BB facilities (both related): byte vectors, and string indirection (UG pages 409-413). The local variable BYTES is set equal to &D00 (the start of a patch of memory available for user supplied resident routines, UG page 501), and the local variable STORE is set equal to 8 more than BYTES. BYTES will be used to hold the values of 8 bytes, and STORE will be used to store 8 characters which will be then converted into byte values.

There are two loops: one — with index I — corresponds to the fines in the character design; and the other — with index J — builds up the BYTES value for line I. Consider (Figure 4.2) what happens for line 5, once we have input the binary number 00001000 into the string $STORE which starts eight bytes on from the start of BYTES.

We now have to relate this arrangement to the routine PROC—INBYTES. When the user types in 00001000, after the request LINE 4?, the eight characters are stored in the string starting at &0D08 (ie by $STORE). $STORE is not the same as STORE$.

STORE$ is a string variable, and $STORE points to a series oflocations starting at STORE, where the locations are filled with the ASCII codes of the characters. We know where $STORE ends because that location contains 13 (&0D). The ASCII code for 0 is 48, and the ASCII code for 1 is 49 (see page 490 of the UG), and so if the values of the locations from &0D08 to &ODOF are examined (48,48,48,48,49,48,48,48) then the correspondence with the input line 00001000 is clear to see. Note that location &0D10 contains 13, to indicate the end of the string.

Thus far we will presume that the correct values have been input to the first four locations of BYTES, and now the next problem is to input the correct value into the fifth location (BYTES?4). BYTES?I is first set to zero (note that I = 4), and in the J loop the value of BYTES?I is doubled. As the loop counter J is increased from 0 to 7 a check is made to see if that element of STORE (ie STOREAJ) is equal to the ASCII value of I (ie STORE?J = 49). If there is a I in the input number at this place in the order, 1 is added to BYTES?I.

Figure 4.2 Indirection Example

LOCATION		NAME	VALUE
&0D00	3584	BYTES?0	28
&0D01	3585	BYTES?1	28
&0D02	3586	BYTES?2	8
&0D03	3587	BYTES?3	127
&0D04	3588	BYTES?4	UNDEFINED
&0D05	3589	BYTES?5	UNDEFINED
&0D06	3590	BYTES?6	UNDEFINED
&0D07	3591	BYTES?7	UNDEFINED
&0D08	3592	STORE?0	48
&0D09	3593	STORE?1	48
&0D0A	3594	STORE?2	48
&0D0B	3595	STORE?3	48
&0D0C	3596	STORE?4	49
&0D0D	3597	STORE?5	48
&0D0E	3598	STORE?6	48
&0D0F	3599	STORE?7	48
&0D10	3600	STORE?8	13 (String end)

A binary number is formed in BYTES? I, and at the end of the routine the numbers needed to send the shape to a character code are in the eight bytes from &0D00.

Numbers to characters

To set up a byte vector of eight elements all that is needed is a

```
DIM VV 7
```

statement in direct/immediate mode. This declaration will remain operative until the computer resets the BASIC variables (eg some errors, or a new fine is entered into a program).

If we want to assign the character definition stored away at &0D00, then PROC_CHRASN is used. The parameters of the routine are the character number (I) and an eight element byte vector (NEWB) — this byte vector is used as a temporary storage medium for the design numbers (used in some later routines.) All the routine does is simply send the numbers stored from &0D00 to the character I, by

```
VDU 23,I
```

and then eight

```
VDU BYTES?I
```

statements (storing in NEWB?I at the same time).

So we design the shape, and then input the binary numbers (by use of PROC_INBYTES): next we decide where to send the design, ie to which character, by use of PROC_CHRASN. Suppose we turn the design upside down, making line 0 into line 7? (but still keeping the left column on the left). We use PROC_CHREV.

PROC_REV is essentially PROC_ASN, but backwards. It starts with BYTES?7 and goes to BYTES?0, instead of vice versa. The new reversed design is stored in the byte vector NEWB.

Next there is a routine to flip from left to right (and right to left): PROC_CHRFLIP. This routine follows the standard pattern, but some points are of special interest. J and L% are loop counters, J for the line, and L% for the column within the line (and bit within the binary number). L% is an integer variable because FL% then gives an exact result. The new number for the line is compiled in K, in the same manner as the number was compiled from STORE in PROC_INBYTES. The IF statement is replaced by a logical assignment in PROC_CHRFLIP.

The logical assignment

K = K - ((2^L% AND BYTES?J) = 2^L%)

works to see if there is a I in the binary number BYTES? J in L%th position from the right, taking into account the binary number 2^L% will have a 1 in the L%th bit from the right.

If the number in BYTES?J has a 1 in L%th position, then (2^FL% AND BYTES?J) — the logical AND, UG pages 205-206 — will be a binary number with a bit equal to 1 in L%th position (with zeros elsewhere). A number with a 1 in L%th position and zeros elsewhere is equal to 2^L%. jfhe numerical value which results when the logical comparisons are TRUE is -1, and it is 0 when FALSE (UG 257-258, 369). So the value is subtracted to make -1 into +1.

Two examples using the number 28 (ie BYTES?J) and the position 0 and 3 (ie L%=0 and L%=3). First example: L% = 0 so that 2^L% is 1 (or 00000001 in binary), BYTES?J = 28 (ie 00011100 in binary); 2^L% AND BYTES?J is 00000000, which does not equal 2^L%. Second example: when L% = 3 then 2^% = 8 which is 00001000 in binary; 2^L% AND BYTES%} is 00001000 which is equal to 2^L%.

PROC_CHROT takes it all that bit further, it turns upside down and flips, that is, it rotates. The routine is like PROC_CHREV with PROC_CHRFLIP inside.

We have with all these routines the spare byte vector (usually called parameter NEWB). PROC_CHRSEND sends the content of a byte vector to the eight locations from &0D00. This allows more than one design to be examined and manipulated at any one time. The bytes from &0D00 are almost the equivalent of the accumulator in a microprocessor. The accumulator is where most of the actual computations are performed — especially with the BBC computer’s 6502 microprocessor.

The routine PROC_CHRINV inverts the design, ie all zeros become ones while all ones become zeros. If the result of the inversion is then sent back to &0D00, all the manipulation of the design is then available for the inverse design. Finally, PROC—INDEC is just like PROC_INBYTES, only you can enter decimal numbers.

Saving the routines

These routines are quite useful, and so it is sensible to try to incorporate them in any program which uses user-designed shapes. You may have noticed that the line numbers for the routines are very high, higher than any program will ever need: this is so that they can be loaded quite safely with any normal program.

If, however, the routines are saved as a BB program, then to load them by, say,

```
LOAD "SHAPES"
```

will erase the program already in the memory. We need to load the program fines in some different way. As we could enter the lines by typing them in, why do we not get the computer to type them in?

Type NEW and then enter the routines, checking to see they work — and how do you check to see they work? You use a VDU command. To see if the design for character 224 is what you want, you enter

```
VDU 224
```

which prints out the character corresponding to the ASCII value 224. LIST to check that all is well.

The next stage is to send the contents of the program to a file: but not as a program file. When we type in at the keyboard we enter a character at a time, or do we? We actually send a value when the key is depressed, that value is then interpreted to appear as a character on the screen, or elsewhere. The value sent by the key depression is an ASCII value — to read in ASCII values, therefore, does not clear out the BASIC program. We seem to require a file which has the program stored as ASCII values — sometimes called an ASCII file.

To send the program to an ASCII file we use *SPOOL (U G page 402-403). To save the ASCII version of the routines we enter

```
*SPOOL "SHAPES"
```

and switch on the tape recorder to record, though with disks we do not have to bother.

```
LIST
```

which lists out the program, and sends it in ASCII form to the file SHAPES. To terminate we enter

```
*SPOOL
```

and the file closes.

To load the routines on top of a program we use

```
*EXEC "SHAPES"
```

and the *EXEC command acts as if the information coming from the file (in ASCII form) were being typed in at the keyboard. This means that it is possible to use these routines in any program. It would be even simpler still to have one or more sets of designed characters which one could pick and choose, depending on the program.

Saving the designs

The user designed shapes are located between &0D00 and &()DFF unless the characters are 'exploded', (U G page 427). Exploding means that more than the 32 characters (ASCII codes 224 to 255) are available to be redesigned: the ASCII codes from 32 to 255 (ie &20 to &FF) can be redesigned. The exploding of the memory allocation uses the command * FX 20,1, but the allocation of memory over-runs the beginning of BASIC (usually at &0E00, ie PAGE — UG page 414).

The sequence to explode the memory for a full set of user designed characters is (all in immediate mode)

```
PRINT PAGE : PAGE = PAGE + &600 : *FX 20,1
```

This explosion must occur before the BASIC program is entered. The designs occupy memory from locations &()C00 to &CFF, and &()E00 to &0A00 if all characters are to be redesigned. The gap at &0D00 to &0DFF (which I used for temporary storage) is left for user-defined routines (see UG page 501 — though this conflicts with page 502). Personally I think that 32 characters are sufficient for most purposes, so I will only consider the default 'imploded' 32 characters.

To save the user-designed characters, therefore, enter

```
*SAVE "CHARS" 0C00 0CFF
```

and then to load these new designs into any program all that is needed is Chapter 4 Driving Graphics *LOAD "CHARS" The extension to explosion is simple enough, That the characters occupy fixed positions also means that the first location of any character is at (CHRLOCN - 224)*8 + &0C00. The character routines can easily be modified to take this arrangement of characters and locations into account.

Text in graphics

All Icons in this book are printer dumps taken directly from the screen, including the Icon label and any other labels (eg Hypotenuse). The labels were printed and positioned by a routine called PROC_PRINT, which comes in two versions. The first version

```
1000 DEF PROC_PRINT(A$,X,Y,CLR)
1010 VDU 5 : REM Write text at graphics cursor
1020 GCOL 0,CLR : REM Set text colour
1030 MOVE X,Y : REM Move to X,Y without plotting
1040 PRINT A$ : REM Write the text
1050 VDU 4 : REM Set text back to text cursor
1060 ENDPROC : REM PRINT Version 1
```

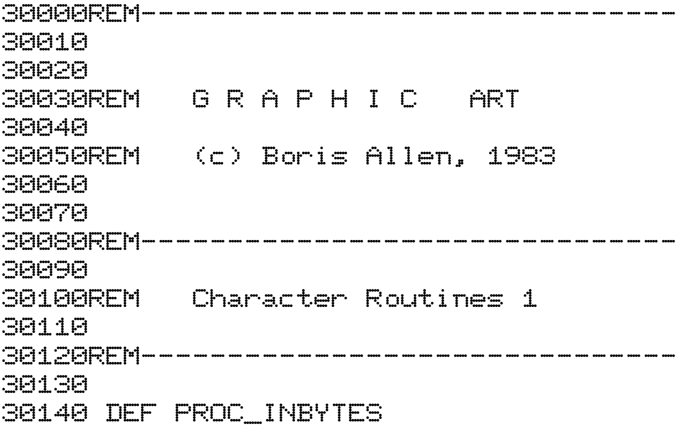
uses only two VDU commands. However, GCOL and MOVE can be expressed as VDU commands (refer to page 378 of the UG). The second version uses many more VDU commands, to the same effect.

```
1000 DEF PROC_PRINT(A$,X,Y,CLR)
1010 VDU 5,18,0,CLR, 25,0,X;Y;
1020 PRINT A$ : VDU4
1030 ENDPROC : REM PRINT Version 2
```

I am ambivalent about the second routine. The first routine is clear, explicit and well documented. The second routine is only clear to those who wish to find out, but it has a coherence — due to the use of the VDU commands as a long fine. The second routine is, I suspect, rather more computationally efficient.

Corresponding to GCOL there is VDU 18 (U Gpage 381-382) with the same parameters (ie 0 and CLR), both of which are of one byte (the comma ","). Move X,Y is the same as PLOT 4,X,Y and (page 386) this is the same as VDU 25 with the equivalent parameters. The first parameter is one byte (","), and the other two parameters are two bytes (";").

On balance, I prefer the second version.



```
30000REM-----------------------------
30010
30020
30030REM   G R A P H I C   ART
30040
30050REM   (c) Boris Allen, 1983
30060
30070
30080REM-----------------------------
30090
30100REM   Character Routines 1
30110
30120REM-----------------------------
30130
30140 DEF PROC_INBYTES
```

```
30150 LOCAL I,J,STORE,BYTES
30160 BYTES = D00 : STORE = BYTES + 8
30170 FOR I = 0 TO 7
30180 PRINT "LINE ";I;
30190 INPUT $STORE
30200 BYTES?I = 0 : FOR J = 0 TO 7
30210 BYTES?I = BYTES?I*2 : IF STORE?J=4
9 THEN BYTES?I = BYTES?I + 1
30220 NEXT J : NEXT I
30230 ENDPROC : REM INBYTES
30240
30250 DEF PROC_CHRASN(I,NEWB)
30260 LOCAL J,BYTES : BYTES = D00
30270 VDU 23, I : FOR J = 0 TO 7
30280 NEWB?J = BYTES?J : VDU BYTES?J : N
EXT J
30290 ENDPROC : CHRASN
30300
30310 DEF PROC_CHREV(I,NEWB)
30320 LOCAL J,BYTES : BYTES = D00
30330 VDU 23,I : FOR J = 7 TO 0 STEP -1
30340 NEWB?(7-J) = BYTES?J : VDU BYTES?J
 : NEXT J
30350 ENDPROC : REM CHREV
30360
30370 DEF PROC_CHRFLIP(I,NEWB)
30380 LOCAL J,K,L%,BYTES : BYTES = D00
30390 VDU 23,I : FOR J = 0 TO 7
30400 K = 0 : FOR L% = 0 TO 7
30410 K= K*2 : K = K - ((2^L% AND BYTES?
J) = 2^L%) : NEXT L%
30420 NEWB?J = K : VDU K : NEXT J
30430 ENDPROC : REM CHRFLIP
30440
30450 DEF PROC_CHRROT(I,NEWB)
30460 LOCAL J,K,L%,BYTES : BYTES = D00
30470 VDU 23,I : FOR J = 7 TO 0 STEP -1
30480 K = 0 : FOR L% = 0 TO 7
30490 K = K*2 : K = K - ((2^L% AND BYTES
?J) = 2^L%) : NEXT L%
30500 NEWB?J = K : VDU K : NEXT J
30510 ENDPROC : REM CHROT
30520
30530 DEF PROC_CHRSEND(NEWB)
30540 LOCAL I,BYTES : BYES = D00
30550 FOR I= T0 O 7
30560 BYTES?I = NEWB?I : NEXT I
30570 ENDPROC : REM CHRSEND
30580
```

```
30590 DEF PROC_CHRINV(I,NEWB)
30600 LOCAL J, BYTES : BYTES = D00
30610 VDU 23,I : FOR J = 0 TO 7
30620 NEWB?J = 255 - BYTES?J : VDU NEWB?
J : NEXT J
30630 ENDPROC : REM CHRINV
30640
30650 DEF PROC_INDEC
30660 LOCAL I, NUM,BYTES : BYTES = D00
30670 FOR I = 0 TO 7
30680 PRINT "LINE ";I;
30690 INPUT NUM : BYTES?I = NUM
30700 NEXT I
30710 ENDPROC : REM INDEC
```

CHAPTER 5
Drawing Charts and Graphs

> In our view the increasing availability
> of micro-computers and the visual
> display which they provide should also
> offer opportunities to illustrate
> statistical ideas and techniques;
>
> Mathematics counts, *(CockCroft Report, HMSO 1982)*

Turtle graphics provide a flexible means to draw many shapes. Sometimes, however, there are easier ways to draw charts and graphs, and this will be illustrated first by an example taken from elementary statistical theory.

The idea is this: when the heights of a large number of people are measured, and the heights divided into categories, the numbers in the categories approximate to what is known as the ‘normal’ distribution (a bell-shaped distribution).

Morris Kline writes (in Mathematics in Western Culture):

> What is especially significant about the distribution of heights as well as of many other characteristics . . . is that the curve approximates an ideal distribution known to mathematicians as the normal frequency curve. In fact, the larger the group whose heights are included the closer the curve comes to having the ideal shape, just as regular polygons with more and more sides approach the shape of a circle (page 391).

that is, a three sided-regular polygon (a triangle) does not look very much like a circle, but a thirty-sided regular polygon looks very like a circle. The distribution of heights for small numbers of people will not look very like a normal distribution, but the greater the number of people the closer the distribution tends to a normal distribution.

Random additions

On average, the components of a person’s height are made up at random (eg parents, nutrition, illnesses). In theoretical statistics there is a result (the Central Limit Theorem), which says that the sum of random numbers is normally distributed — if enough random numbers are summed.

The greater the number of sums we examine, the closer, again, we get to a normal distribution (the equivalent of the circle). These ideas are applied in the Standard Normal Curve program, written for mode I, so first examine the routine FN_NORMAL.

FN_NORMAL is a function which sums together random numbers from 0 to 1 (ie RND(1)), taking twelve of the random numbers at a time. The random numbers are summed in pairs, one being added to the accumulating total (V), and one being subtracted. There are two reasons for this adding and subtracting.

First, the mean (ie average) of the different sums will be zero in the long run and the standard deviation (ie how much the values vary) will be unity. Second, if there are any consistent biases in the random number (and I am not aware of any), this procedure helps to reduce biases.

FN_NORMAL produces what is known as a ‘standard normal deviate’, and is used in the routine PROC_SAMPLE (note that FN_NORMAL has a dummy parameter, totally unnecessary, but possibly of later use for variants of the function). PROC_SAMPLE is just that: it is a procedure to imitate the taking of a sample of values, the values it terms 1.

The parameter NUM gives the number of values in the sample, and the other parameter (CAT) gives the number of categories into which the values are to be grouped. The maximum of the values will be 6, and the minimum -6 (work that out), and so, if there are CAT categories, each will be 12/CAT units wide (call it WIDTH for the moment). To decide which is the category into which the value is put, the value (ie J) is divided by the WIDTH, I is added, and the result is integerised — ie INT(J*CAT/l2+1).

If the array in which the numbers of values are stored is V , then we increment the appropriate value by I (ie V (J) = V (J) + I). At the end of the routine the largest number in any element of the array is stored in V (0), and the final calculations of mean and standard deviation are made. When we leave this routine there are numbers stored in the array V (with the largest number stored in V(0)), and values for the mean and standard deviation.

These two routines copy (or simulate) the sampling of NUM values from a population of values, whose overall mean is zero, and whose standard deviation is unity.

Drawing the graph

There are two key routines for drawing: PROC_HISTOGRAM draws a histogram (it could be used for a bar chart); and PROC_FREQ draws a frequency polygon (the line joining the mid-points of the tops of the bars): both are called in PROC_HIST. The full gory details behind histograms and frequency polygons appear in most elementary statistics books.

PROC_HIST has four parameters: LOWER gives the vertical coordinate of the bottom of the graphs; UPPER gives the upper limit to the histogram; NUMBER gives the number of categories into which the values were placed; and SWITCH indicates whether the histogram and/or frequency polygon are to be printed (1 and 3 for histogram, 2 and 3 for frequency polygon).

The local variable ST is the horizontal coordinate at which the graphs start (39 for this example); WI gives the width (in coordinate units) of each bar, if the total width of the graphs is 1200; and HI is the height of the graphs from base to top. H SWITCH is 1 or 3 then a histogram is drawn by PROC—HISTOGRAM, and if SWITCH is 2 or 3 then a frequency polygon is drawn. This brings PROC_HIST to an end.

PROC_HISTOGRAM takes as parameters the left start, the width of the categories, the base and height of the graph, and the number of categories. For each category (ie I = 1 TO NUMBER) there is a call to PROC_BAR — then the routine ends with the graphics window being reset to the whole screen.

PROC BAR draws bars for charts and graphs, and the parameters are (in order): left coordinate, width, bottom coordinate, and height of bar. The parameters are modified to produce the correct parameters for VDU 24, ie set up a graphics window. The window is cleared with logical colour 3 (background 131), and a bar appears. This is the quickest way to draw rectangles for charts and similar designs.

PROC_FREQ had also been called by PROC_HIST. This routine calculates the midpoints of the bar tops, and joins them by a fine (apart from the first PLOT). The routine does not use relative plots, but rather absolute plots to preserve accuracy. One of the first actions of the routine is the resetting of the graphics window to the whole screen.

Initialization

To use all these routines we need to know how many are to be ‘sampled’, how many categories there are, and what graphs are wanted. We have to know SIZE, CATS, and SWITCH, as they are called in PROC_INIT.

PROC_INIT sets the text colour to logical 2 (which for modes I and 5 is yellow), and the background to logical I (or 129) which is red. (In PROC_HISTOGRAM the histogram is in white, and the frequency polygon is drawn in black). When the screen is cleared we have yellow writing on a red background.

At the top of the screen we have heading output, and then the user is asked for the sample size, followed by the number of categories, and then the value of the switch. The text screen is then set to the lower lines.

In the main program, after PROC_HIST, there is an *FX15,0 call, to flush buffers. I have found that, with programs which take some time to produce a result, there is a tendency to idly tap the keyboard — *FX15,0 removes idle taps.

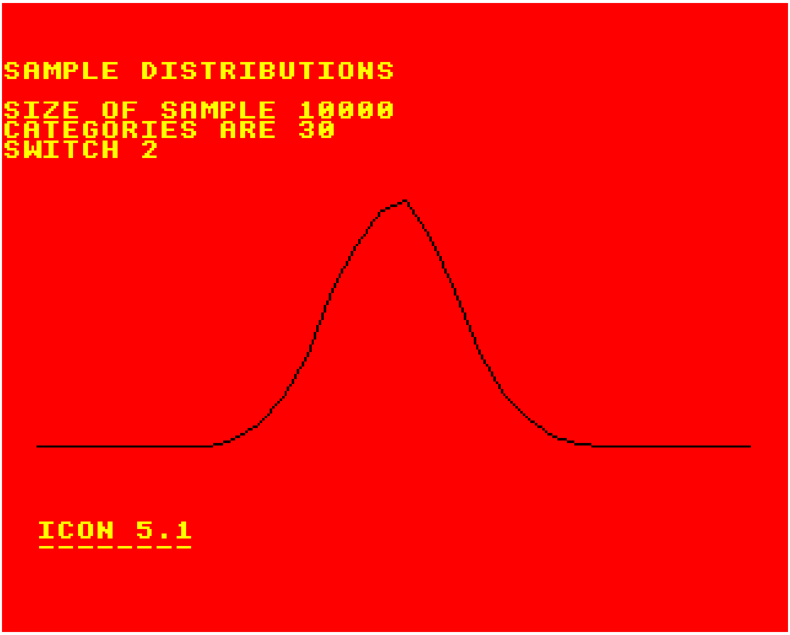
The last line of the main program (before END) sets the formatter (UG page 70, 325-327) @% to &01020307, ie

01 Strings formatted
02 Fixed format — fixed number of decimal places
03 Number of digits after decimal point
07 Field width for number

and then after the printing it is reset to &10 — the default. The mean and standard deviation are printed towards the bottom of the screen.

Icon 5.1 is an example of a very large sample (10000) and it is possible to see that for the simulation shown the result was a close approximation to a normal curve.

Experiment with the effects of different size samples, and different numbers of categories.



```
SAMPLE DISTRIBUTIONS

SIZE OF SAMPLE 10000
CATEGORIES ARE 30
SWITCH 2
```

```
ICON 5.1
--------
```

The real thing

The normal distribution has an exact mathematical form, the ‘height’ of the bar depending upon how far away from the mean is the bar (compared to the standard deviation).

As the standard deviation for the curve we are examining is unity and the mean is zero, the formula is very simple, and given as

```
1370 DEF FN_NORMAL(X) = EXP(-(X^2)/2)/SQR(2*PI)
```

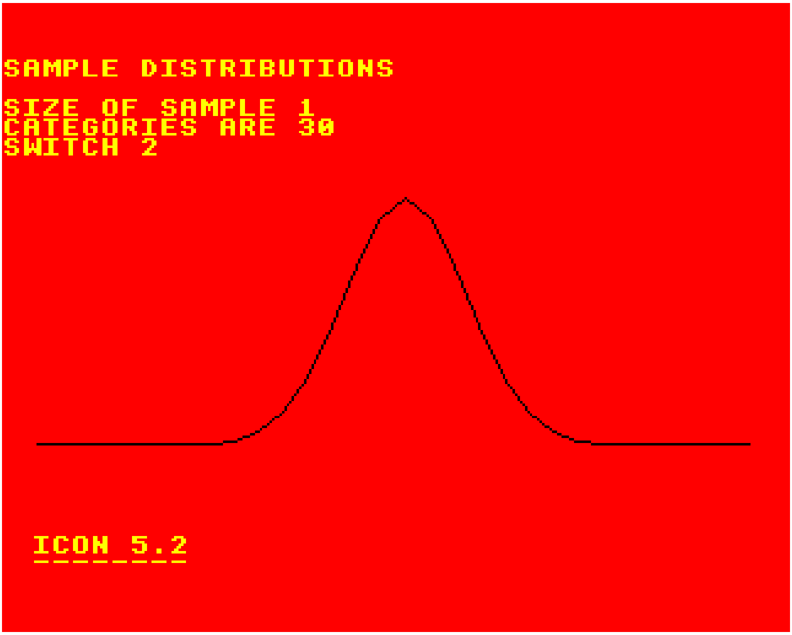
which explains why I had the dummy parameter X — I find that it is slightly tidier. The function now does not give the value sampled, but the probability (the height of the bar) that a value X will occur in a normal distribution.

The two routines PROC_HISTOGRAM and PROC_FREQ are highly general: PROC_HISTOGRAM can be used for bar charts other than histograms, for example; and PROC_FREQ can be used for the plotting of ordinary graphs. We will now see what this implies. PROC_SAMPLE has to be altered to

```
1420 DEF PROC_SAMPLE(NUM,CAT)
1430 LOCAL I,J,K: MEAN = 0 : SD = 0 : NUM = 0
1440 FOR I = -6 TO 6 STEP 12/CAT : J = FN—NORMAL(I) :
MEAN = MEAN + .1*1
1450 SD = SD + J*I*I: NUM= NUM + J :K = INT((I+6)*CAT
/12+ I)
1455 V (K) = J : NEXT I
1460 FOR J = 1 TO CAT : V(J) = V(J)/NUM : IF V(0)< V (J)
THEN V(0) = V (J)
1470 NEXT J
1480 MEAN = MEAN/NUM : SD = SD/NUM - MEAN*
MEAN
1490 ENDPROC : REM SAMPLE Version 2
```

and in this case the heights of the bars (ie J) are stored directly in the array (ie V(K)). The calculation of the mean and standard deviation has also to be modified (we have to cumulate the total of all the heights in NUM). Apart from that there is little real change.

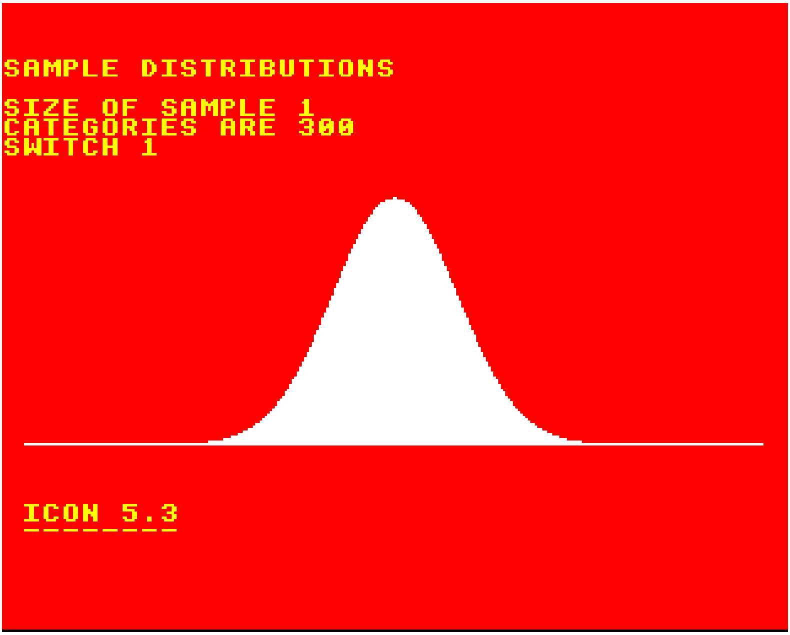
Icon 5.1 (that for a sample of 10000) is fairly close to a ‘proper’ normal distribution: how close is the ‘proper’ version? **Icon 5.2** shows the result of the proper version for the same number of categories as those used in **Icon 5.1** (ie 30 categories). Remembering that the scale goes from -6 to +6, this means that each category is 12/30, or .4 units ‘wide’ , but as the values rarely go beyond -3 to + 3, only about 15 categories are really used.



```
SAMPLE DISTRIBUTIONS

SIZE OF SAMPLE 1
CATEGORIES ARE 30
SWITCH 2


  ICON 5.2
  --------
```

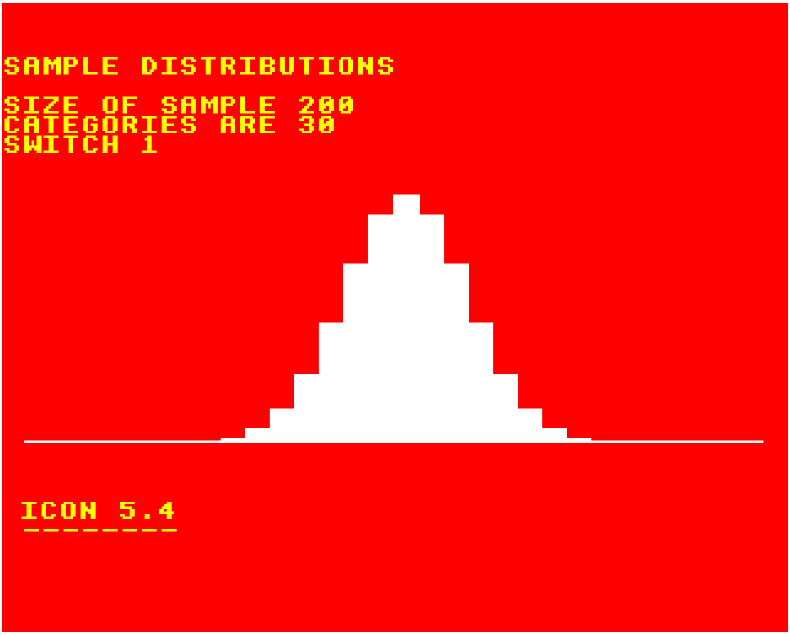


```
SAMPLE DISTRIBUTIONS

SIZE OF SAMPLE 1
CATEGORIES ARE 300
SWITCH 1




 ICON 5.3
 --------
```

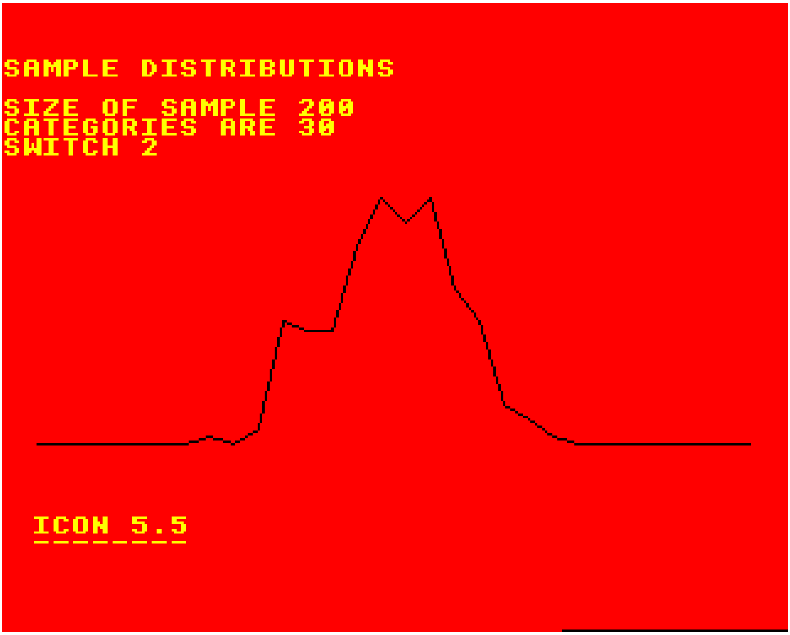


```
SAMPLE DISTRIBUTIONS

SIZE OF SAMPLE 200
CATEGORIES ARE 30
SWITCH 1




 ICON 5.4
 --------
```



```
SAMPLE DISTRIBUTIONS

SIZE OF SAMPLE 200
CATEGORIES ARE 30
SWITCH 2
```

```
ICON 5.5
--------
```

If we want to achieve a higher resolution for the graph (ie thinner bars) we can increase the number of categories. For mode 1, however, (and see Introduction) the maximum discrimination on the screen is four graphical units: this means that for a total width of 1200, we can have a maximum resolution of 1200/4 = 300. **Icon 5.3** shows the effects of the maximum resolution.

The chart in **Icon 5.3** differs from the two preceding icons because in this case I asked for the histogram (ie switch 1). If you compare **Icon 5.2**, in particular, to **Icon 5.1**, the differences are minor — though even **Icon 5.2** is slightly pointed itself, compared to **Icon 5.3** (that of the maximum resolution). Comparing **Icons 5.4** and **5.5** to **Icons 5.1** and **5.2** shows the effects of smaller samples.

With a sample of 10000 the result is close to the theoretical shape, but with either of the two different samples of 200 the matching is poor. **Icons 5.4** and **5.5** display a histogram and a frequency polygon (in that order) to show the ease of interpretation by the two methods. Remember that PROC_FREQ could just as easily be set up to plot a sine curve.

Oblique rectangles

To draw bars by use of the VDU 24 command is fine, and the best way, when the bars (or rectangles) are aligned along the horizontal and vertical axes. There is often a need to draw rectangles (filled in with colour) at angles to the axes. To draw these rectangles all we need are the Turtle Routines Version 1.2. Here is how to draw a rectangle

```
2000 DEF PROC_RECT ANGLE(BASE,HEIGHT)
2010 PROC_TURN(-90) : PROC_MOVE(BASE/2,0)
2020 PROC_TURN(l80) : PROC_MOVE(BASE,1)
2030 PROC_TURN(-90) : PROC_MOVE(HEIGHT,11) :
PROC_TURN(-90) : PROC_MOVE(BASE,11)
2040 PROC_TURN(-90) : PROC_MOVE(HEIGHT,11) :
PROC_TURN(-90) : PROC_MOVE(BASE/2,11)
2050 PROC_TURN(-90)
2060 ENDPROC : REM RECTANGLE
```

where the drawing starts at the middle of one of the bases. The turtle is turned through -90 degrees (ie directly right), and moves forwards through a distance equal to half the base (without plotting). The turtle is then turned directly totally around (ie 180 degrees) and then the rectangle is drawn (using the fill triangles style, 11). The final turn through -90 is to point in the original direction. To show how this routine can be used to produce effects, try

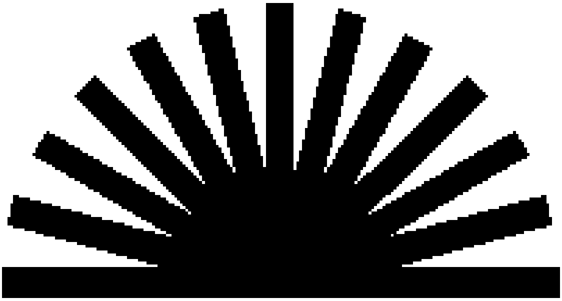
```
3000 DEF PROC_SHOWERS(LGTH)
3010 LOCAL BREADTHJ : BREADTH = LGTH/10
```

```
3020 PROC_TURN(90) : FOR I = 0 TO 12
3030 PROC_RECT ANGLE(BREADTH,LGTH) :
PROC_TURN(-15)
3040 NEXT I
3050 ENDPROC : REM SHOWERS
```

to be activated by

```
PROC START : PROC_SHOWERS(400)
```

which produces the effect of **Icon 5.6**. Remember that these routines are designed for mode 4, and so colours are not possible. Work out why the routine is called PROC_£HOWERS (or what a shower).



ICON_5.6

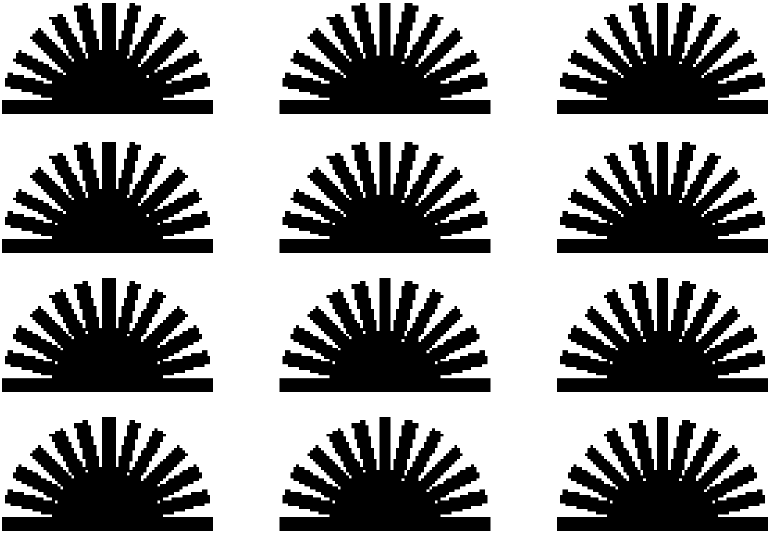
A further modification might be

```
4000 DEF PROC_DOWNPOUR
4010 LOCAL I,J
4020 FOR I = -400 TO 400 STEP 400
4030 FOR J = -400 TO 200 STEP 200
4040 PROC_MOVETO(I,J,0) : PROC_TURNTO(0)
4050 PROC_SHOWERS(150)
4060 NEXT J : NEXT I
4070 ENDPROC : REM DOWNPOUR
```

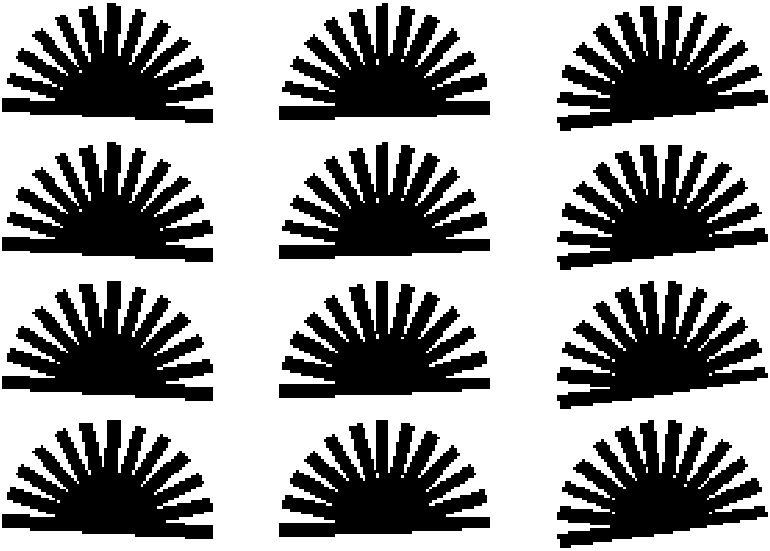
and the effects of the downpour are shown in **Icon 5.7**. The result in **Icon 5.8** is slightly different from that obtained from the above routine. I decided to make the shower slightly drunken, and so used PROC TURNTO(6*I/400 + 6*(100 + 5)/300. All that this shows is how easy it is to modify turtle routines, to produce a drunken shower.

With the correct approach, graphs and charts present no difficulty to the programmer — the difficult aspect is the understanding of the problem in the first case. Remember, it was Disraeli who first said ‘There are lies, damned lies, and statistics’.

ICON_5.7



ICON_5.8



```
1000REM-------------------------------
```

```
 1010
 1020
 1030REM   G R A P H I C   ART
 1040
 1050REM   (c) Boris Allen, 1983
 1060
 1070
 1080REM----------------------------
 1090
 1100REM   Standard Normal Curve
 1110
 1120REM----------------------------
 1130
 1140 MODE 1
 1150 PROC_INIT
 1160 PROC_SAMPLE(SIZE,CATS)
 1170 PROC_HIST(300,700,CATS,SWITCH)
 1180 *FX15,0
 1190 @% = &01020307 : PRINT'"MEAN IS ";
MEAN" SD IS ";SQR(SD)' : @%=10
 1200 END
 1210
 1220 DEF PROC_BAR(A,B,C,D)
 1230 LOCAL a,b,c,d
 1240  a = A : b = C : c = A+B : d = C+D
 1250 VDU 24,a;b;c;d;
 1260 GCOL 0,131
 1270 CLG
 1280 ENDPROC : REM BAR
 1290
 1300 DEF PROC_FREQ(X,INC,NUM,BASE,ROOF)
 1310 LOCAL I,H : H = ROOF-BASE
 1320 VDU 24,0;0;1279;1023; : PLOT4,X,(B
ASE+H*(V(1)/V(0)))
 1330 GCOL 0,0
 1340 FOR I=2 TO NUM : PLOT5,X+INC*(I-1)
,H*V(I)/V(0)+BASE : NEXT I
 1350 ENDPROC : REM FREQ
 1360
 1370 DEF FN_NORMAL(X)
 1380 LOCAL V,I : V = 6
 1390  FOR I = 1 TO 6 : V = V +RND(1)-RN
D(1) : NEXT I
 1400 = V : REM NORMAL
 1410
 1420 DEF PROC_SAMPLE(NUM,CAT)
 1430 LOCAL I,J : MEAN = 0 : SD = 0
 1440 FOR I = 1 TO NUM : J = FN_NORMAL(I
) : MEAN = MEAN + J
```

```
 1450 SD = SD + J*J : J = INT(J*CAT/12+1
) : V(J) = V(J) + 1 : NEXT I
 1460 FOR J = 1 TO CAT : V(J) = V(J)/NUM
 : IF V(0)<V(J) THEN V(0) = V(J)
 1470 NEXT J
 1480 MEAN = MEAN/NUM : SD = SD/NUM - ME
AN*MEAN : MEAN = MEAN - 6
 1490 ENDPROC : REM SAMPLE
 1500
 1510 DEF PROC_INIT
 1520 COLOUR 2 : COLOUR 129 : CLS
 1530 PRINT '''"SAMPLE DISTRIBUTIONS "
 1540 INPUT '"SIZE OF SAMPLE "SIZE
 1550 INPUT "CATEGORIES ARE "CATS : DIMV
(CATS)
 1560 INPUT "SWITCH "SWITCH
 1570 VDU 28,0,31,39,26
 1580 ENDPROC : REM INIT
 1590
 1600 DEF PROC_HIST(LOWER,UPPER,NUMBER,S
WITCH)
 1610 LOCALST,WI,HI:ST=39:WI=1200/NUMBER
:HI=UPPER-LOWER
 1620  IF SWITCH MOD 2 = 1 THEN PROC_HIS
TOGRAM(ST,WI,LOWER,HI,NUMBER)
 1630  IF SWITCH DIV 2 = 1 THEN PROC_FRE
Q(ST+WI/2,WI, NUMBER, LOWER, UPPER)
 1640 ENDPROC : REM HIST
 1650
 1660 DEF PROC_HISTOGRAM(ST,WI,LOWER,HI,
NUMBER)
 1670 LOCAL I : FOR I = 1 TO NUMBER
 1680 PROC_BAR(ST+(I-1)*WI,WI,LOWER,HI*(
V(I)/V(0)))
 1690 NEXT I : VDU 24,0;0;1279;1023;
 1700 ENDPROC : REM HISTOGRAM
 1710
```

CHAPTER 6

Turtle Graphics III

> Treat nature in terms of the cylinder,
> the sphere, the cone, all in perspective.
>
> *Paul Cézanne*

You will have noticed that my icons are not exactly a true representation of the pictures you see on the screen. In **Icon 5.6** the rays should describe an exact semicircle, but my version is slightly squashed.

The simplest, and most logical, way of transforming shapes is therefore to change the axes, and their relative scaling. In the drawing of a rectangle (as in the last chapter), to draw a rectangle by PROC_RECTANGLE (SIDE,RATIO*SIDE) is to produce a square if RATIO is unity.

Rectilinear coordinates

If our axes are at right angles, and the distance between P and P + INC is a\ways the same as the distance between Q and Q + INC (for any values of P an& Q), then the axes axes called rectilinear.

This does not mean that the scale of the axes tor both horizontal and vertical is the same, just that for each axis the scale is regular — rectilinear coordinates are the ones we use when we start coordinate geometry. Such axes are very simple to implement, as long as we concentrate on the intrinsic, and do not get carried away with extrinsic considerations.

Normally, to implement even simple transformations (when a transformation is simply a stretch for the moment) requires the use of transformation matrices — far too tedious, unartistic, almost arthritic. Examine **Turtle Graphics 3.1**.

```
1000 REM------------------------------
1010
1020
1030 REM   G R A P H I C   ART
1040
1050 REM   (c) Boris Allen, 1983
1060
1070
1080 REM------------------------------
1090
1100 REM   Turtle Graphics : 3.1
1110
```

```
 1120 REM-------------------------------
 1130
 1140 DEF PROC_CLRSCR
 1150 PROC_CLS : PROC_CLG
 1160 ENDPROC : REM CLRSCR
 1170
 1180 DEF PROC_CLG
 1190 GCOL 0,PEN : GCOL 0,129-PEN
 1200 VDU 24,0;128;1279;1023; : CLG
 1210 REM Clears an upper graphics windo
w
 1220 VDU 29,640;566;
 1230 REM Sets the origin to centre of g
raphics window
 1240 ENDPROC : REM CLG
 1250
 1260 DEF PROC_CLS
 1270 COLOUR 1-PEN : COLOUR 128+PEN
 1280 VDU 28,0,31,39,28 : CLS
 1290 REM Clears lower text window
 1300 ENDPROC : REM CLS
 1310
 1320 DEF PROC_COL(PE)
 1330 PEN=PE
 1340 ENDPROC : REM COL
 1350
 1360 DEF PROC_CENTRE
 1370 MOVE 0,0 : ANGLE=0 : X=0 : Y=0
 1380 ENDPROC : REM CENTRE
 1390
 1400 DEF PROC_RESTART
 1410 PROC_CLG : PROC_CENTRE
 1420 ENDPROC : REM RESTART
 1430
 1440 DEF PROC_START
 1450 PROC_COL(0) : PROC_CLRSCR : PROC_C
ENTRE
 1460 PROC_TRANSFORMATION(1)
 1470 ENDPROC : REM START
 1480
 1490 DEF PROC_INVERT
 1500 PEN=1-PEN : GCOL 0,PEN
 1510 ENDPROC : REM INVERT
 1520
 1530 DEF PROC_TURNTO(A)
 1540 ANGLE=FN_ANGLE(A)
 1550 ENDPROC : REM TURNTO
 1560
 1570 DEF PROC_TURN(A)
 1580 ANGLE = FN_ANGLE(ANGLE+A)
 1590 ENDPROC : REM TURN
 1600
 1610 DEF PROC_LOC
 1620 PRINT "COORDINATES ARE ";X,Y'"ANGL
E IS "ANGLE
 1630 ENDPROC : REM LOC
 1640
 1650 DEF PROC_MOVE(DISTANCE,STYLE)
 1660 X=X - DISTANCE*SIN(RAD(ANGLE))
 1670 Y=Y + DISTANCE*COS(RAD(ANGLE))
 1680 IF STYLE=1 THEN DRAW STRETCH*X,Y E
LSE MOVE STRETCH*X,Y
 1690 ENDPROC : REM MOVE
 1700
 1710 DEF PROC_MOVETO(XN,YN,STYLE)
 1720 LOCAL XDIF,YDIF : XDIF=XN-X : YDIF
=Y-YN
 1730 IF YDIF<>0 THEN PROC_TURNTO(DEG(AT
N(XDIF/YDIF))+180*(YN<Y)) ELSE PROC_TURN
TO(SGN(-XDIF)*90)
 1740 X=XN : Y=YN
 1750 IF STYLE=1 THEN DRAW STRETCH*X,Y E
LSE MOVE STRETCH*X,Y
 1760 ENDPROC : REM MOVETO
 1770
 1780 DEF FN_ANGLE(A)
 1790 IF A MOD 360 <0 THEN =A MOD 360 +
360 ELSE =A MOD 360
 1800 REM ANGLE
 1810
 1820 DEF PROC_NEW
 1830 VDU 26 : CLS
 1840 ENDPROC : REM NEW
 1850
 1860 DEF PROC_TRANSFORMATION(RATIO)
 1870 STRETCH = RATIO
 1880 ENDPROC : REM TRANSFORMATION
 1890
 1900 DEF PROC_SQUARE(SIDE)
 1910 LOCAL I : FOR I = 1 TO 4
 1920 PROC_MOVE(SIDE,1) : PROC_TURN(90)
 1930 NEXT I
 1940 ENDPROC : REM SQUARE
 1950
 1960 DEF PROC_SQUAREROT(SIDE,INC)
 1970 LOCAL I : FOR I = 0 TO 360 STEP IN
C
 1980 PROC_SQUARE(SIDE) : PROC_TURN(INC)
```

```
1990 NEXT I
2000 ENDPROC : REM SQUAREROT
2010
2020 DEF PROC_CIRCLE(INC)
2030 LOCAL I : FOR I = 1 TO 30
2040 PROC_MOVE(INC,1) : PROC_TURN(12)
2050 NEXT I
2060 ENDPROC : REM CIRCLE
2070
```

Essentially **Version 3.1** is a simple modification of **Version 1.1** (though it is possible to modify **Version 1.2** — try it). The principal differences are to the PROC MOVE and PROC_MOVETO routines, with the addition of a new routine PROC_TRANSFO RMATION. These routines include also three example routines: PROC_SQUARE, PROC_SQUAREROT, and PROC_CIRCLE.

First, the totally new routine PROC_TRANSFO RMATION. This routine has a parameter RATIO, the value of which is assigned to the global variable STRETCH. As it is not possible to use a variable until it has been initialised to some value, PROC_START has to be modified to include the call PROC TRANSFORMATION(0).

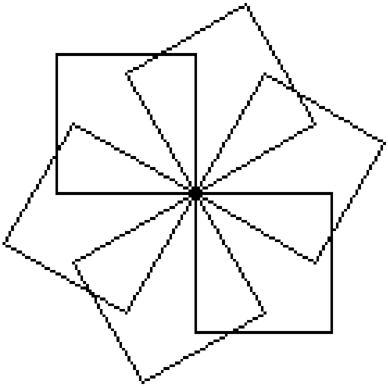
The purpose of STRETCH becomes clearer if PROC MOVE is studied. The values of X and Y are calculated as normal, but the plotting is to values which are functions of X and Y. In the case of Y, the function is merely Y, but for X the function is STRETCH*X. The angle is not altered, because X and Y are not altered. This is a ‘linear’ transformation because a straight line in the old coordinate system is changed to a straight line in the new system.

The changes to PROC_MOVETO mirror the changes to PROC MOVE.

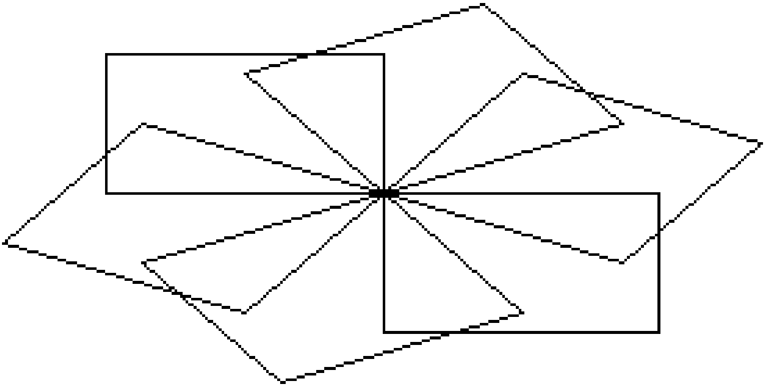
Slipping and sliding

Icon 6.1 shows the operation of PROC_SQUAREROT(200,60), that is, squares of side 200 each turning through 60 degrees from the previous square. A simple enough effect.

If a call is made to PROC_TRANSFORMATION(2), and the squares routine is repeated with the same parameters, then we find the result as shown in **Icon 6.2**. There are two distinct rectangles (where in **Icon 6.1** the squares were parallel to the axes), and several parallelograms.



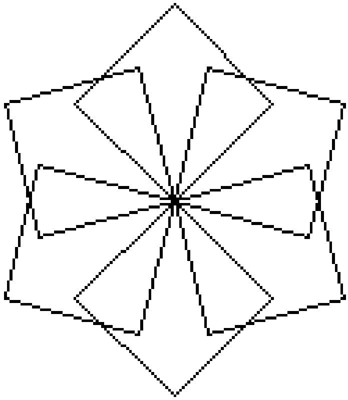
ICON_6.1



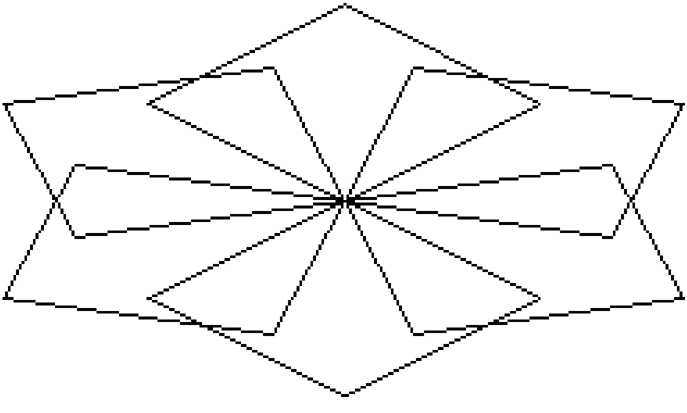
ICON_6.2

In linear transformation, squares become parallelograms because straight fines are still straight lines.

To study a slightly different effect, before PROC_SQUAREROT is called we PROC_TURN(-45) to produce **Icon 6.3**. When transformed as before, the result is as in **Icon 6.4**. The two squares which in **Icon 6.3** are symmetrical about the Y axis are (in **Icon 6.4**) rhombuses (ie equilateral parallelograms.

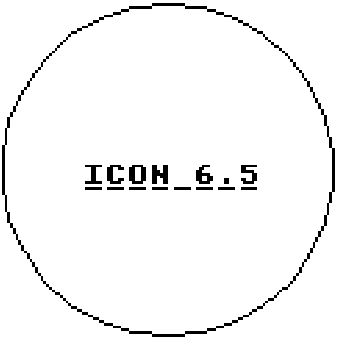


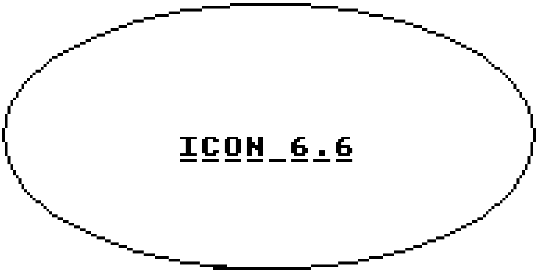
ICON_6.3



ICON_6.4

When a circle is drawn by PROC_CIRCLE, a circle is drawn when the transformation is 1 (in **Icon 6.5** the circle is a little squashed). As soon as the transformation is 2, the result is that of **Icon 6.6**. An ellipse is no more than a stretched (or squashed) circle.





Icon 6.7 shows another transformed circle, when the transformation is not linear.

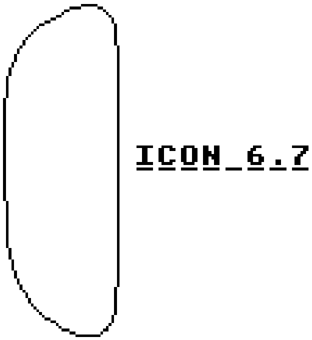
Complex transformations

The ‘circle’ in **Icon 6.7** was drawn using the **Turtle Graphics 3.2** routines. These routines are again modifications to **Version 1.1**, but with a great number of changes.

```
 1000 REM-------------------------------
 1010
 1020
 1030 REM   G R A P H I C   ART
 1040
 1050 REM   (c) Boris Allen, 1983
 1060
 1070
 1080 REM-------------------------------
 1090
 1100 REM   Turtle Graphics : 3.2
 1110
 1120 REM-------------------------------
 1130
 1140 DEF PROC_CLRSCR
 1150 PROC_CLS : PROC_CLG
 1160 ENDPROC : REM CLRSCR
 1170
 1180 DEF PROC_CLG
 1190 GCOL 0,PEN : GCOL 0,129-PEN
 1200 VDU 24,0;128;1279;1023; : CLG
 1210 REM Clears an upper graphics windo
w
 1220 VDU 29,640;566;
 1230 REM Sets the origin to centre of g
raphics window
 1240 ENDPROC : REM CLG
 1250
 1260 DEF PROC_CLS
 1270 COLOUR 1-PEN : COLOUR 128+PEN
 1280 VDU 28,0,31,39,28 : CLS
 1290 REM Clears lower text window
 1300 ENDPROC : REM CLS
 1310
 1320 DEF PROC_COL(PE)
 1330 PEN=PE
 1340 ENDPROC : REM COL
 1350
 1360 DEF PROC_CENTRE
 1370 X=0 : Y=0 : MOVE FN_XAXIS(0),FN_YA
XIS(0) : ANGLE=0
 1380 ENDPROC : REM CENTRE
 1390
 1400 DEF PROC_RESTART
 1410 PROC_CLG : PROC_CENTRE
 1420 ENDPROC : REM RESTART
 1430
 1440 DEF PROC_START
 1450 PROC_COL(0) : PROC_CLRSCR : PROC_C
ENTRE
 1460 ENDPROC : REM START
 1470
 1480 DEF PROC_INVERT
 1490 PEN=1-PEN : GCOL 0,PEN
 1500 ENDPROC : REM INVERT
 1510
 1520 DEF PROC_TURNTO(A)
 1530 ANGLE=FN_ANGLE(A)
 1540 ENDPROC : REM TURNTO
 1550
 1560 DEF PROC_TURN(A)
 1570 ANGLE = FN_ANGLE(ANGLE+A)
 1580 ENDPROC : REM TURN
 1590
 1600 DEF PROC_LOC
 1610 PRINT "COORDINATES ARE ";X,Y'"ANGL
E IS "ANGLE
 1620 ENDPROC : REM LOC
 1630
 1640 DEF PROC_MOVE(DISTANCE,STYLE)
 1650 LOCAL I,IX,IY,SX,SY : SX = X : SY
= Y
 1660 SX=X - DISTANCE*SIN(RAD(ANGLE))
 1670 SY=Y + DISTANCE*COS(RAD(ANGLE))
 1680 IF ABS(X-SX)>ABS(Y-SY) THEN D = IN
T(ABS(X-SX)/12)+1 ELSE D = INT(ABS(Y-SY)
/12)+1
 1690 IX = (SX-X)/D : IY = (SY-Y)/D
 1700 FOR I = 1 TO D
 1710 X = X+IX : Y = Y + IY
 1720 IF STYLE=1 THEN DRAW FN_XAXIS(X),F
N_YAXIS(Y) ELSE MOVE FN_XAXIS(X),FN_YAXI
S(Y)
 1730 NEXT I
 1740 X = SX : Y = SY
 1750 ENDPROC : REM MOVE
 1760
 1770 DEF PROC_MOVETO(XN,YN,STYLE)
 1780 LOCAL XDIF,YDIF,D,I : XDIF=XN-X :
YDIF=Y-YN
 1790 IF YDIF<>0 THEN PROC_TURNTO(DEG(AT
N(XDIF/YDIF))+180*(YN<Y)) ELSE PROC_TURN
TO(SGN(-XDIF)*90)
 1800 IF ABS(XDIF)>ABS(YDIF) THEN D = IN
T(ABS(XDIF)/12)+1 ELSE D = INT(ABS(YDIF)
/12)+1
```

```
 1810 XDIF = (XN-X)/D : YDIF = (YN-Y)/D
 1820 FOR I = 1 TO D
 1830 X = X + XDIF : Y = Y + YDIF
 1840 IF STYLE=1 THEN DRAW FN_XAXIS(X),F
N_YAXIS(Y) ELSE MOVE FN_XAXIS(X),FN_YAXI
S(Y)
 1850 NEXT I
 1860 X=XN : Y=YN
 1870 ENDPROC : REM MOVETO
 1880
 1890 DEF FN_ANGLE(A)
 1900 IF A MOD 360 <0 THEN =A MOD 360 +
360 ELSE =A MOD 360
 1910 REM ANGLE
 1920
 1930 DEF PROC_NEW
 1940 VDU 26 : CLS
 1950 ENDPROC : REM NEW
 1960
 1970 DEF FN_XAXIS(COORD)
 1980 =COORD
 1990 REM XAXIS - modify preceding line
 2000
 2010 DEF FN_YAXIS(COORD)
 2020 =COORD
 2030 REM YAXIS - modify preceding line
 2040
 2050 DEF PROC_SQUARE(SIDE)
 2060 LOCAL I : FOR I = 1 TO 4
 2070 PROC_MOVE(SIDE,1) : PROC_TURN(90)
 2080 NEXT I
 2090 ENDPROC : REM SQUARE
 2100
 2110 DEF PROC_TRIANGLE(SIDE)
 2120 LOCAL I : FOR I = 1 TO 3
 2130 PROC_MOVE(SIDE,1) : PROC_TURN(120)
 2140 NEXT I
 2150 ENDPROC : REM TRIANGLE
 2160
 2170 DEF PROC_CIRCLE(INC)
 2180 LOCAL I : FOR I = 1 TO 30
 2190 PROC_MOVE(INC,1) : PROC_TURN(12)
 2200 NEXT I
 2210 ENDPROC : REM CIRCLE
 2220
 2230 DEF PROC_LINES
 2240 LOCAL I
 2250 FOR I = 0 TO 90 STEP 2
 2260 PROC_MOVETO(-450,-300,0) : PROC_TU
```

```
RNTO(-I) : PROC_MOVE(1500,1)
 2270 NEXT I
 2280 ENDPROC : REM LINES
```



What has to be performed is actually rather mundane. All that is needed is to set up two functions (one for the X axis and one for the Y axis) which convert the actual values of X and Y to screen coordinates. The change from the actual value of X to the screen value of STRETCH*X is a simple example.

Start at the beginning. Going through the routines, the first to be altered is PROC_CENTRE and it is altered in two ways. The initialisations of X and Y are placed at the beginning of the routine (rather than at the end) because sometimes the function routines use X and Y explicitly. The function routines FN_XAXIS and FN_YAXIS initialise the cursor to the centre of the transformed screen.

The real complexities arise in the coding of PROC_MOVE — complexities which ease the way for the user. We need many more local variables, because of the storage of many more interim values. The variables SX and SY are calculated to be the endpoints of the line to be drawn, by the same method as before.

To draw a fine between two points in a rectilinear coordinate geometry is to draw a straight fine (a pretentious way of saying that in ordinary geometry the shortest distance between two points is a straight fine). To draw a line between two points on the surface of the Earth is to draw a curve — though at each point you may think that you are following a straight line. (See for example, Klein's (1953) chapter on 'New Geometries, New Worlds'.)

To draw the shortest fine between two points is to follow a 'geodesic', and the path depends on the geometry. In ordinary geometry the path is a straight fine, in some of the other geometries we will investigate, it is anything but straight. The way we draw a line is a portion at a time, at each time producing a straight fine. As we found with a circle in ordinary geometry, a series of straight fines can produce a curve.

We have to decide on how many little straight fines we need to draw. We decide this by noting that if we move (slowly!) at about three pixels at a time, the fine is an almost perfect curve. if the difference in the X direction is greater than the difference in the Y direction, then the X difference is divided by 12 units to produce the number of steps (D) (else the difference is used).

When D has been calculated, D is used to work out the increments in the X and Y directions (ie IX and IY). These increments are then used to plot D straight fines, where the plotting is to FN_XAXIS(X),FN_YAXIS(Y). The end values (SX and SY) are then assigned to X and Y, to prevent the overaccumulation of rounding errors.

PROC_MOVETO in this version is far simpler to understand than PROC_MOVE, particularly when PROC_MOVE has been studied.

The transformations appear at FN_XAXIS and FN_YAXIS, and by default they do nothing other than return the value of X and Y.

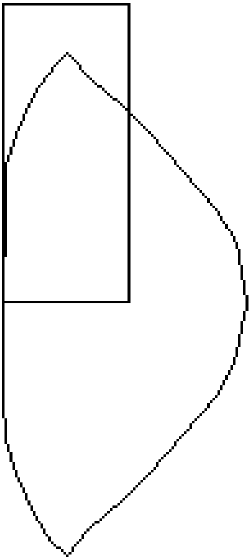
A new geometry

Icon 6.7 was drawn with a strange geometry:

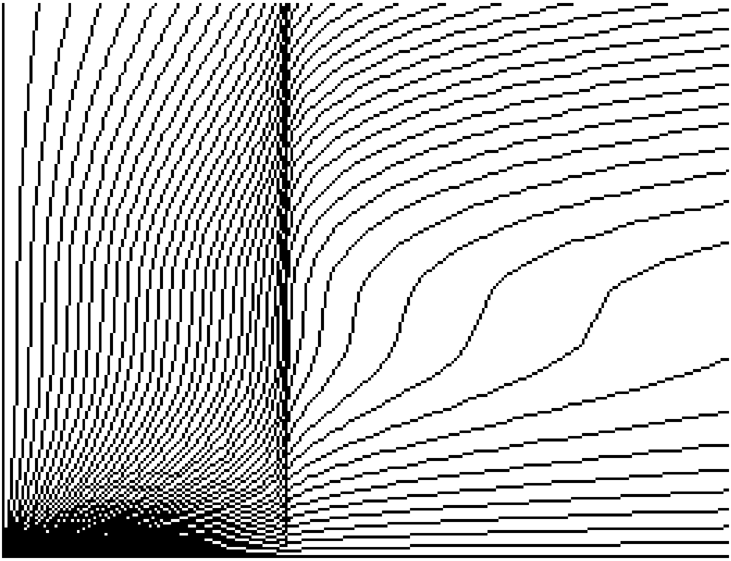
```
DEF FN_XAXIS(COORD) = SGN(COORD)*COOREY2/500
DEF FN_YAXIS(COORD) = SGN(COORD)*SQR(ABS(COOR
D)) *20
```

which stretches the scale more at larger values of X, and squashes the scale more at larger values of Y. These two coordinate transformations give a very non-rectilinear geometry.

Icon 6.8 shows two squares. One appears as a rectangle, and this is a square parallel to the axes. The other is a square which is symmetrical around the X axis. The routine to draw the square is PROCL-SQUARE.



ICON_6.8



ICON_6.9

If PROC_LINES is activated when the coordinates are rectilinear (ie both functions = COORD) then we have a series of straight lines radiating from the point -400,-300 in the upper right quadrant. It is a trifle like the showers example, and has similarities to the Moire example in the **Version 2.1** graphics.

Icon 6.9 has no resemblance to anything, so it would appear. Each fine radiating from the point -400,-300 is a geodesic — the shortest distance between two points in the geometry defined above. What happens when a line is drawn?

At each stage of the curve there is a short straight line which follows in the direction given by the geometry at that point. As with the circular geometry (straight line, turn, straight line, turn) a curve then appears — with the straight portions carefully hidden by our eyes. Of course, a straight fine is only a special form of curve.

With this pair of axis transformations we have effectively defined a ‘force field’, and our turtle follows the fines of force between points. Get the axes correctly defined, and we might have an Einsteinian force field. (Remember Abelson and diSessa?) Let us examine this force field.

The lines radiate from the bottom left corner, starting straight upwards and moving clockwise. Until the fines reach the Y axis they look reasonably uncurved. They look mainly straight until the line crosses the axis (note that crossing the X axis does not seem to be at all traumatic). At the Y axis something strange happens.

As the fines become closer to the Y axis, so they become more bent and seem almost determined not to cross the axis. The fines come very close together, and once past the axis they change direction, and appear more at fight angles to the axis. The effect is strange to watch.

Once over the other side, there seems to be abnormal behaviour about the region of the X axis (though is any behaviour normal?). To go through the X axis on the left side of the Y axis does not produce any kinks in the fines. For some fines, a kink appears on the positive side. Setting the starting point to other coordinates will produce different effects.

All this reveals the truth of Kasner and Newman’s quote (1949 page 163) ‘. . . — our intuitive notions about space almost invariably lead us astray’. Funnily, they were talking of geodesics, but they were interested in spiders not turtles.

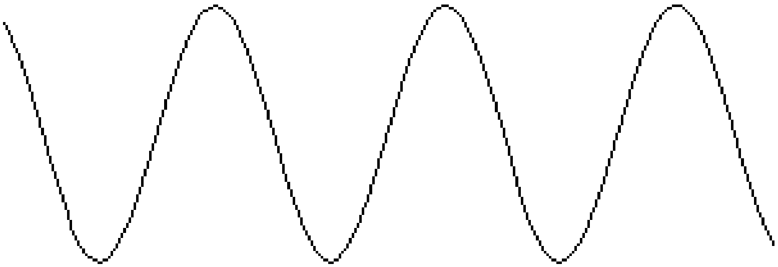
Parametric functions

Icon 6.10 is merely a sine curve. The interesting thing about that sine curve is the way in which it was drawn.

The functions were defined by

```
DEF FN_XAXIS(COORD) = COORD
DEF FN_YAXIS(COORD) = 200*SIN(RAD(X))
```

or, perhaps more illuminatingly,



ICON_6.10

```
DEF FN_XAXIS(COORD) = X
DEF FN_YAXIS(COORD) = 200*SIN(RAD(X))
```

and to draw that line I entered

```
PROC_MOVETO(-600,0,0) : PROC_MOVETO(600,0,1)
```

which moved the turtle/cursor to X equal to -600, Y equal to 200* SIN(RAD(X)), and not Y = 0. The value in the second parameter of PROC_MOVETO is a dummy, that is, it is not used as such, it has been by-passed by the function definition.

A line was then drawn to X = 600, a geodesic in the sine geometry.

We now have a way to draw graphs of functions, but very easily. To draw a parabola (ie X = k*Y^2) we use the definitions

```
DEF FN_XAXIS(COORD) = Y
DEF FN_YAXIS(COORD) = Y^2/400
```

and produce **Icon 6.11**. This is an extremely flexible way of drawing graphs. To produce the parabola I entered

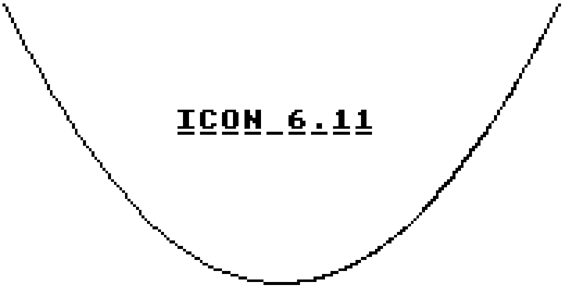
```
PROC_MOVETO(0,400,1)
```

and half of the parabola (the upper half) is drawn. When I then entered

```
PROC_MOVETO(0,-400,1)
```

the upper is retraced and the lower half drawn in. This way of analysing functions is clearly of great assistance, and fits well with turtle graphics, especially Version 3.2. We use the fact that the graph of a function is geodesic, the fine between two points, as given by the geometry (ie the axes).

The technique we are now using is called the ‘parametric form’ of displaying graphs. This technique, which in many ways is simpler than other forms, is not taught at lower levels in schools — being somehow more ‘difficult’ or ‘esoteric’. The parametric form is used quite extensively in ‘advanced’ graphics, it is one way of simplifying, or, as we would say, accentuating the intrinsic and minimising the effects of the extrinsic.


ICON_6.11

CHAPTER 7
Generative Graphics

> . . . the coming of semi-intelligent
> machines into business and technology
> had created a second Industrial
> Revolution, in which only the most
> highly creative human beings, and those
> most gifted at administration, found
> themselves with any skills to sell which
> were worth the world's money to buy.
>
> James Blish, A Life for the Stars

Computers allow us to be creative — Seymour Papert calls them the proteus of machines, that is, machines which are more flexible and more adaptable than any others have been, or could be.

I have tried to show how the BBC computer can be used in a highly creative manner. The creativity comes from the individual, and the way to enhance one's own creativeness. is to try to be creative. Creativity does not come from following rules, it comes from trying to extend and improve, as I hope you improve upon my efforts.

The saintly books which try to teach good programming practice by the application of rules of (say) structured programming often annoy me. Books which are forever denigrating the use of GOTO are more concerned with academic ideals than actual practice. Sometimes the contortions needed to get round the use of a GOTO have to be read to be believed.

A careful examination of my routines will not reveal a GOTO, there was no need for a GOTO and — in creative programming on the BBC computer — there never need be. I do not use GOTO because in complex systems such as **Turtle Graphics 3.2** a GOTO would probably create more havoc than it saved. Note, however, that in PROC_NORT I effectively had a jump out of a routine to END — rules are there to be broken.

My advice, for what it is worth, is to steer clear of rules, try to get a feel for the topic, try to understand what are really the essential elements — accentuate the intrinsic, eliminate the extrinsic (if at all possible). The key to successful thought, never mind programming, is to divine the essence of the problem. Aristotle said that (sort of).

End of sermon.

Further work

Obvious extensions are the multi-coloured graphics to sixteen colours (that will produce fun and games when you try). To extend to the sixteen colours of mode 2 means that space will become at a premium, so the economy of the turtle graphics/intrinsic approach will become even more valuable.

PROC_NORT (I nearly called it PROC_FFORT) can easily be improved, in particular the keyboard sensing - eg using *FX11 and *FX12 commands. You might wish to change the angle turned from 90 degrees - that, at least, is easily done.

You will have most fun with the routines in **TG 3.2**. Why not implement a game in a new geometry, instead of boring Euclidean geometry (ie with rectilinear coordinates?). PROC_NORT in a multi-coloured strange geometry would be a game and a half. It would also be a worthwhile programming exercise.

My next BBC graphics book will move on to three (and more dimensions) for even more flamboyant games effects, and it will analyse animation in far greater detail: what is to stop you getting there first?

What to read

I have referred to several books within the body of this book, and I list them here - in sufficient detail that you may if you wish order them from a library, or possibly buy. I have not yet found a reasonable book on mathematics applicable to computers, and so — until I write it — do the best you can. The Kasner and Newman is an old book, but it has a lot to offer.

As I am not too good on the alphabet (though I know it goes from ASCII 65 to ASCII 90, in capitals) the list is given in almost an order of reading: I assume you have the User Guide.

Seymour PAPERT, *Mindstorms*, The Harvester Press, 1980.

Edward KASNER and James NEWMAN, *Mathematics and the Imagination*, Penguin Books, 1968 (originally published in 1940).

Morris KLINE, *Mathematics in Western Culture*, Penguin Books, 1972 (originally published in 1953).

Harold ABELSON and Andrea diSESSA, *Turtle Geometry*, MIT Press, 1980.

The last is last because it is very expensive, and more difficult to follow in later chapters.

My final recommendation is the Cities in Flight quartet by James Blish (see the above quotation). Reading it might explain the cities in flight on my covers (Collected edition, Arrow, 1981).

```
   10 REM*******************************
```

```
*************
   20 REM B O R I S ' S   P A R T I N G
  G I F T
   30 REM******************************
*************
   40
   50
31000 DEF PROC_SCANDUMP(XL%,YL%,XU%,YU%)
31010 LOCAL X%,Y%,I%,J%,B%
31020 XL% = 32*(XL% DIV 32 - 1) : YL% =
4*(YL% DIV 4 - 1)
31030 XU% = 32*(XU% DIV 32 + 1) : YU% =
4*(YU% DIV 4 + 1)
31040 J% = (YU% - YL%) DIV 4 : DIM N J%
31050 REM Bit images for a vertical line
 will be stored in N
31060 VDU 2, 1,12, 1,27, 1,51, 1,23
31070 REM Printer on, linefeed, and 23/2
16 inches per line
31080 FOR X% = XL% TO XU% STEP 32
31090 FOR Y% = YL% TO YU% STEP 4
31100 B% = 0 : FOR I% = 0 TO 31 STEP 4
31110 REM Store the bit image of 8 horiz
ontal pixels
31120 B% = B%*2 : IF POINT(X%+I%,Y%) >0
THEN B% = B% + 1
31130 NEXT I% : N%?((Y%-YL%) DIV 4) = B%
31140 REM The bit image is stored in suc
cessive elements of N
31150 NEXT Y% : VDU 1,27, 1,75, 1,(J%+1)
 MOD 256, 1, (J%+1) DIV 256
31160 REM Prepare printer for J%+1 bit i
mages from N
31170 FOR I% = 0 TO J% : VDU 1,N?I% : NE
XT I%
31180 VDU 1,13 : NEXT X%
31190 REM End the line with a return
31200 VDU 1,27, 1,50, 3 : REM Switch off
 printer
31210 ENDPROC : REM SCANDUMP
>
```

Other titles from Sunshine

THE WORKING SPECTRUM
David Lawrence
0 946408 00 9

THE WORKING DRAGON 32
David Lawrence
0 946408 01 7

THE WORKING COMMODORE 64
Keith Brain/Steven Brain
0 946408 02 5
0 946408 04 1

DRAGON 32 GAMES MASTER
Keith Brain/Steven Brain
0 946408 03 03

FUNCTIONAL FORTH
for the BBC Computer
Boris Allan

COMMODORE 64 MACHINE CODE MASTER
David Lawrence and Mark England

For further informaton contact:
Sunshine
12-13 Little Newport Street
London WC2R 3LD
01-734 3454

ADVANCED SOUND AND GRAPHICS
for the Dragon computer
Keith and Steven Brain
0 946408 06 8

SPECTRUM ADVENTURES
Tony Bridge and Roy Carnell
0 946408 07 6

THE DRAGON TRAINER
a handbook for beginners
Brian Lloyd
0 946408 09 2

COMMODORE 64 ADVENTURES
Mike Grace
0 946408 11 4

MASTER YOUR ZX MICRODRIVE
Programs, machine code and networking
Andrew Pennell
0 946408 19 X

Printed in England by Commercial Colour Press, London E7.